

Streaming Data Flow: A Story About Performance, Programmability, and Correctness

Albert Cohen

with Léonard Gérard, Adrien Guatto, Nhat Minh Lê, Feng Li,
Cupertino Miranda, **Antoni Pop**, Marc Pouzet

PARKAS Team
INRIA and ENS Paris

Lyon, July 1st, 2013

Funded by: TERAFLUX & PHARAON FP7, and ManycoreLabs “investissements d’avenir” grants



1. Stream Processing?

Stream Processing?

Driving Force: Correct Concurrency by Construction

The Hammer: Language Design

The Anvil: Runtime System Design

Wrap-Up

Stream Processing?

“A model that uses sequences of data and computation kernels to expose and exploit concurrency and locality for efficiency [of execution and programmability].”

Stream Processing?

“A model that uses sequences of data and computation kernels to expose and exploit concurrency and locality for efficiency [of execution and programmability].”

Two Workshops on Streaming Systems

WSS03: <http://groups.csail.mit.edu/cag/wss03>

WSS08: [http://people.csail.mit.edu/rabbah/
conferences/08/micro/wss](http://people.csail.mit.edu/rabbah/conferences/08/micro/wss)

Stream Processing – In This Talk

Application domains

1. High-productivity computing systems
2. Efficient runtimes, task-level pipelines
3. Embedded control, safety-critical, certified systems

Stream Processing – In This Talk

Application domains

1. High-productivity computing systems
2. Efficient runtimes, task-level pipelines
3. Embedded control, safety-critical, certified systems

Some influential languages

- ▶ Block-diagram: STATECHARTS, SCADE, SIMULINK, LabVIEW
- ▶ Synchronous: LUSTRE, ESTEREL, SIGNAL
- ▶ Data-flow: Lucid, Linda, SISAL, pH, SAC, CnC
- ▶ Kahn network APIs: YAPI, CAL
- ▶ Cyclo-static data flow: StreamIt, SigmaC

Stream Processing – Missing Something?

Stream Processing – Missing Something?

Bill Dally's streaming processors

- ▶ Bulk-Synchronous Parallelism? Vector processing pipelines?
- ▶ Brook, Cg? CUDA, OpenCL?

Stream Processing – Missing Something?

Bill Dally's streaming processors

- ▶ Bulk-Synchronous Parallelism? Vector processing pipelines?
- ▶ Brook, Cg? CUDA, OpenCL?

3-phase, “hardware-centric” decoupling: *load*→*compute*→*store*

Stream Processing – Missing Something?

Bill Dally's streaming processors

- ▶ Bulk-Synchronous Parallelism? Vector processing pipelines?
- ▶ Brook, Cg? CUDA, OpenCL?

3-phase, “hardware-centric” decoupling: *load*→*compute*→*store*

- ▶ Special case of Kahn networks
- ▶ Implicitly relies on chaining/fusion to save on memory transfers

Foundations: Kahn Process Networks



Kahn networks, 1974

Gilles Kahn (1946–2006)

Denotational: least fixpoint of a *system of equations* over *continuous* functions, for the Scott topology lifted to unbounded *streams*

$$s \sqsubseteq s' \implies f(s) \sqsubseteq f(s') \quad + \text{ lifted to the limit}$$

Operational: communicating processes over *unbounded FIFOs* with *blocking reads*

- *Deterministic* by design
- General recursion (dynamic process creation), *parallel composition*, *reactive systems*
- Distribute and decouple computations from communications

2. Driving Force: Correct Concurrency by Construction

Stream Processing?

Driving Force: Correct Concurrency by Construction

The Hammer: Language Design

The Anvil: Runtime System Design

Wrap-Up

Challenges

HPC Between the hammer of *programmability* and the anvil of *performance*



Embedded Program, test, verify, simulate, compile a single source code, serving as

- ▶ an abstract model for *static analysis*
- ▶ a concrete model for *simulation*
- ▶ the actual implementation from which *sequential/parallel code* can be generated
- ▶ Ensure strong properties of safety/efficiency at compile-time

Both Rely on efficient, proven runtime execution primitives

- ▶ lightweight scheduling
- ▶ *ring buffers*

Questions

- ▶ **What are the semantic requirements for source programs?**
- ▶ **Should programmers care**
 - About parallelism?
 - About the memory and power walls?**Which programmers?**
- ▶ **What role for the software stack?**
 - Compilers
 - Runtime systems
 - Libraries, library generators
 - Auto-tuning, dynamic optimization
 - Operating system, virtual machine monitor

3. The Hammer: Language Design

Stream Processing?

Driving Force: Correct Concurrency by Construction

The Hammer: Language Design

- OpenStream: dealing with the Von Neumann bottlenecks

- Heptagon: safety-critical embedded systems

The Anvil: Runtime System Design

Wrap-Up

3. The Hammer: Language Design

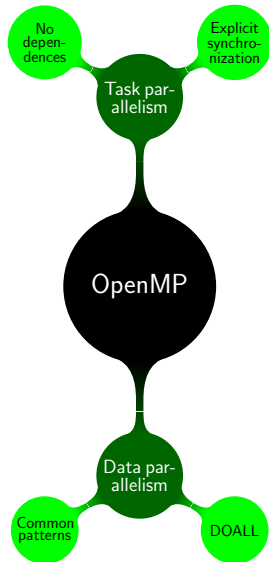
OpenStream: dealing with the Von Neumann bottlenecks

Heptagon: safety-critical embedded systems

OpenStream

OpenMP extension

- ▶ leverage existing toolchains and knowledge
- ▶ maximize *productivity*



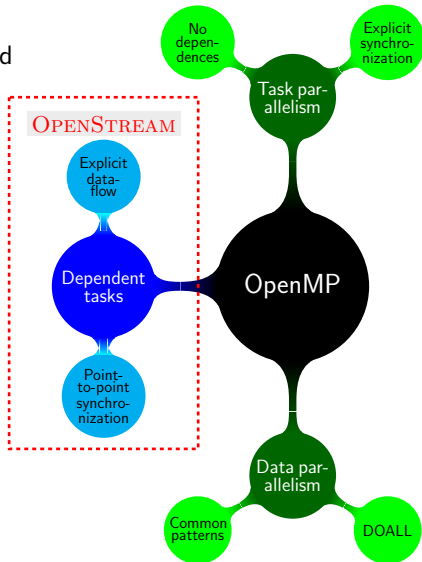
OpenStream

OpenMP extension

- ▶ leverage existing toolchains and knowledge
- ▶ maximize *productivity*

Parallelize irregular, dynamic codes

- ▶ no static/periodic restriction
- ▶ maximize *expressiveness*



OpenStream

OpenMP extension

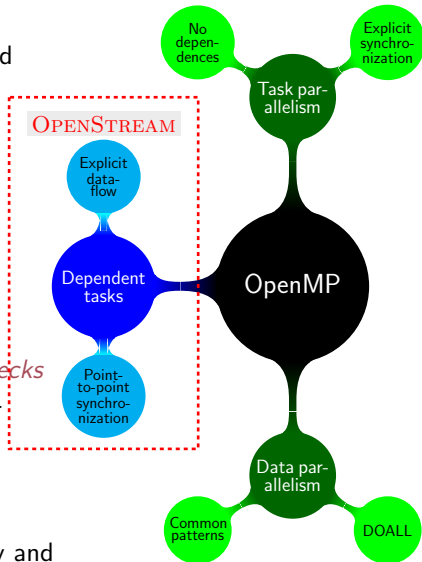
- ▶ leverage existing toolchains and knowledge
- ▶ maximize *productivity*

Parallelize irregular, dynamic codes

- ▶ no static/periodic restriction
- ▶ maximize *expressiveness*

Mitigate the von Neumann bottlenecks

- ▶ decoupled, producer/consumer task-parallel pipelines
- ▶ fight Amdahl's law, scavenge parallelism from all sources
- ▶ but also preserve local memory and communication bandwidth



OpenStream Introductory Example

```
for (i = 0; i < N; ++i) {  
  
    #pragma omp task firstprivate (i) output (x) // T1  
    x = foo (i);  
  
    #pragma omp task input (x) // T2  
    print (x);  
}
```

OpenStream Introductory Example

```
for (i = 0; i < N; ++i) {  
    #pragma omp task firstprivate (i) output (x) // T1  
    x = foo (i);  
  
    #pragma omp task input (x) // T2  
    print (x);  
}
```

Control program sequentially creates **N** instances of **T1** and of **T2**

Firstprivate clause privatizes variable **i** with initialization at task creation

Output clause gives write access to stream **x**

Input clause gives read access to stream **x**

Stream **x** has FIFO semantics

Stream FIFO Semantics

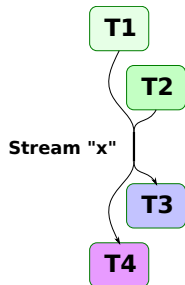
```
#pragma omp task output (x)                // Task T1
    x = ...;
for (i = 0; i < N; ++i) {
    int window_a[2], window_b[2];

    #pragma omp task output (x << window_a[2]) // Task T2
        window_a[0] = ...; window_a[1] = ...;
    if (i % 2) {
        #pragma omp task input (x >> window_b[2]) // Task T3
            use (window_b[0], window_b[1]);
    }
    #pragma omp task input (x)                // Task T4
        use (x);
}
```

Stream FIFO Semantics

```
#pragma omp task output (x)                // Task T1
    x = ...;
for (i = 0; i < N; ++i) {
    int window_a[2], window_b[2];

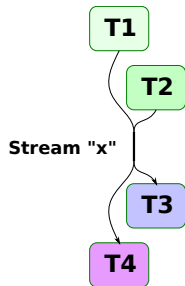
    #pragma omp task output (x << window_a[2]) // Task T2
        window_a[0] = ...; window_a[1] = ...;
    if (i % 2) {
        #pragma omp task input (x >> window_b[2]) // Task T3
            use (window_b[0], window_b[1]);
    }
    #pragma omp task input (x)                // Task T4
        use (x);
}
```



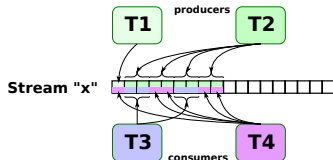
Stream FIFO Semantics

```
#pragma omp task output (x) // Task T1
x = ...;
for (i = 0; i < N; ++i) {
    int window_a[2], window_b[2];

    #pragma omp task output (x << window_a[2]) // Task T2
    window_a[0] = ...; window_a[1] = ...;
    if (i % 2) {
        #pragma omp task input (x >> window_b[2]) // Task T3
        use (window_b[0], window_b[1]);
    }
    #pragma omp task input (x) // Task T4
    use (x);
}
```



Interleaving of stream accesses



<http://openstream.info>

OpenStream

ACM TACO'13, IJPP'11, HiPEAC'11

Foster collaborations and technology transfers

IBM Haifa, U. of Sienna, Thales CS, Thales RT, CAPS Enterprise, Kalray, UPMC, BSC

Open
source
implem. in

GCC4.7

~30 kLoC

*GCC
summit
'08, '09*

New
stream
synchro-
nization
algorithms

*CASES'10,
submitted
EMSOFT'13*

Bench-
marks
~27 kLoC

*with third
party
contributions*

Profiling
and
visualiz.
infrastr.

*collaboration
with UPMC*

<http://openstream.info>

OpenStream

ACM TACO'13, IJPP'11, HiPEAC'11

Foster collaborations and technology transfers

IBM Haifa, U. of Sienna, Thales CS, Thales RT, CAPS Enterprise, Kalray, UPMC, BSC

Open
source
implem. in

GCC4.7
~30 kLoC

*GCC
summit
'08, '09*

New
stream
synchro-
nization
algorithms

*CASES'10,
submitted
EMSOFT'13*

Control-
Driven
Data Flow

CDDF

*PhD thesis,
submitted
research
report*

Bench-
marks
~27 kLoC

*with third
party
contributions*

Profiling
and
visualiz.
infrastr.

*collaboration
with UPMC*

<http://openstream.info>

OpenStream

ACM TACO'13, IJPP'11, HiPEAC'11

Control-Driven Data Flow

Define the formal semantics of imperative programming languages with dynamic, dependent task creation

- ▶ control flow: dynamic construction of task graphs
- ▶ data flow: decoupling dependent computations (Kahn)

CDDF model

▶ *Control program*

- ▶ imperative program
- ▶ creates tasks
- ▶ model: execution graph of *activation points*, each generating a *task activation*

▶ *Tasks*

- ▶ imperative program with a dynamic stream access signature
- ▶ becomes executable once its dependences are satisfied
- ▶ recursively becomes the control program for tasks created within
 - ▶ work in progress
 - ▶ link with synchronous languages
- ▶ model: *task activation* defined as a set of *stream accesses*

▶ *Streams*

- ▶ Kahn-style unbounded, indexed channels
- ▶ multiple producers and/or consumers
- ▶ specify dependences and/or communication
- ▶ model: indexed set of memory locations, defined on a finite subset

Control-Driven Data Flow – Results

- ▶ Deadlock classification
 - ▶ insufficiency deadlock: missing producer before a barrier or control program termination
 - ▶ functional deadlock: dependence cycle
 - ▶ spurious deadlock: deadlock induced by CDDF semantics on dependence enforcement (Kahn prefixes)
- ▶ Conditions on program state allowing to prove
 - ▶ deadlock freedom
 - ▶ compile-time serializability
 - ▶ functional and deadlock determinism

| Condition on state $\sigma = (k_e, \mathcal{A}_e, \mathcal{A}_o)$ | Deadlock Freedom properties | | | | Serializability | | Determinism Func ^{al} & Deadlock |
|--|-----------------------------|-------------------|-----------------------------------|-------------------|----------------------|----------|---|
| | $\neg D(\sigma)$ | $\neg ID(\sigma)$ | $\neg FD(\sigma) \vee ID(\sigma)$ | $\neg SD(\sigma)$ | Dyn. order | CP order | |
| $TC(\sigma) \wedge \forall s, \neg MPMC(s)$ Weaker than Kahn monotonicity | no | no | yes | yes | if $\neg ID(\sigma)$ | no | yes |
| $SCC(H(\sigma)) = \emptyset$ Common case, static over-approx. | no | no | yes | yes | if $\neg ID(\sigma)$ | no | yes |
| $SC(\sigma) \vee \Omega(k_e) \in \Pi$ Less restrictive than strictness | yes | yes | yes | yes | yes | no | yes |
| $\forall \sigma, SC(\sigma)$ Relaxed strictness | yes | yes | yes | yes | yes | yes | yes |

Foster collaborations and technology transfers

IBM Haifa, U. of Sienna, Thales CS, Thales RT, CAPS Enterprise, Kalray, UPMC, BSC

Open
source
implem. in
GCC4.7
~30 kLoC

*GCC
summit
'08, '09*

New
stream
synchro-
nization
algorithms

*CASES'10,
submitted
EMSOFT'13*

Feed-
Forward
Data Flow

Control-
Driven
Data Flow
CDDF

*PhD thesis,
submitted
research
report*

Bench-
marks
~27 kLoC

*with third
party
contributions*

Profiling
and
visualiz.
infrastr.

*collaboration
with UPMC*

<http://openstream.info>

OpenStream

ACM TACO'13, IJPP'11, HiPEAC'11

Runtime Design and Implementation

Efficiency requirements

1. eliminate false sharing
2. use software caching to reduce cache traffic
3. avoid atomic operations on data that is effectively shared across many cores
4. avoid *effective sharing* of concurrent structures

Feed-Forward Data Flow

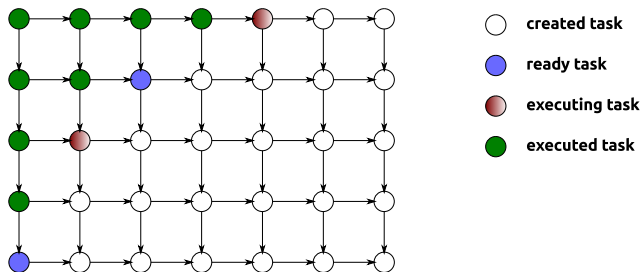
1. Resolve dependences at task creation
2. Producers know their consumers before executing

Feed-forward, a.k.a. argument fetch data flow

Feed-Forward Data Flow

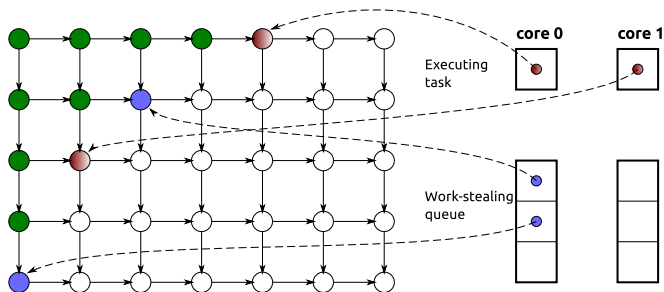
1. Resolve dependences at task creation
2. Producers know their consumers before executing

Feed-forward, a.k.a. argument fetch data flow



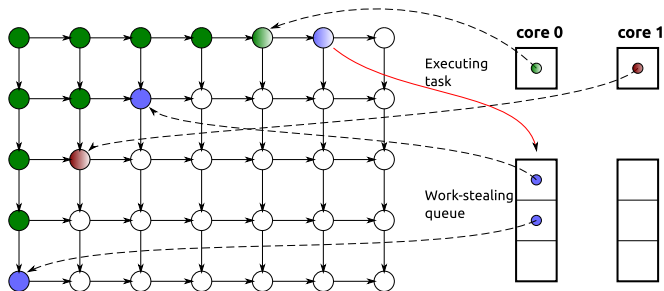
Feed-Forward Data Flow

1. Resolve dependences at task creation
2. Producers know their consumers before executing
Feed-forward, a.k.a. argument fetch data flow
3. Local work-stealing queue for ready tasks



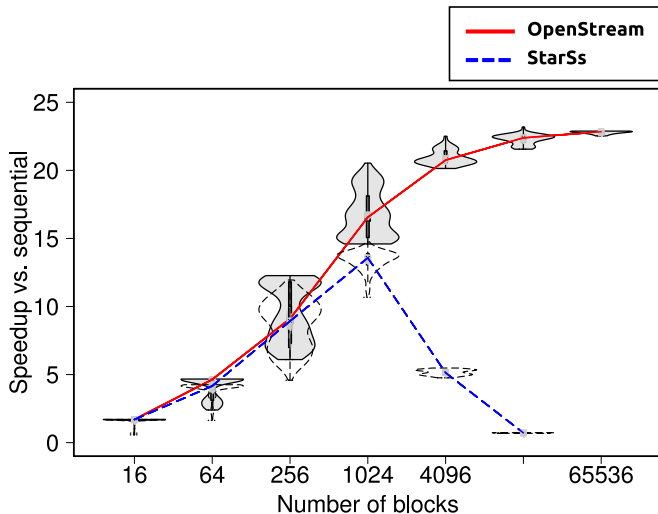
Feed-Forward Data Flow

1. Resolve dependences at task creation
2. Producers know their consumers before executing
Feed-forward, a.k.a. argument fetch data flow
3. Local work-stealing queue for ready tasks
4. Producer decides which consumers become executable
 - ▶ *local* consensus among producers providing data to the same task
 - ▶ without traversing *effectively* shared data structures



Comparison to Polling-Based Runtime

Block-sparse LU factorization on StarSs (block size 128×128)



Alternative: Using Explicit FIFOs

Arvind and Nikhil's I-structures

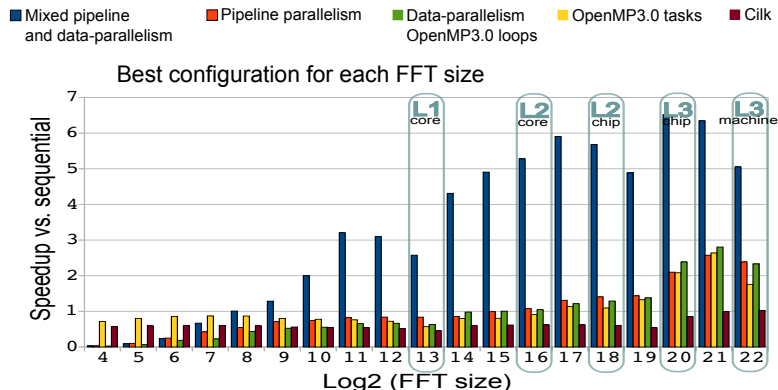
Book: *Implicit parallel programming in pH*, 2001.

I-structures: infinite, operational indexed streams, with *full/empty* bits, Single-Producer-Multiple-Consumers (SPMC)

- ▶ I-structures were introduced for in-place operations in data-flow programs
- ▶ Kahn networks provide a functional definition (and denotational semantics)
- ▶ Does *not* depend on a task scheduler to enforce dependences
- ▶ Caveats:
 - ▶ No load balancing
 - ▶ Need to derive a bounded implementation
 - ▶ Cumbersome interaction with user-level thread scheduling: blocking a task blocks the underlying worker (POSIX) thread! Need a mechanism for lightweight task suspension

Implementation: ring buffer, vector of futures

Streaming With FIFOs Wins: Evaluation on FFT



4-socket Opteron – 16 cores

How does it work? Please wait for implementation details

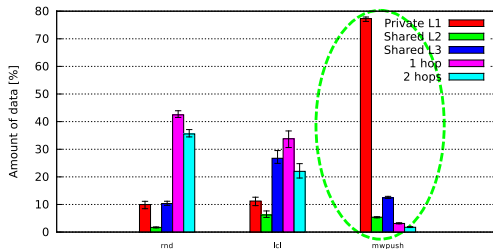
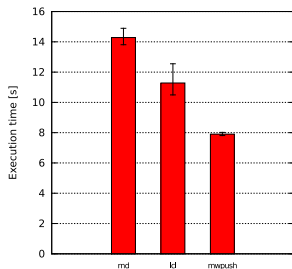
Ongoing Work

Runtime optimization

Ongoing Work

Runtime optimization

- ▶ task placement and topology-aware stealing



Ongoing Work

Runtime optimization

- ▶ task placement and topology-aware stealing
- ▶ automatic task aggregation

Ongoing Work

Runtime optimization

- ▶ task placement and topology-aware stealing
- ▶ automatic task aggregation
- ▶ runtime deadlock detection in presence of speculative aggregation

Ongoing Work

Distributed memory and heterogeneous platform execution

- ▶ Owner Writable Memory (OWM)
 - ▶ software coherence protocol
 - ▶ code-generation geared towards *Distributed Shared Memory*
 - ▶ explicit *cache/publish* operations
 - ▶ leverage one-sided communications and rDMA
- ▶ Nesting transactions and dynamic, task-level data flow
- ▶ Compiler transformations for dynamic, task-level data flow

Foster collaborations and technology transfers

IBM Haifa, U. of Sienna, Thales CS, Thales RT, CAPS Enterprise, Kalray, UPMC, BSC

Open
source
implem. in
GCC4.7
~30 kLoC

*GCC
summit
'08, '09*

New
stream
synchro-
nization
algorithms

*CASES'10,
submitted
EMSOFT'13*

Feed-
Forward
Data Flow

Control-
Driven
Data Flow
CDDF

*PhD thesis,
submitted
research
report*

Proof
techniques
for
concurrent
algorithms

PPoPP'13

Bench-
marks
~27 kLoC

*with third
party
contributions*

Profiling
and
visualiz.
infrastr.

*collaboration
with UPMC*

<http://openstream.info>

OpenStream

ACM TACO'13, IJPP'11, HiPEAC'11

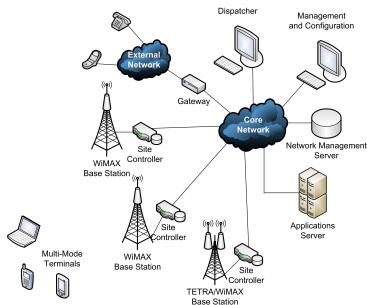
3. The Hammer: Language Design

OpenStream: dealing with the Von Neumann bottlenecks

Heptagon: safety-critical embedded systems

Application Challenges

Computational applications with real-time control aspect

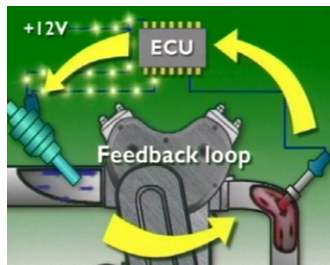


Embedded control systems running on more and more complex computer architectures, with compiler optimizations, parallel execution, and dynamic power management

Application Challenges

Safety-critical applications with simulation in the loop

Consolidating applications in mixed-critical systems, enabling communications between critical and non-critical components



Heptagon Goals

Generate *efficient parallel* code

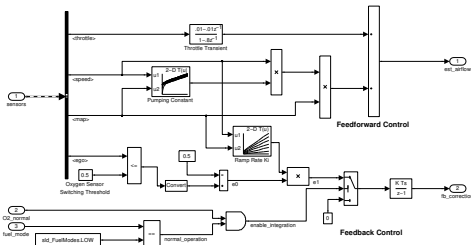
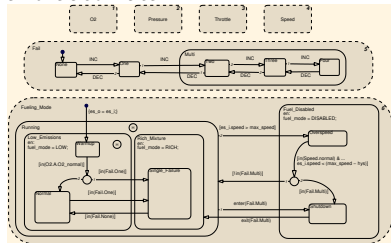
Preserving...

- ▶ the *functional semantics* of the program
- ▶ non-functional properties like *static and bounded memory*
- ▶ existing sequential compilation algorithms
- ▶ existing certification methodologies for embedded software

[Gérard et al., EMSOFT 2012]

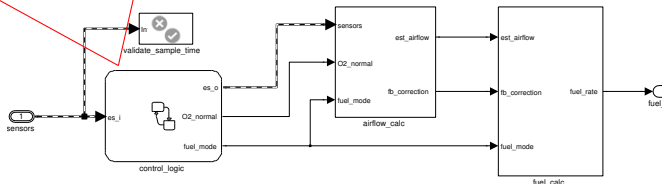
Common Practice in Embedded System Design

Matlab Simulink/StateFlow: mixed continuous/discrete signals, data-flow and automata



automata

dataflow



Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  y = x + m;
  m = 0 fby y;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow

Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow

Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register `fby`

| | |
|---|----------|
| m | 0 |
| x | |
| y | |

Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register `fby`

| | |
|---|---|
| m | 0 |
| x | 0 |
| y | |

Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register `fby`

| | |
|---|---|
| m | 0 |
| x | 0 |
| y | 0 |

Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register fby

| | | |
|---|---|---|
| m | 0 | 0 |
| x | 0 | |
| y | 0 | |

Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register `fby`

| | | |
|---|---|---|
| m | 0 | 0 |
| x | 0 | 1 |
| y | 0 | |

Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register `fby`

| | | |
|---|---|---|
| m | 0 | 0 |
| x | 0 | 1 |
| y | 0 | 1 |


Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register fby

| | | | |
|---|---|---|---|
| m | 0 | 0 | 1 |
| x | 0 | 1 | 0 |
| y | 0 | 1 | |



Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register `fby`

| | | | |
|---|---|---|---|
| m | 0 | 0 | 1 |
| x | 0 | 1 | 0 |
| y | 0 | 1 | 1 |

Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register fby

| | | | | |
|---|---|---|---|---|
| m | 0 | 0 | 1 | 1 |
| x | 0 | 1 | 0 | 2 |
| y | 0 | 1 | 1 | |

Heptagon a LUSTRE/SCADE-like language

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

Heptagon in short:

- ▶ Functional synchronous
- ▶ Declarative data-flow
- ▶ Values are streams
- ▶ Types and operators are lifted pointwise
- ▶ The synchronous register `fby`

| | | | | |
|---|---|---|---|---|
| m | 0 | 0 | 1 | 1 |
| x | 0 | 1 | 0 | 2 |
| y | 0 | 1 | 1 | 3 |

Heptagon a LUSTRE/SCADE-like language

Translate into JAVA syntax:

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- Modular compilation, each node is compiled into a class.

Heptagon a LUSTRE/SCADE-like language

Translate into JAVA syntax:

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- ▶ Modular compilation, each node is compiled into a class.
- ▶ Synchronous registers are instance variables.

Heptagon a LUSTRE/SCADE-like language

Translate into JAVA syntax:

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- ▶ Modular compilation, each node is compiled into a class.
- ▶ Synchronous registers are instance variables.
- ▶ Initialisation (and reinitialisation) method.

Heptagon a LUSTRE/SCADE-like language

Translate into JAVA syntax:

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- ▶ Modular compilation, each node is compiled into a class.
- ▶ Synchronous registers are instance variables.
- ▶ Initialisation (and reinitialisation) method.
- ▶ Step method, with in place update of the state.

Heptagon a LUSTRE/SCADE-like language

Translate into JAVA syntax:

```
node sum(x:int)=(y:int)
var m :int;
let
  m = 0 fby y;
  y = x + m;
tel
```

```
class Sum {
  int m;
  void reset(){ m = 0; }
  int step(int x){
    int y;
    y = x + m;
    m = y;
    return y;
  }
}
```

- ▶ Modular compilation, each node is compiled into a class.
- ▶ Synchronous registers are instance variables.
- ▶ Initialisation (and reinitialisation) method.
- ▶ Step method, with in place update of the state.

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- ▶ `when`: the sampling operator

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- ▶ `when`: the sampling operator
- ▶ `merge`: the (lazy) complementing operator

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- ▶ `when`: the sampling operator
- ▶ `merge`: the (lazy) complementing operator

| | |
|----------------------------------|-------------|
| <code>x</code> | 0 |
| <code>big = period3()</code> | <i>true</i> |
| <code>xt = x when big</code> | |
| <code>xf = x whenot big</code> | |
| <code>y = merge big xt xf</code> | |

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- ▶ `when`: the sampling operator
- ▶ `merge`: the (lazy) complementing operator

| | |
|----------------------------------|-------------|
| <code>x</code> | 0 |
| <code>big = period3()</code> | <i>true</i> |
| <code>xt = x when big</code> | 0 |
| <code>xf = x whenot big</code> | . |
| <code>y = merge big xt xf</code> | |

- ▶ `whenot = when not`
- ▶ `(.)` = absence of value

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- ▶ `when`: the sampling operator
- ▶ `merge`: the (lazy) complementing operator

| | |
|----------------------------------|-------------|
| <code>x</code> | 0 |
| <code>big = period3()</code> | <i>true</i> |
| <code>xt = x when big</code> | 0 |
| <code>xf = x whenot big</code> | . |
| <code>y = merge big xt xf</code> | 0 |

- ▶ `whenot = when not`
- ▶ `(.)` = absence of value

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- ▶ `when`: the sampling operator
- ▶ `merge`: the (lazy) complementing operator

| x | 0 | 1 |
|----------------------------------|-------------|--------------|
| <code>big = period3()</code> | <i>true</i> | <i>false</i> |
| <code>xt = x when big</code> | 0 | |
| <code>xf = x whenot big</code> | . | |
| <code>y = merge big xt xf</code> | 0 | |

- ▶ `whenot = when not`
- ▶ `(.)` = absence of value

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- ▶ `when`: the sampling operator
- ▶ `merge`: the (lazy) complementing operator

| x | 0 | 1 |
|----------------------------------|-------------|--------------|
| <code>big = period3()</code> | <i>true</i> | <i>false</i> |
| <code>xt = x when big</code> | 0 | . |
| <code>xf = x whenot big</code> | . | 1 |
| <code>y = merge big xt xf</code> | 0 | |

- ▶ `whenot = when not`
- ▶ `(.)` = absence of value

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- ▶ `when`: the sampling operator
- ▶ `merge`: the (lazy) complementing operator

| x | 0 | 1 |
|----------------------------------|-------------|--------------|
| <code>big = period3()</code> | <i>true</i> | <i>false</i> |
| <code>xt = x when big</code> | 0 | . |
| <code>xf = x whenot big</code> | . | 1 |
| <code>y = merge big xt xf</code> | 0 | 1 |

- ▶ `whenot = when not`
- ▶ `(.)` = absence of value
- ▶ `merge` is *lazy*, its inputs have to arrive only when needed.

Sampling and complementing streams: *when* and *merge*

Two core data-flow operators to manipulate streams:

- ▶ *when*: the sampling operator
- ▶ *merge*: the (lazy) complementing operator

| x | 0 | 1 | 2 |
|----------------------------------|-------------|--------------|--------------|
| <code>big = period3()</code> | <i>true</i> | <i>false</i> | <i>false</i> |
| <code>xt = x when big</code> | 0 | . | . |
| <code>xf = x whenot big</code> | . | 1 | 2 |
| <code>y = merge big xt xf</code> | 0 | 1 | 2 |

- ▶ *whenot* = *when* not
- ▶ *(.)* = absence of value
- ▶ *merge* is *lazy*, its inputs have to arrive only when needed.

Sampling and complementing streams: `when` and `merge`

Two core data-flow operators to manipulate streams:

- ▶ `when`: the sampling operator
- ▶ `merge`: the (lazy) complementing operator

| x | 0 | 1 | 2 | 3 | 4 | ... |
|----------------------------------|-------------|--------------|--------------|-------------|--------------|-----|
| <code>big = period3()</code> | <i>true</i> | <i>false</i> | <i>false</i> | <i>true</i> | <i>false</i> | ... |
| <code>xt = x when big</code> | 0 | . | . | 3 | . | ... |
| <code>xf = x whenot big</code> | . | 1 | 2 | . | 4 | ... |
| <code>y = merge big xt xf</code> | 0 | 1 | 2 | 3 | 4 | ... |

- ▶ `whenot = when not`
- ▶ `(.)` = absence of value
- ▶ `merge` is *lazy*, its inputs have to arrive only when needed.
- ▶ The compiler computes correct rhythm for every stream.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);
```

| | | | | | | |
|----|---|------|------|------|-------|-----|
| ys | 0 | 3.14 | 6.28 | 9.42 | 12.56 | ... |
|----|---|------|------|------|-------|-----|

- slow: step integration with horizon of 1 second.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);  
yf = 0 fby fast(1, yf);
```

| | | | | | | |
|----|---|------|------|------|-------|-----|
| ys | 0 | 3.14 | 6.28 | 9.42 | 12.56 | ... |
| yf | 0 | 3 | 6 | 9 | 12 | ... |

- ▶ slow: step integration with horizon of 1 second.
- ▶ fast: fast approximate

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);  
yf = 0 fby fast(1/3, yf);
```

| | | | | | | | | | | | |
|----|---|------|------|------|-------|-----|---|---|---|---|-------|
| ys | 0 | 3.14 | 6.28 | 9.42 | 12.56 | ... | | | | | |
| yf | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10... |

- ▶ slow: step integration with horizon of 1 second.
- ▶ fast: fast approximate with horizon of 1/3 second.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);  
yf = 0 fby fast(1/3, yf);  
big = period3();  
y = merge big ys (yf whenot big);
```

| big | true | false | false | true | false | false | true | false | ... |
|-----|------|-------|-------|------|-------|-------|------|-------|-----|
| ys | 0 | . | . | 3.14 | . | . | 6.28 | . | ... |
| yf | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
| y | 0 | 1 | 2 | 3.14 | 4 | 5 | 6.28 | 7 | ... |

- ▶ slow: step integration with horizon of 1 second.
- ▶ fast: fast approximate with horizon of 1/3 second.
- ▶ We use the correct value when possible.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);  
yf = 0 fby fast(1/3, yf);  
big = period3();  
y = merge big ys (yf whennot big);
```

| big | true | false | false | true | false | false | true | false | ... |
|-----|------|-------|-------|------|-------|-------|------|-------|-----|
| ys | 0 | . | . | 3.14 | . | . | 6.28 | . | ... |
| yf | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
| y | 0 | 1 | 2 | 3.14 | 4 | 5 | 6.28 | 7 | ... |

- ▶ slow: step integration with horizon of 1 second.
- ▶ fast: fast approximate with horizon of 1/3 second.
- ▶ We use the correct value when possible.
- ▶ And complement with the approximate one.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);  
yf = 0 fby fast(1/3, y);  
big = period3();  
y = merge big ys (yf whenot big);
```

| big | true | false | false | true | false | false | true | false | ... |
|-----|------|-------|-------|------|-------|-------|------|-------|-----|
| ys | 0 | . | . | 3.14 | . | . | 6.28 | . | ... |
| yf | 0 | 1 | 2 | 3 | 4.14 | 5.14 | 6.14 | 7.28 | ... |
| y | 0 | 1 | 2 | 3.14 | 4.14 | 5.14 | 6.28 | 7.28 | ... |

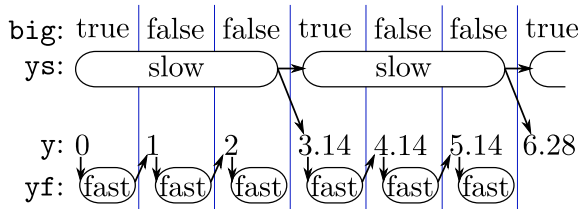
- ▶ slow: step integration with horizon of 1 second.
- ▶ fast: fast approximate with horizon of 1/3 second.
- ▶ We use the correct value when possible.
- ▶ And complement with the approximate one.

The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);  
yf = 0 fby fast(1/3, y);  
big = period3();  
y = merge big ys (yf whennot big);
```

| big | true | false | false | true | false | false | true | false | ... |
|-----|------|-------|-------|------|-------|-------|------|-------|-----|
| ys | 0 | . | . | 3.14 | . | . | 6.28 | . | ... |
| yf | 0 | 1 | 2 | 3 | 4.14 | 5.14 | 6.14 | 7.28 | ... |
| y | 0 | 1 | 2 | 3.14 | 4.14 | 5.14 | 6.28 | 7.28 | ... |

We would like to run them in parallel:

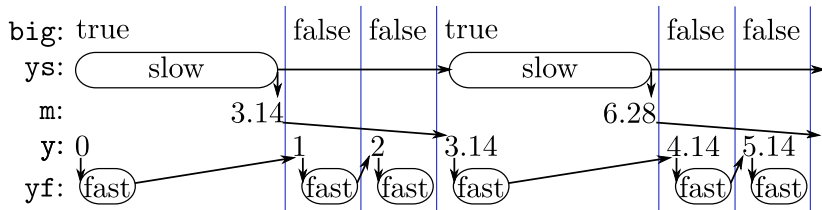


The slow_fast classical exemple

```
ys = 0 fby slow(1, ys);  
yf = 0 fby fast(1/3, y);  
big = period3();  
y = merge big ys (yf whenot big);
```

| big | true | false | false | true | false | false | true | false | ... |
|-----|------|-------|-------|------|-------|-------|------|-------|-----|
| ys | 0 | . | . | 3.14 | . | . | 6.28 | . | ... |
| yf | 0 | 1 | 2 | 3 | 4.14 | 5.14 | 6.14 | 7.28 | ... |
| y | 0 | 1 | 2 | 3.14 | 4.14 | 5.14 | 6.28 | 7.28 | ... |

This is what happens, unfortunately:



Synchronous register are synchronous

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m;
  float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big;
    big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f,y);
    return y;
  }
}
```

Reminder:

- ▶ y gets the value of the register m.
- ▶ During the same step, m is *updated for the next time*.

Synchronous register are synchronous

```
class Slow_fast {
    Fast fast;
    Slow slow;
    Period3 period3;
    float m;
    float m2;
    void reset () {
        period3.reset();
        slow.reset();
        fast.reset();
        m = 0.f;
        m2 = 0.f;
    }
    float step () {
        float y;
        boolean big;
        big = period3.step();
        if (big) {
            y = m;
            m = slow.step(1.f, y);
        } else {
            y = m2;
        }
        m2 = fast.step(0.3f,y);
        return y;
    }
}
```

Reminder:

- ▶ y gets the value of the register m.
- ▶ During the same step, m is *updated for the next time*.

This sequential compilation is:

- ▶ very efficient and simple
- ▶ traceable
- ▶ used and certified in Scade 6

But it *prevents parallelization across step boundaries*.

Synchronous register are synchronous

```
class Slow_fast {
  Fast fast;
  Slow slow;
  Period3 period3;
  float m;
  float m2;
  void reset () {
    period3.reset();
    slow.reset();
    fast.reset();
    m = 0.f;
    m2 = 0.f;
  }
  float step () {
    float y;
    boolean big;
    big = period3.step();
    if (big) {
      y = m;
      m = slow.step(1.f, y);
    } else {
      y = m2;
    }
    m2 = fast.step(0.3f, y);
    return y;
  }
}
```

Reminder:

- ▶ y gets the value of the register m.
- ▶ During the same step, m is *updated for the next time*.

This sequential compilation is:

- ▶ very efficient and simple
- ▶ traceable
- ▶ used and certified in Scade 6

But it *prevents parallelization across step boundaries*.

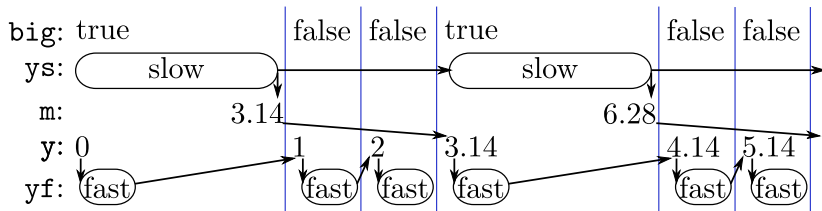
OCREP by A. Girault

The distributed imperative code is *optimized to bypass* the synchronous register.

Decoupling slow_fast with *futures*

```
node slow_fast() = (y :float)
var big :bool; yf :float; ys :float
let
  ys = 0 fby slow(1, ys);
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whennot big);
tel
```

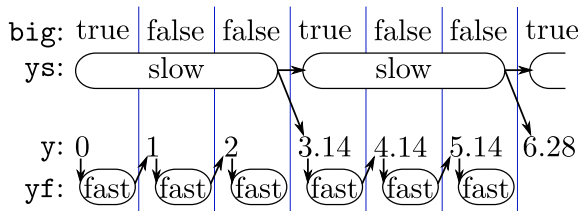
We had this:



Decoupling slow_fast with *futures*

```
node slow_fast() = (y :float)
var big :bool; yf :float; ys :float
let
  ys = 0 fby slow(1, ys);
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whenot big);
tel
```

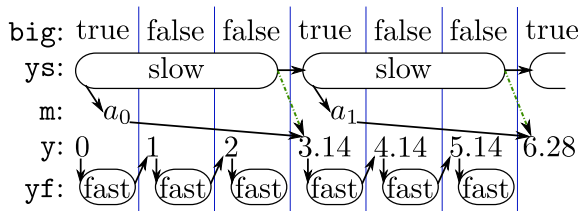
We want this:



Decoupling slow_fast with *futures*

```
node slow_fast() = (y :float)
var big :bool; yf :float; ys :float
let
  ys = 0 fby slow(1, ys);
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whenot big);
tel
```

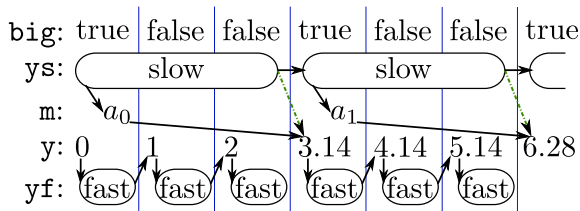
We can use *futures* as proxies:



Decoupling slow_fast with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :float
let
  ys = 0 fby (async slow(1, ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whenot big);
tel
```

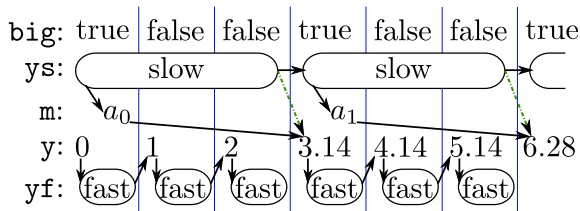
We can use *futures* as proxies:



Decoupling slow_fast with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :future float
let
  ys = (async 0) fby (async slow(1, ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big ys (yf whenot big);
tel
```

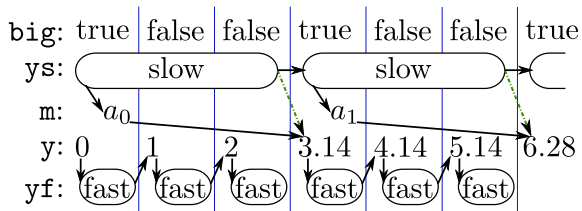
We can use *futures* as proxies:



Decoupling slow_fast with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :future float
let
  ys = (async 0) fby (async slow(1, ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big !ys (yf whennot big);
tel
```

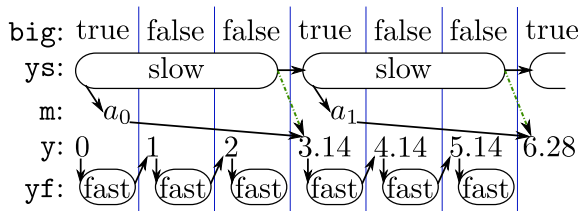
We can use *futures* as proxies:



Decoupling slow_fast with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :future float
let
  ys = (async 0) fby (async slow(1, !ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big !ys (yf whennot big);
tel
```

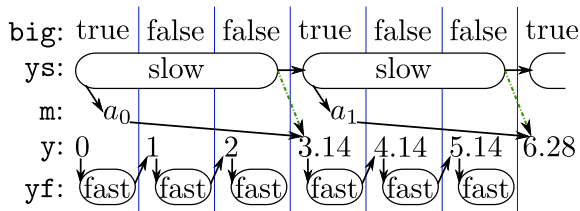
We can use *futures* as proxies:



Decoupling slow_fast with *futures*

```
node slow_fast_a() = (y :float)
var big :bool; yf :float; ys :future float
let
  ys = (async 0) fby (async slow(1, !ys));
  yf = 0 fby fast(1/3, y);
  big = period3();
  y = merge big !ys (yf whenot big);
tel
```

We can use *futures* as proxies:



Futures

- ▶ Futures appeared in MULTILISP [Halstead, 1985].
- ▶ Are now in most functional languages and Java, C++, etc.
- ▶ Depending on language integration, it can be a mere library.

Futures

- ▶ Futures appeared in MULTILISP [Halstead, 1985].
- ▶ Are now in most functional languages and Java, C++, etc.
- ▶ Depending on language integration, it can be a mere library.

What is the future?

Futures

- ▶ Futures appeared in MULTILISP [Halstead, 1985].
- ▶ Are now in most functional languages and Java, C++, etc.
- ▶ Depending on language integration, it can be a mere library.

What is a future?

Futures

- ▶ Futures appeared in MULTILISP [Halstead, 1985].
- ▶ Are now in most functional languages and Java, C++, etc.
- ▶ Depending on language integration, it can be a mere library.

What is a future?

It is *a value*, which will hold the result of a *closed term*.

Futures

- ▶ Futures appeared in MULTILISP [Halstead, 1985].
- ▶ Are now in most functional languages and Java, C++, etc.
- ▶ Depending on language integration, it can be a mere library.

What is a future?

It is *a value*, which will hold the result of a *closed term*.

Intuitively, it is a *promise* of result that is *bound to come*.

Futures

- ▶ Futures appeared in MULTILISP [Halstead, 1985].
- ▶ Are now in most functional languages and Java, C++, etc.
- ▶ Depending on language integration, it can be a mere library.

What is a future?

It is *a value*, which will hold the result of a *closed term*.

Intuitively, it is a *promise* of result that is *bound to come*.

To guarantee futures integrity in Heptagon:

- ▶ future t is an *abstract type*, with t being the result type.
- ▶ A future may *only* be created from:
 - ▶ Constants: `async 42`
 - ▶ Asynchronous function calls: `async f(x,y)`
- ▶ `!x` “get” the result held by the future x — it is *blocking*.

Unchanged compilation: find the 4 differences

```
class Slow_fast {
    Fast fast;
    Slow slow;
    Period3 period3;
    float m; float m2;
    void reset () {
        period3.reset();
        slow.reset();
        fast.reset();
        m = 0.f;
        m2 = 0.f;
    }
    float step () {
        float y;
        boolean big = period3.step();
        if (big) {
            y = m;
            m = slow.step(1.f, y);
        } else {
            y = m2;
        }
        m2 = fast.step(0.3f,y);
        return y;
    }
}
```



```
class Slow_fast_a {
    Fast fast;
    Async<Slow> slow;
    Period3 period3;
    Future<float> m; float m2;
    void reset () {
        period3.reset();
        slow.reset();
        fast.reset();
        m = new Future(0.f);
        m2 = 0.f;
    }
    float step () {
        float y;
        boolean big = period3.step();
        if (big) {
            y = m.get();
            m = slow.step(1.f, y);
        } else {
            y = m2;
        }
        m2 = fast.step(0.3f,y);
        return y;
    }
}
```

Unchanged compilation: find the 4 differences

```
class Slow_fast {  
    Fast fast;  
    Slow slow;  
    Period3 period3;  
    float m; float m2;  
    void reset () {  
        period3.reset();  
        slow.reset();  
        fast.reset();  
        m = 0.f;  
        m2 = 0.f;  
    }  
    float step () {  
        float y;  
        boolean big = period3.step();  
        if (big) {  
            y = m;  
            m = slow.step(1.f, y);  
        } else {  
            y = m2;  
        }  
        m2 = fast.step(0.3f,y);  
        return y;  
    }  
}
```



```
class Slow_fast_a {  
    Fast fast;  
    Async<Slow> slow;  
    Period3 period3;  
    Future<float> m; float m2;  
    void reset () {  
        period3.reset();  
        slow.reset();  
        fast.reset();  
        m = new Future(0.f);  
        m2 = 0.f;  
    }  
    float step () {  
        float y;  
        boolean big = period3.step();  
        if (big) {  
            y = m.get();  
            m = slow.step(1.f, y);  
        } else {  
            y = m2;  
        }  
        m2 = fast.step(0.3f,y);  
        return y;  
    }  
}
```


Unchanged compilation: find the 4 differences

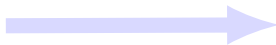
```
class Slow_fast {  
    Fast fast;  
    Slow slow;  
    Period3 period3;  
    float m; float m2;  
    void reset () {  
        period3.reset();  
        slow.reset();  
        fast.reset();  
        m = 0.f;  
        m2 = 0.f;  
    }  
    float step () {  
        float y;  
        boolean big = period3.step();  
        if (big) {  
            y = m;  
            m = slow.step(1.f, y);  
        } else {  
            y = m2;  
        }  
        m2 = fast.step(0.3f,y);  
        return y;  
    }  
}
```



```
class Slow_fast_a {  
    Fast fast;  
    Async<Slow> slow;  
    Period3 period3;  
    Future<float> m; float m2;  
    void reset () {  
        period3.reset();  
        slow.reset();  
        fast.reset();  
        m = new Future(0.f);  
        m2 = 0.f;  
    }  
    float step () {  
        float y;  
        boolean big = period3.step();  
        if (big) {  
            y = m.get();  
            m = slow.step(1.f, y);  
        } else {  
            y = m2;  
        }  
        m2 = fast.step(0.3f,y);  
        return y;  
    }  
}
```

Unchanged compilation: find the 4 differences

```
class Slow_fast {
    Fast fast;
    Slow slow;
    Period3 period3;
    float m; float m2;
    void reset () {
        period3.reset();
        slow.reset();
        fast.reset();
        m = 0.f;
        m2 = 0.f;
    }
    float step () {
        float y;
        boolean big = period3.step();
        if (big) {
            y = m;
            m = slow.step(1.f, y);
        } else {
            y = m2;
        }
        m2 = fast.step(0.3f,y);
        return y;
    }
}
```



```
class Slow_fast_a {
    Fast fast;
    Async<Slow> slow;
    Period3 period3;
    Future<float> m; float m2;
    void reset () {
        period3.reset();
        slow.reset();
        fast.reset();
        m = new Future(0.f);
        m2 = 0.f;
    }
    float step () {
        float y;
        boolean big = period3.step();
        if (big) {
            y = m.get();
            m = slow.step(1.f, y);
        } else {
            y = m2;
        }
        m2 = fast.step(0.3f,y);
        return y;
    }
}
```

Unchanged compilation: find the 4 differences

```
class Slow_fast {
    Fast fast;
    Slow slow;
    Period3 period3;
    float m; float m2;
    void reset () {
        period3.reset();
        slow.reset();
        fast.reset();
        m = 0.f;
        m2 = 0.f;
    }
    float step () {
        float y;
        boolean big = period3.step();
        if (big) {
            y = m;
            m = slow.step(1.f, y);
        } else {
            y = m2;
        }
        m2 = fast.step(0.3f,y);
        return y;
    }
}
```



```
class Slow_fast_a {
    Fast fast;
    Async<Slow> slow;
    Period3 period3;
    Future<float> m; float m2;
    void reset () {
        period3.reset();
        slow.reset();
        fast.reset();
        m = new Future(0.f);
        m2 = 0.f;
    }
    float step () {
        float y;
        boolean big = period3.step();
        if (big) {
            y = m.get();
            m = slow.step(1.f, y);
        } else {
            y = m2;
        }
        m2 = fast.step(0.3f,y);
        return y;
    }
}
```

Implementation: the `async` wrapper

The `async` wrapper

- ▶ runs asynchronously a node in a worker thread.
- ▶ behaves like a node:
 - ▶ `step`
 - ▶ At each input a future is returned.
 - ▶ Inputs are fed to the wrapped node through a buffer.
 - ▶ `reset` is done so as to allow data-parallelism.

Implementation: the `async` wrapper

The `async` wrapper

- ▶ runs asynchronously a node in a worker thread.
- ▶ behaves like a node:
 - ▶ `step`
 - ▶ At each input a future is returned.
 - ▶ Inputs are fed to the wrapped node through a buffer.
 - ▶ `reset` is done so as to allow data-parallelism.

Important observations

- ▶ the need of an input buffer to allow *decoupling*
- ▶ the use of reset to enable *data-parallelism*

Memory management

A future

- ▶ is a shared object with one producer, multiple consumers
- ▶ may be stored and used later on
- ▶ might not be used at all
- ▶ typically depends on the evaluation of upstream futures

Memory management

A future

- ▶ is a shared object with one producer, multiple consumers
- ▶ may be stored and used later on
- ▶ might not be used at all
- ▶ typically depends on the evaluation of upstream futures

Without restrictions, the live-range of a future is *undecidable* and a *concurrent gc* is needed, as the one of java.

Memory management

A future

- ▶ is a shared object with one producer, multiple consumers
- ▶ may be stored and used later on
- ▶ might not be used at all
- ▶ typically depends on the evaluation of upstream futures

Without restrictions, the live-range of a future is *undecidable* and a *concurrent gc* is needed, as the one of java.

Memory boundedness

Alive futures are bounded by the number of synchronous registers.
A slab allocator is possible with *static allocation* and reuse.

Memory management

A future

- ▶ is a shared object with one producer, multiple consumers
- ▶ may be stored and used later on
- ▶ might not be used at all
- ▶ typically depends on the evaluation of upstream futures

Without restrictions, the live-range of a future is *undecidable* and a *concurrent gc* is needed, as the one of java.

Memory boundedness

Alive futures are bounded by the number of synchronous registers.
A slab allocator is possible with *static allocation* and reuse.

Scope restriction for node level memory management

Preventing futures to be returned or passed to an *async* call, allows gc and slab to be *synchronous* and *node local*.

Backends

Existing JAVA backend

- ▶ Futures are the ones of JAVA
- ▶ Static queues and worker threads
- ▶ But dynamic allocation of futures

Backends

Existing JAVA backend

- ▶ Futures are the ones of JAVA
- ▶ Static queues and worker threads
- ▶ But dynamic allocation of futures

Existing C backend, aiming for embedded systems

- ▶ Hand tailored futures, queues and threads
- ▶ Slab allocator local to each node
- ▶ Futures have scope restrictions

Backends

Existing JAVA backend

- ▶ Futures are the ones of JAVA
- ▶ Static queues and worker threads
- ▶ But dynamic allocation of futures

Existing C backend, aiming for embedded systems

- ▶ Hand tailored futures, queues and threads
- ▶ Slab allocator local to each node
- ▶ Futures have scope restrictions

Work-in-progress OPENSTREAM backend

- ▶ Data-flow parallel runtime with high-performance task scheduler
- ▶ Handle a large number of `async`

Code generation for embedded systems

`async` may be *annotated* with any needed static arguments.

Code generation for embedded systems

`async` may be *annotated* with any needed static arguments.

Location annotations for distribution without scheduler:

- ▶ One thread per computing unit
- ▶ No surprise
- ▶ Usually not efficient

Code generation for embedded systems

`async` may be *annotated* with any needed static arguments.

Location annotations for distribution without scheduler:

- ▶ One thread per computing unit
- ▶ No surprise
- ▶ Usually not efficient

Priority annotations for EDF scheduling:

- ▶ Well known
- ▶ May be optimal
- ▶ Existing tools need to be adapted

Wrap-Up on Heptagon

Semantics

Same semantics as the sequential program without `async` and `!`

Expressivness

- ▶ Synchronous language: *time programming*
- ▶ Futures: decouple and make explicit *beginning* and *end* of computations
- ▶ Together they allow for *programing parallelism*:
 - ▶ decoupling, partial-decoupling
 - ▶ data-parallelism
 - ▶ fork-join, temporal fork-join
 - ▶ pipeline, etc.

Safety

- ▶ Statically serializable: futures in a pure language
- ▶ No dynamic memory allocation or thread creation
- ▶ Proven runtime system (scheduler, FIFO ring buffer)
- ▶ Proven compilation flow (part of it)

4. The Anvil: Runtime System Design

Stream Processing?

Driving Force: Correct Concurrency by Construction

The Hammer: Language Design

The Anvil: Runtime System Design

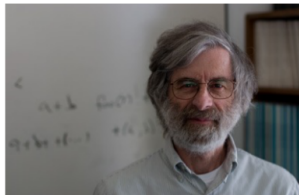
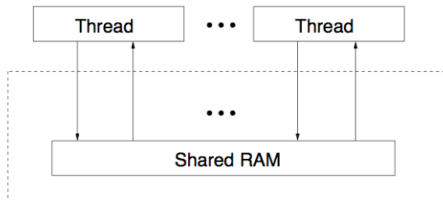
Wrap-Up

Concurrent Programming: Which Abstraction?

Simplest asynchronous model: sequential consistency

...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program...

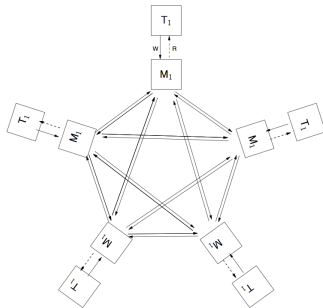
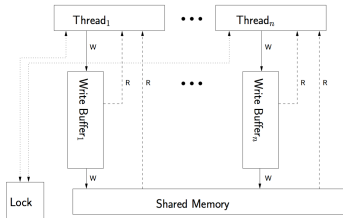
Lamport, 1979.



Still... Non-Determinism, Data Races, Contention



Real World is Worse: Relaxed Memory Models



Shared-Memory Concurrency in the Real World

```
void __lockfunc __op##_lock(locktype##_t *lock)
{
    for (;;) {
        preempt_disable();
        if (likely(__raw__op##_trylock(lock)))
            break;
        preempt_enable();

        if (!(lock)->break_lock)
            (lock)->break_lock = 1;
        while (!op##_can_lock(lock) && (lock)->break_lock)
            __raw__op##_relax(&lock->raw_lock);
    }
    (lock)->break_lock = 0;
}
```

excerpt from Linux spinlock.c

Shared-Memory Concurrency in the Real World

```
void __lockfunc __op##_lock(locktype##_t *lock)
{
    for (;;) {
        preempt_disable();
        if (likely(__raw__op##_trylock(lock))
            break;
        preempt_enable();

        if (!(lock->break_lock)
            (lock->break_lock = 1;
        while (!op##_can_lock(lock) && (lock->break_lock)
            __raw__op##_relax(&lock->raw_lock);
        }
        (lock->break_lock = 0;
    }
}
```

excerpt from Linux spinlock.c

```
int steal(Deque *q) {
    size_t t = load_explicit(&q->top, acquire);
    thread_fence(seq_cst);
    size_t b = load_explicit(&q->bottom, acquire);
    int x = EMPTY;
    if (t < b) {
        /* Non-empty queue. */
        Array *a = load_explicit(&q->array, relaxed);
        x = load_explicit(&a->buffer[t % a->size], relaxed);
        if (!compare_exchange_strong_explicit(&q->top, &t, t + 1, seq_cst, relaxed))
            /* Failed race. */
            return ABORT;
    }
    return x;
}
```

Shared-Memory Concurrency in the Real World

```
void __lockfunc __op##_lock(locktype##_t *lock)
{
```

```
    for (;;) {
```

```
        preempt_disable(
```

```
            if (likely(!_raw_
```

```
                break;
```

```
        preempt_enable()
```

```
        if (!(<lock>->bre
```

```
            (<lock>->|
```

```
        while (!<op>##_can
```

```
            _raw_##O|
```

```
    }
```

```
    (<lock>->break_lock = 0;
```

```
}
```

excerpt from Linux spinlock.c

Lemma 3. *The following properties involving barriers apply:*

(i) $(Wx, - \xrightarrow{\text{sync}} Wy, - \xrightarrow{\text{pp-sat}} Rz, - \vee Wx, - \xrightarrow{\text{pp-sat}} Ry, - \xrightarrow{\text{sync}} Rz, -)$

$\implies Wx, - \xrightarrow{\text{pp-sat}} Rz, -$

(ii) $A.Wx, - \xrightarrow{\text{rf}} B.Rx, - \xrightarrow{\text{sync}} B.Wy, - \xrightarrow{\text{pp-sat}} C.Rx, -$

$\implies A.Wx, - \xrightarrow{\text{pp-sat}} C.Rx, -$

(iii) *Let X stand for $A.Wx, - \xrightarrow{\text{rf}} B.Rx, -$ or $(A \sim B).Wx, -$*

and Y stand for $C.Wy, - \xrightarrow{\text{rf}} D.Ry, -$ or $(C \sim D).Wy, -$

then the following holds:

$\neg (X \xrightarrow{\text{sync}} B.Ry, - \xrightarrow{\text{fr}} C.Wy, - \wedge Y \xrightarrow{\text{sync}} D.Rx, - \xrightarrow{\text{fr}} A.Wx, -)$

```
int steal(Deque *q) {
    size_t t = load_explicit(&q->top, acquire);
    thread_fence(seq_cst);
    size_t b = load_explicit(&q->bottom, acquire);
    int x = EMPTY;
    if (t < b) {
        /* Non-empty queue. */
        Array *a = load_explicit(&q->array, relaxed);
        x = load_explicit(&a->buffer[t % a->size], relaxed);
        if (!compare_exchange_strong_explicit(&q->top, &t, t + 1, seq_cst, relaxed))
            /* Failed race. */
            return ABORT;
    }
    return x;
}
```

Shared-Memory Concurrency in the Real World

```
void __lockfunc_ ##op##_lock(locktype##_t *lock)
{
```

```
    for (;;) {
```

```
        preempt_disable(
```

```
            if (likely(!_raw_
```

```
                break;
```

```
        preempt_enable()
```

```
        if (!(<lock>->bre
```

```
            (<lock>->
```

```
        while (!op##_can
```

```
            _raw_##o
```

```
    }
```

```
    (<lock>->break_lock = 0;
```

```
}
```

excerpt from Linux spinlock.c

Lemma 3. *The following properties involving barriers apply:*

(i) $(Wx, - \xrightarrow{\text{sync}} Wy, - \xrightarrow{\text{pp-sat}} Rz, - \vee Wx, - \xrightarrow{\text{pp-sat}} Ry, - \xrightarrow{\text{sync}} Rz, -)$

$\implies Wx, - \xrightarrow{\text{pp-sat}} Rz, -$

(ii) $A.Wx, - \xrightarrow{\text{rf}} B.Rx, - \xrightarrow{\text{sync}} B.Wy, - \xrightarrow{\text{pp-sat}} C.Rx, -$

$\implies A.Wx, - \xrightarrow{\text{pp-sat}} C.Rx, -$

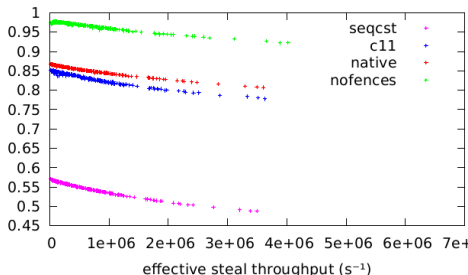
(iii) *Let X stand for $A.Wx, - \xrightarrow{\text{rf}} B.Rx, -$ or $(A \sim B).Wx, -$ and Y stand for $C.Wy, - \xrightarrow{\text{rf}} D.Ry, -$ or $(C \sim D).Wy, -$ then the following holds:*

$\neg(X \xrightarrow{\text{sync}} B.Ry, - \xrightarrow{\text{fr}} C.Wy, - \wedge Y \xrightarrow{\text{sync}} D.Rx, - \xrightarrow{\text{fr}} A.Wx, -)$

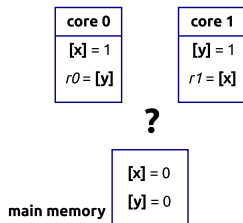
```
int steal(Deque *q) {
    size_t t = load_explicit(&q->top, acquire);
    thread_fence(seq_cst);
    size_t b = load_explicit(&q->bottom, acquire);
    int x = EMPTY;
    if (t < b) {
        /* Non-empty queue. */
        Array *a = load_explicit(&q->array, read);
        x = load_explicit(&a->buffer[t % a->size], read);
        if (!compare_exchange_strong_explicit(
            /* Failed race. */
            return ABORT;
        }
    }
    return x;
}
```

normalized push/take throughput

b=3, d=15 on ARM (Tegra 3) 2 threads

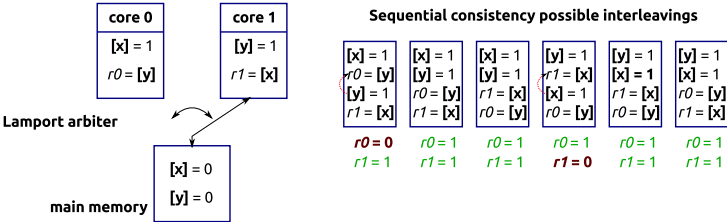


Memory Consistency



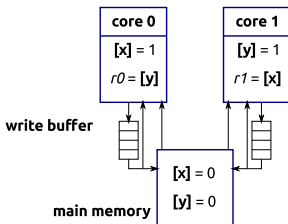
Memory Consistency

- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
 - ▶ Will necessarily read $r0 = 1$ or $r1 = 1$



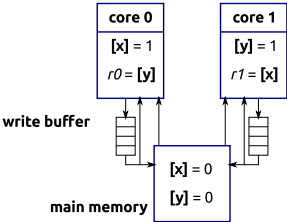
Memory Consistency

- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
- ▶ Total Store Order (x86): write buffer delays visibility of stores from other processors



Memory Consistency

- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
- ▶ Total Store Order (x86): write buffer delays visibility of stores from other processors



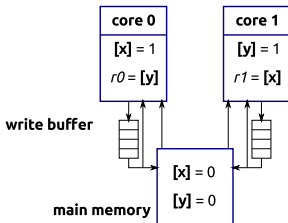
Sequential consistency possible interleavings

| | | | | | |
|--|--|--|--|--|--|
| <code>[x] = 1</code> <code>r0 = [y]</code> <code>[y] = 1</code> <code>r1 = [x]</code> | <code>[x] = 1</code> <code>[y] = 1</code> <code>r0 = [y]</code> <code>r1 = [x]</code> | <code>[x] = 1</code> <code>[y] = 1</code> <code>r1 = [x]</code> <code>r0 = [y]</code> | <code>[y] = 1</code> <code>r1 = [x]</code> <code>[x] = 1</code> <code>r0 = [y]</code> | <code>[y] = 1</code> <code>[x] = 1</code> <code>r1 = [x]</code> <code>r0 = [y]</code> | <code>[y] = 1</code> <code>[x] = 1</code> <code>r0 = [y]</code> <code>r1 = [x]</code> |
| <code>r0 = 0</code> <code>r1 = 1</code> | <code>r0 = 1</code> <code>r1 = 1</code> | <code>r0 = 1</code> <code>r1 = 1</code> | <code>r0 = 1</code> <code>r1 = 0</code> | <code>r0 = 1</code> <code>r1 = 1</code> | <code>r0 = 1</code> <code>r1 = 1</code> |

Additional TSO "perceived" interleavings... all permutations

Memory Consistency

- Sequential consistency: behavior equivalent to serial interleaving of accesses.
- Total Store Order (x86): write buffer delays visibility of stores from other processors



Sequential consistency possible interleavings

| | | | | | |
|--|--|--|--|--|--|
| <code>[x] = 1</code> <code>r0 = [y]</code> <code>[y] = 1</code> <code>r1 = [x]</code> | <code>[x] = 1</code> <code>[y] = 1</code> <code>r0 = [y]</code> <code>r1 = [x]</code> | <code>[x] = 1</code> <code>[y] = 1</code> <code>r1 = [x]</code> <code>r0 = [y]</code> | <code>[y] = 1</code> <code>r1 = [x]</code> <code>[x] = 1</code> <code>r0 = [y]</code> | <code>[y] = 1</code> <code>[x] = 1</code> <code>r1 = [x]</code> <code>r0 = [y]</code> | <code>[y] = 1</code> <code>[x] = 1</code> <code>r0 = [y]</code> <code>r1 = [x]</code> |
| <code>r0 = 0</code> <code>r1 = 1</code> | <code>r0 = 1</code> <code>r1 = 1</code> | <code>r0 = 1</code> <code>r1 = 1</code> | <code>r0 = 1</code> <code>r1 = 0</code> | <code>r0 = 1</code> <code>r1 = 1</code> | <code>r0 = 1</code> <code>r1 = 1</code> |

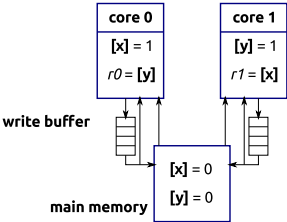
Additional TSO "perceived" interleavings... all permutations

```

r0 = [y]
r1 = [x]
[x] = 1
[y] = 1
r0 = 0
r1 = 0
    
```

Memory Consistency

- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
- ▶ Total Store Order (x86): write buffer delays visibility of stores from other processors
 - ▶ Can read **r0 = 0** and **r1 = 0**



Sequential consistency possible interleavings

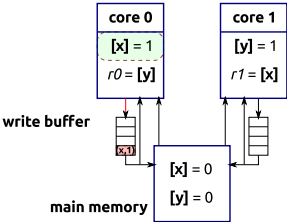
| | | | | | |
|--|--|--|--|--|--|
| $[x] = 1$ $r0 = [y]$ $[y] = 1$ $r1 = [x]$ | $[x] = 1$ $[y] = 1$ $r0 = [y]$ $r1 = [x]$ | $[x] = 1$ $[y] = 1$ $r1 = [x]$ $r0 = [y]$ | $[y] = 1$ $r1 = [x]$ $[x] = 1$ $r0 = [y]$ | $[y] = 1$ $[x] = 1$ $r1 = [x]$ $r0 = [y]$ | $[y] = 1$ $[x] = 1$ $r0 = [y]$ $r1 = [x]$ |
| r0 = 0 r1 = 1 | r0 = 1 r1 = 1 | r0 = 1 r1 = 1 | r0 = 1 r1 = 0 | r0 = 1 r1 = 1 | r0 = 1 r1 = 1 |

Additional TSO "perceived" interleavings... all permutations

| | |
|--|--|
| $r0 = [y]$ $r1 = [x]$ $[x] = 1$ $[y] = 1$ | $[x] = 1$ $[y] = 1$ $r0 = [y]$ $r1 = [x]$ |
| r0 = 0 r1 = 0 | |

Memory Consistency

- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
- ▶ Total Store Order (x86): write buffer delays visibility of stores from other processors
 - ▶ Can read **r0 = 0** and **r1 = 0**



Sequential consistency possible interleavings

| | | | | | |
|---|---|---|---|---|---|
| <div><div>[x] = 1</div><div>r0 = [y]</div><div>[y] = 1</div><div>r1 = [x]</div></div> <div>r0 = 0</div> <div>r1 = 1</div> | <div><div>[x] = 1</div><div>[y] = 1</div><div>r0 = [y]</div><div>r1 = [x]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> | <div><div>[x] = 1</div><div>[y] = 1</div><div>r1 = [x]</div><div>r0 = [y]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> | <div><div>[y] = 1</div><div>r1 = [x]</div><div>[x] = 1</div><div>r0 = [y]</div></div> <div>r0 = 1</div> <div>r1 = 0</div> | <div><div>[y] = 1</div><div>[x] = 1</div><div>r1 = [x]</div><div>r0 = [y]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> | <div><div>[y] = 1</div><div>[x] = 1</div><div>r0 = [y]</div><div>r1 = [x]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> |
|---|---|---|---|---|---|

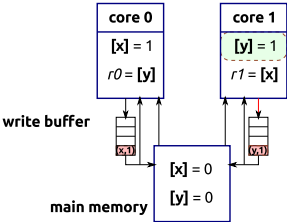
Additional TSO "perceived" interleavings... all permutations

| |
|---|
| <div><div>r0 = [y]</div><div>r1 = [x]</div><div>[x] = 1</div><div>[y] = 1</div></div> <div>r0 = 0</div> <div>r1 = 0</div> |
|---|

| |
|---|
| <div><div>[x] = 1</div><div>[y] = 1</div><div>r0 = [y]</div><div>r1 = [x]</div></div> <div>goes to write buffer</div> |
|---|

Memory Consistency

- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
- ▶ Total Store Order (x86): write buffer delays visibility of stores from other processors
 - ▶ Can read $r0 = 0$ and $r1 = 0$



Sequential consistency possible interleavings

| | | | | | |
|--|--|--|--|--|--|
| $[x] = 1$ $r0 = [y]$ $[y] = 1$ $r1 = [x]$ | $[x] = 1$ $[y] = 1$ $r0 = [y]$ $r1 = [x]$ | $[x] = 1$ $[y] = 1$ $r1 = [x]$ $r0 = [y]$ | $[y] = 1$ $r1 = [x]$ $[x] = 1$ $r0 = [y]$ | $[y] = 1$ $[x] = 1$ $r1 = [x]$ $r0 = [y]$ | $[y] = 1$ $[x] = 1$ $r0 = [y]$ $r1 = [x]$ |
| $r0 = 0$ $r1 = 1$ | $r0 = 1$ $r1 = 1$ | $r0 = 1$ $r1 = 1$ | $r0 = 1$ $r1 = 0$ | $r0 = 1$ $r1 = 1$ | $r0 = 1$ $r1 = 1$ |

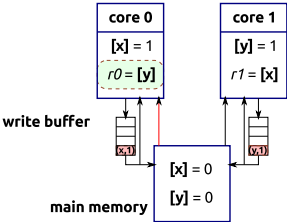
Additional TSO "perceived" interleavings... all permutations

| |
|--|
| $r0 = [y]$ $r1 = [x]$ $[x] = 1$ $[y] = 1$ |
| $r0 = 0$ $r1 = 0$ |

| | |
|--|--|
| $[x] = 1$ $[y] = 1$ $r0 = [y]$ $r1 = [x]$ | goes to write buffer goes to write buffer |
|--|--|

Memory Consistency

- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
- ▶ Total Store Order (x86): write buffer delays visibility of stores from other processors
 - ▶ Can read **r0 = 0** and **r1 = 0**



Sequential consistency possible interleavings

| | | | | | |
|--|--|--|--|--|--|
| [x] = 1 r0 = [y] [y] = 1 r1 = [x] | [x] = 1 [y] = 1 r0 = [y] r1 = [x] | [x] = 1 [y] = 1 r1 = [x] r0 = [y] | [y] = 1 r1 = [x] [x] = 1 r0 = [y] | [y] = 1 [x] = 1 r1 = [x] r0 = [y] | [y] = 1 [x] = 1 r0 = [y] r1 = [x] |
| r0 = 0 r1 = 1 | r0 = 1 r1 = 1 | r0 = 1 r1 = 1 | r0 = 1 r1 = 0 | r0 = 1 r1 = 1 | r0 = 1 r1 = 1 |

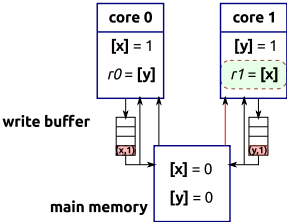
Additional TSO "perceived" interleavings... all permutations

| |
|--|
| r0 = [y] r1 = [x] [x] = 1 [y] = 1 |
| r0 = 0 r1 = 0 |

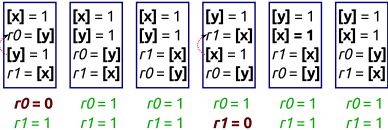
| | |
|--|---|
| [x] = 1 [y] = 1 r0 = [y] r1 = [x] | goes to write buffer goes to write buffer reads 0 from memory |
|--|---|

Memory Consistency

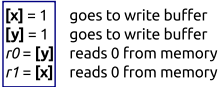
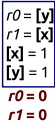
- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
- ▶ Total Store Order (x86): write buffer delays visibility of stores from other processors
 - ▶ Can read **r0 = 0** and **r1 = 0**



Sequential consistency possible interleavings

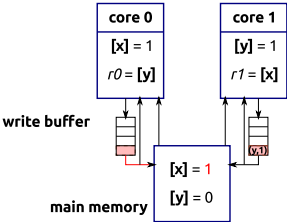


Additional TSO "perceived" interleavings... all permutations



Memory Consistency

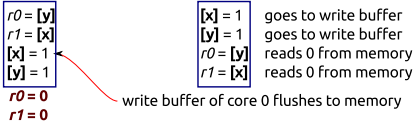
- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
- ▶ Total Store Order (x86): write buffer delays visibility of stores from other processors
 - ▶ Can read **r0 = 0** and **r1 = 0**



Sequential consistency possible interleavings

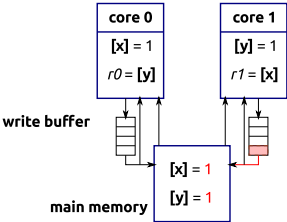
| | | | | | |
|---|---|---|---|---|---|
| <div><div>[x] = 1</div><div>r0 = [y]</div><div>[y] = 1</div><div>r1 = [x]</div></div> <div>r0 = 0</div> <div>r1 = 1</div> | <div><div>[x] = 1</div><div>[y] = 1</div><div>r0 = [y]</div><div>r1 = [x]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> | <div><div>[x] = 1</div><div>[y] = 1</div><div>r1 = [x]</div><div>r0 = [y]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> | <div><div>[y] = 1</div><div>r1 = [x]</div><div>[x] = 1</div><div>r0 = [y]</div></div> <div>r0 = 1</div> <div>r1 = 0</div> | <div><div>[y] = 1</div><div>[x] = 1</div><div>r1 = [x]</div><div>r0 = [y]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> | <div><div>[y] = 1</div><div>[x] = 1</div><div>r0 = [y]</div><div>r1 = [x]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> |
|---|---|---|---|---|---|

Additional TSO "perceived" interleavings... all permutations



Memory Consistency

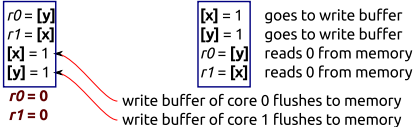
- ▶ Sequential consistency: behavior equivalent to serial interleaving of accesses.
- ▶ Total Store Order (x86): write buffer delays visibility of stores from other processors
 - ▶ Can read **r0 = 0** and **r1 = 0**



Sequential consistency possible interleavings

| | | | | | |
|---|---|---|---|---|---|
| <div><div>[x] = 1</div><div>r0 = [y]</div><div>[y] = 1</div><div>r1 = [x]</div></div> <div>r0 = 0</div> <div>r1 = 1</div> | <div><div>[x] = 1</div><div>[y] = 1</div><div>r0 = [y]</div><div>r1 = [x]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> | <div><div>[x] = 1</div><div>[y] = 1</div><div>r1 = [x]</div><div>r0 = [y]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> | <div><div>[y] = 1</div><div>r1 = [x]</div><div>[x] = 1</div><div>r0 = [y]</div></div> <div>r0 = 1</div> <div>r1 = 0</div> | <div><div>[y] = 1</div><div>[x] = 1</div><div>r1 = [x]</div><div>r0 = [y]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> | <div><div>[y] = 1</div><div>[x] = 1</div><div>r0 = [y]</div><div>r1 = [x]</div></div> <div>r0 = 1</div> <div>r1 = 1</div> |
|---|---|---|---|---|---|

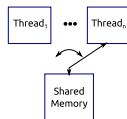
Additional TSO "perceived" interleavings... all permutations



Memory Consistency and Relaxation

Sequential consistency – interleaving

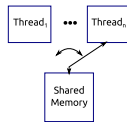
- + total order of all memory operations
- no longer a valid hypothesis: performance bottleneck



Memory Consistency and Relaxation

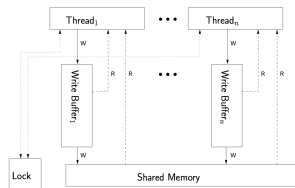
Sequential consistency – interleaving

- + total order of all memory operations
- no longer a valid hypothesis: performance bottleneck



Total store order (x86)

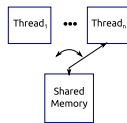
- + total store order: reason about global invariants
- does not scale well



Memory Consistency and Relaxation

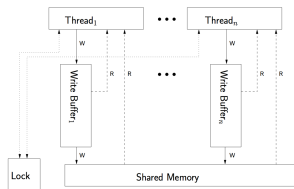
Sequential consistency – interleaving

- + total order of all memory operations
- no longer a valid hypothesis: performance bottleneck



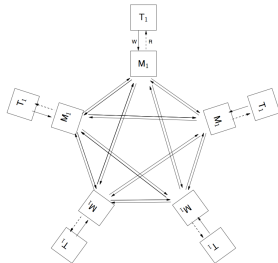
Total store order (x86)

- + total store order: reason about global invariants
- does not scale well



POWER, ARM, C/C++11, RC, WC, DAG, PC, LC...

- partial order of memory operations
- processors may have conflicting views of memory
- + **better scalability at a lower power price tag**



Example: First Provably Correct Work-Stealing Scheduler

A relaxed lock-free work-stealing algorithm.

Two implementations: C11 and ARM inline assembly.

Based on state-of-the-art sequentially consistent algorithm (Chase and Lev, 2005).

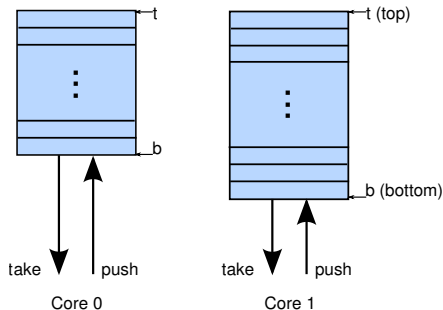
Proven for the POWER/ARM relaxed memory model (axiomatic POWER model by Mador-Haim et al., 2012). POWER and ARM have the same memory model.

[Lê et al., PPOPP 2013]

Work stealing (1)

Each core has an associated double-ended queue (deque)

- ▶ New tasks are *pushed* to the core's deque
- ▶ When a task finishes, another is *taken* from the core's deque

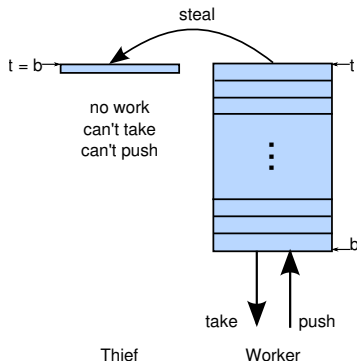


Work stealing (2)

If a task finishes, and the queue is empty, the core *steals* from another deque

Cores alternate between the *worker* and *thief* roles

We focus on the study of a single deque (hereafter, *the* deque)



Expected properties

Legal reads Only tasks pushed are taken or stolen

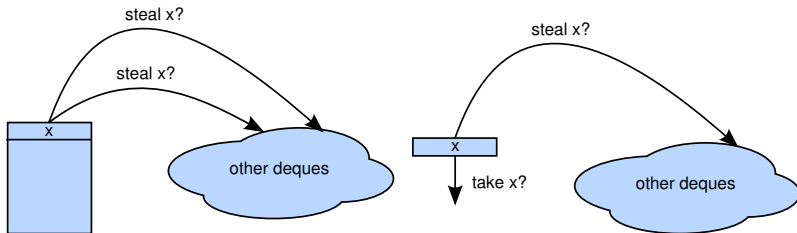
Uniqueness A task pushed into the deque cannot be taken or stolen more than once

Existence Tasks are not lost because of concurrency

Progress Given a finite number of tasks, the deque is eventually emptied

Concurrency problems

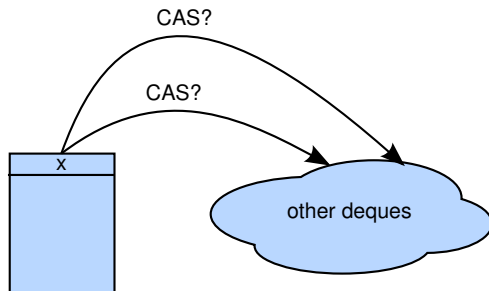
Two thieves attempt to steal the same task
Worker and thief contend for the same task



Steal/steal resolution in SC

Proven by Chase and Lev for sequential consistency (SC)

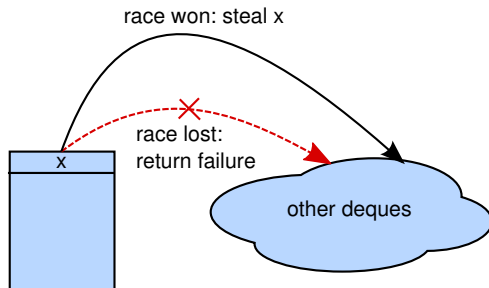
Steal/steal resolution with compare-and-swap (CAS)



Steal/steal resolution in SC

Proven by Chase and Lev for sequential consistency (SC)

Steal/steal resolution with compare-and-swap (CAS)

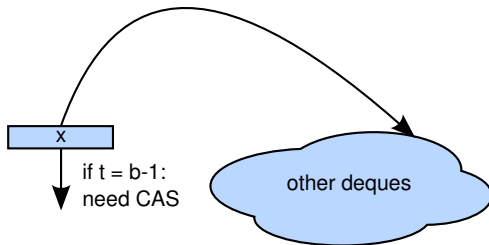


Take/steal resolution in SC

Proven by Chase and Lev for sequential consistency (SC)

Potential take/steal races only if one task left; detected through comparison of indices (t and b)

If one task: worker “self steals” from its own deque with a CAS

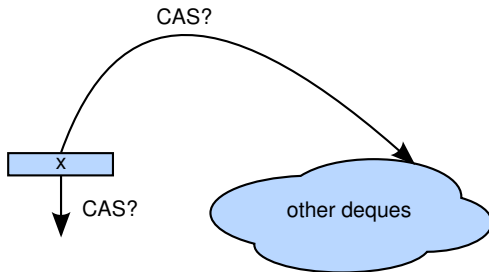


Take/steal resolution in SC

Proven by Chase and Lev for sequential consistency (SC)

Potential take/steal races only if one task left; detected through comparison of indices (t and b)

If one task: worker “self steals” from its own deque with a CAS

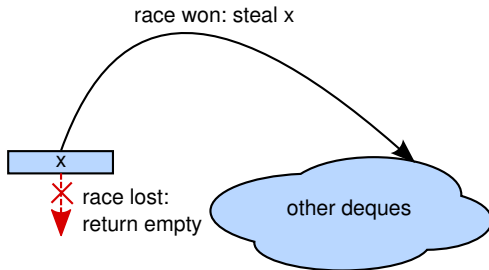


Take/steal resolution in SC

Proven by Chase and Lev for sequential consistency (SC)

Potential take/steal races only if one task left; detected through comparison of indices (t and b)

If one task: worker “self steals” from its own deque with a CAS



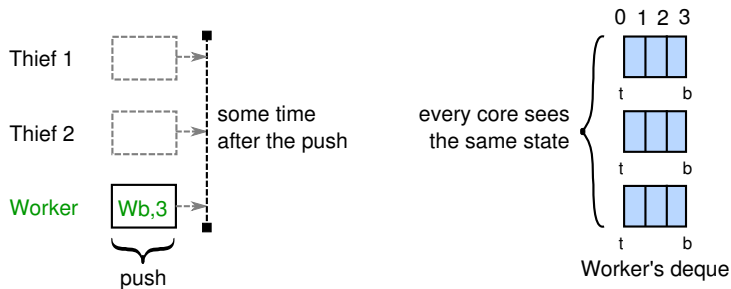
Summary of operations in SC

Table below shows how operations affect indices

| | | |
|-----------------------|-----|-------------------|
| push | ++b | |
| take (deque size > 1) | --b | |
| take (deque size = 1) | ++t | if CAS successful |
| steal | ++t | if CAS successful |

A relaxed concurrency problem (1)

1. Let's assume the deque starts with three tasks

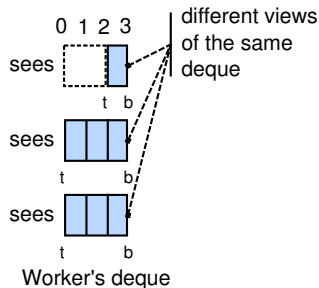
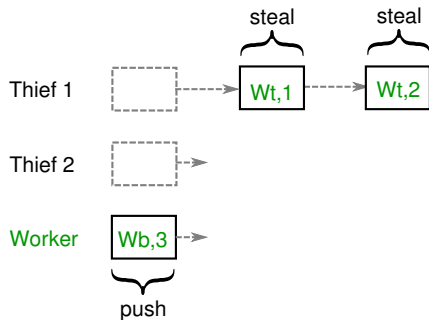


$W_{x,v}$ denotes a write of the value v to the variable x

Blue bins represent the views of Worker's deque from each core

A relaxed concurrency problem (1)

1. Let's assume the deque starts with three tasks
2. Thief 1 steals two tasks; the others don't know yet

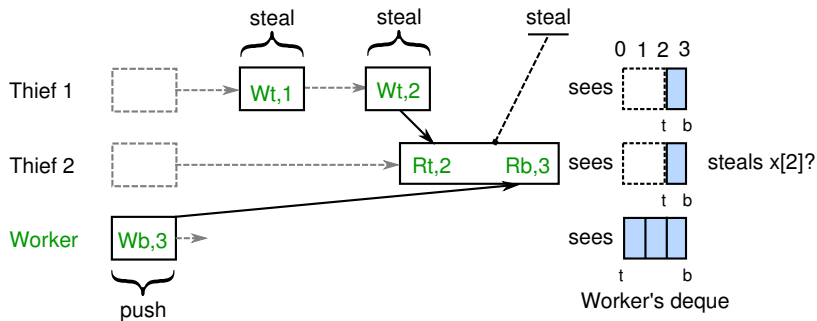


$W_{x,v}$ denotes a write of the value v to the variable x

Blue bins represent the views of Worker's deque from each core

A relaxed concurrency problem (1)

1. Let's assume the deque starts with three tasks
2. Thief 1 steals two tasks; the others don't know yet
3. Thief 2 sees the two steals and attempts to steal $x[2]$

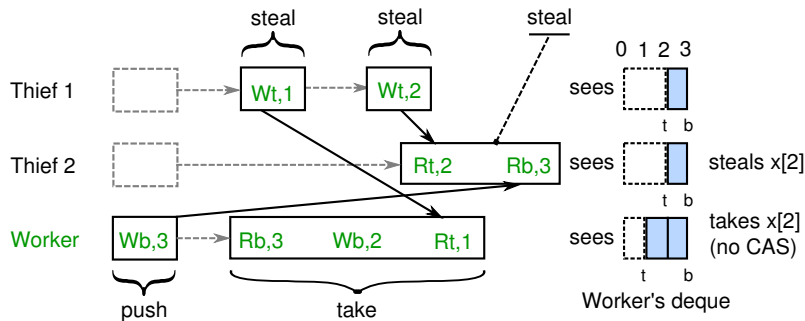


$R_{x,v}$ denotes a read of the value v from the variable x

Black arrows represent communications

A relaxed concurrency problem (1)

1. Let's assume the deque starts with three tasks
2. Thief 1 steals two tasks; the others don't know yet
3. Thief 2 sees the two steals and attempts to steal $x[2]$
4. Worker sees *only the first steal* and takes $x[2]$



$R_{x,v}$ denotes a read of the value v from the variable x

Black arrows represent communications

A relaxed concurrency problem (2)

Why does it happen?

Different views of the indices in each core

The state of the deque is *relative* to the core that observes it
(\neq SC where the state of the deque is the same for all)

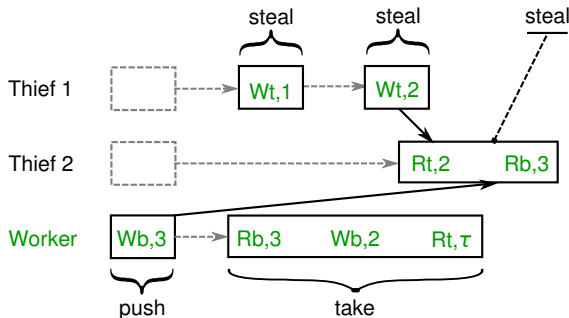
The worker does not realize that it is taking the last element
(*from its viewpoint*, $t \neq b-1$)

Hence no CAS to resolve the conflict

Sequentially consistent ideas (1)

Why is it different in SC?

In SC, all memory events are totally ordered
Transitively so

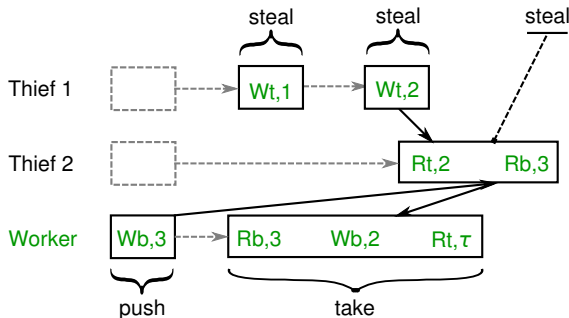


Operations on the same variable are ordered by *coherency*

Sequentially consistent ideas (1)

Why is it different in SC?

In SC, all memory events are totally ordered
Transitively so

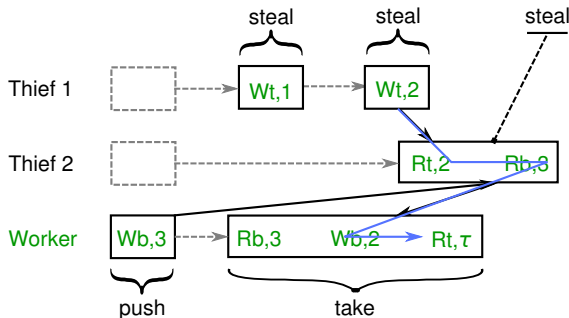


Operations on the same variable are ordered by *coherency*
 $Rb,3$ in Thief 2 occurs before $Wb,2$; otherwise, it would read 2

Sequentially consistent ideas (1)

Why is it different in SC?

In SC, all memory events are totally ordered
Transitively so



Transitively, Worker has already seen $Wt,2$ overwrite $Wt,1$
Hence, Rt,τ cannot read 1

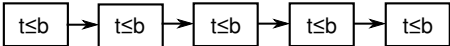
Sequentially consistent ideas (2)

In SC, a test in one core gives information about *other* cores: the state is the same for all at any given time

Can test for special cases (e.g., single-task queue $t = b-1$)

Can test for invariants (e.g., well-formed queue $t \leq b$)

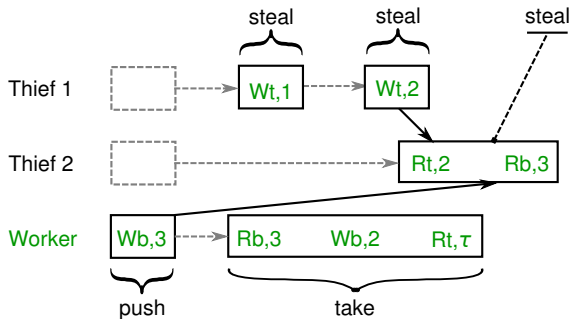
Can use induction on these invariants

SC (all cores)  implies
no take/steal conflict

Does not hold in a relaxed memory model.

POWER/ARM barriers and cumulativity

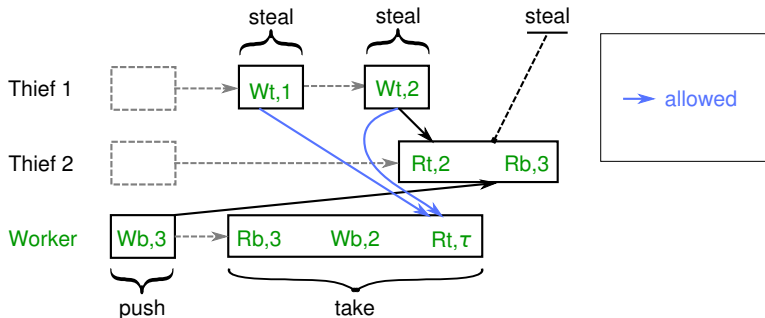
Recall the situation in the previous example



$Rb,3$ in Thief 2 reads from $Wb,3$ in Worker
 $Rt,2$ in Thief 2 reads from $Wt,2$ in Thief 1

POWER/ARM barriers and cumulativity

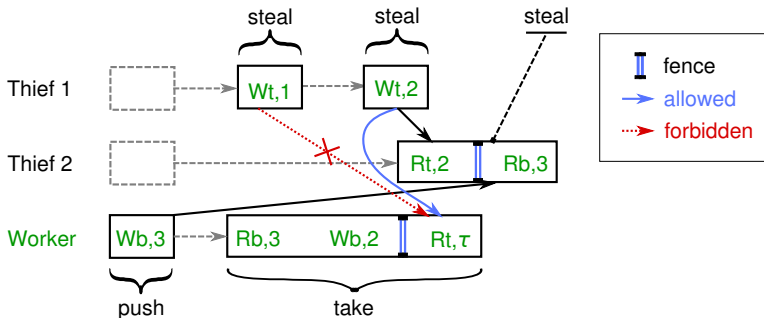
Recall the situation in the previous example



What values of t can we read in Worker?
With relaxed semantics, τ can be either 1 or 2

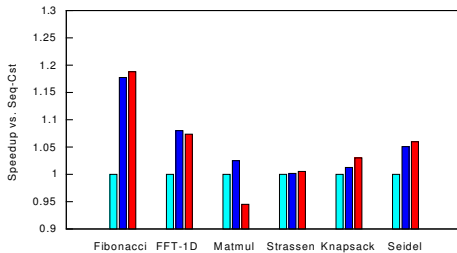
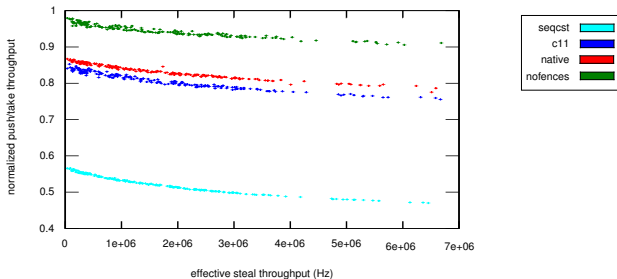
POWER/ARM barriers and cumulativity

A **sync** memory barrier instruction guarantees that *all the writes that have been observed* by the core issuing the barrier instruction are propagated to all the other cores before the core can continue



With memory fences, however, Rt,τ cannot read 1
The two fences stall each other until one has finished

Experimental results (on Tegra 3, 4-core ARM)



More on Weak Memory Models

- ▶ First application of a formal relaxed memory model to the manual proof of a moderately complex real-world algorithm
- ▶ Two efficient implementations: C11 and inline assembly; tested on ARM, POWER and x86

Recent and ongoing work:

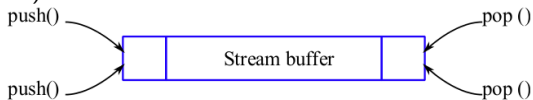
- ▶ Proof of a fast SPSC FIFO ring buffer in C11
- ▶ Index-based MPMC FIFO: “Erbium”
- ▶ Combined FIFO and scheduling with lightweight suspension and wake-up

Perspectives:

- ▶ Global address space models for software caches/DSMs
- ▶ Application to manycore processors with on-chip distributed memory
- ▶ E.g., Kalray MPPA

FIFO Ring Buffer: Lock-Free Implementations

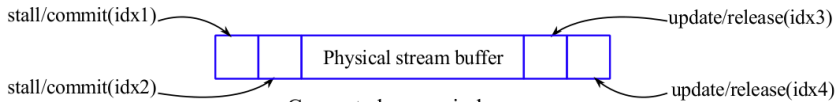
- ▶ SPSC: only needs release/acquire (free on x86)
Lamport's FIFO, Lee et al.'s MCRB
- ▶ SPMC/MPSC: needs an atomic operation
- ▶ MPMC: often implemented using SPMC + MPSC (useful for data parallelism)



Consensus required

FastFlow: <http://mc-fastflow.sourceforge.net>

- ▶ Indexed-based MPMC: only needs release/acquire (free on x86)

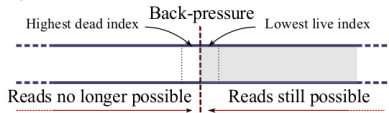
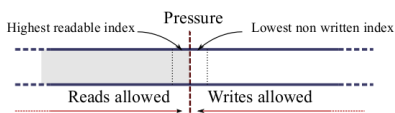


Computed access indexes

Fast Implementations of SPSC Streams

Ring buffer with two pointers/indices

- Issue: detect full buffers with back-pressure



- Simplest and efficient solution for powers of two: absolute indices
Need arithmetic care to deal with wrap-arounds and (32bit) overflows
- Cache-aware optimization
 - Manual/software caching of the last index

Lamport's Lock-Free FIFO Queue in C11

```
atomic_size_t front;
atomic_size_t back;
T data[SIZE];

void init(void) {
    atomic_init(&front, 0);
    atomic_init(&back, 0);
}

bool push(T elem) {
    size_t b, f;
    b = atomic_load(&back, seq_cst);
    f = atomic_load(&front, seq_cst);
    if ((b + 1) % SIZE == f)
        return false;
    data[b] = elem;
    atomic_store(&back, (b+1)%SIZE, seq_cst);
    return true;
}

bool pop(T *elem) {
    size_t b, f;
    b = atomic_load(&back, seq_cst);
    f = atomic_load(&front, seq_cst);
    if (b == f)
        return false;
    *elem = data[b];
    atomic_store(&front, (f+1)%SIZE, seq_cst);
    return true;
}
```

Optimization: WeakRB

```
atomic_size_t front;
size_t pfront;
atomic_size_t back;
size_t cback;

_Static_assert(SIZE_MAX % SIZE == 0,
    "SIZE div SIZE_MAX");
T data[SIZE];

void init(void) {
    atomic_init(&front, 0);
    atomic_init(&back, 0);
}

bool push(const T *elems, size_t n) {
    size_t b, f;
    b = atomic_load(&back, relaxed);
    if (pfront + SIZE - b < n) {
        pfront = atomic_load(&front, acquire);
        if (pfront + SIZE - b < n)
            return false;
    }
    for (size_t i = 0; i < n; i++)
        data[(b+i) % SIZE] = elems[i];
    atomic_store(&back, b + n, release);
    return true;
}

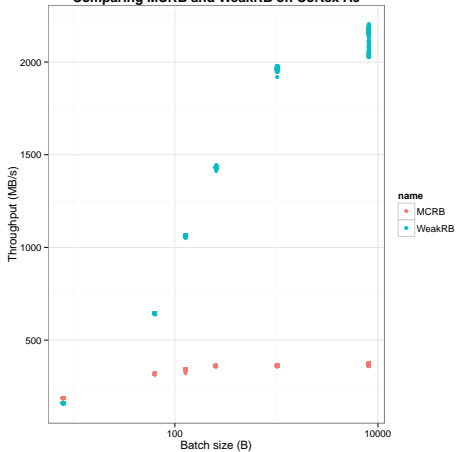
bool pop(T *elems, size_t n) {
    size_t b, f;
    f = atomic_load(&front, relaxed);
    if (cback - f < n) {
        cback = atomic_load(&back, acquire);
        if (cback - f < n)
            return false;
    }
    for (size_t i = 0; i < n; i++)
        elems[i] = data[(f+i) % SIZE];
    atomic_store(&front, f + n, release);
    return true;
}
```

Evaluation Platforms

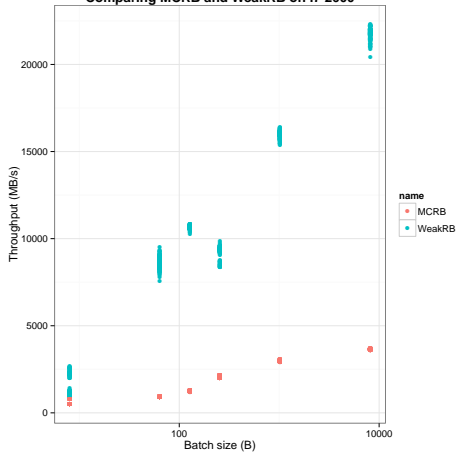
| Machine | Cortex A9 | Core i7 |
|-----------------|------------------|----------------|
| Manufacturer | Samsung | Intel |
| ISA | ARMv7 | x86_64 |
| Number of cores | 4 | 4 (8 logical) |
| Clock frequency | 1.3 GHz | 3.4 GHz |
| Best throughput | 2.2 ,GB/s | 22 GB/s |

Performance Results

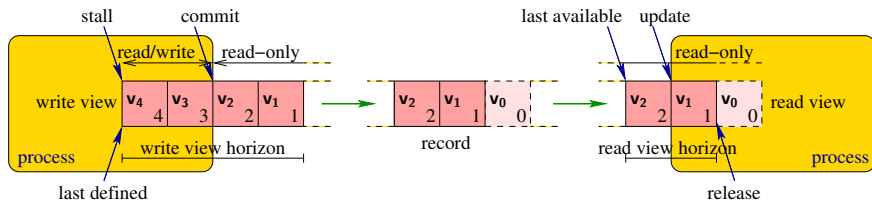
Comparing MCRB and WeakRB on Cortex A9



Comparing MCRB and WeakRB on i7 2600



Erbium: Fast Index-Based MPMC Streams

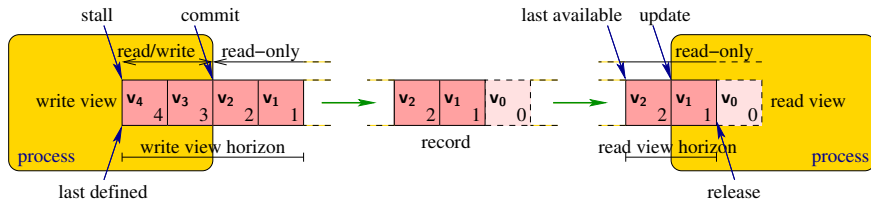


Stream synchronization primitives

- ▶ `commit()`/`update()`: pressure
- ▶ `release()`/`stall()`: back-pressure
- ▶ `receive()`: prefetch on a sliding window
- ▶ Deterministic initialization protocol and garbage collection of dead sliding windows

[Miranda et al., CASES 2010]

Erbium: Fast Index-Based MPMC Streams



Erbium: Lightweight runtime

Ring buffer with two sets of indices, for the producer and consumer sides

- ▶ Cache the minimum of the producer (resp. consumer) indices
- ▶ Cache-aware optimization
 - ▶ Alignment of shared indices to avoid false sharing
 - ▶ Manual/software caching of the last index
 - ▶ Monotonicity tolerates races on minimum index computation
- ▶ Lock-free, consensus-free implementation
 - ▶ No HW atomic instruction, no fence on x86
- ▶ ≈ 10 cycles per streaming communication cycle

5. Wrap-Up

Stream Processing?

Driving Force: Correct Concurrency by Construction

The Hammer: Language Design

The Anvil: Runtime System Design

Wrap-Up

Streaming Data Flow

Kahn networks

- ▶ *Deterministic parallelism*
- ▶ Abstract description and/or distributed implementation
- ▶ Mathematical (ideal) model: stream equations, differential equations, automata
- ▶ The *compiler* plays a central role: restrict to executable specifications, optimization, exposing task parallelism
- ▶ Inspiration for a scalable and efficient *runtime system* for dynamic dependence resolution and task scheduling

Synchronous Kahn parallelism

- ▶ As a programming model for dealing with time and parallelism
- ▶ And as an *internal representation* in optimizing compilers
- ▶ And for *code generation* down to sequential and parallel code
- ▶ Full *traceability* between the source and target code

OpenStream Impact and Dissemination

1. Used in 3 collaborative research projects
2. Used in 3 ongoing *PhD theses* (ÉNS, INRIA and UPMC)
3. Used in a *parallel programming course* project at Politecnico di Torino
4. Ongoing work to port OpenStream on Kalray MPPA
5. Used for developing and evaluating communication channel synchronization algorithms by Preud'Homme et al. in "*An Improvement of OpenMP Pipeline Parallelism with the BatchQueue Algorithm*," ICPADS 2012
6. Proven work-stealing implementation discussed within OpenJDK and help debug SAP's distributed GC
7. OpenStream featured in the *HPCwire* magazine
www.hpcwire.com/hpcwire/2013-01-24/the_week_in_hpc_research.html?page=3
8. Source code publicly available on *Sourceforge*
<http://sourceforge.net/p/open-stream/>
9. Project website: <http://openstream.info>