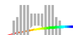
Polyhedral Streaming
The Communicating Regular Processes Approach

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr

April 12, 2014



Outline

Streaming Languages: Characterization
- Principles
- Alternatives and Constraints

Communicating Regular Processes
- CRP Example
- Modularity
- Code Generation
- Implementation

Conclusion and Future Work

Parallel Programming Models: Dataflow

A program in which no control is specified: an operation is "fired" as soon as its operands are available.

- The "grain" (size of an atomic operation) is a design choice in the model (made by the programmer):
  - for `pmake`, compilation of one file,
  - for a dataflow processor, one machine instruction.
- Parallelism: several operations may be ready to fire at the same time and can execute in parallel (resources permitting).
- Some examples:
  - Alpha (language of Quinton, Mauras, Risset, Rajopadhye),
  - Lucid, Sisal,
  - Synchronous languages such as Lustre and Signal.

Programming Model: Streaming

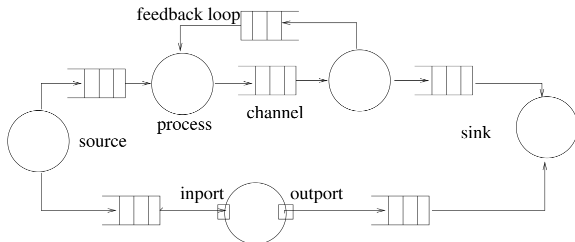
A streaming program is simply a program with an outermost infinite loop, consuming and producing data (i.e., streams).

- Depending on the model, this loop can be:
  - implicit: KPN, SDF, Lustre, Signal, StreamIt, Faust, Σ-C (?),
  - explicit: CRP (see later), OpenStream.
- Alpha (at least its theoretical framework) can have unbounded polyhedra as domains, thus can represent streaming.

Streaming Dataflow Programs

A dataflow program in which the set of operations is infinite.

- Especially adapted to signal processing.
- Usually presented as a network of processes (or filters) interconnected by channels. Processes consume data (tokens) from their input ports and create tokens on their output ports.



Processes

- Depending on the model, processes can be treated as:
  - black boxes: the compiler knows only the pattern of production and consumption,
  - grey boxes: some processing is hidden into library functions,
  - white boxes: all the process code is available to the compiler.
- In the last two cases, the process body must be written in some host language:
  - usually a restriction of a familiar sequential language, e.g., C or Java, sometime a language of guarded actions (Caltrop).

Communication

Communication style:

- Channels: sends & receives via a communication medium.
  - FIFO (e.g., KPN).
  - Sliding window (e.g., Stream-It, OpenStream).

  Explicit synchronizations but implicit information on data.
- Shared memory: read & write via addressable memory regions. Implicit synchronizations but explicit data accesses.

Communication implementation: each kind of communication can emulate the other.

Questions and Answers

Some properties that are obvious for sequential programs are conjectural for process networks:

- Determinism: does the network always give the same result, or always have the same behaviour? Sub-question: what is a result, or a behaviour?
- Deadlocks: does the network stall for ever?
- Does or can the network run in bounded memory?
- Synchronization/communication: how to implement channels?
- Optimization: can the grain be changed? Can the degree of parallelism be changed?

All these problems may be solved by the compiler, or left to the programmer, or solved – at least partially – by a library.

Communicating Regular Processes (CRP): Rationale

The problems with KPN:

- For KPN analysis, one has to pair a send and a receive: the send must be executed earlier than the corresponding receive.
- This is done by counting, but if the send or the receive is in a multidimensional loop nest, the count is a polynomial.
- The same problem occurs for OpenStream.

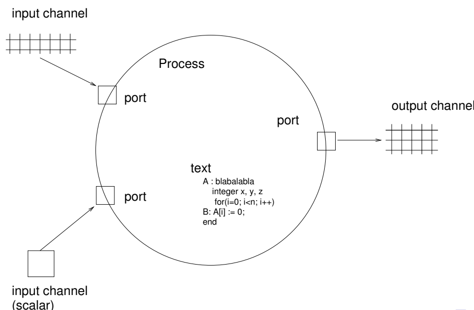
The CRP approach:

- Instead of `send` and `receive`, write or read to an unbounded array of (usually) the same dimension as the loop nest.
- It is the responsibility of the compiler to shrink each array as much as possible.
- Determinism guaranteed when in single assignment (SA).

Communicating Regular Processes

CRP are similar to Kahn Process Networks but:

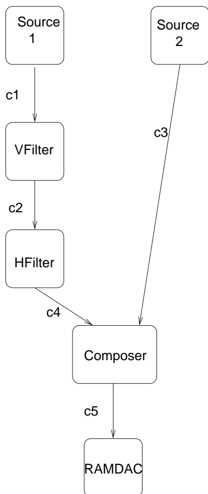
- Channels: unbounded shared arrays.
- Synchronizations: non-blocking write, blocking read. This can be implemented with full/empty bits, but there are other methods.
- write once, read many. Implies determinism.
- Affine subscripts & loops: SA can be checked mechanically.

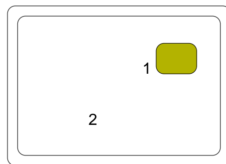


Problems and Solutions

- Analysis: static analysis and static scheduling are possible.
- Determinism: CRP are deterministic if SA property holds.
- Deadlocks: if scheduling succeeds, no deadlock in the network.
- Buffer size:
  - A partial solution: increase the size until deadlocks disappear.
  - Incidentally, resizing buffers adjusts the degree of parallelism.
  - Use array contraction and code transformations (e.g., tiling).

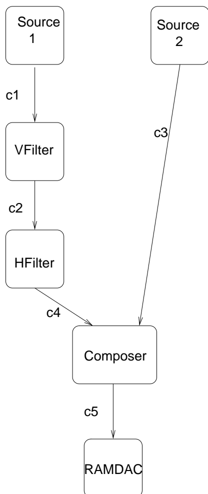
A Video Example, I





- Picture-in-picture.
- Two video sources.
- Source 1 is scaled down.
- Composer: for each screen pixel, select source 1 or 2.
- RAMDAC: paint the screen.

A Video Example, II

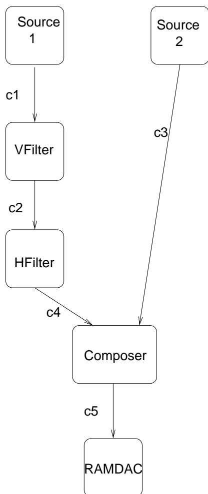


The HFilter

```
struct bigLine {
  char pixel[960];
}
struct smallLine {
  char pixel[120];
}
process HFilter(
    inport struct bigLine x[],
    outport struct smallLine y[]) {
 int i, j;

 for(i=0;;i++)
   for(j=0; j<120; j++)
     y[i].pixel[j] = x[i].pixel[8*j];
}
```

A Video Example, III



The Glue Code

```
void main(){
  channel struct bigLine c1[];
  channel struct bigLine c2[];
  channel struct bigLine c3[];
  channel struct smallLine c4[];
  channel struct bigLine c5[];

  source(c1, 1);
  source(c3, 2);
  VFilter(c1,c2);
  HFilter(c2, c4);
  composer(c3, c4, c5);
  ramdac(c5);
}
```

Schedules

- A schedule θ is a function that gives the start time of each operation.
- A schedule is specially useful for *synchronous* architectures, where time can be expressed in clock cycles.
- Schedules directly express parallelism: if $\theta(u) = \theta(v)$, u and v can be executed in parallel.
- Scheduling is a well known tool for acyclic programs. It can be extended to loop programs.

Scheduling Methods

Rule of Causality: if u and v are in dependence, v cannot start until u is finished.

- For regular loop programs, this rule generates a very large (even infinite) system of linear inequalities ...
- ... which can be compacted into a small finite set and solved by linear programing.
- One can easily add temporal constraints: latency or throughtput.
- Unsolved Problem: Handling resource constraints.

Scheduling Basics

- Dependence constraints come in the form:

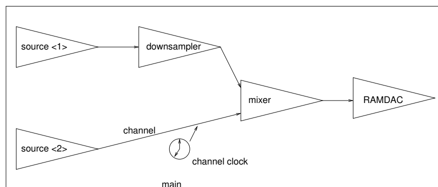
$$\forall i, j : \langle i, j \rangle \in D \Rightarrow \theta(R, i) + \delta \leq \theta(S, j),$$

  where R, S are instructions, i, j are iteration vectors, D is the dependence relation, and δ is the duration of instruction R.
- The schedule is assumed to be affine in the iteration vector.
- If one substitutes numerical values for the iteration vectors, one obtains a linear constraint for the coefficients of θ.
- By enumerating all values in D, one obtain a large (possibly infinite) linear program.
- If D is a polyhedron, this system can be compressed into a small finite set and solved by linear programming.

Modularity

- Scheduling is not scalable:
  - Number of dependences $\approx$ square of the size of the program;
  - Simplex $\approx$ cube of the number of constraints.
- Modularity promote reuse.
- The trick: divide an application into *processes* with multidimensional *write-once read-many* channels (the analysis is simpler than for Kahn Process Netwoks).
- The application stays deterministics.

Modular Scheduling

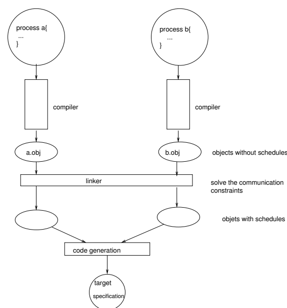


- Introduce channel clocks.
- Schedule `source`, `downsampler`, `mixer` `RAMDAC` independently, with the channel clocks as parameters.
- Schedule `main` (i.e., compute the channel clocks).
- Substitute the solution into the schedules for `downsampler` and `mixer`. The `source` processes are probably software and the `RAMDAC` is an IP.

Channel Clocks

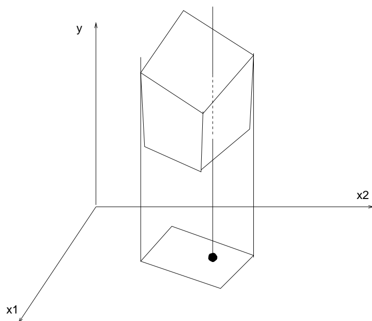
- The process schedules are not independant: they are linked by *communication dependences*.
- To restore a degree of independence, one introduces *channel clocks*. A being a channel, $\theta(A, x)$ is the date at which $A[x]$ is guaranteed to be available.
- A process schedule now depends only on the clocks of its incoming and outgoing ports.

Modular Scheduling



- Local scheduling for each process, communication schedules being kept as parameters
- Communication scheduling
- Back substitution into local schedules
- The complexity of scheduling becomes almost linear in the number of processes

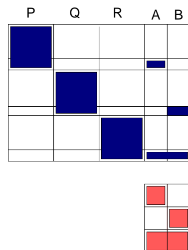
The Projection Method

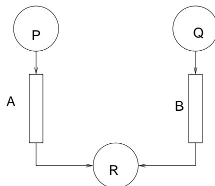


The trick: eliminate all inner schedules and get constraints involving only channel clocks.

- The projection of a polyhedron is a polyhedron.
- There are many projection algorithms:
  - Fourier-Motzkin (superexponential, redundant, easy to program).
  - Pip (fast, redundant).
  - Chernikova (fast, no redundancy).

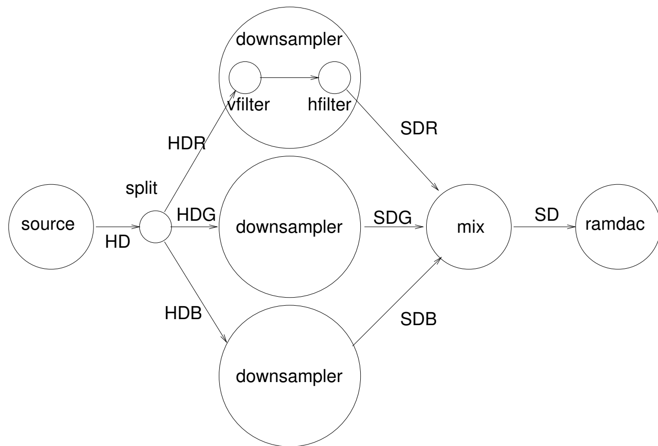
Modular Scheduling



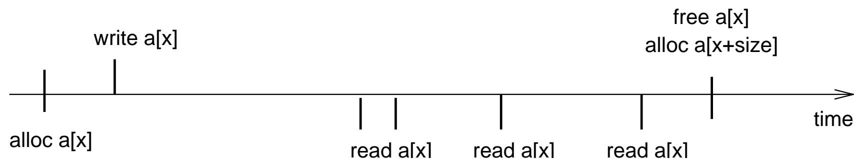


- One can eliminate the local schedule of each process independently.
- The result is a relation between the clocks of its input and output ports (the input/ouput constraints).
- One can then interconnect the channels (i.e. identify variables in the channel clocks) and solve the global scheduling problem.
- Once the global schedule is known, one can find the local schedules by backpropagation.

Structured Scheduling



Buffer Size



$$\begin{aligned} \theta(\text{write } a[x]) &\geq \theta(\text{alloc } a[x]), \\ \theta(\text{read } a[x]) &> \theta(\text{write } a[x]), \\ \theta(\text{free } a[x]) = \theta(\text{alloc } a[x + \text{size}]) &\geq \theta(\text{read } a[x]). \end{aligned}$$

Apply Farkas and solve.

Code Generation

Reconstructing a loop program from a schedule.

- Easy in theory. One just has to invert the schedule:

$$\begin{array}{lll} \mathrm{do} \;\; t & = & 0, L \\ \quad \mathrm{doall} & & \{u \mid \theta(u) = t\} \end{array}$$

- Difficult in practice. One must avoid complex control structures, which may offset the advantages of optimization or parallelization.
- There are good stand-alone implementations: [Pugh, Quilleré, Bastoul].
- Problem: identify parallel loops.

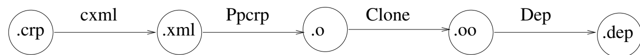
Another Code Generation Approach

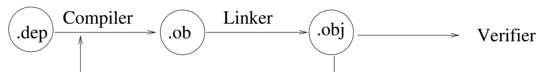
- Let E be the set of all operations in $2d+1$ notation,
- Let $u \prec v \equiv \theta(u) \ll \theta(v) \vee (\theta(u) = \theta(v) \wedge u \ll v)$, θ a schedule and $\ll$ the lexicographic order, $\prec$ is a total order,
- Define:

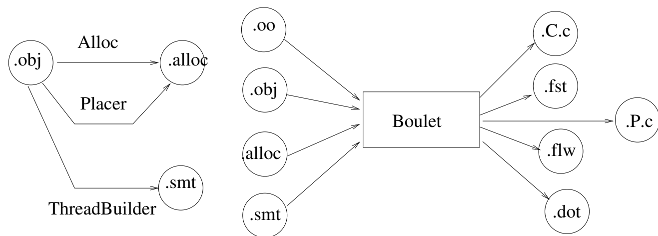
$$\begin{aligned} \mathrm{first}() &= \min_{\prec} E, \\ \mathrm{next}(u) &= \min_{\prec} \{v \in E \mid u \prec v\} \end{aligned}$$

- next and first are conditional expressions which translate directly into a Finite State Machine,
- Transitions such that $\theta(u) = \theta(v)$ denote parallelism.

Implementation Status / I







Implementation Status / II

- Code Generation for one thread per process
- Runtime

Future work.

- Several threads per process,
- Multi-dimensional communications,
- Black-box functions,
- Trivial pointers.

Conclusion

- A design can be divided in many processes, which can be reused elsewhere.
- Each process can be scheduled independently. The result is a set of constraints on its port clocks.
- A linker then solve the communication constraints, finalize the process constraints, and generate the object code.
- No recompilation is needed for unmodified processes.
- One can add constraints on the size of channel buffers.

If you want to know more

Paul Feautrier: Scalable and Structured Scheduling, Int. J. of Parallel Programming, 35, 5, pp. 459 – 487.

`http://perso.ens-lyon.fr/paul.feautrier/Syntol/notes`

or you can ask QUESTIONS