Synthesis of Data-Flow Interfaces for Regular Parallel Programs

Franck Bardoult, Patrice Quinton, Sanjay Rajopadhye, Tanguy Risset

INRIA
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

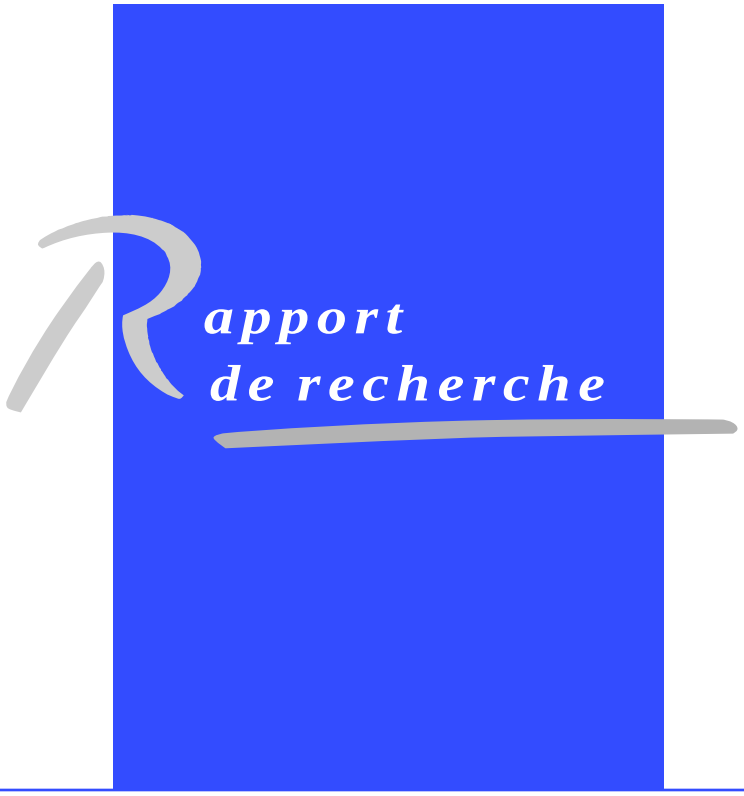
Synthesis of Data-Flow Interfaces for Regular Parallel Programs

Franck Bardoult, Patrice Quinton, Sanjay Rajopadhye and Tanguy Risset

N˚3760

Septembre 1999

THÈME 1



ISSN 0249-6399

RENNES

Synthesis of Data-Flow Interfaces for Regular Parallel Programs

Franck Bardoult, Patrice Quinton, Sanjay Rajopadhye and Tanguy Risset

Thème 1 — Réseaux et systèmes
Projet Cosi

Rapport de recherche n˚3760 — Septembre 1999 — 13 pages

Abstract: We present a method for generating an interface between an architecture executing a regular program and a host processor, during an hardware-software co-design process. The interface is generated by static analysis of a single assignment Alpha program and of its scheduling. This method is implemented in the MMAlpha design environment, and was experimented on a H261 image coder.

Key-words: Alpha, cosimulation, interface, data-flow

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00

Synthèse d'interfaces data-flow pour programmes réguliers parallèles

Résumé : Nous présentons une méthode pour générer une interface entre une architecture exécutant un programme régulier et un processeur hôte. Ceci s'effectue pendant une conception conjointe logiciel-matériel. L'interface est générée par une analyse statique d'un programme Alpha à assignation unique, et de son ordonnancement. La méthode s'applique dans l'environnement de conception MMAlpha, et a été expérimentée sur un codeur image H261.

Mots-clé : Alpha, cosimulation, interface, data-flow

1 Introduction

Regular parallel programs appear in many embedded applications. For example, most of the signal processing tasks in telecommunication appliances are organized around a kernel of regular algorithms such as matrix operations, linear system solvers, FFT, etc. In order to implement such systems on silicon, it is often needed to distribute the code between several processors. Typically, a control component is implemented as a software program on a Risc core or a DSP, and is interfaced to a computation intensive component which is implemented as dedicated hardware. This situation, although simplified, is a good generic framework for studying the general case of systems having a more complex distribution.

Designing such systems makes use of co-design techniques. Among the problems relevant to co-design, generating the interface between the software component and the hardware is certainly one of the most difficult. Ismail and Jerraya[8] use abstract communication units to exchange data, and associate protocols and controllers to these units during communication synthesis. In [11], Wenba, O'Leary and Brown describe and generate communication protocols by co-design. Processes interact via channels, and whenever an hardware interface is needed, the user has to provide its description and its driver. Chou, Ortega and Borriello[2] use detailed lists of processor and device ports to create the software and hardware glue between components. Gogniat et al.[6] propose a communication synthesis method for generic telecommunication architectures.

Here, we consider the situation where the hardware component implements a regular algorithm, and we choose to create a protocol that is directed by the scheduling of the algorithm and its input/output. In such a case, communication between the software and the hardware components consists of supplying the latter with the data needed to perform the next computation, and to receive the results: we call such an interface a *data-flow interface*. In a data-flow interface for a regular algorithm, the regularity of the communication pattern is such that one must be able to design it in a systematic way. Our goal is to try to communicate only what is needed for each step of the calculation.

In this paper, we show that it is possible to create a data-flow interface automatically, by collecting information in the regular program. We place ourselves in the framework where the regular algorithm – called *co-program* in the following, – is described by means of the data-parallel functional language Alpha[4] and is called using an imperative language such as C, – called *host program* in what follows. As explained in [3], this host program can also be derived from a real-time data-flow language such as Signal[7]. The reason for choosing Alpha to describe the regular part is two-fold. First, such a language allows information to be extracted more easily than in a conventional imperative language. Second, methods for deriving parallel architectures from Alpha are available (see [9]) and the co-design is therefore easier.

In our approach, semantic preserving transformations, in particular scheduling, are applied to the Alpha program in order to obtain the description of an architecture [9]. Scheduling is especially important, as it defines the communication pattern of the architecture: calculations are splitted in several steps, and at each step, different data are supplied to the co-processor. Once the scheduling is defined, the host program is modified in order to implement the interface to the hardware supporting the co-program.

The organization of this paper is as follows. In section 2, we present a motivating example which illustrates the main concepts of our approach. Section 3 provides the necessary background

on Alpha. In section 4, we show how the interface is generated. The protocol and its correction are described in section 5. Implementation details, applications, and a discussion are in section 6. Section 7 concludes the paper.

2 A motivating example

Consider a system in which a matrix-vector multiplication (MV in the following) has to be parallelized. In this system, MV is called at successive instants of time $t = 0, 1, ..., \tau$, etc., and we assume that a call to MV takes no time: in other words, the initial specification of this system is purely functional.

We assume MV to be expressed as an Alpha program, called the co-program. This Alpha program can be translated into a C program, and linked to the host program, thus providing a means to check its correctness by simulation. An alternative, not considered here, would be to have MV expressed using an imperative language, and translated into single-assignment code such as Alpha by means of a method such as that presented in [5].

From the initial Alpha program, a parallel architecture can be derived, using techniques presented in [4]. Here, we assume that the target is the systolic array shown in figure 1. The arguments of MV are a 4×4 matrix A, a vector B of size 4, and an output vector C. A four cell array is used to compute this product. Cell i computes the value $C(i)$. Each element of B is needed only once by a given cell, and coefficients $B(i)$ are pipelined from cell to cell as shown in the figure. Similarly, the coefficients $A(i, j)$ are input in the array in a regular fashion. At a given instant of time, a vector of coefficients $A(i, j)$ such that $i + j$ is constant enters the 4 cells. Such a vector of data will correspond to *a slice* in the following. Clearly, 8 steps are needed to perform the full computation (7 steps to input the $A(i, j)$, and one more step to produce the final result $C(4)$).

The operation of this architecture can be described by a new Alpha program where a re-indexing of the calculations has been performed: this re-indexing can be computed automatically by means of a scheduling algorithm. After this transformation, a call to MV is no longer instantaneous. To model this change in the timing, we consider that 8 sub-clock ticks are introduced in-between two calls to MV. Let us call each of these clock ticks a *micro-step*. The Alpha program is now considered as a process which communicates with the host program, receiving the input data needed to perform the current micro-step, and providing the results at the time when they are produced. This new representation is not purely functional, and describes, at a rather high level of abstraction, the interfacing of the host program and the co-processor.

In order to generate automatically such a new specification, we need to solve the following problems:

- Given a scheduled version of the initial Alpha program, determine the number of micro-steps necessary for doing the total calculation.
- For each micro-step, find out what data the host has to send to the co-program, and what data it needs to receive from it.
- Define a communication protocol between the host and the co-program, and show that this protocol is correct.

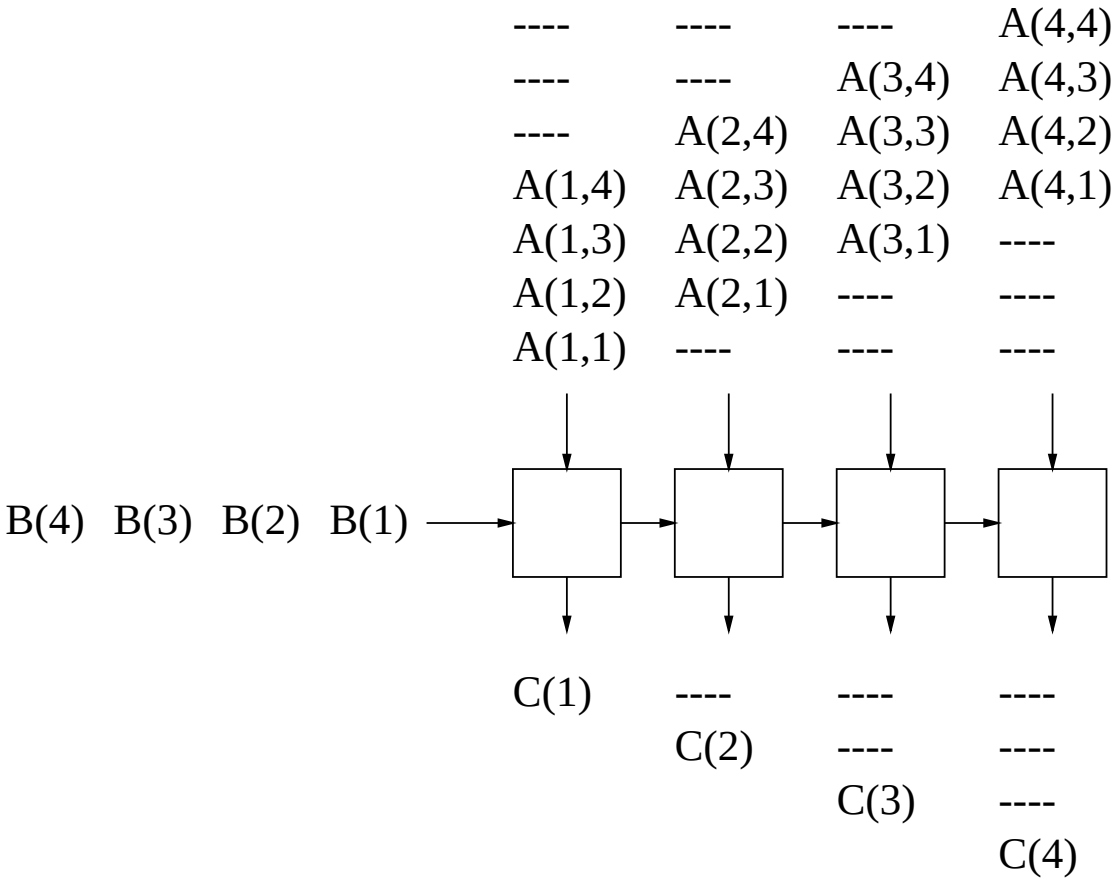


Figure 1: Systolic architecture for matrix-vector product, obtained by high level synthesis of a functional Alpha representation. For sake of concision, inputs (A and B) and output (C) are represented together. Their position indicates their input dates in (resp. output dates from) the array.

- Generate an interface which implements the protocol.

Notice that we do not consider here how to generate a scheduled program, how to prove that the new system is functionally equivalent to the initial one, neither do we explain how to generate a C program that simulates the Alpha scheduled program: these steps rely upon well-known techniques (see [10])

3 Background

In this section, we describe the structure of an Alpha program, and we provide the background information necessary to understand the remaining of this paper.

system simple (I : $Dom(I)$ **of integer**)
    returns (O : $Dom(O)$ **of integer**);
var
  C : $Dom(C)$ **of integer**;
let
  $C = f(I)$;
  $O = g(C)$;
tel;

Figure 2: Generic Alpha system

Figure 2 shows the sketch of an Alpha program, with one single input I and one single output O. This program has one local variable C. Variables are defined on a set of integral coordinate points of some n-dimensional space, which forms a *polyhedron*. The domain of variable V is denoted $\mathrm{Dom}(V)$. An equation such as $C = f(I)$ can thus be read as $\forall \vec{i} \in \mathrm{Dom}(C), C(\vec{i}) = f(I(\vec{i}))$. Such a program defines a functional relation between the output O and the input I, here $O = g(f(I))$.

Scheduling the Alpha program `simple` of figure 2 consists of the following operations:

- Introduce local variables I' and O' (called *shadow variables*) for the input and the output: input I (resp. output O) is copied into a new local variable I' (resp. O') by means of an additional equation $I' = I$ (resp. $O = O'$). This obviously correctness preserving transformation will be needed to reflect the re-indexing of input and output variables by the scheduling.

- A time function is computed for every local variable, I', C and O'.

- These time functions are used to re-index the variables. This process is called *change of basis*. The effect of a change of basis is to map a point $\vec{i}$ of a domain to a pair $(t, \vec{p})$, where t represents a micro-step number, and $\vec{p}$ a processor number. Let us name P the change of basis applied to I' and Q that of O'. In Alpha, change of basis matrices are *unimodular*, i.e., they are integral matrices with determinant ± 1. Once the changes of basis applied, local variables such as the shadow variables have new domains, and are defined by new equations.

system sched_simple (I : $Dom(I)$ **of integer**)
returns (O : $Dom(O)$ **of integer**);
var
C : $Dom(C)$ **of integer**;
I' : $P(Dom(I))$ **of integer**;
O' : $Q(Dom(O))$ **of integer**;
let
$I' = I.P^{-1}$;
$C = f(I'.P)$;
$O' = g(C).Q^{-1}$;
$O = O'.Q$;
tel;

Figure 3: Scheduled Alpha system

Figure 3 shows the effect of the scheduling process on program `simple`. The "." operator corresponds to function composition: in other words, $I' = I.P^{-1}$ means that, for all $\vec{\imath} \in Dom(I')$, $I(\vec{\imath}) = I(P^{-1}(\vec{\imath}))$.

In the sequel, we will use the following definitions:

Slice : Let V be a scheduled Alpha variable with a domain $\mathrm{Dom}(V)$. We denote by $\mathrm{Slice}(V)_\tau$ the set of instances $V(t, \vec{p})$ such that $t = \tau$, that is

$$\mathrm{Slice}(V)_\tau = \{V(\tau, \vec{p}) \mid (\tau, \vec{p}) \in \mathrm{Dom}(V)\}.$$

Functions first and tail: Given an affine function f, $\mathrm{first}(f)$ defines the projection of f on its first component, and $\mathrm{tail}(f)$ the projection of f on its remaining components. For example, if $f(i, j, k) = (i + 1, j + 2, k)$, then $\mathrm{first}(f)(i, j, k) = (i + 1)$ and $\mathrm{tail}(f)(i, j, k) = (j + 2, k)$.

4 Computing the interface

In this section, we describe the interface. In particular, we explain how to compute the number of micro-steps, what data are transmitted, how they are transmitted. Rather than giving full details of the interface calculations, we consider the case of the generic program `simple`: all results that we present can be easily generalized to more than one input, one output and one local variable.

4.1 Structure of the communication protocol

After scheduling, the behaviour of the co-program can be represented by the following parallel program, where synchronization is done by message-passing:

```
/* Co-program */
doseq t = Min(t), step 1 to Max(t)
{
   receive{I'(t,p⃗) | ∀p⃗ s.t. (t,p⃗) ∈ Dom(I')};
   dopar p⃗ s.t. (t,p⃗) ∈ Dom(C)
      {Compute C(t,p⃗)};
   dopar p⃗ s.t. (t,p⃗) ∈ Dom(O')
      {Compute O'(t,p⃗)};
   send{O'(t,p⃗) | ∀p⃗ s.t. (t,p⃗) ∈ Dom(O')};
}
```

At each micro-step t, the co-program receives the set of data $I'(t,\vec{p})$ necessary to feed the processors; then it performs one step of evaluation, and finally returns the results $O'(t,\vec{p})$ produced by the processors.

To meet this structure, the host program must behave as follows:

```
/* Host program */
doseq t = Min(t), step 1 to Max(t)
{
   send{I'(t,p⃗) | ∀p⃗ s.t. (t,p⃗) ∈ Dom(I')};
   receive{O'(t,p⃗) | ∀p⃗ s.t. (t,p⃗) ∈ Dom(O')};
}
```

However, the host program does not see the transmitted data in the same referential that the co-program: rather than addressing the I' and O' variables, it addresses the I and O variables. In order to implement in full detail the interface, the relationship between the input variable I (resp. the output variable O) and the shadow variable I' (resp. O') has to be defined.

We observe that the number of micro-steps is given by the minimum value Min(t) and the maximum value Max(t) of the index t. As domains are polyhedra, these values can be obtained easily by solving linear programs.

4.2 Data sent by the host program

The set of data received by the co-program is

$$\mathrm{Slice}(I')_t = \{I'(t,\vec{p}) \mid (t,\vec{p}) \in \mathrm{Dom}(I')\}.$$

by definition of Slice. As I' is defined from I after scheduling, $\mathrm{Dom}(I') = P(\mathrm{Dom}(I))$, thus

$$\forall (t,\vec{p}) \in \mathrm{Dom}(I'), I'(t,\vec{p}) = I(P^{-1}(t,\vec{p})) \quad .$$

Therefore:

$$\mathrm{Slice}(I')_t = \{I(P^{-1}(t,\vec{p})) \mid (t,\vec{p}) \in \mathrm{Dom}(I')\}. \tag{1}$$

Equation (1) has two consequences. First, it gives the relationship between the $\vec{\imath}$ index of the I variable in the host program and the corresponding $(t,\vec{p})$ indexes of the I' variable in the program.

Second, it shows that the subset of the $\vec{i}$ indexes which correspond to data sent to the co-program is a polyhedron. Therefore, this subset can be scanned with a do loop, using well-known methods [1]. All these informations allow one to implement the filling of $\mathrm{Slice}(I')_t$ from I in the host program, and its reading by the co-program.

4.3 Data received by the host program

In a similar way, we notice that the set of values sent at micro-step t by the Alpha program to the host is

$$\mathrm{Slice}(O')_t = \{O'(t,\vec{p}) \mid (t,\vec{p}) \in \mathrm{Dom}(O')\}.$$

But O' is obtained from a change of basis Q on O, thus $\mathrm{Dom}(O') = Q(\mathrm{Dom}(O))$. The relation between O and O' is therefore:

$$\forall \vec{i} \in \mathrm{Dom}(O), \quad O(\vec{i}) = O'(Q(\vec{i})).$$

Now, let us find out which indexes $\vec{i}$ of O correspond to data sent by the co-program. If we note $t = \mathrm{first}(Q(\vec{i}))$ and $\vec{p} = \mathrm{tail}(Q(\vec{i}))$, we have

$$\forall \vec{i} \in \mathrm{Dom}(O), \quad O(\vec{i}) = O'(t,\vec{p}).$$

Therefore,

$$\mathrm{Slice}(O')_t = \{O(\vec{i}) \mid t = \mathrm{first}(Q(\vec{i})), (\mathrm{first}(Q(\vec{i})), \mathrm{tail}(Q(\vec{i}))) \in \mathrm{Dom}(O')\}. \quad (2)$$

Again, this relationship gives all the details needed to fill, in the host program, the variable O with the values received from the co-program.

4.4 Final host program

In summary, the host program is:

```
/* Final Host program */
doseq t = Min(t), step 1 to Max(t)
{
    send{I(P^{-1}(t,p)) | ∀p s.t. (t,p) ∈ Dom(I')};
    receive{O(i) | ∀i s.t. t = first(Q(i)),
                (first(Q(i)), tail(Q(i))) ∈ Dom(O')};
}
```

5 Correction of the protocol

Programs *Final Host Program* and *Co-Program* define precisely the communication protocol between the host program and the Alpha co-program.

To show the correction of this protocol, we need to prove that, provided that the slices are correctly transmitted between Alpha and host program, we have the following properties:

1. Any data $I'(t, \vec{p})$ needed by the Alpha program at micro-step t, is available.

2. Any data $O(\vec{i})$ of the host program is set to its final value at micro-step $\mathrm{first}(Q(\vec{i}))$.

The proof is done by induction on t. Assume that the properties are true for all micro-steps up to t.

Property 1: we have to guarantee that $I'(t, \vec{p})$ is filled and keeps its value from micro-step t on, that is:

$$\forall \tau \geq t, \forall \vec{p} \ : \ (t, \vec{p}) \in \mathrm{Dom}(I') \Rightarrow I'(t, \vec{p}) = I(P^{-1}(t, \vec{p})).$$

Our protocol says that, at micro-step t, the host program fills $\mathrm{Slice}(I')_t$, according to equation (1). This set of data is then sent to and received by the co-program. Therefore $I'(t, \vec{p})$ is available at time t. Moreover, since Alpha is a single assignment language, this value is kept until the end of the process. Hence $I'(t, \vec{p})$ is also available for all $\tau > t$.

Property 2: After receiving $\mathrm{Slice}(I')_t$, the Alpha program evaluates C, O', and fills $\mathrm{Slice}(O')_t$ which is then sent to and received by the host program. Then the host program sets O according to equation (2):

$$\mathrm{Slice}(O')_t = \{O(\vec{i}) \mid t = \mathrm{first}(Q(\vec{i})), (\mathrm{first}(Q(\vec{i})), \mathrm{tail}(Q(\vec{i}))) \in \mathrm{Dom}(O')\}.$$

hence, for all $\vec{i}$ such that $t = \mathrm{first}(Q(\vec{i}))$, $O(\vec{i})$ is set to its final value if $Q(\vec{i}) \in \mathrm{Dom}(O')$. Because of the change of basis property, we know that $Q(\vec{i}) \in \mathrm{Dom}(O') \Leftrightarrow \vec{i} \in \mathrm{Dom}(O)$. Therefore, for all $\vec{i}$ such that $t = \mathrm{first}(Q(\vec{i}))$, $O(\vec{i})$ is set to its final value at step t. Hence the property.

This proof highlights the fact that no deadlock can occur provided that the channels between the host program and co-program are safe.

6 Practical implementation and results

The previous section gave an abstract description of the communication protocol, and showed its correction. Here, we deal we practical implementation issues, we present an application of this technique to the simulation of a H261 image coder, and we discuss the interest of our method.

6.1 Implementation issues

We have seen that the messages (slices of data) exchanged by the host program and the Alpha program are sets of indexed variables whose indexes form convex polyhedra. To implement this in C, we encode these values in enclosing arrays, using a technique described in [10]. The size of these arrays is chosen in such a way that the array attributed to a given input/output variable can contain any slice of it.

For example, given a scheduled variable I with a domain $\mathrm{Dom}(I) = \{t, p \mid p \leq t \leq 2+p, 1 \leq p \leq 4\}$, I corresponds to six slices, one for each value of t. Each slice has a different number of points. But the array defined by $array(p) = \{p \mid 1 \leq p \leq 4\}$ contains any of the slices.

As a slice is only a subset of the encoding array, one needs to make sure that the host and the co-program do not access incorrect elements, while filling this array.

On the co-program side, this property is enforced by a static analysis and code generation tools. The static analyzer of MMAlpha checks domains compatibility in an Alpha system. In the host program, accesses are controlled by conditions on the indexes. These conditions are trivially derived from the ones found in the co-program to define polyhedra domains.

6.2 Application to a H261 image coder

This approach was used to simulate a large application using jointly the Signal and MMAlpha systems. The MMAlpha system [4] is used for manipulating Alpha programs. Transformations of Alpha programs are written in Mathematica, and can be executed interactively in the Mathematica environment.

The application considered was a H261 image coder. This chain compresses a video signal in the 34Mbits, MPEG norm. The initial application was written using Signal, with some functions, such as motion estimation, quantization, and Discrete Cosine Transform (DCT), written in C. These functions are those which are candidate to hardware implementation. Our first step was to rewrite the DCT in Alpha. By generating C code from Alpha, a simulation of the complete system was done.

The second step was to schedule the DCT using MMAlpha. The MMAlpha commands for this are:

```
load["dct.alpha"];
schedule[];
```

Then, the interface generator was called, for particular values of parameters (here, both are set to 8):

```
genInterface["-p 8 8"];
```

It produced a new C program implementing the DCT, together with its interface. It also produced the new Signal program, modified in order to generate the micro-ticks between two calls to the DCT, and the C code corresponding to this new Signal specification. The complete system was then simulated.

6.3 Discussion

The method we have presented is very flexible. It is driven by the scheduling, and any modification of the scheduling can be immediately reflected in a new, correct interface generation. As scheduling is a fundamental step in the choice of an efficient architecture, the designer has full freedom to select the most appropriate one. Moreover, the scheduling itself can be driven by the resource constraints of the physical interface, which will in turn influence the interface generation.

The heart of our method is the so-called “polyhedral model” of Alpha, where regular calculations can be expressed as operations on collections of data of various shapes. The advantage of this approach over expressing calculations using imperative programs is the computational power of the tools developed for it: they allow communication protocols to be calculated automatically

and quickly, as shown in section 4. The drawback is clearly the need to express the computations using this rather unconventional model. However, methods to translate imperative loops into single assignment code exist.

7 Conclusion

We have shown how a data-flow interface between a regular program expressed in Alpha and its host program can be designed automatically by analysis of the code of the Alpha program given a scheduling of this program. The communication protocol and the exact data that are exchanged have been defined, and the correction of the interface generator has been proven. An implementation of this interface generator is available in the MMAlpha software, and allows one to produce the C program simulating the scheduled Alpha program, and the C program of its interface.

We are currently pursuing this research in order to generate an implementation of the Alpha program on configurable hardware, and produce automatically the interface between the host and the reconfigurable platform.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. *ACM Sigplan Notices*, 26(7):39–50, July 1991.

[2] P. Chou, R. B. Ortega, and G. Borriello. Interface co-synthesis techniques for embedded systems. In *International Conference on Computer Aided Design*, pages 280–287, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[3] C.T.I. Comete. *Codesign – Conception conjointe logiciel-matériel*, chapter 5: Le projet Cairn : vers la conception d'architectures à partir de Signal et Alpha. Collection Technique et scientifique des Télécommunications. Eyrolles, 1998.

[4] F. Dupont de Dinechin, P. Quinton, and S. Rajopadhye T. Risset. First steps in alpha. Publication Interne 1244, Irisa, 1999.

[5] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[6] G. Gogniat, M. Auguin, L. Bianco, and A. Pegatoquet. A Codesign Back End Approach for Embedded System Design. *ACM Transaction on Design Automation of Embedded Systems*, 1999.

[7] Paul Le Guernic and Thierry Gautier. Data-flow to von Neumann: the SIGNAL approach. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 15, pages 413–438. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[8] Tarek Ben Ismail and Ahmed Amine Jerraya. Synthesis steps and design models for codesign. *Computer*, 28(2):44–53, February 1995.

[9] P. Le Moënner, L. Perraudeau, S. Rajopadhye, T. Risset, and P. Quinton. Generating regular arithmetic circuits with alphard. In *2th International on Massively Parallel Computing Systems (MPCS'96)*, May 1996.

[10] P. Quinton, S. Rajopadhye, and D. Wilde. Deriving imperative code from functional programs. In *7th Conference on Functional Programming and Computer Architecture*, pages 36–44, La Jolla, Ca, USA, June 1995. SIGPLAN/SIGARCH/WG2.8.

[11] Alan S. Wenban, John W. O'Leary, and Geoffrey M. Brown. Codesign of communication protocols. *Computer*, 26(12):46–52, December 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399