# Leveraging Streaming for Deterministic Parallelization

## an Integrated Language, Compiler and Runtime Approach

*Antoniu Pop*

Centre de recherche en informatique, MINES ParisTech

*PhD Defence*

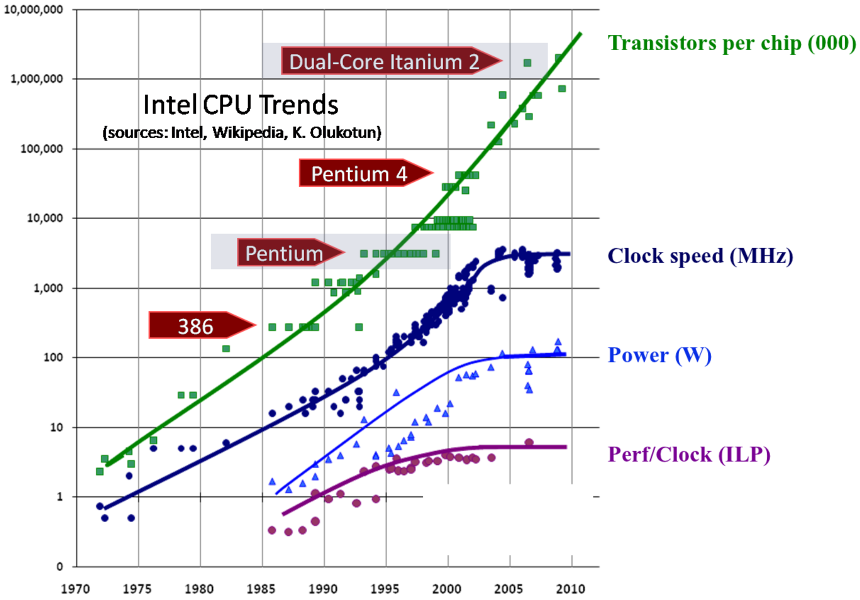
30 September 2011, MINES ParisTech, Paris, France

| | |
|---|---|
| **Philippe CLAUSS**, Université de Strasbourg | Rapporteur |
| **Albert COHEN**, INRIA | Examinateur |
| **François IRIGOIN**, MINES ParisTech | Directeur de thèse |
| **Paul H J KELLY**, Imperial College London | Rapporteur |
| **Fabrice RASTELLO**, INRIA | Examinateur |
| **Pascal RAYMOND**, CNRS | Examinateur |
| **Eugene RESSLER**, United States Military Academy | Examinateur |

*"Power Wall + Memory Wall + ILP Wall = Brick Wall"*

*"Increasing parallelism is the primary method of improving processor performance."*

David A. Patterson (2006)



Herb Sutter, *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software* (2009)

# Introduction

**No surprise the memory wall issue is getting worse**

**Possible solution: stream-computing**

- Memory latency: decoupling
- Off-chip bandwidth: local, on-chip communication
- False sharing and spatial locality: aggregation of communications

# Stream programming models and languages

## Kahn Process Networks (1974)

- Data-driven deterministic processes
- Unbounded single-producer single-consumer FIFO channels
- Cyclic communication can lead to deadlocks
- UNIX pipes

## Synchronous Data-Flow (1987)

- Statically defined, periodic behaviour
- Production/consumption rates known at compile time
- Ptolemy (1985-96), StreamIt language (2001)

## Synchronous languages

- Reactive systems and signal processing networks
- Deterministic and deadlock-free
- Sampled signals instead of streams
- Signal (1986), LUSTRE (1987), Lucid Synchrone (1996), Faust (2002)

# Can streaming help to efficiently exploit non-streaming applications?

**Existing streaming models**

- Regular streams of data
- Single-producer single-consumer FIFO queues
- Restricted to specific classes of applications

**General-purpose parallel programming**

- Irregular communication patterns
- Control flow cannot be ignored
- Multi-producer multi-consumer FIFO queues
- Express control-dependent irregular data flow
- Efficiency is an issue

# Is a new stream programming language necessary? Desirable?

## New stream programming language

- Adopting yet another new language
- New compilation and debugging tool-chains
- Mixing different programming styles and parallel constructs

## Providing stream-computing semantics to a well-established language

- Incremental adoption
- Integration with existing parallel constructs: data-parallel loops, tasks

## Pragmatic choice: OpenMP 3.0

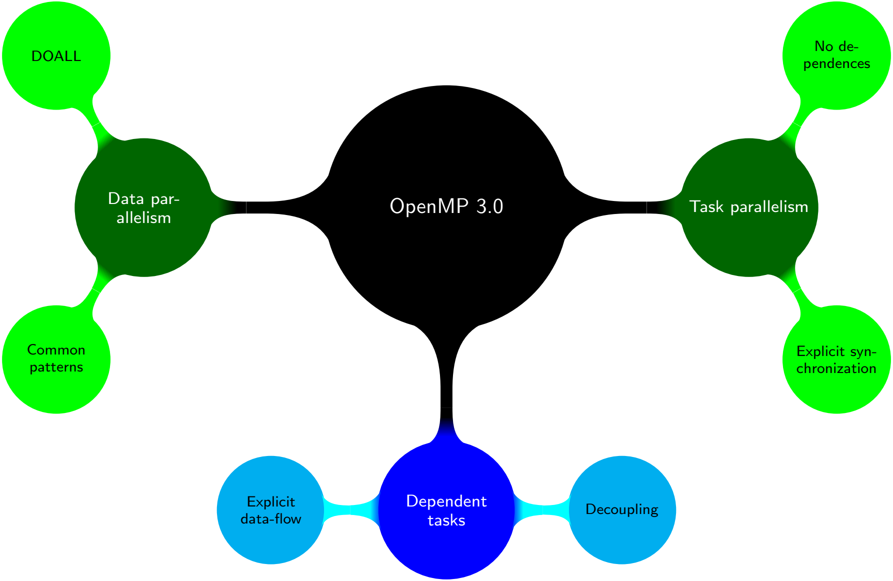
- De facto standard for shared memory parallel programming
- Widely available and used
- *Any language that provides support for task parallelism*

# Presentation and Thesis Outline

1. **Generalized, Dynamic Stream Programming Model for OpenMP**
   - **Ch 2.** A Stream-Computing Extension to OpenMP
   - **Ch 8.** Experimental Evaluation
2. **Compilation and Execution of Generalized Streaming Programs**
   - **Ch 6.** Runtime Support for Streamization
   - **Ch 7.** Work-Streaming Compilation
3. **Contributions and Perspectives**
   - **Ch 3.** Control-Driven Data-Flow (CDDF) Model of Computation
   - **Ch 4.** Generalization of the CDDF Model
   - **Ch 5.** CDDF Semantics of Dependent Tasks in OpenMP

# 1. Generalized, Dynamic Stream Programming Model for OpenMP

# Bird’s Eye View of OpenMP


OpenMP 3.0

Data parallelism

DOALL

Common patterns

Task parallelism

No dependences

Explicit synchronization

Dependent tasks

Explicit data-flow

Decoupling

# OpenMP through examples I

## Data-parallel loops

```
#pragma omp parallel for shared (A)
for(i = 0; i < N; ++i)
  A[i] = ...;
```

```
#pragma omp parallel for shared (B)
for(i = 1; i < N; ++i)
  B[i] = ... B[i-1] ...;
```

- No verification of validity of annotations

# OpenMP through examples II

## OpenMP 3.0 tasks

```
p = ...;

while (p != NULL) {
  #pragma omp task firstprivate (p)
  {
    do_work (p->data);
  }
  p = p->next;
}
```

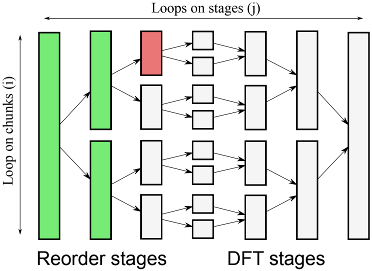
- No order can be assumed on the execution of tasks
- Dependences must be synchronized by hand

# Motivation for Streaming

## Sequential FFT implementation

```
float A[2 * N];
for(i = 0; i < 2 * N; ++i)
  A[i] = ...;

// Reorder
for(j = 0; j < log(N)-1; ++j)
{
  chunks = $2^{j}$;
  size = $2^{(\log(N)-j+1)}$;

  for (i = 0; i < chunks; ++i)
    reorder (A[i*size .. (i+1)*size-1]);
}
```

```
// DFT
for(j = 1; j <= log(N); ++j) {
  chunks = $2^{(\log(N)-j)}$;
  size = $2^{(j+1)}$;

  for (i = 0; i < chunks; ++i)
    compute_DFT (A[i*size .. (i+1)*size-1]);
}

// Output the results
for(i = 0; i < 2 * N; ++i)
  printf ("%f\t", A[i]);
```

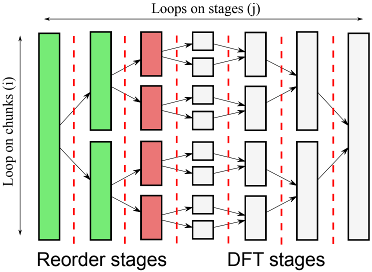


# Example: FFT Data Parallelization

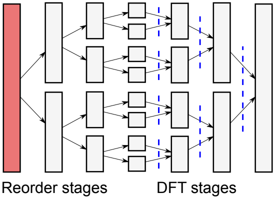
## OpenMP `parallel loop` implementation

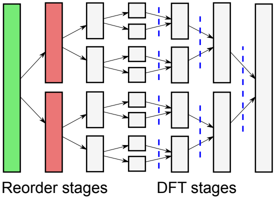
```
float A[2 * N];
for(i = 0; i < 2 * N; ++i)
  A[i] = ...;

// Reorder
for(j = 0; j < log(N)-1; ++j)
{
  chunks = $2^{j}$;
  size = $2^{(\log(N)-j+1)}$;

#pragma omp parallel for
  for (i = 0; i < chunks; ++i)
    reorder (A[i*size .. (i+1)*size-1]);
}
```

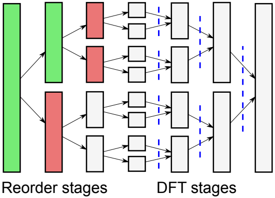
```
// DFT
for(j = 1; j <= log(N); ++j) {
  chunks = $2^{(\log(N)-j)}$;
  size = $2^{(j+1)}$;

#pragma omp parallel for
  for (i = 0; i < chunks; ++i)
    compute_DFT (A[i*size .. (i+1)*size-1]);
}

// Output the results
for(i = 0; i < 2 * N; ++i)
  printf ("%f\t", A[i]);
```

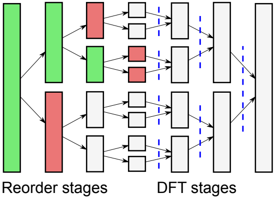


# Example: FFT Task Parallelization

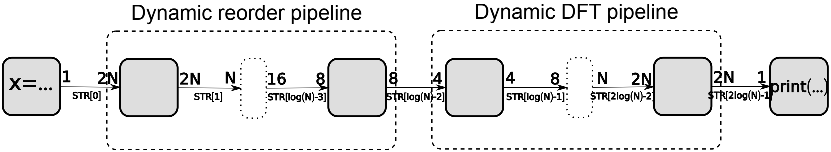


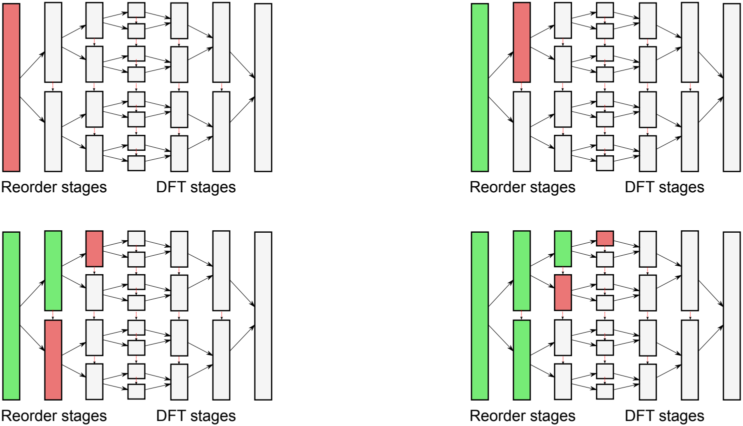




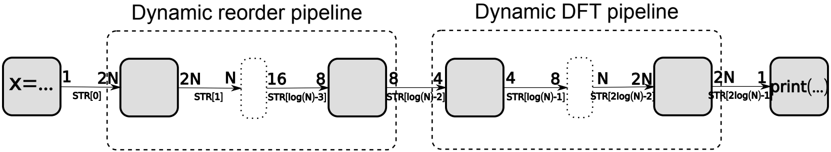


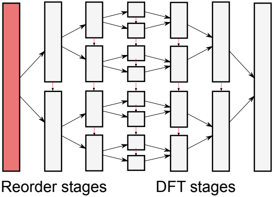
# Example: FFT Pipeline Parallelization

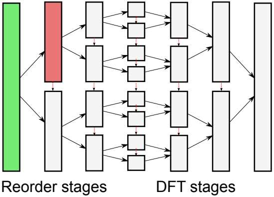


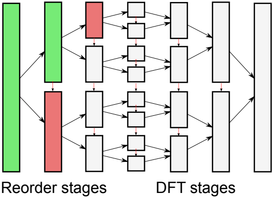


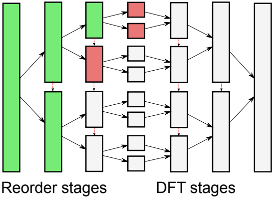
# Example: FFT Streamization (pipeline and data-parallelism)



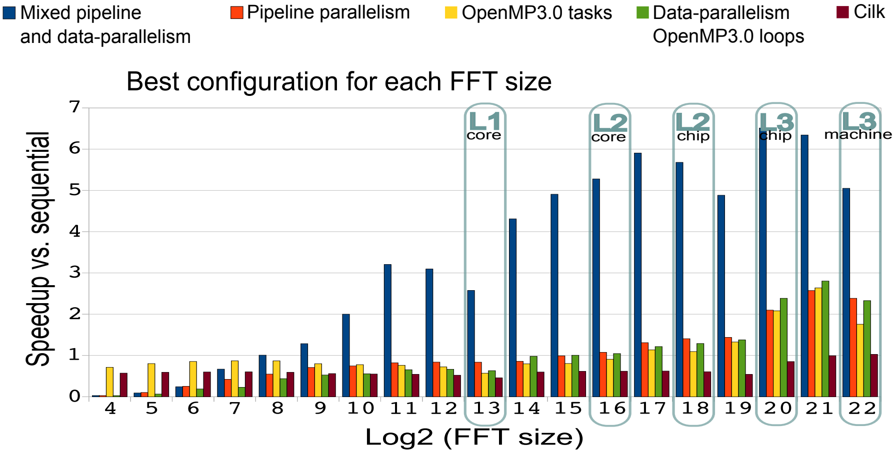








# Single FFT Performance



4-socket Opteron – 16 cores

# Performance evaluation of streaming applications

## FMradio

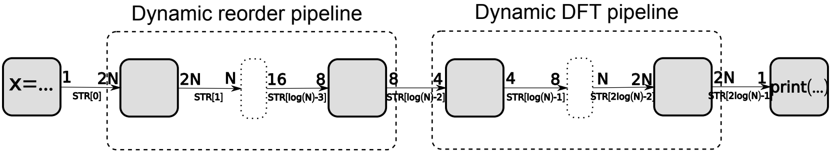
- high amount of data-parallelism, fairly well-balanced
- little effort to annotate with our streaming extension
- $\mathbf{12.6}\times$ speedup on 16-core Opteron ($\mathbf{10.5}\times$ automatic code generation – $\mathbf{20\%}$)
- *StreamIt:* $\mathbf{8.6}\times$ *speedup on 16-core Raw architecture (different implementations)*

## IEEE802.11a

- complicated to parallelize, more unbalanced
- complex code refactoring is necessary to expose data parallelism
- annotating the program is straightforward to exploit pipeline parallelism
- annotating while enabling data-parallelism is difficult
- $\mathbf{13}\times$ speedup on 16-core Opteron ($\mathbf{6}\times$ automatic code generation – $\mathbf{55\%}$)

# Design of the Streaming Extension: FFT Case Study

## What needs to be expressed?



- Producer-consumer relations (flow dependences)
- Variable amount of data produced/consumed
- Dynamic pipeline

## How can it be expressed?

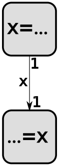
- Coding patterns
- Syntax

# Coding Patterns

## Producer-consumer relation

- Add `input` and `output` clauses to OpenMP tasks

```
int x;

for (i = 0; i < N; ++i)
{
#pragma omp task output (x)
  x = ... ;

#pragma omp task input (x)
  ... = ... x ...;
}
```



## Decoupling through privatization

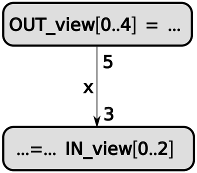
- Eliminate anti/output dependences
  - equivalent to scalar expansion on `x`
- Streams naturally map on communication channels

# Coding Patterns

## Variable amount of data produced/consumed

- Enable tasks to consume or produce multiple values at a time: “burst” rates
- Rename the stream variable within the task: “view”
- Use the C++-flavoured << and >> stream operators to connect a view to a stream

```
int x, IN_view[5], OUT_view[5];

for (i = 0; i < N; ++i)
{
#pragma omp task output (x << OUT_view[5])
  for (int j = 0; j < 5; ++j)
    OUT_view[j] = ... ;

#pragma omp task input (x >> IN_view[3])
  for (int j = 0; j < 5; ++j)
    ... = ... IN_view[j] ...;
}
```



OUT_view[0..4] = ...

5

x

3

...=... IN_view[0..2]

## Monotonic stream accesses

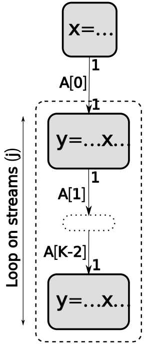
- Memory accesses are serialized in the stream
  - Contiguous memory accesses by design
  - Cache locality with memory re-organisation (explicit in the task body)
- Deterministic concurrency semantics
- No periodicity requirement

# Coding Patterns

## Dynamic pipeline of filters

- Arrays of streams
- Dynamic connection of streams/tasks

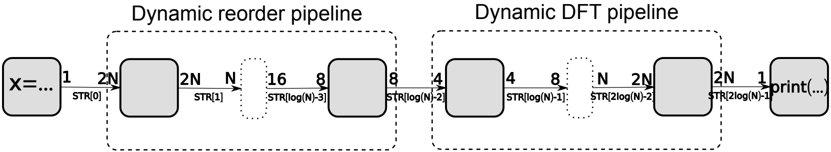
```
int x, y, A[K];

for (i = 0; i < N; ++i)
{
#pragma omp task output (A[0] ≪ x)
  x = ... ;
}

for (j = 0; j < K-1; ++j) // Task graph construction loop
{
  for (i = 0; i < N; ++i)
  {
#pragma omp task input (A[j] ≫ x) output (A[j+1] ≪ y)
    y = ... x ...;
  }
}
```



## Explicit dynamic construction of dynamic task graphs

- Dynamic dependences define the producer-consumer relations
- **Not limited to streaming applications: arbitrary dependences and control**
  - Flexible and expressive, but can we preserve the streaming properties

# Streamized FFT Implementation with the OpenMP Extension



```
float x, STR[2*(int)(log(N))];

for(i = 0; i < 2 * N; ++i)
#pragma omp task output (STR[0] << x)
  x = ...;

// Reorder
for(j = 0; j < log(N)-1; ++j) {
  int chunks = 2^j;
  int size = 2^(log(N)-j+1);
  float X[size], Y[size];

  for (i = 0; i < chunks; ++i)
#pragma omp task input (STR[j] >> X[size]) \
                 output (STR[j+1] << Y[size])
    {
      Y[0..size-1] = reorder (X[0..size-1]);
    }
}
```

```
// DFT
for(j = 1; j <= log(N); ++j) {
  int chunks = 2^(log(N)-j);
  int size = 2^(j+1);
  float X[size], Y[size];

  for (i = 0; i < chunks; ++i)
#pragma omp task input (STR[j+log(N)-2] >> X[size]) \
                 output (STR[j+log(N)-1] << Y[size])
    {
      Y[0..size-1] = compute_DFT (X[0..size-1]);
    }
}

for(i = 0; i < 2 * N; ++i)
#pragma omp task input(STR[2*log(N)-1] >> x)\
                 input (stdout) output (stdout)
  printf ("%f\t", x);
```

# 2. Compilation and Execution of Generalized Streaming Programs

1. Generalized, Dynamic Stream Programming Model for OpenMP
2. **Compilation and Execution of Generalized Streaming Programs**
3. Contributions and Perspectives

# Execution of Generalized Streaming Programs

### Pure streaming applications

- Synchronous Data-Flow invariants
- Periodic behaviour
- Statically optimized static schedule

### Generalized streaming applications

- Dynamic behaviour (unknown at compile time)
- Run-time scheduling

# Work-Streaming Code Generation: naive expansion

## Example: streaming task

```
float x, y;
for (i = 0; i < N; ++i) {
  // Do non-streaming work
  if (condition ()) {
#pragma omp task input(x) output(y)
    y = f (x);
  }
}
```

↓ Work-streaming compilation and runtime ↓

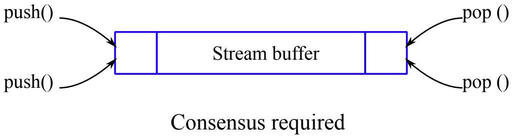
```
GOMP_stream_id id_x, id_y;

for (i = 0; i < N; ++i)
{
  // Do non-streaming work

  if (condition ()) {
    GOMP_activate_stream_task
      (stream_task_wf, id_x, id_y);
  }
}
```

```
void stream_task_wf (&params) {
  GOMP_stream s_x = params->x, s_y = params->y;
  float *view_x, *view_y;
  int current;

  while (get_activation (&current)) {
    view_y = stall (s_y, current); // blocking
    view_x = update (s_x, current); // blocking

    *view_y = f (*view_x);

    commit (s_y, current); // non-blocking
    release (s_x, current); // non-blocking
  }
}
```

# Synchronization constraints

## Multi-producer multi-consumer streams

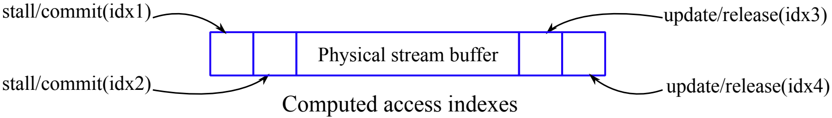
- FIFO queues: non-deterministic interleaving
- Requires atomic operations



Consensus required

## Compute access indexes based on control flow

- Synchronize only producers with consumers
- No need to reach a consensus between producers or consumers



Computed access indexes

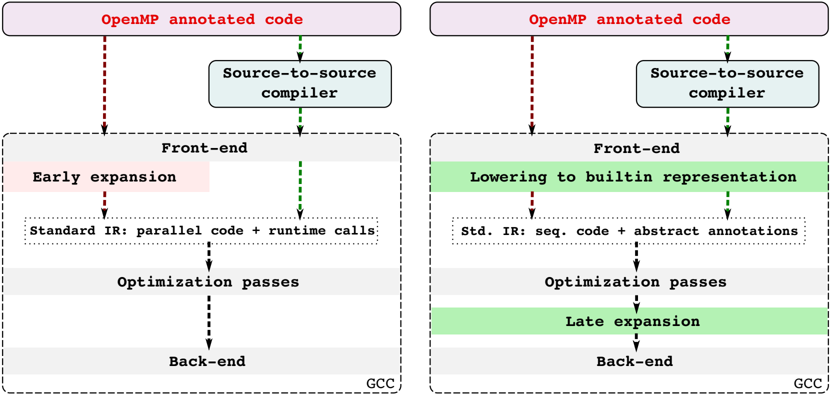
# Work-Streaming Code Generation: optimized case

```
GOMP_stream_id id_x, id_y;
for (i = 0; i < N; ++i) {
  // Do non-streaming work
  if (condition ()) {
    GOMP_activate_stream_task
      (stream_task_wf, id_x, id_y);
  }
}
```

```
void stream_task_wf (&params) {
  GOMP_stream s_x = params->x, s_y = params->y;
  float *view_x, *view_y;
  int beg, end, beg_s, end_s;

  while (get_activation_range (&beg, &end)) {
    for (beg_s=beg; beg_s<=end; beg_s += AGGREGATE) {
      end_s = MIN (beg_s + AGGREGATE, end);
      view_y = stall (s_y, end_s); // blocking
      view_x = update (s_x, end_s); // blocking

      // Automatic vectorized version
      for (i=0; i<end_s-beg_s; i+=4)
        view_y[i..i+3] = f_v4f_clone (view_x[i..i+3]);

      // Fall-back version
      for (MAX (0, i-4); i<end_s-beg_s; i++)
        view_y[i] = f (view_x[i]);

      commit (s_y, end_s); // non-blocking
      release (s_x, end_s); // non-blocking
    }
  }
}
```

- Views directly access stream buffers: no unwarranted memory copy
- Optimization example: automatic vectorization

# On-going work: OpenMP late expansion



OpenMP annotated code

Source-to-source compiler

Front-end

Early expansion

Standard IR: parallel code + runtime calls

Optimization passes

Back-end

GCC

OpenMP annotated code

Source-to-source compiler

Front-end

Lowering to builtin representation

Std. IR: seq. code + abstract annotations

Optimization passes

Late expansion

Back-end

GCC

# 3. Contributions and Perspectives

1. Generalized, Dynamic Stream Programming Model for OpenMP
2. Compilation and Execution of Generalized Streaming Programs
3. **Contributions and Perspectives**

# Contributions of this thesis I

1. Integration of the streaming paradigm in a high-level, general-purpose parallel programming language, OpenMP
    - no need for a domain specific language (e.g., StreamIt)
    - no access barrier for application programmers
    - no loss of expressiveness, preserving the existing parallel and sequential constructs
    - no loss of efficiency
2. Extension of the streaming paradigm with irregular accesses to streams and dynamically defined task graphs
    - dynamically allocated streams and arrays of streams
    - dynamic subscripting of arrays of streams for dynamically connecting tasks with streams
    - dynamically created tasks
3. Minimal syntactic extension and maximal semantic compatibility with OpenMP, offering functional determinism and all the expressiveness of dependent tasks with streaming computations

# Contributions of this thesis II

4. Control-Driven Data-Flow: formal model of computation
    - proofs of statically analyzable conditions for dead-lock freedom and compile-time serializability
    - proof of functional and deadlock determinism
    - generalization to execution in bounded memory and extension of proofs
5. Algorithmic support for performance and debugging
    - Stream synchronization algorithm proved to require no atomic operations and no memory fences
    - Runtime deadlock detection algorithm with support for bounded memory execution
6. Code generation and runtime implemented as a prototype in GCC
7. Experimental evaluation
    - streaming applications can be efficiently exploited
    - non-streaming applications can be (concisely) expressed and efficiently exploited
    - evidence of the usefullness of the extension to generalize the streaming paradigm

# Perspectives and Open Questions

- Dataflow analysis of streaming applications
  - Can stream access patterns be captured by dataflow analysis techniques?
  - Is it possible to statically enable task-level optimizations on generalized streaming programs?
- Desynchronization of the LUSTRE synchronous language
- Generation of code for distributed memory systems
- Extending other parallel programming models with streaming

# Leveraging Streaming for Deterministic Parallelization

## an Integrated Language, Compiler and Runtime Approach

*Antoniu Pop*

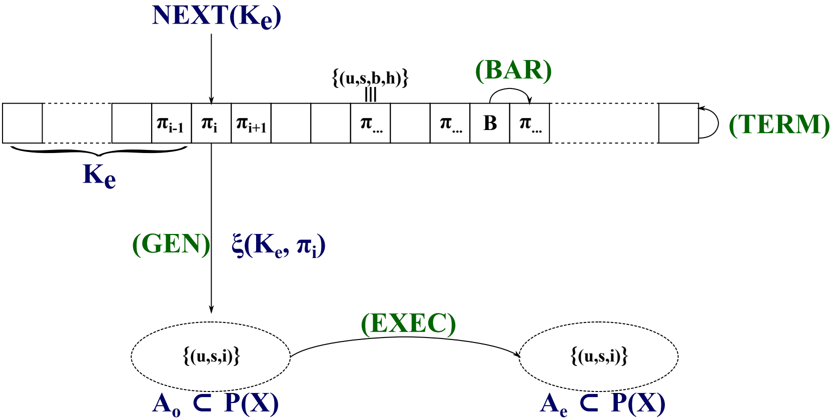
30 September 2011, MINES Paristech, Paris, France

**Contributions:**

1. Integration of the streaming paradigm in a high-level, general-purpose parallel programming language, OpenMP
2. Extension of the streaming paradigm with irregular accesses to streams and dynamically defined task graphs
3. Minimal syntactic extension and maximal semantic compatibility with OpenMP, offering functional determinism and all the expressiveness of dependent tasks with streaming computations
4. Control-Driven Data-Flow: formal model of computation
5. Algorithmic support for performance and debugging
6. Code generation and runtime implemented as a prototype in GCC
7. Experimental evaluation

# Control-Driven Data-Flow Execution Model

$$\sigma = (K_e , A_e , A_o) \xrightarrow{\text{(GEN)} \lor \text{(EXEC)} \lor \text{(BAR)}} \sigma'$$



# Properties of CDDF Programs

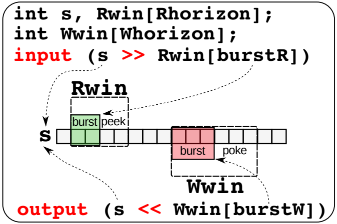
| Condition on state | Deadlock Freedom properties | | | | Serializability | |
|---|---|---|---|---|---|---|
| $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ | $\neg D(\sigma)$ | $\neg ID(\sigma)$ | $\neg FD(\sigma)$ | $\neg SD(\sigma)$ | Dyn. order | CP |
| $TC(\sigma) \wedge \forall s \in SCC(H(\sigma)), \neg MPMC(s)$ | no | no | yes | yes | if $\neg ID(\sigma)$ | no |
| $TC(\sigma) \wedge \forall s, \neg MPMC(s)$ | no | no | yes | yes | if $\neg ID(\sigma)$ | no |
| $SCC(H(\sigma)) = \varnothing$ | no | no | yes | yes | if $\neg ID(\sigma)$ | no |
| $SC(\sigma) \vee NEXT(\mathcal{K}_e) \in \Pi$ | yes | yes | yes | yes | yes | no |
| $\forall \sigma, SC(\sigma)$ | yes | yes | yes | yes | yes | yes |

# Properties of Generalized CDDF Programs

| Condition on state<br>$\boldsymbol{\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)}$ | $\neg D(\sigma)$ | $\neg ID(\sigma)$ | $\neg FD(\sigma)$ | $\neg SD(\sigma)$ | $\neg LD(\sigma)$ | $\neg LSD(\sigma)$ |
|---|---|---|---|---|---|---|
| $TC(\sigma)\ \wedge\ \forall s \in SCC(H(\sigma)), \neg MPMC(s)$ | no | no | yes | yes | no | no |
| $\forall a \in \mathcal{A}_o, LP([a]_\sim)$ `not` $= \varnothing$,<br>$\forall s \in \mathcal{I}^+(a) \cup SCC(H(\sigma))\ \neg MPMC(s)$<br>$TC(\sigma)$ | no | no | yes | yes | no | yes |
| $TC(\sigma)\ \wedge\ \forall s, \neg MPMC(s)$ | no | no | yes | yes | no | yes |
| $SCC(H(\sigma)) = \varnothing$ | no | no | yes | yes | no | no |
| $SC(\sigma)\ \vee\ NEXT(\mathcal{K}_e) \in \Pi$ | yes | yes | yes | yes | no | no |
| $SC(\sigma)\ \vee\ NEXT(\mathcal{K}_e) \in \Pi$<br>$\vee\ \forall a \in \mathcal{A}_o, LP([a]_\sim) = \varnothing$ | yes | yes | yes | yes | yes | yes |
| $\forall \sigma, SC(\sigma)$ | yes | yes | yes | yes | yes | yes |

# OpenMP Extension for Stream-Computing: Syntax

```
input/output (list)
    list   ::= list, item
             | item
    item   ::= stream
             | stream >> window
             | stream << window
    stream ::= var
             | array[expr]
    expr   ::= var
             | value
```



```
int s, Rwin[Rhorizon];
int Wwin[Whorizon];
input (s >> Rwin[burstR])
output (s << Wwin[burstW])
```

```
int S[K];          // Array of streams
int X[horizon];  // View

#pragma omp task output (S[0] << X[burst])
  // task code block
  // burst <= horizon
  for (int i = 0; i < burst; ++i)
    X[i] = ...;

#pragma omp task input (S[0] >> X[burst])
  // task code block
  // burst <= horizon
  for (int i = 0; i < horizon; ++i)
    ... = ... X[i];
```

```
int A[5];    // Stream of arrays

#pragma omp task output (A)
  // task code block
  // Each element is an array
  for (int i = 0; i < 5; ++i)
    A[i] = ...

#pragma omp task input (A)
  // task code block
  for (int i = 0; i < 5; ++i)
    ... = ... A[i];
```

In general, annotations alter the semantics of the underlying sequential code

# Stream Causality I

### Serialization by ignoring annotations

- Each state of the program is stream causal

```
int x;

for (i = 0; i < N; ++i)
{
#pragma omp task output (x)
  x = ... ;

#pragma omp task input (x)
  ... = ... x ...;
}
```

# Stream Causality II

## Underlying program has different semantics than streaming program

- Only some states are stream causal

```
int x;

for (i = 0; i < N; ++i)
{
#pragma omp task input (x)
  ... = ... x ...;

#pragma omp task output (x)
  x = ... ;
}
```

```
int x;

for (i = 0; i < N; ++i)
{
#pragma omp task output (x)
  x = ... ;
}

for (i = 0; i < N; ++i)
{
#pragma omp task input (x)
  ... = ... x ...;
}
```

# Selected Publications

F. Li, A. Pop, and A. Cohen.
Advances in parallel-stage decoupled software pipelining.
In F. Bouchez, S. Hack, and E. Visser, editors, Proceedings of the Workshop on Intermediate Representations, pages 29–36, 2011.

A. Pop and A. Cohen.
Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers.
In Proceedings of the 15th Workshop on Compilers for Parallel Computers, CPC '10, Vienna, Austria, 07 2010.

A. Pop and A. Cohen.
A stream-computing extension to openmp.
In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC '11, pages 5–14, New York, NY, USA, 2011. ACM.