Scalable and Modular Scheduling

Paul Feautrier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Scalable and Modular Scheduling

Paul Feautrier

N° 5180

Mai 2004

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5180--FR+ENG

Scalable and Modular Scheduling

Paul Feautrier

Thème Com — Réseaux et systèmes
Projet Compsys

Rapport de recherche n° 5180 — Mai 2004 — 25 pages

Abstract: Scheduling a program (i.e. constructing a timetable for the execution of its operations) is one of the most powerful methods for automatic parallelization. A schedule gives a blueprint for constructing a synchronous program, suitable for an ASIC or VLIW processor. However, constructing a schedule entails solving a large linear program. Even if one accept the (experimental) fact that the Simplex is almost always polynomial, the scheduling time is of the order of a large power of the program size. Hence, the method does not scale well. The present paper proposes two methods for improving the situation. Firstly, a big program can be divided in smaller units (processes) which can be scheduled separately. This is *modular scheduling* Second, one can use projection methods for solving linear programs incrementatly. This is specially efficient if the dependence graph is sparse.

Key-words: scheduling, modularity, parallelization

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Ordonnancement modulaire

Résumé : Ordonnancer un programme est l'une des méthodes les plus puissantes en parallélisation automatique. Un ordonnancement fournit un schéma de construction pour un programme synchrone, bien adapté à un circuit spécialisé (ASIC) ou à un processeur VLIW. Cependant, pour construire un ordonnancement il faut en général résoudre un programme linéaire de grande taille. Même si l'onaccepte le fait expérimental que le Simplex est presque toujours de complexité polynomiale, le temps d'ordonnancement est de l'ordre d'une puissance élevée de la taille du programme. En conséquence, la méthode ne passe pas bien à l'échelle. Cet article propose deux méthodes pour améliorer la situation. On montre tout d'abord comment diviser un programme en petites unités (processus) qui peuvent être ordonnancées individuellement. D'autre part, les méthodes de projection permettent de résoudre les programmes linéaires de façon incrémentale, ce qui est spécialement efficace quand le graphe de dépendance est creux.

Mots-clés : ordonnancement, modularité, parallélisation

1 Introduction

One of the challenges in the design of embedded system is to devise methods for the automatic or semi-automatic construction of application-specific devices from a behavioral specification. This is a very difficult problem, since one has to take into account many evaluation functions (design cost, fabrication cost, performance, power consumption, time-to-market among others) and find a compromise between conflicting requirements.

The first step toward a solution has been the division of the field according to the type of the application. In control intensive applications, the amount of computation is small, the realtime constraints (if any) are easily met, and the emphasis is on the safety of the resulting systems. This subfield has lead to the design of the very successful synchronous languages [3].

The situation is different for compute intensive systems, which are mostly found in signal processing applications (audio and video processing, radar software, telephony, etc.). Here the computing time cannot be neglected, the amount of data is huge, and the real time constraints are soft. For instance, in domestic TV applications, one can tolerate missing a very small proportion of frames. This subfield is less well understood than the preceding one. At present, applications (or parts thereof) are first modeled in very high level languages (mostly, Matlab), then mapped by hand on a variety of architectures, and then implemented in a mixture of medium level code (C) and assembly code. The design process is lengthy, complex, error-prone, and does not lend itself to the exploration of the solution space.

The aim of this paper is to sketch another approach, in which the application is specified as a system of communicating processes, each process being written in a medium-level language like C. I will explain how such a specification can be converted to a synchronous program, suitable for instance for a VLIW processor or as a first step in the design of a specialized circuit. The first step in this conversion is the construction of a schedule, which gives the epoch at which each operation in the program is executed. The problem of regenerating a program from a schedule has been first studied by Irigoin [1] and considered by many other scholars. Very efficient solutions (with associated software) [14, 2] are available today.

The situation for scheduling is less favorable. Finding legal schedules entails solving large linear programs. In [7], I reported scheduling times of the order of several tens of minutes! One may imagine that the situation has improved for three reasons:

- Moore's law has given us a factor of a thousand in compilation power.
- Linear programming software has improved, but by a much smaller factor.
- Embedded systems designers tolerate much longer compilation times than high-performance programmers.

However, Moore's law cuts both ways. Present day embedded systems are able to host much more complex applications than ten years ago, hence the programs to be compiled today are much larger than those I used for benchmarks in [8].

The aim of this paper is to propose two methods for applying scheduling to large applications. The first method consists in modifying the basic scheduling algorithm to achieve better scalability. In the second method, I investigate under which conditions a program can be divided in independent modules which can be, at least partially, scheduled independently. This second method has the added advantage that it may be the key to reuse of hardware or software components in parallel applications.

In the next sections I define which type of modules are suitable for parallel programming and review the basic scheduling algorithm. In section 4, I explain how to improve the scheduling time of one process provided that the dependence graph is sparse. Section 5 explains how to do modular scheduling. In the conclusion, I present some open problems and discuss future work.

2 Communicating Regular Processes

The model of Communicating Regular Processes has been designed with the following requirements in mind:

- The model must allow the decomposition of a large application into small modules, thus promoting reuse and readability. These modules – in fact,

processes –, communicate through channels; the resulting system allows a visual representation and looks familiar to electronic designers.

- Many systems of communicating processes have been designed, both in theory (CSP [10], KPN [11]) and in practice (the Unix system with pipes or sockets is a system of communicating processes). However, these systems are too liberal and do not allow much in the way of automatic analysis.

- Among the properties that one would like to check more or less automatically are the absence of deadlocks, the boundedness of the channel buffers, and the fact that no undefined value is ever used in a computation. Obviously, simulation and testing may pinpoint some errors of this kind. It is well known, however, that testing is efficient only in the first steps of a design, and that formal methods are necessary to find the last bugs.

2.1 Definition

Let us first emphasize the fact that the language of CRPs is not a programming language but a specification language. For instance, it is said down below that a process is a sequential program. This does not mean that a process must be executed sequentially; it just says that the observable effects of a process must be the same as if it were executed sequentially – performance excepted. The degree of parallelism of a CRP system bears no relation to the number of its processes, and is mostly under control of its implementor.

2.1.1 Processes

A process is a sequential program which can communicate with other processes through channels (see below). With the exception of channels, all variables are local to one and only one process and are not visible from other processes[1]. The code of a process can be written in any convenient algorithmic language. I use C here, but other choices are possible: Pascal, Fortran and others.

[1] The model can tolerate read-only global variables (e.g. tables of constants). This facility is not discussed here for brevity sake.

The code of a process is *regular*, or has static control [7] in the following sense:

- Statements are assignments statements and bounded loop statements. All variables are considered part of some array, scalars being zero-dimensional arrays. Each statement modifies only one memory cell, and each array in the process has at most one occurrence in the right hand side (rhs) of the assignment[2].
- Loops are of the arithmetics progression variety (exactly the `for` loops of Pascal), and the loop upper and lower bounds are affine forms in numerical or symbolic constants and surrounding loop counters.
- The only method of address calculation is subscripting into arrays of arbitrary dimension. The subscripts must be affine forms in constants and surrounding loop counters.

Some of these restrictions are quite natural when one is designing compute-intensive embedded systems with real time constraints. It is difficult, for instance, to predict the execution time of a while loop or of the traversal of a truly dynamic data structure. Other restrictions can be lifted by preprocessing (`goto` removal, inductive variable detection, subscript-like pointer detection, function inlining).

The iteration vector of a statement is a list of its surrounding loop counters, from outside inward. An iteration vector for S cannot take arbitrary values. It must belong to the iteration domain of S, which is obtained by stating that each counter is within the bounds of the corresponding loop. Under the assumption that the program is regular, iterations domains are convex polyhedra (or, more precisely, sets of integral points inside polyhedra). In the presence of conditionals, an iteration domain may be a union of polyhedra instead of a single polyhedron. I will ignore this complication in what follows.

Let D_S be the iteration domain of statement S. An iteration of S or *operation* is written $\langle S, x\rangle, x \in D_S$ where x is the *iteration vector*. The set of operations of a process P is the disjoint union:

$$E_P = \bigcup_{S\in P} \{\langle S, x\rangle \mid x \in D_S\},$$

[2]This restriction is there just to simplify notations.

and the set of operation of a process system is $\cup_P E_P$. In more abstract contexts, I may simply write $u \in E$ for an arbitrary operation.

2.1.2 Channels

A channel is an array of arbitrary dimension which is used as a communication medium from a process to another process. Channels are unidirectional. One process is declared as the writer to a channel. Considered as an array, each cell of the channel must be written only once by its writer: this is the single assignment property. Writing to a channel is non-blocking.

On the other hand, a channel may have any number of readers, and there are no constraints on the pattern of reading. Reading is not destructive: a value remains in a channel at least as long as some process may have some use for it. If a process reads a cell which has not yet been defined, it blocks until a definition happens.

$\mathcal{W}(A)$ denotes the set of operations that write into channel A with subscript function ω_A, and $\mathcal{R}(A)$ denote the set of operations that read from A with subscript function ρ_A. Clearly, $\mathcal{W}(A) \subseteq E$ and $\mathcal{R}(A) \subseteq E$. The set:

$$\mathcal{F}(A) = \{\omega_A(u) \mid u \in \mathcal{W}(A)\}$$

is the *footprint* of A. If the following constraint:

$$\mathcal{G}(A) = \{\rho_A(u) \mid u \in \mathcal{R}(A)\} \subseteq \mathcal{F}(A) \tag{1}$$

is not satisfied, it is clear that some process will block for ever when accessing a memory cell in $\mathcal{G}(A) - \mathcal{F}(A)$.

2.1.3 Connections

It is possible to assume that processes have direct access to channels. However, in real life applications, it is better to assume that processes access *ports*, and that ports are connected by channels. This allows, among other possibilities, that a process be reused several time with different channel connections.

When connecting ports, one must verify (statically) that each channel has only one writer, that the single assignment property is verified, that the two ports have the same (data) type and dimension, and that the constraint (1) is

satisfied. The connections are specified by a glue language yet to be specified (however, see the example below).

In what follows, and for the sake of simplicity, I will omit this connection step (which poses no theoretical problem) and assume that processes are directly connected to channels, and that all necessary verifications have already been done successfully.

2.2 An Example

The following trivial example specify a system in which a producer generates an infinite stream of values which are sent to a consumer process which compute a sliding mean.

```
process producer(outport X[]){
  int i;
  for(i=0;;i++)
    X[i] = f(i);
}
```

```
process consumer(inport Y[]){
  float s;
  int i;
  s = 0.0;
  for(i=0;;i++)
    s = 0.5*(s + Y[i]);
}
```

```
/* the glue code */

channel float A[];

main(void){
  producer(A);
  consumer(A);
}
```

The new keywords `process, inport, outport` and `channel` are self-explanatory. Technically, they appears as new storage specifiers in the C grammar. In the glue code, one starts a process with the same syntax as for a function invocation. However, the process call returns immediately. Processes can have ordinary parameters, but this facility has not been used here.

2.3 Data Dependences

Data dependences were defined, as early as 1966, for the purpose of parallelization [4]. Two operations are in dependence if interchanging them in the execution order changes the final result of the program. This is a global definition, which in general is too complex to be usable. A more local definition is: two consecutive operations are in dependence if interchanging them change the history of some variable. This definition involves semantics considerations. For instance, to see that the two operations `x = x+1` and `x = x+2` are (locally) independent, one needs some knowledge of elementary arithmetics. The merit of Bernstein is to have given a purely syntactical criterion for dependence. An operation u being given, let $R(u)$ be the set of memory cells that are read by u (on which the effect of u depends) and $W(u)$ be the set of cells which are modified by u. Without loss of generality, we will suppose in this paper

that $W(u)$ always is a singleton. Then u and v are in dependence if at least one of the three sets $W(u) \cap W(v)$ (output dependence), $W(u) \cap R(v)$ (flow dependence) and $R(u) \cap W(v)$ (anti-dependence) is not empty.

Data dependences are concerned with the case where the memory cells under consideration are local to some process. It follows that u and v belong to the same process and that sequential order is well defined. One says that v depends on u (in symbols $u \, \delta \, v$) if u and v are in dependence and if $u <_{\text{seq}} v$.

2.4 Communication Dependences

Assume now that the variable which causes the dependence is a channel cell. We can still says that two operations may be in dependence, with the same definition as above. It is clear that the dependence cannot be an output dependence, since each channel cell is written only once. One must impose a flow dependence (no read can be executed before the first and only write), and this is sufficient to eliminate all anti-dependences. Hence, each dependence involving a channel cell (a communication dependence) is a flow dependence and is oriented from the write operation to the read operations. These operations clearly belong to different processes, hence this ordering does not conflict *directly* with any other ordering in a CRP system.

In what follows, I will use the same symbol, δ, for data and communication dependences.

2.5 The Programming Model of CRP Systems

One may imagine the execution of a CRP system in the following way:

- Each process is executed sequentially on a separate processor. This processor has access to a private memory which holds its local variables.
- Each processor can also access a global memory which holds the channels. Each cell of this memory is associated to a full/empty bit. Initially, this bit is set to "empty". When the cell is written for the first time, this bit is reset to "full". A write to a full cell is an error.
- A processor which attempt a read to an empty cell is stalled until the cell is filled.

Observe that this model is completely asynchronous. The relative speeds of the processors are not specified, and may, in fact, be arbitrary. Despite this basic asynchrony, it can be proved that the behavior of the system is determinate in the following sense: for all legal executions, all memory cells have the same history. This property is similar to the famous result of Kahn: in a Kahn Process Network, each channel history is the same for all legal executions. The proof is rather technical and will be given elsewhere.

3 Scheduling

3.1 Target Architectures

In contrast to the above programming model, most of today electronic systems are synchronous: there is one global clock, and all changes of state occur in relation to the clock. More precisely, these systems are "globally asynchronous and locally synchronous" (GALS); there are several unrelated clocks, and different *clock domains* communicate through synchronization protocols, like handshake or bus arbitration. The theory of multiple clock systems is still in infancy. We will postulate here that the target system is fully synchronous. This model fits well with the structure of VLIW processors or ASIC/FPGA special purpose circuits. An unspecified VLIW processor will be the main target architecture in what follows.

3.2 Schedules

A schedule is a function which assign a starting time to all operations in a program. In other words, a schedule is a function from E to the set of time values, T. But what is time? One possibility is to consider physical time. In that case, T is the set of non-negative integers, time being measured in clock cycles. This approach is suitable to deal with fine-grain systems in which execution time is well defined (typically 1 clock cycle), and with real time problems.

Another possibility is to consider a schedule as just a way of specifying an execution order. In that case, T is any ordered set. θ being a schedule, the

associated order is:

$$u <_\theta v = \theta(u) < \theta(v).$$

The favorites for T are again $\mathbb{N}$ and $\mathbb{N}^d$, lexicographically ordered. This second case gives rise to the so-called multidimensional schedules.

The execution order which is defined by a schedule must be legal, i.e. it must extend the dependence relation:

$$\forall u, v \in E : u \,\delta\, v \Rightarrow \theta(u) < \theta(v). \tag{2}$$

To solve this functional inequality, one has to postulate a shape for θ. The usual choice is that $\theta(\langle S, x\rangle)$ is an affine form in the iteration vector, x:

$$\theta(\langle S, x\rangle) = h_S.x + k_S, \tag{3}$$

where h_S is the timing vector of S and k_S is a scalar. For regular programs, this choice has the advantage that everything in (2) become affine, and that powerful results from the theory of linear inequalities, like Farkas lemma [17], can be used to characterize the solutions. The reader is refered to [7, 8] for details. A short review of the method will be given below.

3.3 Solving the Scheduling Constraints

The first step of the solution consists in splitting formula (2) according to the source and sink of dependences. For a given pair of statements, S and T, the constraint now reads:

$$\forall x \in D_S, y \in D_T : \langle S, x\rangle \,\delta\, \langle T, y\rangle \Rightarrow \theta(\langle S, x\rangle) < \theta(\langle T, y\rangle). \tag{4}$$

Each such constraint represents in fact $O(\text{Card } D_S \times \text{Card } D_T)$ linear constraints on the coefficients of θ. This number is usually enormous, or even infinite in the presence of unbounded parameters or non-terminating loops. However, thanks to the fact that the schedules are affine, and that the constraints defining δ are affine, these constraints can be compressed into a short finite set.

This compression can be done either by the vertex method [16] or by making use of the following version of Farkas lemma:

Lemma 1 *The formula:*

$$\forall x : Ax + b \geq 0 \Rightarrow c.x + d \geq 0$$

is equivalent to:

$$\exists \lambda_0 \geq 0, \lambda \geq 0 : \lambda.b + \lambda_0 = d,\ \lambda A = c.$$

provided that the system $Ax + b \geq 0$ *is feasible.*

In this formula, A is an $m \times n$ matrix, x is an n-vector, b is an m-vector, c is an n-vector and d is a scalar.

To apply this result, let x be the concatenation x_S, x_T of the iteration vectors of S and T. Let $Ax + b \geq 0$ be the system of constraints that define the dependence relation from S to T. One first checks that this system is feasible. If not, the dependence does not exists and impose no constraints on the schedules.

The inequality $c.x + d \geq 0$ is taken as the *delay* between execution of $\langle S, x\rangle$ and $\langle T, t\rangle$:

$$c.x + d = (-h_S, h_T).(x_S, x_T)^T + k_t - k_S - 1 \geq 0,$$

which gives the equivalent formulas:

$$\lambda A = (-h_S, h_T), \tag{5}$$

$$\lambda b + \lambda_0 = k_T - k_S - 1. \tag{6}$$

For regular programs, A and b can be extracted from the program text by a simple analysis. Hence, (5) is a system of linear equations in positive variables. There is such a system for each dependence, and the schedules must satisfy all of them. Hence, one has to gather all such constraints, and submit the grand system thus constructed to some linear programming tool. Most of the time, such a system has many solutions (i.e., many legal schedules). One can introduce a linear objective function and select the best solution in some sense (minimum length of the critical path, for instance).

However, in some cases, the system (5) is not feasible. This may be due to the presence of deadlocks in the interconnection system. But the failure

may sometime be traced to complexity reasons. A program that has an affine schedule can be executed in linear time when enough processors are available. It is clear that there exists programs for which this is impossible. One can resort in this case to multidimensional schedules, whose parallel latency is polynomial. The construction of multidimensional schedules is explained in [8]. We will ignore this difficulty in this preliminary paper.

4 Scalability

The number of unknown in a scheduling problem is of the order of the number of statements times the mean depth of loop nests. The number of dependences is in general quadratic in the program size, and the number of constraints per dependences is again proportional to the mean nesting depth. Lastly, the Simplex algorithm, while exponential in the worst case, has a high probability of being cubic in the number of unknowns or constraints, when these two numbers are of the same order of magnitude. Hence, the direct solution of the scheduling constraints by linear programming does not scale well.

4.1 Elimination of the Farkas Multipliers

The first step in improving the scalability of the method consists in eliminating the Farkas multipliers. The important point is that there is one independent set of Farkas multipliers per dependence. Hence, the elimination can proceed one dependence at a time. The complexity of the elimination is linked to the maximum nesting level of the program, a small integer. The number of eliminations is equal to the number of dependences, which is at most quadratic in the size of the program.

Since the Farkas multipliers occurs in linear equations, one can start by using Gaussian elimination. In general, there are more unknowns than equations: all Farkas multipliers cannot be eliminated. The resulting constraints express the fact that the eliminated multipliers must be positive. This trick has been proposed in [7] and has proved very efficient in practice.

But one can go farther than that. The remaining Farkas multipliers can be eliminated by the Fourier-Motzkin method, thus leaving as sole unknowns the coefficients of the schedules. It is well known that Fourier-Motzkin elimination

has a tendency to create new constraints in large number. This does not happen here, because the number of eliminated variables is small and the constraint matrix is sparse. In fact, my experience shows that, most of the time, the constraint system after Fourier-Motzkin elimination is smaller than the original.

4.2 Stepwise Scheduling

After the elimination of the Farkas multipliers, the number of unknowns has roughly been divided by two, and the number of constraints has stayed the same or may have increased slightly. Scheduling is still not scalable. To go further, one has to observe that the constraint matrix is sparse, or, rather, block sparse. In fact, a dependence from S to T being given, the resulting constraints can be written as:

$$M_{ST}(h_S, k_S)^T + N_{ST}(h_T, k_T)^T \geq 0. \tag{7}$$

If one compress each block M_{ST} or N_{ST} to a single cell, one gets the incidence matrix of the dependence graph.

If the scheduling problem is solved by a variant of the simplex algorithm, one cannot make use of this sparsity to speed up the resolution: the simplex has fillup. In fact, the simplex algorithm is very similar to Gaussian elimination, with the exception that the choice of the pivot, which is almost arbitrary in Gaussian elimination, is highly constrained in the simplex. Hence, one cannot choose the pivot that generates the less fillup, as in direct methods for sparse linear systems solution [18].

The solution is to use projection algorithms. The projection of a set D in $\mathbb{R}^{n+1}$ along its first dimension is:

$$P = \{x \mid \exists y : y.x \in D\}. \tag{8}$$

It is well known that if D is a polyhedron, so is P. For polyhedra, there are several projection algorithms:

- The simplest one is the Fourier-Motzkin algorithm. Its complexity is super exponential. Part of this complexity is due to the fact that the resulting system of constraints contains many redundant inequalities.

- One can also use parametric linear programming as in PIP [6]. The complexity is less, but the result still has many redundancies.

- Lastly, if one knows the Minkowski representation of D, it is easy to find the Minkowski representation of P. From that, one can reconstruct an irredundant constraint system with the Chernikova algorithm.

The last solution is probably the best one, especially since there exists an efficient implementation [20]. However, for the preliminary experiments that are reported here, I have used the Fourier-Motzkin algorithm coupled to a naive redundancy elimination method.

Whatever the projection algorithm, whenever one has chosen a point $x \in P$, one can find – in time linear in the number of constraints of D – a segment $[a, b]$ such that if $a \leq y \leq b$, then $y.x \in D$. We can thus solve a system of affine constraints by successively eliminating all unknowns, selecting a point in the last – one dimensional – projection, and then back propagating the result until all unknowns have been valued. One can show that all feasible points for the initial constraints can be obtained in this way.

This suggest the use of the following algorithm:

- For each statement S:
  - Collect all the rows of M where h_S has a non-zero coefficient.
  - Eliminate h_S.
  - Remember the bounds for h_S.

- If the resulting system is trivially unfeasible ($-1 \geq 0$) stop. No schedule exists.

- For each statement S in reverse order:
  - The bounds for h_S are constants. Select a value within the bounds for h_S (e.g. the lower bound).
  - Substitute these values in all other bounds.

4.3 Choosing the Next Victim

Obviously, whether this algorithm is scalable depends both on the way the variable to be eliminated is chosen and on the shape of the dependence graph. If for instance the dependence graph is complete, at each elimination all constraints are involved, and there will be no improvement. Fortunately, the dependence graphs one encounter in practice are far from being complete, simply because programmers cannot manage code in which everything depends on everything. One can model the elimination process by a hypergraph on the statements of the program. A hypergraph is a generalization of a non-oriented graph. While in an ordinary graph, an edge is a set of exactly two vertices, in a hypergraph, the number of vertices per hyperlink is arbitrary.

Initially, the hypergraph is the Dependence Graph. To understand how to mimics the elimination process, suppose we have decided to eliminate statement S. We have to take into account all constraints in which h_S has a non-zero coefficient, i.e. all dependences which have S as a source or a sink, i.e., all hyperlinks which contain S. After the elimination, we are left with a system of constraints which may include h_T for all T adjacent to S, and S has disappeared. The new hyperlink is given by: $\cup_{e \ni S} e - \{S\}$, where e is any hyperlink. The resulting constraint matrix will still be block sparse. The new hyperlink gives an estimates of the set of vertices which participate in the new block. The estimate is conservative: some vertices may disappear due to the vagaries of projection algorithms.

Based on this simulation we may devise several heuristics. For instance, one may select the vertex which results in the smallest hyperlink. A simpler method is to select the vertex with the smallest degree. For the moment, I have used the dumbest heuristic: eliminate the first remaining statement. Experience with a limited set of programs shows that while this technique does not reduce much the number of constraints, the number of unknown at each elimination step decreases sharply, which is a big improvement since the Fourier-Motzkin algorithm is super exponential in the number of unknowns.

5 Modularity

In language and compiler design, the standard definition of a module is "a part of a program which can be *partially* compiled without reference to other parts". Traditionally, the result of partial compilation is called an *object*. When all modules have been compiled, another processor, the *linker* is needed to finish the construction of the program. Modularity has many advantages. Modules promote reuse. Also, in case of a modification, one recompile only the affected module(s). As we have seen earlier, the natural unit of compilation for a parallel program is the *process*.

5.1 Channel Schedules

Going back to the scheduling constraints (2), one can see that processes are not isolated from each others, as there will be relations between the schedules of the writer and the readers of each channel. This does not allow modular scheduling. The solution is to provide some "insulation" between processes.

Observe that each cell in a channel A is written only once at a definite time by statements from only one process Therefore, one can postulate the existence of a channel schedule $\theta(\langle A, x\rangle)$ such that the value $A[x]$ is guaranteed to be defined at time $\theta(\langle A, x\rangle)$ (and later). For simplicity, I assume here that θ is affine. This is a loss of generality. Even when all statements have affine schedules, since a channel can be divided in parts, each part being written by a different statement, the channel schedule may be only piecewise affine. This problem can be taken into account along the lines of [9] and is left for future work.

The value of a channel schedule is clearly not defined for $x \notin \mathcal{F}(A)$, but it may be that the linear formula for θ nevertheless gives spurious values beyond that domain, by a process of extrapolation. This is why the property (1) must be checked independently.

With this definition, a dependence on a channel array can be split in two parts:

- On the write side, a cell is not available before it has been written. Let $S : A[\omega_A(x)] := \cdots$ be a statement that writes into A:

$$\theta(\langle A, \omega_A(x)\rangle) \geq \theta(\langle S, x\rangle) + 1 \tag{9}$$

- On the other side, a cell cannot be read before it is available. Let $R : \cdots := \cdots A[\rho_A(x)] \cdots$ be a statement that read A:

$$\theta(R, x) \geq \theta(\langle A, \rho_A(x) \rangle). \tag{10}$$

The 1 in formula (9) is intended to represent a propagation delay through the channel. I have arbitrarily inserted this delay on the write side, but many other configurations can be used without changing the overall method.

5.2 The Modular Algorithm

Let h_P be the concatenation of the timing vectors for all statements in process P, and let h_A be the timing vector for array A. After application of the Farkas algorithm to (9) or (10) and elimination of the Farkas multipliers, the shape of the constraint matrix is as follows.

For each process P there is a system $U_P h_P \geq 0$ which represents the constraints generated by the inner dependences in P. The matrix U_P is block sparse, and each of its blocks is one of the M_S or N_S blocks in formula (7). For each process P and each channel A which is connected to P there is a system $V_{AP} h_P + W_{AP} h_A \geq 0$ which represents the constraints generated by the communication dependences of the system. These observations suggest the following modular scheduling algorithm.

1. Construct the constraint matrix for each process and its adjacent channels.

2. For each process P eliminate h_P from the constraints:

$$U_P h_P \geq 0,\ V_{PA} h_P + W_{PA} h_A \geq 0,\ \text{for all } A \text{ connected to } P \tag{11}$$

   This first pass of compilation is modular, in so far as this can be done one process at a time, without reference to other processes. The result is a system of constraints on channel schedules.

3. When all such *communication constraints* have been computed (or collected from a repository), they can be solved as a whole, giving a solution for the channel schedules. Again, the communication constraints matrix

is block-isomorphic to the communication graph of the whole system, and has a high probability of being sparse. This is the only place where the system has to be considered *in toto*.

4. The solution for the channel schedules can then be substituted in the bounds for the coefficients of the schedules, and these coefficients can be recovered by back-substitution.

5. It remains to gather all schedules and submit them to a code generator. With present day tools [2], there is no hope of staying modular there, unless one deals with highly specialized architectures. However, tools like CLooG are quite efficient and can handle very large programs.

Consider the example of Sect. 2.2. The first step is to compile the two processes. Let:

$$\theta(\langle W, i\rangle) = \alpha i,\ \theta(\langle Z\rangle) = \beta,\ \theta(\langle R, i\rangle) = \gamma i + \delta,\ \theta(\langle A, x\rangle) = \epsilon x + \phi.$$

The producer has no data dependence, hence the only constraint is a communication constraint:

$$i \geq 0 \Rightarrow \epsilon i + \phi \geq \alpha i + 1.$$

Application of the Farkas algorithm gives $\phi \geq 1$ and $\epsilon \geq \alpha$ after elimination of the multipliers. After elimination of α, the only remaining constraint is $\phi \geq 1$.

In the consumer there is a flow dependence from Z to R, which gives $\delta \geq \beta + 1$, and a flow dependence from R to itself, which gives $\gamma \geq 1$. Lastly, there is a communication dependence from A to R which entails $\phi \leq \delta$ and $\epsilon \leq \gamma$. The next step is the elimination of β, γ and δ from the system of constraints:

$$\delta - \beta - 1 \geq 0,\ \gamma \geq 1,\ \phi - \delta \geq 0,\ \gamma - \epsilon.$$

The resulting system is empty. The only communication constraint is $\phi \geq 1$ whose smallest solution is $\phi = 1$. From there, one may reconstruct the schedules:

$$\theta(\langle W, i\rangle) = 0,\ \theta(\langle Z\rangle) = 0,\ \theta(\langle R, i\rangle) = i + 1,\ \theta(\langle A, x\rangle) = 1.$$

This solution is not satisfactory, since one has to deposit an infinite number of values in A in one clock cycle. An easy way out is to slow down the producer by introducing a dependence:

```
C:  t = f(i);
W:  A[i] = t;
```

The schedules become:

$$\theta(\langle C, i\rangle) = 2i,\ \theta(\langle Z\rangle) = 0,\ \theta(\langle W, i\rangle) = 2i{+}1,\ \theta(\langle A, x\rangle) = 2i{+}2,\ \theta(\langle R, i\rangle) = 2i{+}2.$$

Notice that it was not necessary to recompile the consumer. These schedules correspond to a VLIW program whose kernel is:

clock cycle	C	W	R
even	*		*
odd		*	

6 Related Work

While the literature on automatic parallelization is enormous, and the literature on scheduling is only slightly smaller, the problems of modular parallelization and of modular scheduling have not been extensively considered by the academic community, let alone industry.

In [19], the unit of modularity is the procedure, whose effect is summarized by computing *regions*. The drawback of this method is that one can find parallelism between procedure calls, and also inside procedures, but not parallelism that requires a transformation involving both a procedure and its calling context.

Nearer to the subject of this paper, Risset and Quinton [15] have defined structured scheduling for *systems* in the Alpha specification language [13]. Systems can be scheduled independently. The schedules of several systems are then composed to give the global schedule. This is possible only if somewhat stringent restrictions are imposed on systems.

The use of processes in parallel programming dates back to the commencement of the subject. Kahn Process Networks [11] have been a source of inspiration for the present paper. The main difference is that in KPN, there are no constraints on the definition of each process – which may not be a program in the usual sense – hence their *a priori* analysis and compilation is almost

impossible. This results in the present situation, where KPN are only used for simulation or even direct execution. In contrast, CRP systems can be checked statically or compiled into synchronous programs.

7 Conclusion and Future Work

This paper is very preliminary and many problems have to be solved if our proposal is to become a practical solution for the design of embedded systems. Let us quote some of them:

- In the above description, there is nothing to bound the size of a channel. One needs a way of constructing schedules under the additional constraint that each channel uses no more than a given amount of memory. Let us note that the inverse problem (finding the amount of memory needed to support a given schedule) has been the subject of much research and that good solutions are known [12, 5].

- One may also want to constrain the schedule to use no more than a given number of functional units. It is well known that constraining the amount of memory also limits the degree of parallelism and hence the number of functional units. There might be better solutions than this indirect approach.

- For complexity reasons, as soon as resources are in a fixed finite amount, the restriction to affine schedules is no longer tenable. One has to use *many-dimensional schedules*. While there are methods for constructing such schedules [8], building their modular extension is by no means obvious.

- On a more practical point of view, the use of the Fourier-Motzkin algorithm for doing projections is not possible beyond a given complexity. One must experiment with more efficient methods, like Chernikova's algorithms.

- Many problems in, e.g., image processing, are outside the regular (or polytope) model. One may sometime obviate this difficulty by overestimating dependences, or by encapsulating the irregular program parts, or

by asking for help from the programmer. There is much work to be done in this direction.

References

[1] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50. ACM Press, April 1991.

[2] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing*, October 2003. to appear, `http://www.prism.uvsq.fr/users/cedb/`.

[3] Albert Benveniste, Paul Caspi, Nicolas Halbwachs, and Paul Le Guernic. Dataflow synchronous languages. In *A Decade of Concurrency, Reflexions and Perspective. REX School / Symposium*, pages 1–45, 1993. LNCS 803.

[4] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15, 1966.

[5] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. In 6*th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2003)*, October 2003.

[6] Paul Feautrier. Semantical analysis and mathematical programming; application to parallelization and vectorization. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Workshop on Parallel and Distributed Algorithms, Bonas*, pages 309–320. North Holland, 1989.

[7] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[8] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[9] Martin Griebl, Paul Feautrier, and Christian Lengauer. Index set splitting. *Int. J. of Parallel Programming*, 28(6):607–631, 2000.

[10] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21, 1978.

[11] G. Kahn. The semantics of a simple language for parallel programming. In North Holland, editor, *IFIP'94*, pages 471–475, 1974.

[12] Vincent Lefebvre and Paul Feautrier. Optimizing storage size for static control programs in automatic parallelizers. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Europar'97*, volume 1300 of *LNCS*, pages 356–363. Springer, August 1997.

[13] Hervé Leverge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[14] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of Efficient Nested Loops from Polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[15] P. Quinton and T. Risset. Structured scheduling of recurrence equations: Theory and practice. In *Proc. of the System Architecture MOdelling and Simulation Workshop*, Lecture Notes in Computer Science, 2268, Samos, Greece, 2001. Springer Verlag.

[16] Patrice Quinton. The systematic design of systolic arrays. In F. Fogelman, Y. Robert, and M. Tschuente, editors, *Automata networks in Computer Science*, pages 229–260. Manchester University Press, December 1987.

[17] A. Schrijver. *Theory of linear and integer programming*. Wiley, NewYork, 1986.

[18] Robert E. Tarjan. Graph theory and gaussian elimination. In J. Bunch and D. Rose, editors, *Sparse Matrix Computations*. Academic Press, 1976.

[19] Rémi Triolet, François Irigoin, and Paul Feautrier. Automatic parallelization of FORTRAN programs in the presence of procedure calls. In Bernard Robinet and R. Wilhelm, editors, *ESOP 1986, LNCS 213*. Springer-Verlag, 1986.

[20] D. Wilde. A library for doing polyhedral operations. Technical Report 785, Irisa, Rennes, France, 1993.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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