

An **Intermediate Language** for Productive and Efficient Parallel Programming: A Stream-Computing **Experiment**

Albert Cohen
Albert.Cohen@inria.fr

ALCHEMY Group



And **colleagues**, at NXP Research, INRIA and Paris-Sud 11 University

Dagstuhl Seminar 07361 – September 2007

Productivity **and** Performance?

	Source	Target
Productivity	functional and parallel composition, determinism	modularity (reuse without recompilation)
Scalability	expressive concurrency constructs	facilitate synchronization-light implementation
Efficiency	expressive resource management constructs	resource-bound low-level code optimized across module boundaries

Intermediate Language Approach

“Parallel Efficiency Layer”

Portability of Performance

- Target for multiple parallel programming models
 - ▶ Do not confuse the challenges of (concurrent or not) high-level programming (for expert or average programmers) with the need for parallelism on the target
 - ▶ I am working on my own preferred model, but yours may be fine as well (better if it exposes communications, though)
- **Explicit** functional **and** behavioral representation
 - ▶ Analyses, including your compilers' current data-flow ones
 - ▶ Transformations, including parallelism
 - ▶ Modularity, with adaptive/generative library composition

Related Work

- Message passing
 - Very expressive, leaves no single chance for a compiler
- Process algebra: CSP (Occam), CCS
 - Logical concurrency, rendez-vous only
 - Non-determinate, compiler-unfriendly
- Concurrent CFG + scalar data-flow: PDG, concurrent SSA
 - Scalar only, nasty memory models come into play
- Transactional memory (partial language specification only)
 - Probably better than data-flow + non-determinate select
 - Moderate scalability, at a high complexity/resource price
 - Not very compiler-friendly
- More restricted models: BSP (\approx CUDA)
 - Mostly for data-parallelism
- Even more restricted models: low-level (dwarf?) skeletons, Σ -SPL
 - Very effective for a specific application (sub-)domain
 - Note: MapReduce is not an intermediate language

Related Work in the Polyhedral Model

Multi-Dimensional Affine Scheduling

- θ : iteration vectors \rightarrow lexicographically ordered multidimensional dates
- Two instances (statement iterations) may execute in parallel if they share the same execution date

Multi-Dimensional Affine Scheduling

- Captures most compositions of loop-centric transformations on significant class of computationally intensive codes
- Main algorithms scale to 100s of array references in 5+ nested loops
- Implicit and **explicit** characterization of legal transformations

Expressiveness Limitation(s) of Affine Schedules

- Fine grain synchronous parallelism (or `#pragma ivdep`)

$\theta(S_1, i) = 0, \theta(S_2, i) = 1$

```
forall (i=0; i<n; i++)
| S1
forall (i=0; i<n; i++)
| S2
```

$\theta(S_1, i) = \theta(S_2, i) = 0$

```
forall (i=0; i<n; i++)
| S1 || S2
```

Not an affine schedule

```
forall (i=0; i<n; i++)
| S1
| S2
```

Related Work in the Polyhedral Model

Affine Partitioning

- π : iteration vectors \rightarrow multidimensional execution slices
- Two instances (statement iterations) may execute in parallel if they belong to distinct slices

Expressiveness Limitation(s) of Affine Schedules

- Implicit sequential ordering on slices
 - ▶ ... Or combination of partitioning and scheduling
 - ▶ ... Possibly nested: θ_1 on slices and θ_2 on instances

$$\begin{aligned}\pi(S_1, i) &= i, \\ \pi(S_2, i) &= i, \\ \theta_1(S_1, i) &= 0, \theta_1(S_2, i) = 1\end{aligned}$$

```
forall (i=0; i<n; i++)
| S1
forall (i=0; i<n; i++)
| S2
```

$$\begin{aligned}\pi(S_1, i) &= i, \\ \pi(S_2, i) &= n + i, \\ \theta_1(S_1, i) &= \theta_2(S_2, i) = 0\end{aligned}$$

```
forall (i=0; i<n; i++)
| S1 || S2
```

$$\begin{aligned}\pi(S_1, i) &= \pi(S_2, i) = i, \\ \theta_1(S_1, i) &= 0, \\ \theta_1(S_2, i) &= 1\end{aligned}$$

```
forall (i=0; i<n; i++)
| S1
| S2
```

Related Work in the Polyhedral Model

Caveats

- Lacks flexible expression of multi-level parallelism
- In practice, need to resort to additional synchronization mechanism: typically, generalized notify/wait
- Impractical to isolate synchronizations for a sub-system
 - ▶ One-to-one synchronization in a streaming pipeline
 - ▶ Non-affine ordering (reductions)
- Leads to load imbalance and synchronization overhead

Related Work: Kahn Networks (1974)

- Stream: infinite sequence of values
- Kahn process: monotonic (partial) function on infinite streams
- Process network: equations on such functions
- Informal: (sequential or parallel) processes communicating through FIFOs with blocking read and non-blocking writes
- **Determinate, functional and parallel composition**
- Hard to generalize without losing these properties
- Memory management and run-time overhead, compiler-unfriendly
- Derivative models of computation: data-flow, lazy streams, some Ptolemy II MoCs

Related Work: Synchronous Languages

- Designed for reactive, interactive and critical systems
- SW and HW generation, huge impact on industrial niche
- Static “by construction” guarantees
 - ▶ **Dead-lock free**
 - ▶ **Bounded resources** and reaction time
(zero or unit buffers, depend on the target)
 - ▶ Real-time and correctness properties amenable to automatic verification
 - ▶ Systematic test tools, static (model checking) and dynamic

Related Work: Synchronous Languages

- Synchronous composition in a nutshell
 - ▶ Partially ordered (infinite) set of clock ticks
 - ▶ Clock c : totally ordered subset of ticks
 - ▶ Function (on infinite streams): clock type $c \rightarrow c'$
 - ▶ Unobservable behavior between ticks
 - ▶ Compose functions of matching clock types
 - ▶ Functional rendez-vous, yet allow to decouple communication from synchronization
 - ▶ Multiple (logically) incomparable clocks
 - ▶ Causality enforcement: any inductive definition must pass a clock tick
- Synchrony is ensured by a type-system for **clocks**: a **clock calculus**
- Most general setting: **Synchronous Kahn Networks**

Related Work: Synchronous Languages

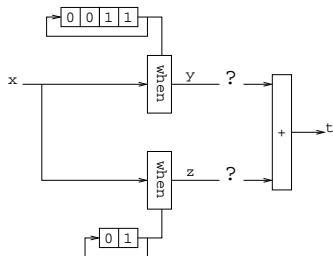
Applications to Real-Time Modeling

- External mapping to physical time: unobservable means “instantaneous”
- Composition need to satisfy constraints on this mapping
- Static or dynamic accounting of execution time

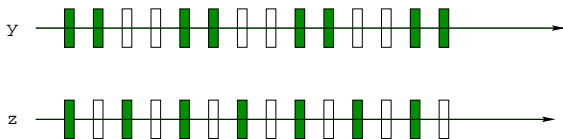
Caveats of Strictly Synchronous Composition

- Risk combinatorial control complexity
- Tends to generate nested `ifs` in a big time loop
- Efficient for circuit parallelism only
- Optimizations to coarsen the grain (nested loops), some success in control codes, fail on more predictable stream computations
- Primitive memory management, explicit copy into processes' buffers

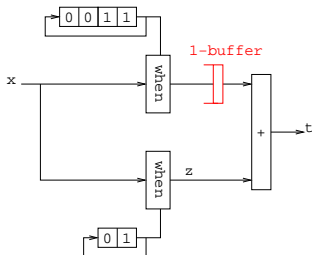
Related Work: Synchronous Languages



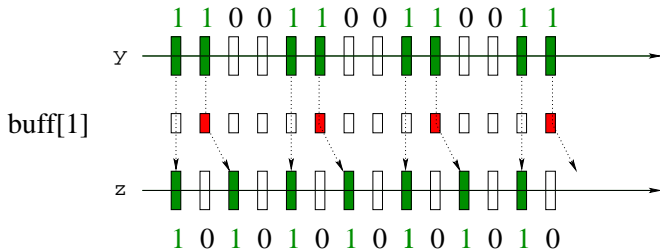
- Streams must be synchronous when composed
 $y+z$ is rejected by the clock calculus
- Relax this to reduce combinatorial complexity (of target code) and manage memory resources rather than registers



Recent Work: n-Synchronous Languages (PoPL'06)



- Relaxed “synchrony up to n ticks”, i.e., up to insertion of a bounded buffer
- Denotational and operational (clocked) semantics, n -synchronous clock calculus, complete type inference for ultimately periodic clocks



Clock Sub-Sampling

Primitive Control-Flow Construct

c_1 on c_2 denotes a **sub-sampled clock**

c_1 on c_2 is the clock obtained in **driving c_2 at the pace of c_1**

base	1	1	1	1	1	1	1	1	1	1	...	(1)
c_1	1	1	0	1	0	1	0	1	0	1	...	1(10)
base on c_1	1	1	0	1	0	1	0	1	0	1	...	1(10)
c_2	0	1		0		1		0		1	...	(01)
(base on c_1) on c_2	0	1	0	0	0	1	0	0	0	1	...	(0100)

Current Work

Goal

- Generate tight nested loops by construction

Key Idea

- Generate coarse grain loop \iff unobservable sequence of computations
- Clock ticks should not only be present or absent
 - \rightarrow They are non-negative integers, counting a **burst of unobservable events**
- Generalize sub-sampling and super-sampling in a single construct

Clock Sub- and Super-Sampling

Primitive Control-Flow Construct

c_1 on c_2 is the clock obtained in driving c_2 at the pace of c_1 , summing burst counts of c_2 according to bursts of c_1

Supports both event-driven reactive control and nested loop control

base	1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...	(1)
c_1	2 2 0 1 0 2 0 1 0 2 0 1 0 2 ...	2(2010)
base on c_1	2 2 0 1 0 2 0 1 0 2 ...	2(2010)
c_2	0 3 0 3 0 3 0 3 0 3 ...	(03)
(base on c_1) on c_2	3 3 0 0 0 3 0 3 0 3 ...	3(30003030)

Low-Level Concurrency Constructs

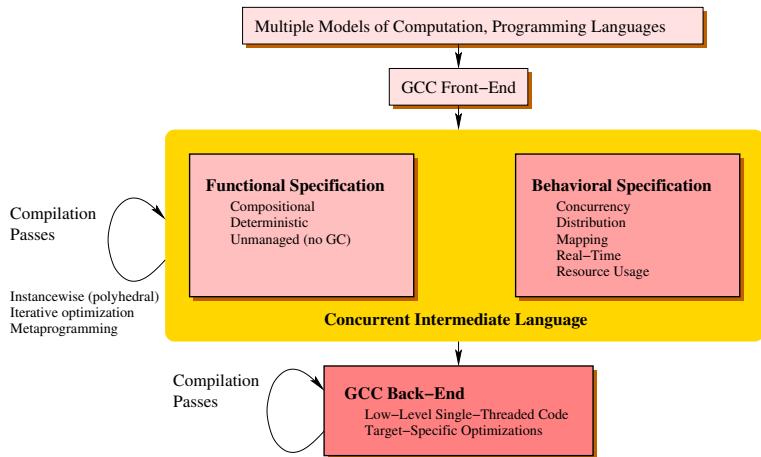
```
timestamp producer;
timestamp consumer;
int main() {
    ...
    clock (producer) {
        while (...) {
            wait(3*consumer-2*producer<=42); // back-pressure
            ...
            advance(3);
        }
    }
    clock (consumer) {
        while (...) {
            timestamp p = producer;
            wait(producer - p == 2);
            ...
            advance(1);
        }
    }
    ...
}
```

Low-Level Concurrency Constructs

```
timestamp producer;
timestamp consumer;
int stream data[42];
boolean stream eos;
int main() {
    ...
    clock (producer) {
        while (...) {
            wait(consumer >= producer - 42);
            for (i=0; i<10; i++)
                data@i = ...
            advance(10);
        }
        eos = TRUE;
    }
    clock (consumer) {
        while (...) {
            wait(producer == consumer);
            if (eos) break;
            ... = data@0 + data@-1 + data@-2;
            advance(2);
        }
    }
}
```

Practical Implementation

Maximize Impact and Mutualization: GCC Platform



- Current status: not started yet...
- ML dialect prototypes (e.g., Marc Pouzet's Lucid Synchronic)

Ongoing and Future Work

- Beyond stream-prefix causality, e.g., model nested doacross loops (notify/wait) to facilitate expression and load-balancing of parallelism not parallel to the axes
- Integrate real-time modeling to clock calculus
- Time-dependent semantics with dead-lock free and (weaker) determinism guarantees?
 - ▶ Execution triggered on (physical) timestamp conditions
 - ▶ Synchronous form of non-deterministic (time-dependent) select
- Coordination with polyhedral compilation techniques
- Implementation targeting the Cell processor, NXP NeXVP, STMicroelectronics xStream
- Design-space exploration of architecture support for clocked-streaming parallelism