Scalable and Structured Scheduling

Paul Feautrier *

July 20, 2006

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 accepts 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 large program can be divided in smaller units (processes) which can be scheduled separately. This is *structured scheduling*. Second, one can use projection methods for solving linear programs incrementally. This is specially efficient if the dependence graph is sparse.

Keywords Structured scheduling, automatic parallelization, scalability.

1 Introduction

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 an abstract and partial representation of the behaviour of a program, and can be converted either into a loop program, suitable for execution on a DSP or VLIW processor, or into a VHDL specification at the RTL level, suitable as the input of a CAD system.

Roughly speaking (the details can be found later or in the original papers [8, 9]), scheduling proceeds in two phases:

*`Paul.Feautrier@ens-lyon.fr`, LIP / project Compsys / Ecole Normale Supérieure de Lyon / INRIA / Université Lyon I

- Each dependence in the program is converted into a set of linear constraints on the coefficients of the schedule;
- These constraints are solved by some linear programming algorithm like Fourier-Motzkin or the Simplex or Chernikova's algorithm.

The problem I want to address in this paper is the scalability of scheduling. As a rule of thumb, the number of constraints is proportional to the number of dependences, which is quadratic in the size of the program. The Simplex can be more than exponential in the worst case, but both experience and theory show that its most likely complexity is $O(m^3)$, where m is the number of constraints. It follows that the complexity of the method is of the order of the sixth power of the size of the program. Hence the interest of *modular* scheduling; if a large program can be split into several modules which can be scheduled almost independently, large speedups can result.

In the next sections I define which type of modules are suitable for parallel programming and review the basic scheduling algorithm. In section 5, I explain how to improve the scheduling time of one module provided that the dependence graph is sparse. Section 6 explains how to do modular and structured scheduling. In Sect. 7, I explain how to bound the size of communication channels. In the conclusion, I report on an experiment, I present some open problems and discuss future work.

2 Communicating Regular Processes

Modular compilation is a very well-known technique, which dates back to the early days of Fortran. For most sequential languages, the module is the function. In fact, provided one has designed a clever calling interface, functions can be compiled independently of each others[1]. Nevertheless, the compiler output is not an executable programs. One needs another tool, the *linker*, whose goal is mainly to plug the addresses of the called functions into the callee code. Compilation is modular, but linking is not.

In the case of parallel programming, functions are not suitable as modules. If functions are handled as black boxes, then one may lose many opportunities for parallelization. If one opens the box, then modularity disappears. There is, however, another possibility: processes and network of processes. Process networks abstract from

[1] In modern languages, the need for accurate type checking induces more complex relations between modules.

the behaviour of message-passing architectures: each process sits in a processor and has its own private memory. Processes communicate only by sending messages over ports and through channels. Message passing libraries and languages abound. Libraries range from the basic socket system of Unix to MPI and BSP. The best known message passing language is Occam. Such systems have almost no restrictions on what the programmer can do and may suffer from non-deterministic behaviour and deadlocks. The analysis and debugging of programs written using these libraries is very difficult.

Kahn Process Networks (KPN) [11] are an attempt to impose determinism by construction: channels are perfect FIFO queues, and each channel can have only one reader and one writer. The static analysis of KPN is still difficult, because send and receive operations can only be correlated by counting messages, which may lead to non-linear counting functions and may even be impossible in the presence of conditionals.

My proposal is to use processes as modules, but to change the semantic of channels. In this paper, a channel is an array of arbitrary dimension, which is used in write once/read many mode. This constraint is enough to insure determinism (the proof is rather technical and will be published elsewhere). Read and write operations are now correlated by comparing array subscripts. To insure the possibility of precise analysis, subscripts must be affine functions of surrounding loop counters, i.e. the processes must be static control programs in the sense of [7].

Let me emphasize the fact that the language of Communicating Regular Processes (CRP) 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.

The emphasis of this paper is not on simulation or direct implementation, but on static analysis and compilation. 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 Program Structure

An application is a collection of function and process definitions. Several definitions can be collected in a module (usually a file); an application can be composed of an arbitrary number of modules. Like in C, process and function definitions are top level objects and do not nest.

2.2 Syntax

The basis of the syntax of CRP is ANSI C[2]. There are, however, a few new keywords: `process, inport, outport, channel`. All these are reserved and are considered as additional "storage class specifiers" in the C grammar.

To get a feeling for the syntax of CRP, the reader is referred to Fig. 1 which describes a producer / consumer system.

2.3 Semantics

2.3.1 Basic Types

The basic types are `void`, `int` and `float` in various sizes (`long, long long, short, char, double`). The addition of another basic type, `fix`, is contemplated for embedded system design.

2.3.2 Arrays and other Data Structures

There is an array constructor, `[]`, with the same properties as in C. However, the rules for dimensions are much more permissive than in C. In fact, in many cases, the compiler infers the size of the array from the way it is used. Similarly, there is a structure constructor, with the same syntax and properties as in C.

In the present version of the compiler, pointers are ignored.

2.3.3 Functions

User-defined functions are inlined. Hence, recursion is forbidden.

One may use blackbox functions, which are handled by the system as if they were pure (no modifications of the actual parameters or of global variables). In embedded system design, such functions may be

[2] This choice is completely arbitrary. Fortran or Pascal would do as well, but C is the language of choice for embedded systems.

useful for representing the use of Intellectual Properties[3]. For instance, instead of writing:

```
x = y + z;
```

one may want to write

```
x = adder_7bits(y, z);
```

if one knows that 7 bits are enough for this particular addition. Another use of blackbox functions is to hide an irregular piece of program.

2.3.4 Processes

A process is a sequential program which can communicate with other processes through channels (see 2.3.5). All variables are local to one and only one process and are not visible from other processes[4].

Besides operative statements, a process can include process start statements, which have the same syntax as a void function call. Process start statements are not part of the control flow of the surrounding process. In effect all the process start statements in an application are collected and executed immediately at application start time. One can define a process start graph, which must be a DAG.

The operating code of a process must be *regular*, or have static control [8] in the following sense:

- Statements are assignments statements and regular loop statements. All variables are considered part of some array, scalars being zero-dimensional arrays.
- 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).

[3] IPs are ready-made hardware modules, usually sold as hardware description language files.

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

2.3.5 Channels

A channel is the only medium of communication between processes. A channel can be viewed as a write-once/read-many array of indefinite dimension. Each cell has a (virtual) full/empty bit. At application start time, all such bits are set to "empty".

- A write to an empty cell defines its value and sets the control bit to "full".
- A write to a full cell generates an error.
- A read of an empty cell stalls the reading process until the cell is filled.
- A read of a full cell is immediately satisfied.

There is no way of emptying a cell.

The reader is cautioned that this is just a specification of the behaviour of a channel. The actual implementation may be quite different. In fact, the target in this work is a synchronous implementation, in which a processor never has to wait for an empty cell.

A channel may have any number of readers, and there are no constraints on the reading patterns. Reading is not destructive: a value remains available at least as long as some process may have some use for it.

2.3.6 Ports

A port is an interface between a process and a channel. It allows, *inter alia*, that a process be instantiated several times, each instance being connected to different channels. Ports are only allowed as formal parameters to processes.

When connecting ports and channels, one must verify (statically) that the two entities have the same (data) type and dimension. Channels play the role of actual parameters to the port formal parameters. 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.

The usual rule of visibility applies to ports and channels. Let `P` be a process in which a channel `c` is defined. The only processes which can access `c` are `P` itself and processes which are started by `P` and have a port connected to `c`.

3 Dependences

3.1 Notations

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:

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

and the set of operation of a CRP system is $E = \cup_P E_P$. In more abstract contexts, I may simply write $u \in E$ for an arbitrary operation. u and v being operations of the same process P, one writes $u <_{\text{seq}} v$ iff u executes before v. $<_{\text{seq}}$ is a strict total order on E_P. On the other hand, there is no *a priori* ordering of operations in different processes.

In this paper, the most important operations are reads and writes to some channel `A`. I assume that in each statement S there is at most[5] one access to A, with subscript function f_{SA}, or simply f_S when A is clear from the context. Namely, operation $\langle S, i\rangle$ access $A[f_{SA}(i)]$, and f_{SA} is affine.

Let $\mathcal{W}(A)$ denotes the set of statements that write into channel A, and $\mathcal{R}(A)$ denotes the set of statements that read from A. The set:

$$\mathcal{F}(A) = \cup_{S \in \mathcal{W}(A)} \{f_{SA}(i) \mid i \in D_S\}$$

is the *write window* of A. Similarly the following set:

$$\mathcal{G}(A) = \cup_{S \in \mathcal{R}(A)} \{f_{SA}(i) \mid i \in D_S\}$$

is the *read window* of A. If the following constraint:

$$\mathcal{G}(A) \subseteq \mathcal{F}(A), \tag{1}$$

[5]The general case (several accesses) poses no theoretical problem, and its cost is just one more loop in the compiler. For instance, the example in Fig. 1 does not observe this restriction. However, lifting it would greatly complicate notations and explanations. The same observation is true for iteration domains.

is not satisfied, it is clear that some process will block for ever when accessing a memory cell in $\mathcal{G}(A) - \mathcal{F}(A)$. This constraint must be checked by the software before scheduling. On the other hand, the cells in $\mathcal{F}(A) - \mathcal{G}(A)$ are useless. In what follows, I will assume that this set is empty, or that all channel cells are useful.

3.2 Data dependences

Data dependences were defined, as early as 1966, for the purpose of parallelization [3]. 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 changes 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 found a purely syntactical criterion for independence. 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 that are modified by u. It is easy to prove that u and v are independent (u being executed first) if 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) are 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 not independent and if $u <_{\text{seq}} v$.

The data dependence relation can be split into pieces according to the source statement S, the sink statement T, the array references and the depth of the dependence. Each piece is a polytope in $D_S \times D_T$ which can be checked for emptiness by any linear programming algorithm, or by more specialized algorithms like Banerjee's test or the gcd test.

3.3 Communication Dependences

Assume now that the variable which causes the dependence is a channel cell. One can still say that some operations are in dependence, with the same definition as above. The presence of an output dependence indicates that the "write once" condition is not respected, and generates an error. One must impose a flow dependence (no read can be executed before the first and only write), and the absence of output dependences, in conjunction with (1), is enough to insure the absence of 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.

4 Scheduling

4.1 Target Architectures

In contrast to the above programming model, most of today digital 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. More precisely, it has a datapath, comprising operators, registers, memory, and some interconnect, and a control part, which can be coded either as a program, as in a VLIW processor, or as a finite state machine, as in an ASIC or FPGA.

4.2 Schedules

A schedule is a function which assigns a starting time to each operation of an application. In other words, a schedule is a function from E to the set of time values, $\mathcal{T}$. But what is time? One possibility is to consider physical time. In that case, $\mathcal{T}$ is the set of non-negative integers, time being measured in clock cycles. This approach is suitable when dealing with fine-grain systems in which execution time is well defined (typically one 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, $\mathcal{T}$ is any ordered set. θ being a schedule, the associated order is:

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

The favorites for $\mathcal{T}$ are again $\mathbb{N}$ and $\mathbb{N}^d$, lexicographically ordered. The 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 $\theta(S, x)$ is an affine form in the iteration vector, x:

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

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) becomes affine, and that powerful results from the theory of linear inequalities, like Farkas lemma [15], can be used to characterize the solutions. The reader is referred to [8, 9] for details. A short review of the method is given below.

4.3 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:

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

Each such constraint represents of the order of Card δ 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 small finite set.

This compression can be done either by the vertex method [14] or by making use of the following version of Farkas lemma [15]:

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. λ_0 and the vector λ are known as the Farkas multipliers.

To apply this result, let x be the concatenation $(x_S, x_T)^T$ of the iteration vectors of S and T. Let $A_{ST}x + b_{ST} \geq 0$ be the system of constraints that defines the dependence relation from S to T. One first checks that this system is feasible. If not, the dependence does not exist and imposes no constraints on the schedules.

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

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

which gives the equivalent formulas:

$$\begin{aligned} \lambda A_{ST} &= (-h_S, h_T), & (5)\\ \lambda b_{ST} + \lambda_0 &= k_T - k_S - 1. & (6) \end{aligned}$$

For regular programs, A_{ST} and b_{ST} 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 source program. But the failure can 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 [9]. I will ignore this difficulty here.

5 Scalability

The number of unknowns 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.

5.1 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 occur 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 [8] and has proved to be very efficient in practice.

But one can go farther than that, by using elimination (or projection) methods to finish the job. The projection of a set D in $\mathbb{R}^{n+1}$ along its first dimension is:

$$P = \{x \mid \exists y : y.x \in D\}. \tag{7}$$

It is well known that if D is a polyhedron, so is P. Projection along y is equivalent to the elimination of y in the constraints that define D. 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 solution is given as a disjoint union of polyhedra; reconstructing P from its subsets is not easy.
- Lastly, if one knows the Minkowski representation of D [15], it is easy to find the Minkowski representation of P by projecting the vertices and rays of D. From that, one can reconstruct an irredundant constraint system with the Chernikova algorithm.

In my implementation, I use the last method, which has the advantage of being much faster than the Fourier-Motzkin algorithm, and of always generating irredundant constraints.

Whatever the projection algorithm, when 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 $y.x \in D$. Therefore, after eliminating all unknowns but one, one can select a point in the last – one dimensional – projection, and then back propagate 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. Usually, selecting the lowest possible value for each unknown is a good choice. This technique is not necessary in the case of the Farkas multipliers, since their value is irrelevant for the final schedule. It will however be used in the next section.

5.2 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 often is sparse, or, rather, block sparse. In fact, a dependence from S to T being given, the resulting constraints can be written as:

$$M_{ST}(h_S, k_S)^T + N_{ST}(h_T, k_T)^T \geq 0. \tag{8}$$

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

If the scheduling problem is solved by any 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 difference 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 system solution [16]. However, if the Simplex is replaced by any kind of successive elimination algorithm, one may hope to control fillup by careful selection of the unknowns to be eliminated at each stage. I propose here the following algorithm:

- Partition the program into groups of statements $\mathcal{S}_k, k = 1, N$. Let h_k be the concatenation of the timing vectors of the statements in $\mathcal{S}_k$.
- For each group $\mathcal{S}_k, k = 1, N$ do:

- Collect all the rows of M where h_k has a non-zero coefficient. Remove these rows from M.
- Eliminate h_k from the collection. Add the result of the elimination to M.
- Remember the bounds for h_k.

- If the final system is trivially infeasible (like $-1 \geq 0$) stop. No schedule exists.
- For each group $\mathcal{S}_k$ in reverse order do:
  - The bounds for h_k are constants. Select a value within the bounds for h_k (e.g. the lower bound).
  - Substitute these values in all other bounds.

5.3 Choosing the next victims

The partitioning of the source program into group of statements is necessarily a heuristic process. In the present version of the software, I have implemented two very simple approaches: one-by-one elimination, the selected statement being the one which generates the smallest system of constraints, and elimination of all statements at the same time. Experimental results show that the best heuristic strongly depends on the shape of the dependence graph. One can in fact observe that loosely coupled statements, like statements in a serie-parallel structure, can be eliminated individually at almost no cost, while there is no advantage in separating statements forming a clique. It should be possible to deduce from these observations rules for constructing an efficient elimination order, but this is left for future work.

6 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, a *linker* is needed to finalize the construction of the target program. Modularity has many advantages. Modules promote reuse. Also, in case of a modification, one recompiles only the affected module(s). But the most important effect of modularity is one of discipline: since one usually forbids access to the local variables of a module from another module, the dependence graph is simplified and its scheduling is easier.

6.1 Channel schedules

As has been said earlier, the natural unit of compilation for a parallel program is the *process*, and dependences between processes occur only when they share a channel. The scheduling constraints (2) are equally applicable to the channels, and show that the schedules of different processes are not independent. This precludes modular scheduling, unless some "insulation" between processes is provided.

Observe that in the CRP model, each cell in channel A is written only once at a definite time by an operation from its unique writer process. Therefore, one can postulate the existence of a channel schedule $\theta(A,x)$ such that the value $A[x]$ is guaranteed to be defined at time $\theta(A,x)$ (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 only be piecewise affine. This problem can be taken into account along the lines of [10] 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 $S : A[f_{SA}(i)] := \cdots$ be a statement that writes into A:

$$i \in D_S \Rightarrow \theta(A, f_{SA}(i)) \geq \theta(S,i). \tag{9}$$

- On the read side, a cell cannot be read before it is available. Let $S : \cdots := \cdots A[f_{SA}(j)] \cdots$ be a statement that read A:

$$j \in D_S \Rightarrow \theta(S,j) \geq \theta(A, f_{SA}(j)) + 1. \tag{10}$$

The 1 in formula (10) is intended to represent a propagation delay through the channel. I have arbitrarily inserted this delay on the read side, but many other configurations can be used without changing the overall method. The present choice reflects the fact that a value can be written in a register at the end of a clock cycle, and is available for reading at the beginning of the next cycle.

6.2 The Modular Scheduling 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_{ST} or N_{ST} blocks in formula (8). 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:

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

   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 library), 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 is likely to be 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 process 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 Fig. 2, where a system composed of 3 processes, `P, Q`, and `R` is represented. `P` and `Q` send information to `R` through channels `A` and `B` respectively. In the constraint matrix, there is a column for the coefficients of the schedules of each of these five objects. The schedule of `P`, for instance, appears in a bloc of inner constraints, and also in a block coupling it to the schedule of `A`. Eliminating `P` gives constraints on the schedule of `A`: for instance, `P` may impose an upper bound to the rate of `A`. Similarly, `R` may impose a relation between the rates of `A` and `B`, e.g. if `R` reads two tokens from `A` to one from `B`. After eliminating `P`, `Q` and `R`, one obtains the communication constraint matrix of Fig. 2, which can be solved for the channel schedules. If the communication constraints are not feasible, it means that there is no affine schedule for the system. This probably indicates a deadlock; note however that the system may have a piecewise affine schedule or even a non linear schedule, which are not found by the proposed method.

Consider now the example of Fig. 1. The first step is to compile the two processes. Let:

$$\theta(W,i) = \alpha i,\ \theta(Z) = \beta,\ \theta(M,i) = \gamma i + \delta,\ \theta(A,x) = ax + b.$$

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

$$i \geq 0 \Rightarrow ai + b \geq \alpha i.$$

Application of the Farkas algorithm gives[6]: $b \geq 0$ and $a \geq \alpha$ after elimination of the multipliers. After elimination of α, the only remaining constraint is $b \geq 0$.

In the consumer there are a flow dependence from Z to M, a flow dependence from R to itself, and two communication dependences from A to R. The corresponding constraints are:

$$\begin{aligned} i \geq 0 \Rightarrow \beta + 1 &\leq \gamma i + \delta, \\ i + 1 \leq i' \Rightarrow \gamma i + \delta + 1 &\leq \gamma i' + \delta, \\ i \geq 0 \Rightarrow ai + b + 1 &\leq \gamma i + \delta, \\ i \geq 0 \Rightarrow a(i+2) + b + 1 &\leq \gamma i + \delta. \end{aligned}$$

Application of Farkas lemma gives:

$$\begin{aligned} \beta + 1 &\leq \delta, \\ 1 &\leq \gamma, \\ a &\leq \gamma, \\ b + 1 &\leq \delta, \\ 2a + b + 1 &\leq \delta. \end{aligned}$$

Elimination of α, β and γ gives an empty system. The only communication constraint is $b \geq 0$ whose smallest solution is $b = 0$. From there, one may reconstruct the schedules:

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

[6] In very simple cases like these, one can apply the vertex method: to be true everywhere, it is sufficient that these inequalities be true at $i = 0$ and $i = \infty$.

This solution is not satisfactory, since one has to deposit an infinite number of values in A in one clock cycle, and hence, the size of A must be infinite. For a practical implementation, one needs a way of bounding *a priori* the channel size. A solution to this problem is presented in Sect. 7.

6.3 Structured scheduling

In software engineering, and also when designing an embedded system, a common practice is to wrap up several modules as a *component* and to reuse it several times in the same or in different designs. For instance, when designing a video streamer, one may build a downsampler, which transforms a high resolution pixel stream into a standard resolution one. One may then instantiate three downsamplers, one for each primary color. The downsampler itself is made of two processes, a vertical filter and a horizontal filter.

Such a structured or hierarchical design can be handled in the CRP model at no extra cost. This is due to the visibility rules of Sect. 2.3.6. Consider for instance the following specification:

```
process down1(inport int s[], outport int t[]){...}
process down2(inport int u[], outport int v[]){...}

process up(inport int A[], outport B[]){
    channel int X[];

    down1(A, X);
    down2(X, B);
}
```

The `up` process receives data through its `A` port, and sends it directly to process `down1`. The output of this process is sent to `down2`, whose output is directly returned by `up` through port `B`.

The important point is that the internal channel `X` is not visible from any process except `up, down1` and `down2`. Hence, once `down1` and `down2` have been compiled, giving as results two sets of communication constraints, one does not have to look anywhere else before eliminating `X`. The result is the set of communications constraints for `up` which can be used bottom-up for further scheduling. One can see that the global schedules are found by a postfix traversal of the process start tree.

7 Bounding the Size of a Channel

The definition of CRP says that a channel is unbounded. This is necessary, since in many embedded systems, one must use an infinite loop, which, in many cases, represents the flow of time and the stream of data. In designing an ASIC or FPGA circuit, or when writing a program, it is not possible to provide an unbounded amount of memory. The solution is to analyze the lifespan of each value, and to assign a memory cell to it only from its definition up to its last use. This can be done *a posteriori*, when schedules are known [5]. The danger is that, when the application has much parallelism, the inferred memory space may be too large or still be infinite.

Here I want to investigate another approach, in which the memory size is given *a priori* (e.g. as part of an architecture exploration process), the problem being that of finding a schedule that can run in the given memory. This is a very difficult problem, which has already been studied in [17]. The solution given there relies on *ad hoc* hypotheses, which may or may not be verified in real life programs. I will use here a similar approach. I will first identify reasonable assumptions on the code of a CRP. I will then solve the problem under these assumptions.

The first observation is that a sequential program[7] can have only one infinite loop. The reason is that an infinite loop never terminates. Hence everything that follows it can be removed as dead code. Similarly, if an infinite loop is enclosed in another loop, the outer loop will only execute one iteration and can be removed[8]. It follows that the shape of a process is as follows:

- First, some finite calculations, perhaps taking care of initialization;
- Then, at most one infinite loop;
- The body of the infinite loop may contain finite loops.

Let i be the counter of the infinite loop. If the process under consideration writes into some channel A, then i must occur in at least one of the subscripts of A; if this were not so, the "write once" condition could not be satisfied. This dependence on i may be indirect, as in:

```
for(i=0;:i++)
  for(j=i;j<=i;j++)
    a[i] = ...;
```

[7]Remember that the semantic of a CRP is defined by its sequential execution.

[8]Take care that the concept of an infinite loop is semantical not syntactical. There is no infinite loop in the following fragment: `for(i=0;;i++) for(j=i; j<n; j++) S;`. A loop nest is infinite if the corresponding iteration domain has a ray.

I assume that such strange constructions have been removed by preprocessing (detection of hidden equations). I also assume that i occurs in the first subscript of A. This can always be obtained by subscript permutation and is not really a restriction. My aim is then to bound the first dimension of A. Remember that this may not always be possible.

The channels in CRP are write once. This means that memory cells are in one-to-one correspondence with the *values* that flow through the channel. If the channel is implemented as an array with bounded first subscript, the runtime system has to allocate a sub-array $A[x_1][*]$ each time x_1 changes, and deallocate it as soon as it becomes useless. In the interest of performance, the mechanism for allocation and deallocation must be as simple as possible. Tools like `malloc` and `free` are out of the question. One option is to use a circular buffer (or modulo allocation). Let `a` be the buffer associated to A, d its dimension, and B be the size of its first dimension. The value $A[x]$ is stored in cell $\mathtt{a}[x_1 \bmod B][x_{2..d}]$. A similar modification must be applied to all read accesses to A. It is clear that the resulting program may not be equivalent to the original, since a value may be overwritten before all reads have been executed. Hence, to avoid such errors, the process and channel schedules must satisfy additional constraints.

To each channel let us associate two functions α and ϕ. $\alpha(A, x_1)$ is the time at which the sub-array $A[x_1][*]$ is allocated. Similarly, $\phi(A, x_1)$ is the time at which $A[x_1][*]$ is deallocated. It is legitimate to assume that α and ϕ are affine. To fulfill the "write once" condition, x_1 must be unbounded, and if α were decreasing, some allocations would occur at negative time, i.e. before the application start time. Hence, α and ϕ are monotone increasing. These two functions are not independent: deallocation of $A[x_1][*]$ occurs at the time $A[x_1+B][*]$ is allocated, since these two array slices occupy the same memory cells. Hence, assuming that allocation and deallocation take one clock cycle:

$$\alpha(A, x_1 + B) + 1 = \phi(A, x_1). \tag{12}$$

This constraint is enough to ensure that α is strictly increasing. Obviously, one cannot write into an array cell before its allocation, and one cannot read it after it has been deallocated. This gives the two constraints:

$$\begin{aligned} \forall i \in D_W : \alpha(A, f_{AW}(i)) &\leq \theta(W, i), & (13)\\ \forall i \in D_R : \theta(R, j) &\leq \phi(A, f_{AR}(j)). & (14) \end{aligned}$$

These constraints are in the same form as (10) and (9) and can be subjected to the Farkas algorithm.

Lemma 2 *The conjunction of (12, 13, 14) is sufficient to insure the correctness of the transformed program.*

> **Proof** Let $A[x]$ be a value which is written to $\mathtt{a}[x_1 \bmod B][x_{2..d}]$. An error occurs if a write to a cell $A[x_1+kB][x_{2..d}]$ is executed between instants $\alpha(A, x_1)$ and $\phi(A, x_1)$. k is an arbitrary integer; observe that k cannot be null: this would imply that $A[x]$ is written twice, and is in contradiction to the "write once" assumption. Assume that k is positive, and let $\langle W', i' \rangle$ be the offending write:
>
> $$\theta(W', i') \leq \phi(A, x_1).$$
>
> By (13), $\alpha(A, x_1+B) \leq \alpha(A, x_1+kB) \leq \theta(W', i')$, since α is increasing, and $\alpha(A, x_1+B) = \phi(A, x_1)+1$, a contradiction.
>
> The proof is similar for the other possibility, $k < 0$. ∎

As a consequence of (12), one can dispense with ϕ in favor of α. (13) is a constraint on the unique writer to A, while (14) pertains to all its readers. Modularity is thus preserved, After elimination of the inner schedules, and before solving the communication constraints, one identifies the allocation coefficients of the several ports of A, in the same way that one identifies the coefficients of A clocks.

Return to the example of Fig. 1, and assume that 3 memory cells are allocated to channel A. Let $\alpha(A, x) = a'x + b'$ be the allocation function for A. In this case, (13) is:

$$i \geq 0 \Rightarrow a'i + b' \leq \alpha i.$$

From this follows $a' \leq a$ and $b' = 0$. There are two instances of (14), one for each occurrence of Y in statement M:

$$\begin{array}{rcl} i \geq 0 & \Rightarrow & \gamma i + \delta \leq a'(i+3) + b' + 1, \\ i \geq 0 & \Rightarrow & \gamma i + \delta \leq a'(i+5) + b' + 1. \end{array}$$

It is easy to see that the second inequality is redundant, simply because, under our hypotheses, a' is positive. Applying Farkas lemma and adding the original constraints give:

$$\begin{array}{rl} \gamma \leq a', & \delta \leq 3a' + b' + 1, \\ \beta + 1 \leq \delta & 1 \leq \gamma, \\ a \leq \gamma & b + 1 \leq \delta, \\ 2a + b + 1 \leq \delta & \end{array}$$

Elimination of α, β and γ gives:

$$\begin{aligned} 1 &\leq 3a' + b' + 1, \\ 2a + b + 1 &\leq 3a' + b' + 1, \\ 1 \leq a', & \quad\quad a \leq a'. \end{aligned}$$

The communication constraints from the producer side are:

$$\begin{aligned} b = 0, & \quad\quad b' = 0, \\ a' &\leq a. \end{aligned}$$

The solution of the system of communication constraints is: $a = a' = 1$, $b = b' = 0$, from which follows $\alpha = 1, \beta = 0, \gamma = 1, \delta = 3, \theta(W, i) = i$ and $\theta(M, i) = i + 3$. Now the producer sends one value at each clock cycle through the channel. This value must be buffered to be used three clock cycles later. The reader may care to redo the computation with other values of the channel size. What is the minimum size for which schedules still exist?

The system of constraints which has been constructed above may be infeasible. This may indicate either a deadlock induced by insufficient buffer space, or an application for which no finite memory implementation exists. In fact, the allocation scheme which is proposed here implies that the lifetime of any value is bounded:

$$\phi(A, x_1) - \alpha(A, x_1) = \alpha(A, x_1 + B) + 1 - \alpha(A, x_1) = \frac{d\alpha}{dx_1} B + 1.$$

Now consider the following piece of code:

```
for(i=0;;i++)
   for(j=0; j<i; j++)
      A[i][j] = ....;
```

Assume that the computation of the values of A enforces sequential execution, and also that each value is useful (i.e. that some reader process accesses it some time). Then the lifespan of $A[i][*]$ extends at least from the creation of $A[i][0]$ to the creation of $A[i][i-1]$, hence takes at least i cycles, which is not bounded.

8 Related Work

To the best of my knowledge, modular scheduling has not been considered in the classical literature. The reason is probably that in applications like job-shop scheduling, there is no reason to assume that the precedence graph has any special property. On the other hand, there has been several attempts at modular automatic parallelization.

In [18], 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 astride a procedure boundary.

Nearer to the subject of this paper, Risset and Quinton [13] have defined structured scheduling for *systems* in the ALPHA specification language [12]. 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 reduction of the memory footprint of an application is important for embedded systems for reasons of cost and energy consumption. Many authors have considered it (see [5] and its list of references) mainly assuming that the schedule is known. Thies et. al. [17] have considered the problem of finding the schedule when the memory size is bounded (as is done here) and the simultaneous optimization of the memory size and the schedule. Their solution relies on the assumption that array subscripts are increasing functions of the loop subscripts; my assumption that the allocation function is increasing is similar but more natural than their hypothesis.

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 problem with KPNs, as with any other system in which processes communicate through message streams, is that analysis is only possible if the compiler is able to infer the correspondence between send and receive operations. In fact, as any practioner of message passing programming knows, having a process receive the wrong message is the main source of errors. In KPNs, this phenomenon is mitigated by having only one reader and one writer per channel. There is still the possibility of an error in message order. CRP goes one step further in this direction by associating messages to locations in memory, thus rendering their order immaterial.

9 Conclusion and future work

In this paper, I have proposed both a language, CRP, and a scheduling method for structured specification of process systems. A prototype scheduler, SYNTOL, has been implemented and was used to run all the examples in this paper. Since processes are scheduled independently, the method promotes reuse, avoids complete recompilation in case of a local modification, and shortens the compilation time.

9.1 Experimental results

To justify the above claim, I have set up the following somewhat artificial experiment. An equalizer is composed of a signal source, a mixer, and several 6-taps FIR filters connected in parallel. The input signal is fed to all filters. The output signal of each filter is sent to a mixer, where it is weighted and added to the outputs of other filters. The number of filters can be easily changed. The wall-clock time for each compilation is given by the second column of Table 1, as a function of the number of filters. These numbers were obtained on a Pentium M 1.4 Mhz processor, running Linux, with 256 MB of memory. The software is mostly written in MuPAD, with the exception of linear programming kernels like PIP and the Polylib, which are written in C. These numbers are reproducible to about 1". As these numbers show, the compilation time grows almost linearly with the number of filters, the cost of one additional filter being about 13" of scheduling time.

As a comparison, I have implemented a one-process version of the application. Not surprisingly, the compilation time is better in the one filter case. This is due to the launching overhead of the compiler, which is paid four times in the many process version against one here. There is a crossover point when the third filter is introduced, and then an explosion in the scheduling time. In fact, the system created for the five filters case is too large for the present version of the Polylib. This phenomenon does not occur in the modular version. Here, the only problems whose size grows with the number of filters are the mixer scheduling system and the system of communication constraints. If these systems grow too large, it is always possible to separate the filters in several banks, and to use a cascading mixer to generate the output signal.

9.2 Future work

However, there are still many problems to be solved if this proposal is to become a practical solution for scheduling and high level synthesis. Let us outline some of them:

- One may want to constrain the schedule to use no more than a given number of functional units. Solutions are known for two particular cases of this problem. One may apply *software pipelining* to the innermost loop of each nest. The problem is that one may have to explore all legal loop permutations until a satisfactory solution is found.

If the loop nest has a high degree of parallelism, one can apply tiling to the parallel loops. Usually, programs obtained in this way have bad locality. More general solutions to this problem are sorely needed.

- 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 [9], building their modular extension is by no means obvious.
- Many problems in, e.g., image processing or software radio, are slightly 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.
- Although the problem of code generation from a schedule has been much studied since the pioneering paper of Ancourt et. al. [1] (see for instance a recent paper by C. Bastoul [2]), the solutions can still be much improved, especially by taking into account the peculiarities of the schedules and/or the underlying hardware. One possibility is to adapt the method of Boulet et. al. [4] to the direct construction of the control automaton.

A Appendix: the equalizer application

The schematics of the application is given in Fig. 3 for the three filter case. Circles and triangles are processes, and links represent channels. Note that the output channel of the source process is directly fed to all three filters. In the CRP formalism, there is no need of a triplicator.

All filters have the same code, which is given below. It is clear that the values in the `poids` parameter will differ from filter to filter.

```
process filtre(inport float entree[], outport float sortie[],
               float poids[7]){
     int i, j;
     float s[7];

     for(i=0;;i++){
Z:       s[0] = 0.;
         for(j=1; j<7; j++)
```

```
MAC:         s[j] = s[j-1] + entree[i+j-1] * poids[j];
W:       sortie[i] = s[6];
     }
}
```

The code of the mixer for the three filters case is:

```
void play(float s);
float w1, w2, w3;

process mixer(inport float x1[],
              inport float x2[],
              inport float x3[]){
 int i;
 for(i=0;;i++)
Q:  play(w1*x1[i]+w2*x2[i]+w3*x3[i]);
}
```

The connection between processes and channels is specified by the `main` process:

```
process source(outport float a[]);
process filtre(inport float x[], outport float y[], float poids[]);
process mixer(inport float u[], inport float v[], inport float w[]);

float p1[7], p2[7], p3[7];

void main(){
    channel float a[], b[], c[], d[];

 SO: source(a);
 F: filtre(a, b, p1);
 G: filtre(a, c, p2);
 H: filtre(a, d, p3);
 SI: mixer(b, c, d);
}
```

Lastly, here is the one process code for the same case:

```
void play(float s);
float f(int i);
float p1[7], p2[7], p3[7];

void main(){
    float a[];
```

```
    int i, j;
    float s1[7], s2[7], s3[7];
    float w1, w2, w3;
    for(i=0; i<5; i++)
Q:      a[i] = f(i);

    for(i=6;;i++){
P:      a[i] = f(i);

Z1:     s1[0] = 0.;
        for(j=1; j<7; j++)
MAC1:       s1[j] = s1[j-1] + a[i+j-1] * p1[j];

Z2:     s2[0] = 0.;
        for(j=1; j<7; j++)
MAC2:       s2[j] = s2[j-1] + a[i+j-1] * p2[j];

Z3:     s3[0] = 0.;
        for(j=1; j<7; j++)
MAC3:       s3[j] = s3[j-1] + a[i+j-1] * p3[j];

Q:      play(w1*s1[6]+w2*s2[6]+w3*s3[6]);
    }
}
```

Note that it would be straightforward to encapsulate the above design into a higher level process and duplicate it to create a stereophonic sound system.

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*, pages 23–30, Ljublana, October 2003.

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

[4] Pierre Boulet and Paul Feautrier. Scanning polyhedra without DO loops. In *PACT'98*, October 1998.

[5] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. In *6th 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. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[8] 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.

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

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A unified framework for schedule and storage optimization. In *ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, jun 2001.

[18] 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.

```
float f(int i);

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

```
process consumer
          (inport float Y[]) {
    float s;
    int i;
Z:  s = 0.;
   for(i=0; ;i++)
M:     s = 0.5*(s + Y[i]*Y[i+2]);
}
```

```
void main(){
    channel float A[];
P:  producer(A);
Q:  consumer(A);
}
```

Figure 1: A producer / consumer application

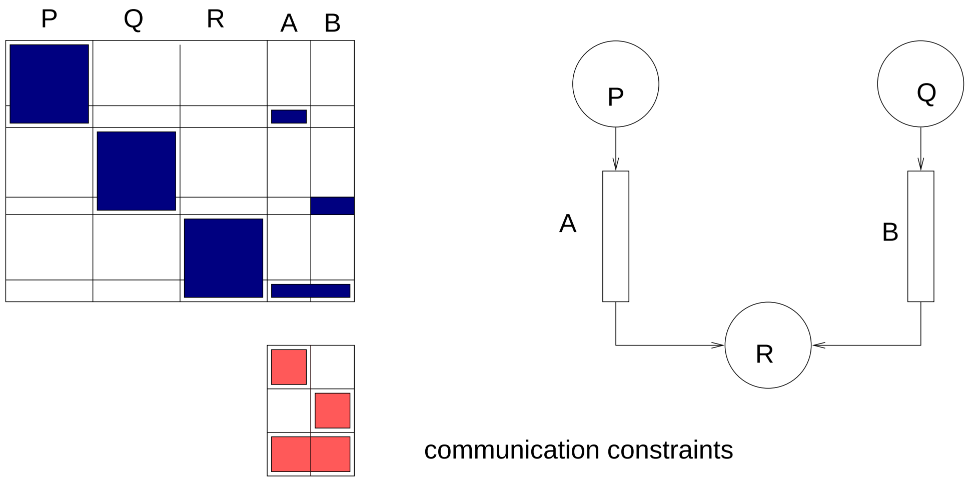

Figure 2: The constraint matrix

Filters	many processes	one process
1	23”	15”
2	34”	27”
3	46”	40”
4	60”	176”
5	73”	—

Table 1: Compilation times for the equalizer application

.

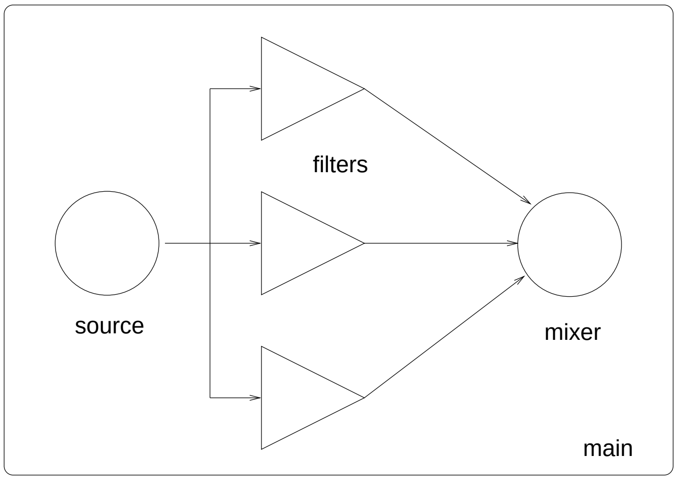

Figure 3: Schematics of the equalizer application