FMCO 2011 - Torino, Italy - October 2nd, 2011

SEVENTH FRAMEWORK PROGRAMME

FastFlow: Performance and Productivity in the Multicore Era

Marco Aldinucci - ParaPhrase project
Computer Science Dept. - University of Torino (Turin) - Italy

a joint work with

Massimo Torquati and Marco Danelutto
Computer Science Dept. - University of Pisa - Italy

Massimiliano Meneghin
IBM Technology Campus, Dublin Software Lab, Ireland

Peter Kilpatrick
Queen's University Belfast, U.K.

PARAPHRASE

Outline

- Issues for multicore era
  - Mostly focusing on shared memory hardware
- High-level patterns & FastFlow
- The challenge for Formal methods.
  - Questions and thoughts in this context (in open order)
  - How our work might benefit or influence on formal method community

ISSUEs for exascale era

UNIVERSITÀ DEGLI STUDI DI TORINO

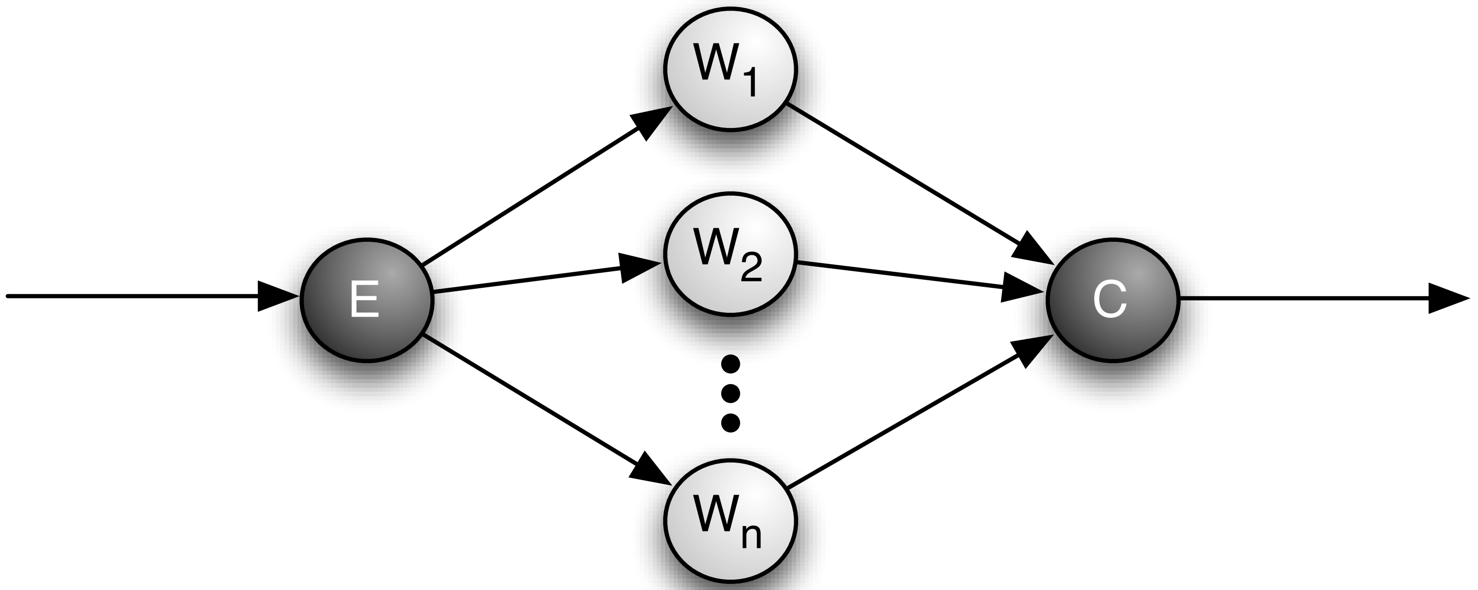
- Increasing scalability requires to decrease concurrency grain
- Programming systems should be designed to support fast data movement and enforce locality
  - It is not about Flops, it is about data movement
- Novel computing models are needed
  - A computer language is not a computing model. A library is not a computing model.

Micro-benchmarks: farm of tasks

Used to implement: parameter sweeping, master-worker, etc.

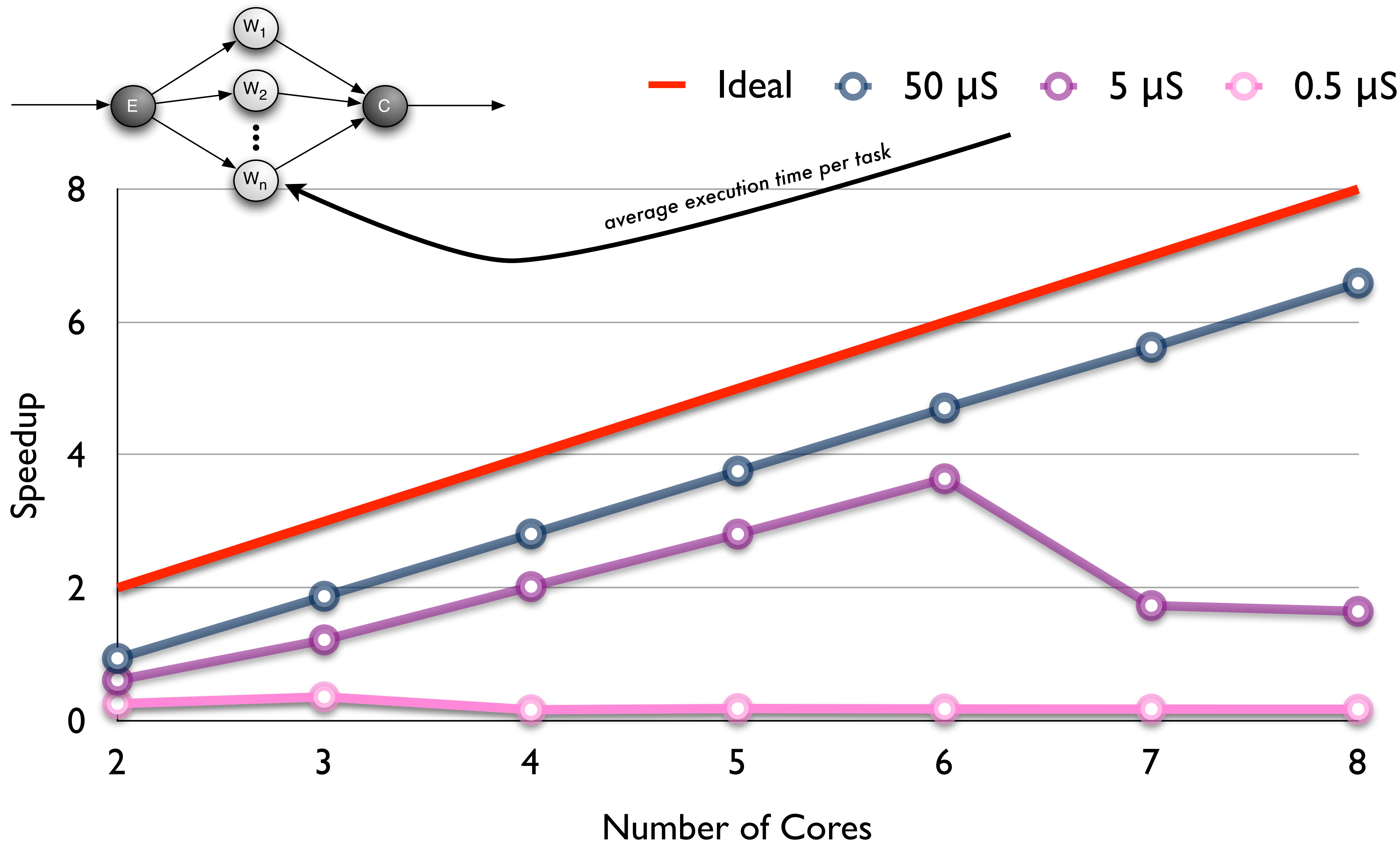
```
void Emitter () {
  for ( i =0; i <streamLen;++i){
    task = create_task ();
    queue=SELECT_WORKER_QUEUE();
    queue ->PUSH(task);
  }
}

void Worker() {
  while (!end_of_stream){
  myqueue ->POP(&task);
  do_work(task) ;
  }
}
```

```
int main () {
  spawn_thread( Emitter ) ;
  for ( i =0; i <nworkers;++i){
    spawn_thread(Worker);
  }
  wait_end () ;
}
```



Task farm with POSIX lock/unlock

UNIVERSITÀ DEGLI STUDI DI TORINO

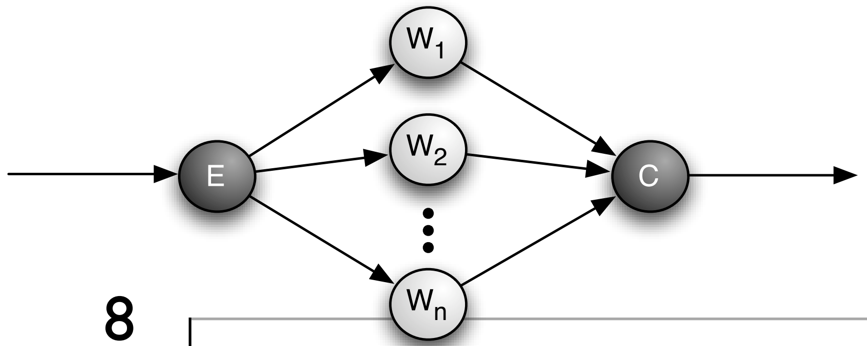


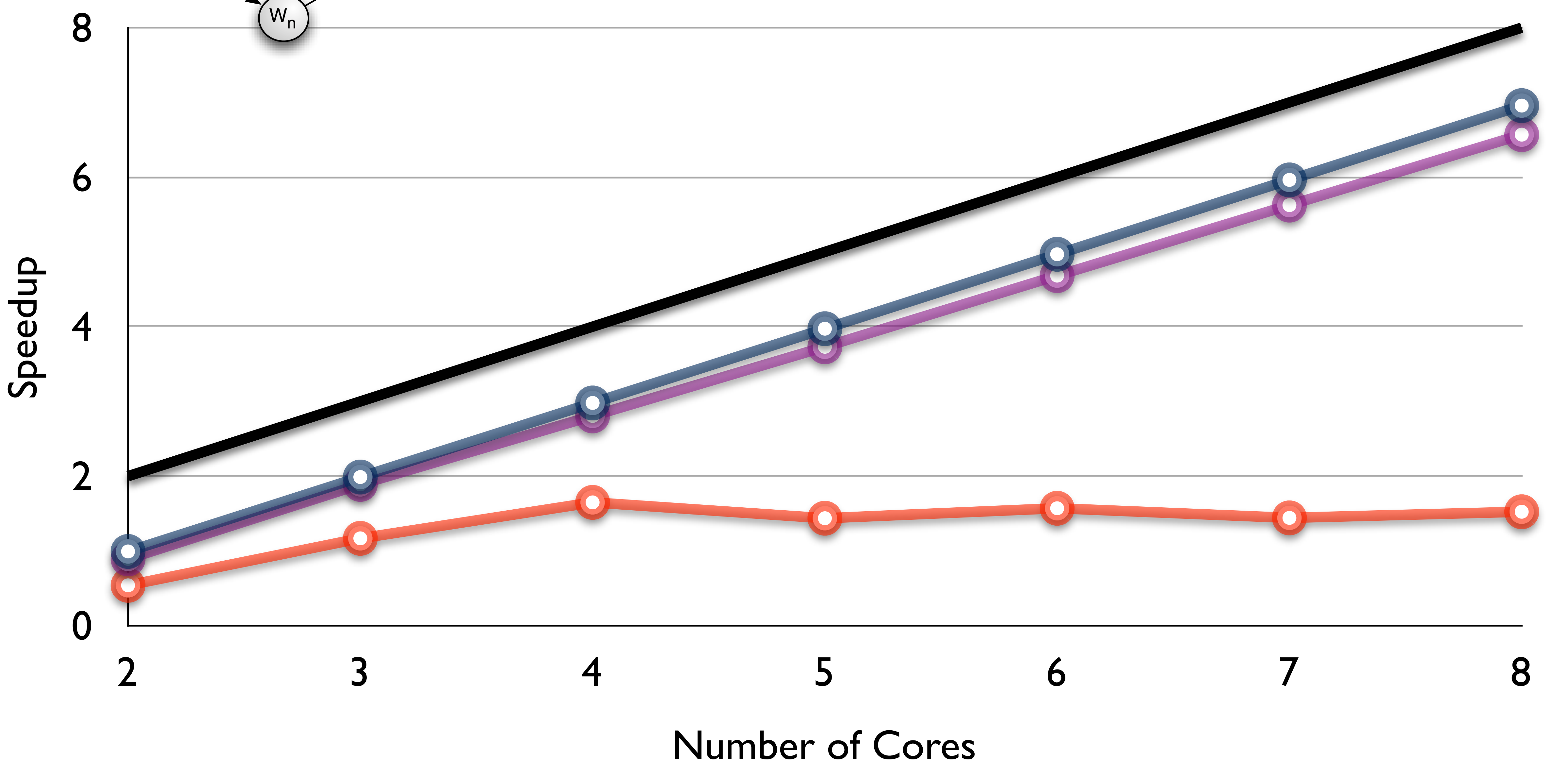
Can we avoid locks?

UNIVERSITÀ DEGLI STUDI DI TORINO

- Under relaxed memory models, using CAS/atomic ops
  - "lock-free" data structures
  - they perform better than lock-based
  - they fence the memory and pay cache coherency reconciliation overhead

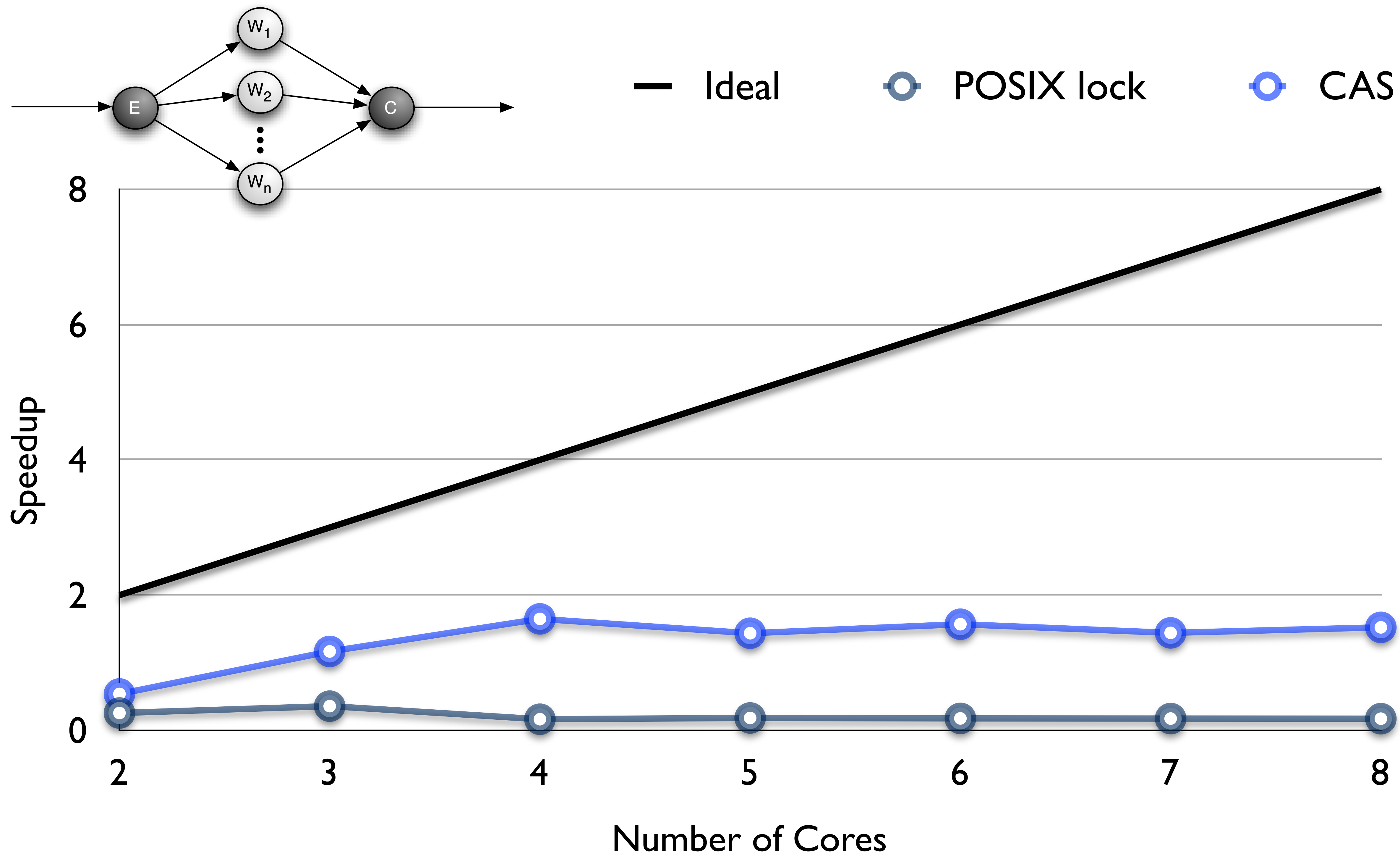
CompareAndSwap queues





Lock vs CAS at fine grain (0.5 μS)

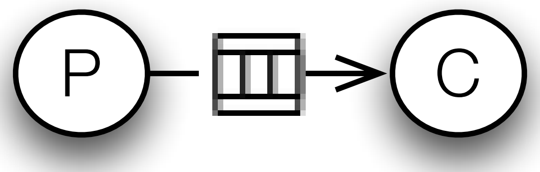
UNIVERSITÀ DEGLI STUDI DI TORINO



Re-starting from the basics

UNIVERSITÀ DEGLI STUDI DI TORINO

- Reducing the problem to the bare bones
  - Producer-Consumer model (streaming)
  - Directly control thread blocking using non-blocking synchronizations
  - Directly design the "data channel"
    - Having clear how data move in the whole memory hierarchy
- Restarting from the FIFO queue



Interaction models: theoretical background

- Low-level synchronisation in the shared memory model
  - Mutual Exclusion (mutex)
    - typically used as basic building block of synchronisations
  - Producer Consumer
- They are not equally demanding
  - Mutual Exclusion is inherently more complex since requires deadlock-freedom
    - require interlocked ops (CAS, ...), that induces memory fences, thus cache invalidation
    - Dekker and Bakery requires Sequential Consistency (++)
  - Producer Consumer is a cooperative (non cyclic) process

FastFlow SPSC queues

UNIVERSITÀ DEGLI STUDI DI TORINO

```
push_nonbocking(data) {
  if (NEXT(head) == tail) {
    return EWOULDBLOCK;
  }
  buffer[head] = data;
  head = NEXT(head);
  return 0;
}

pop_nonblocking(data) {

  if (head == tail) {
    return EWOULDBLOCK;
  }
  data = buffer[tail];
  tail = NEXT(tail);
  return 0;
}
```

Lamport FIFO - 1983

```
push_nonbocking(data) {
  if (NULL != buffer[head]) {
    return EWOULDBLOCK;
  }
  buffer[head] = data;
  head = NEXT(head);
  return 0;
}

pop_nonblocking(data) {
  data = buffer[tail];
  if (NULL == data) {
    return EWOULDBLOCK;
  }
  buffer[tail] = NULL;
  tail = NEXT(tail);
  return 0;
}
```

FastFlow FIFO - derived from P1C1 1997

FastFlow SPSC queues

UNIVERSITÀ DEGLI STUDI DI TORINO

```
push_nonblocking(data) {
  if (NEXT(head) == tail) {
    return EWOULDBLOCK;
  }
  buffer[head] = data;
  head = NEXT(head);
  return 0;
}

pop_nonblocking(data) {

  if (head == tail) {
    return EWOULDBLOCK;
  }
  data = buffer[tail];
  tail = NEXT(tail);
  return 0;
}
```

head and tail are mutually invalidated by producer and consumer
1 cache miss every push and pop (at least)

Lamport FIFO - 1983

```
push_nonblocking(data) {
  if (NULL != buffer[head]) {
    return EWOULDBLOCK;
  }
  buffer[head] = dat
  head = NEXT(head);
  return 0;
}

pop_nonblocking(data)
  data = buffer[tail];
  if (NULL == data) {
    return EWOULDBLOCK
  }
  buffer[tail] = NULL;
  tail = NEXT(tail);
  return 0;
}
```

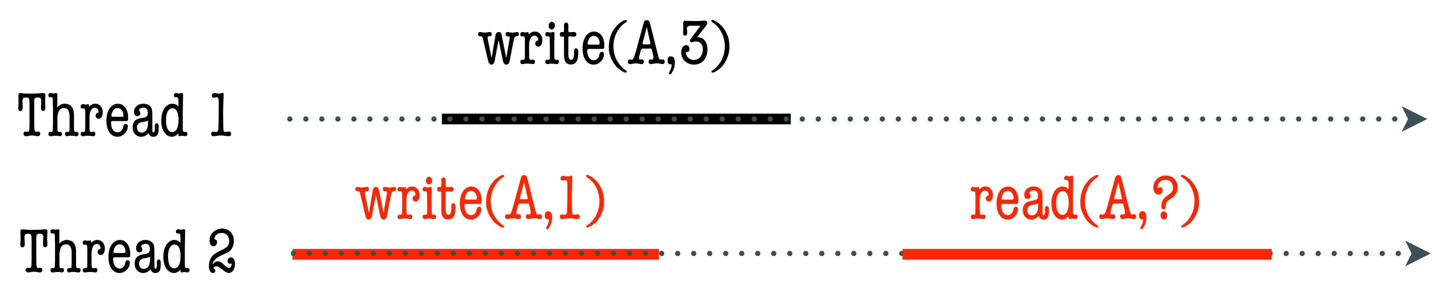
producer read/write head
consumer read/write tail
no misses

excluding “true” deps

extended domain on void *

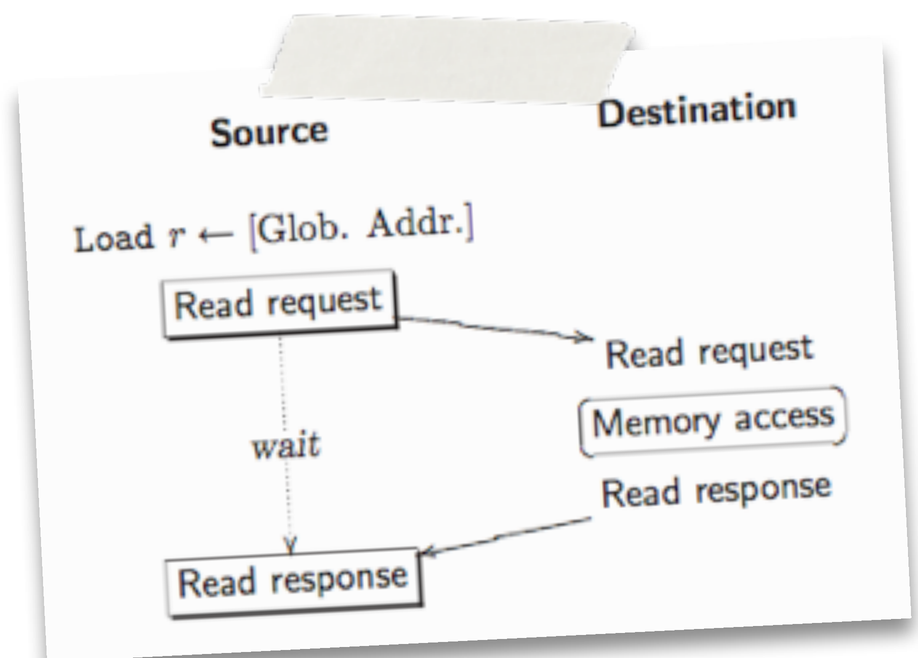
FastFlow FIFO - derived from P1C1 1997

Recall: Two features - two problems

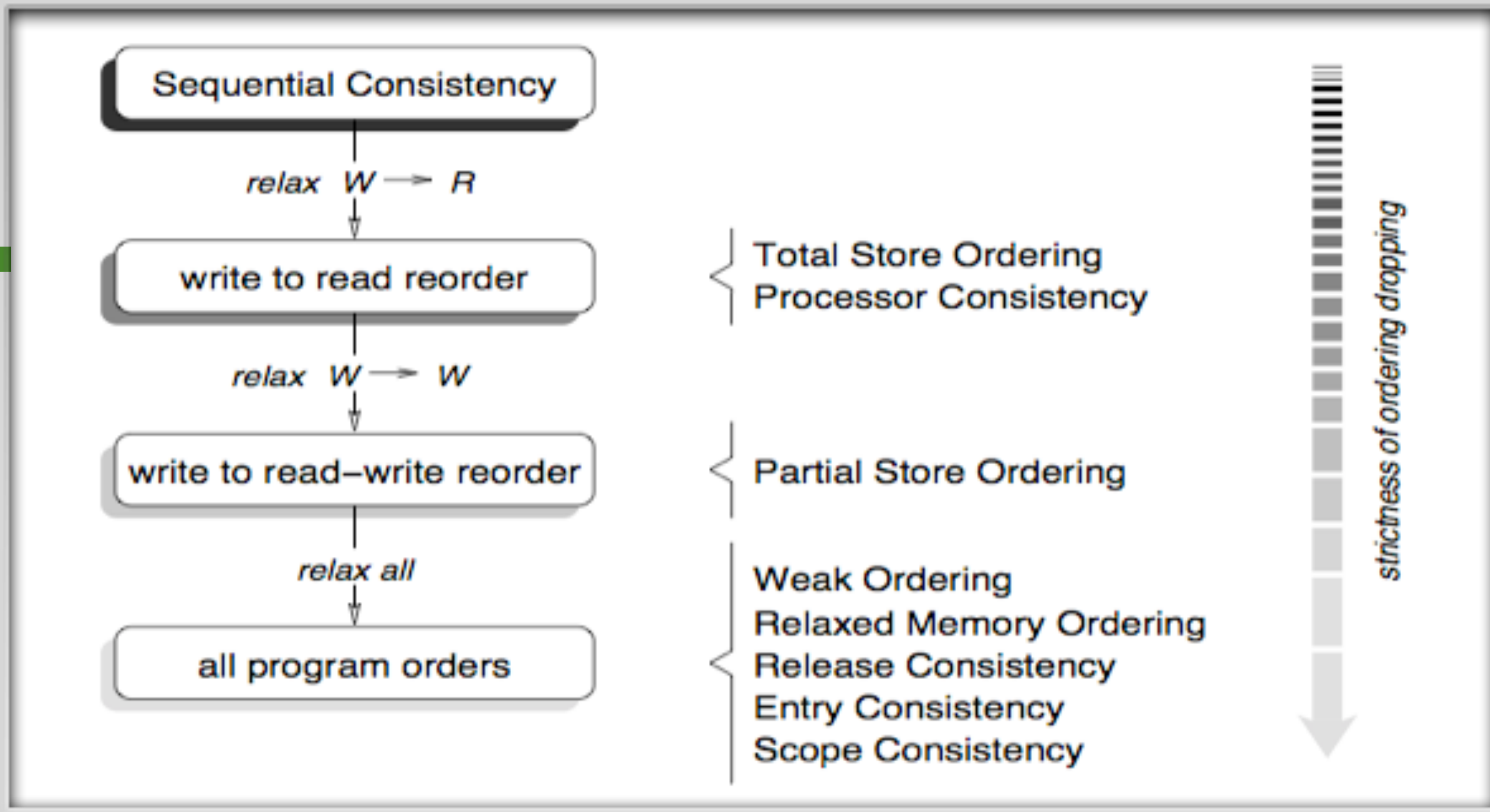


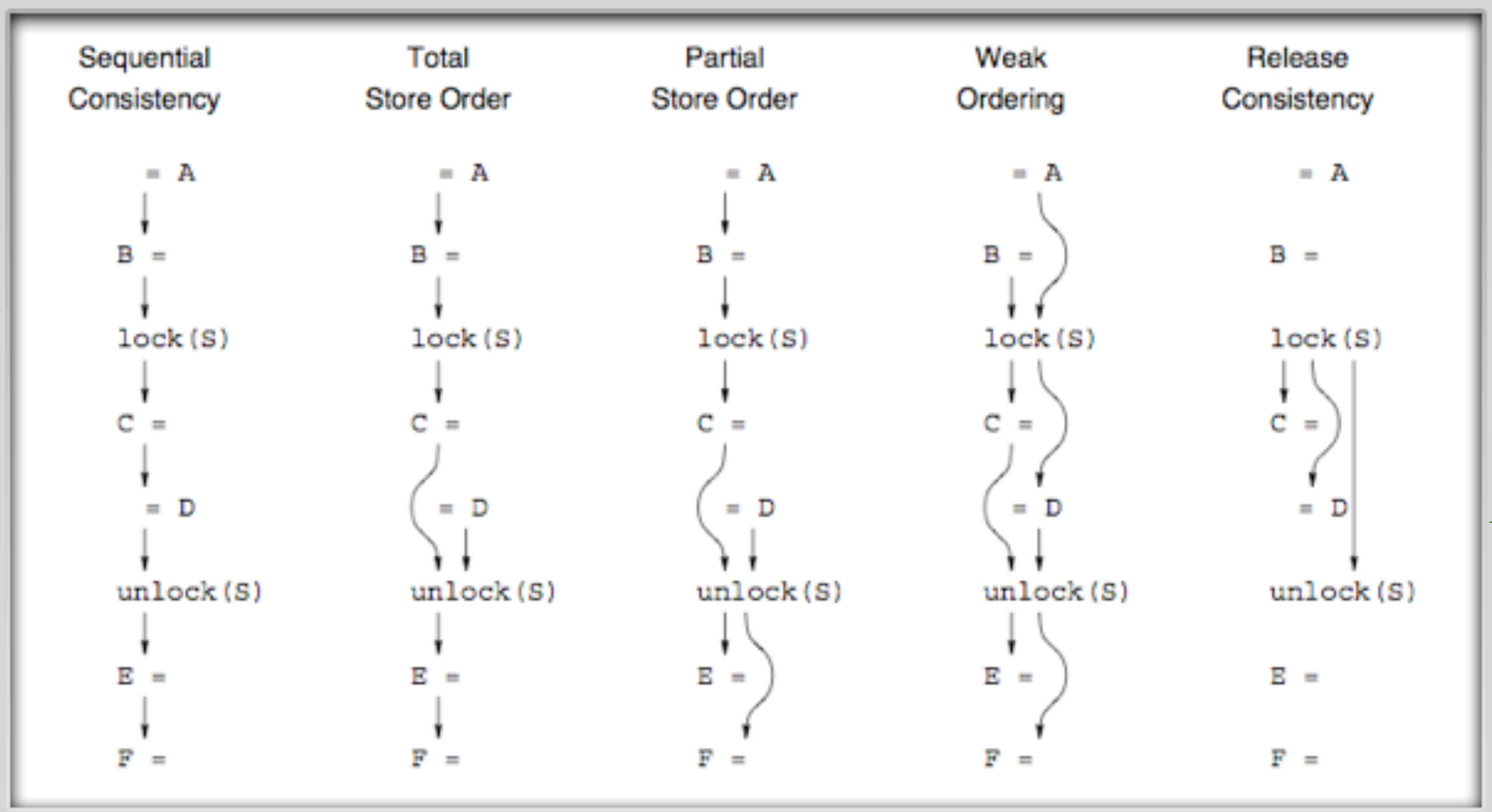
- Memory/Cache Coherence
  - Deal with multiple replicas of the same location in different caches

- Memory Consistency
  - Deal with the ordering in which writes and reads at different locations take effect in memory (issued by either the same or different processors/cores)



Memory Consistency





- Processors:
  - x86, x86_64: Total Store Order
  - PowerPC: Weak Ordering (PowerEN?)
  - ARM Cortex: Weak Ordering
  - Alpha: Release Consistency
- Any Sequential Consistency?
  - No! It is not efficient

Mem Consistency: Seq. Consistency

UNIVERSITÀ DEGLI STUDI DI TORINO

Pi	Pj
write(A,0) ... write(A,1) if (B==0) ...	write(B,0) ... write(B,1) if (A==0) ...

- Can both "if" be evaluated to TRUE?
  - Ideally NO, under Sequential Consistency NO
  - Under more relaxed models? Not guaranteed ...
- Java memory model doesn't expose this complexity
  - at the price of performance

FastFlow SPSC queues

UNIVERSITÀ DEGLI STUDI
DI TORINO

```
push_nonbocking(data) {
  if (NEXT(head) == tail) {
    return EWOULDBLOCK;
  }
  buffer[head] = data;
  head = NEXT(head);
  return 0;
}

pop_nonblocking(data) {

  if (head == tail) {
    return EWOULDBLOCK;
  }
  data = buffer[tail];
  tail = NEXT(tail);
  return 0;
}
```

Lamport FIFO - 1983

- Proved to be correct under SC
  - doesn't work under weaker models

FastFlow SPSC queues


```
push_nonbocking(data) {
  if (NEXT(head) == tail) {
    return EWOULDBLOCK;
  }
  buffer[head] = data;
  head = NEXT(head);
  return 0;
}

pop_nonblocking(data) {

  if (head == tail) {
    return EWOULDBLOCK;
  }
  data = buffer[tail];
  tail = NEXT(tail);
  return 0;
}
```

Lamport FIFO - 1983

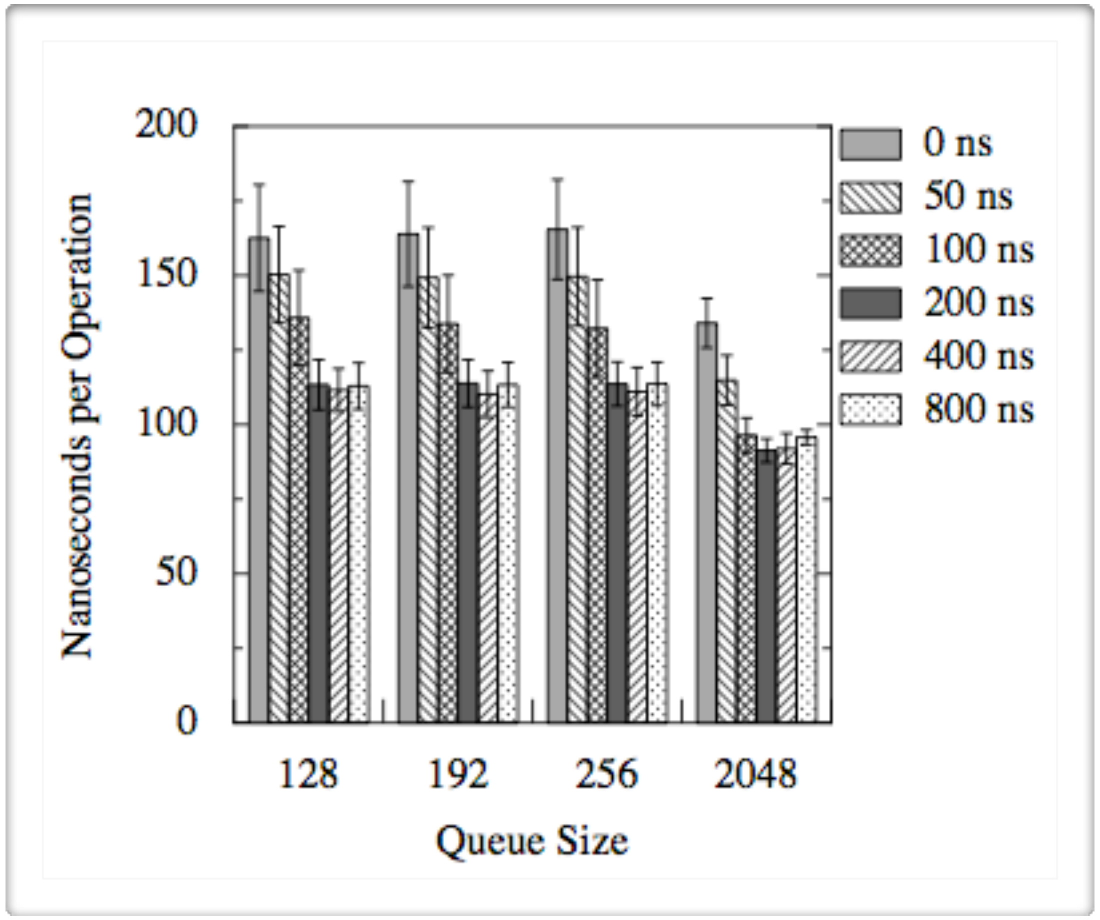
```
push_nonbocking(data) {
  if (NULL != buffer[head]) {
    return EWOULDBLOCK;
  }                                        (WMB)
  buffer[head] = data;
  head = NEXT(head);
  return 0;
}

pop_nonblocking(data) {
  data = buffer[tail];
  if (NULL == data) {
    return EWOULDBLOCK;
  }
  buffer[tail] = NULL;
  tail = NEXT(tail);
  return 0;
}
```

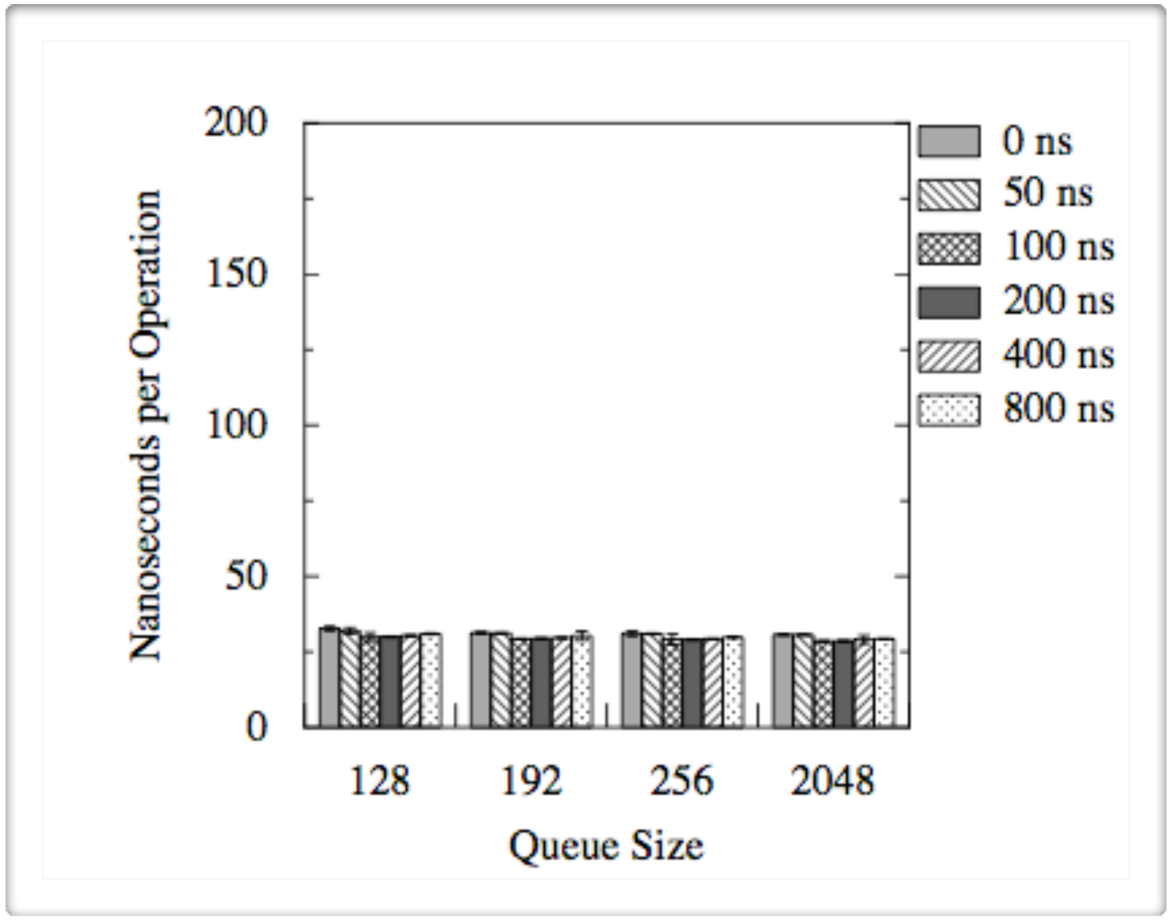
(WMB) For any model weaker than TSO

FastFlow FIFO - derived from P1C1 1997

FastFlow SPSC queues

UNIVERSITÀ DEGLI STUDI DI TORINO





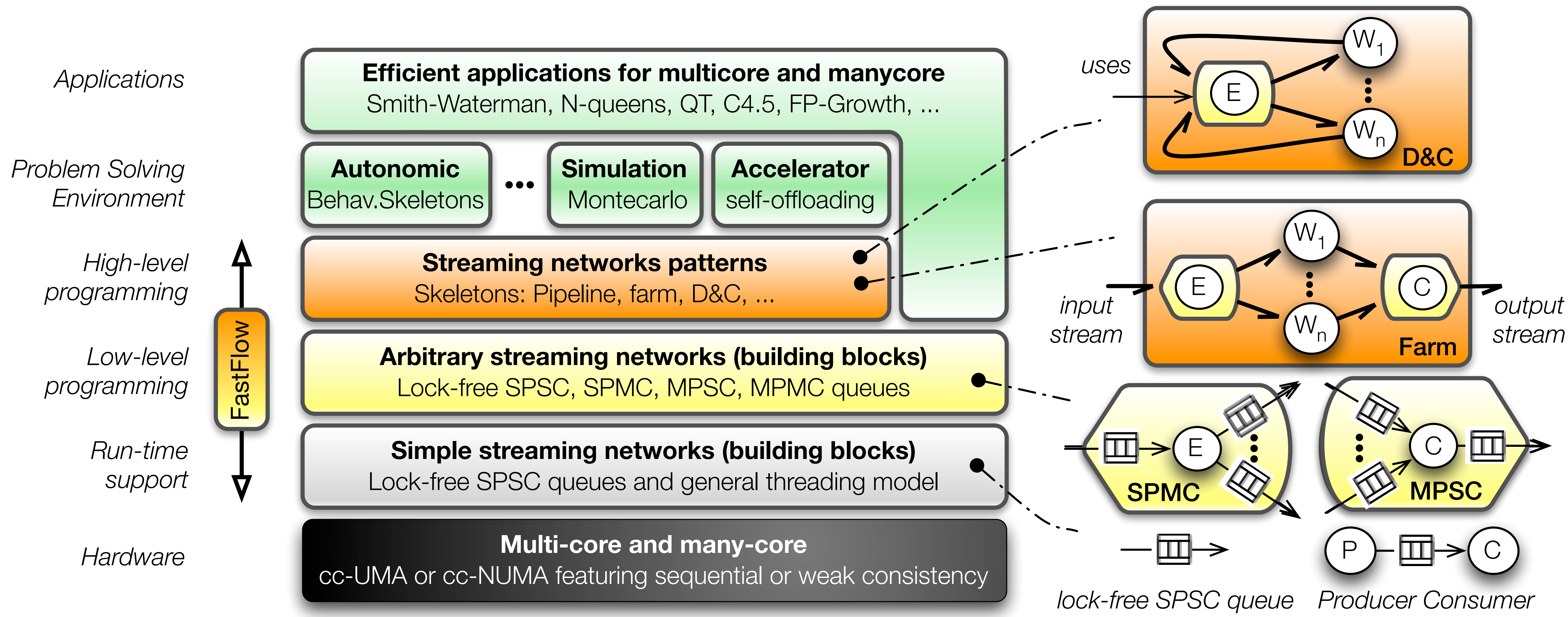
High-level patterns & FastFlow

Pattern-based approach: rationale

- Abstract parallelism exploitation pattern by parametric code
  - e.g. higher order function, code factories, C++ templates, ...
  - Hopefully, in such a way they can composed and nested as programming language constructs
- Provide user with mechanisms to specify the parameters
  - functional (seq code) and extra-functional (QoS) parameters
- Provide state-of-the-art implementation of each parallelism exploitation pattern

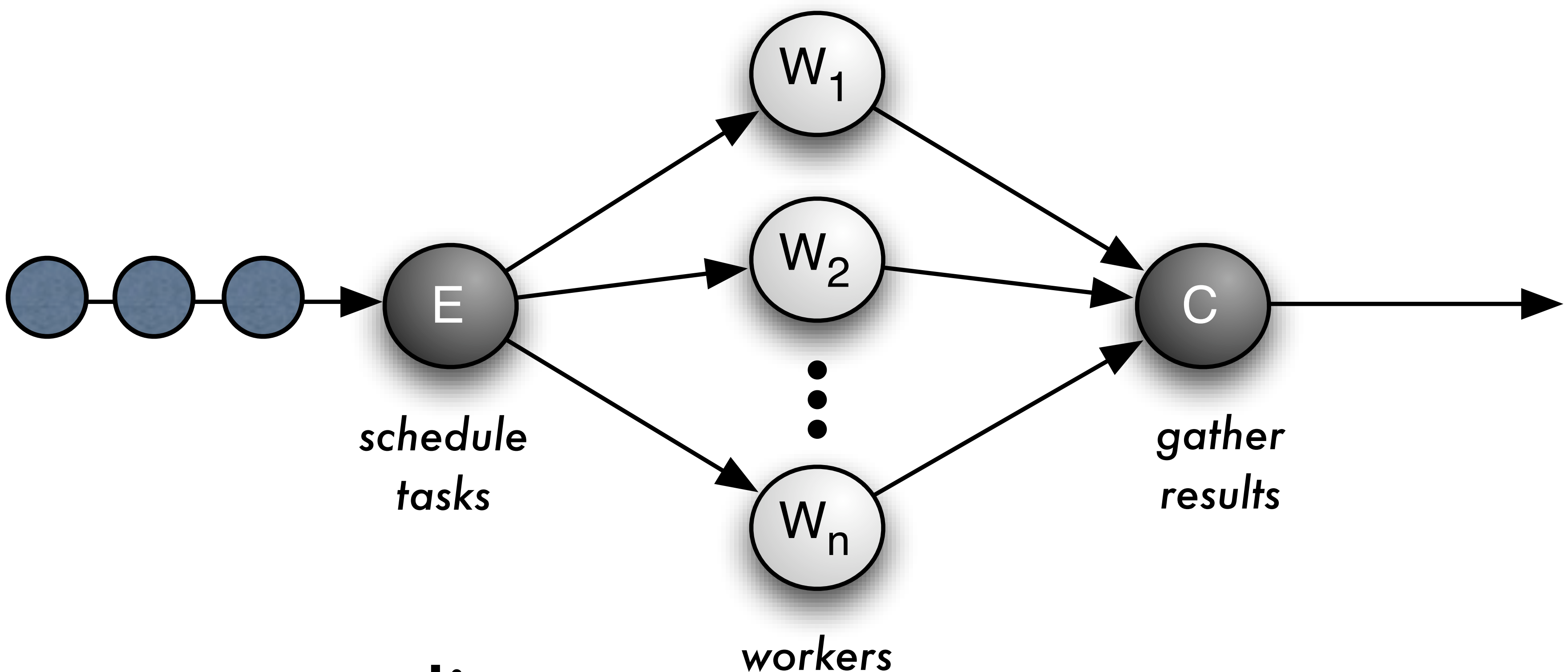
FastFlow: architecture

UNIVERSITÀ DEGLI STUDI DI TORINO



- High-level programming
  - Lock-free/fence-free non-blocking synchronisations
  - C++ STL-like implementation

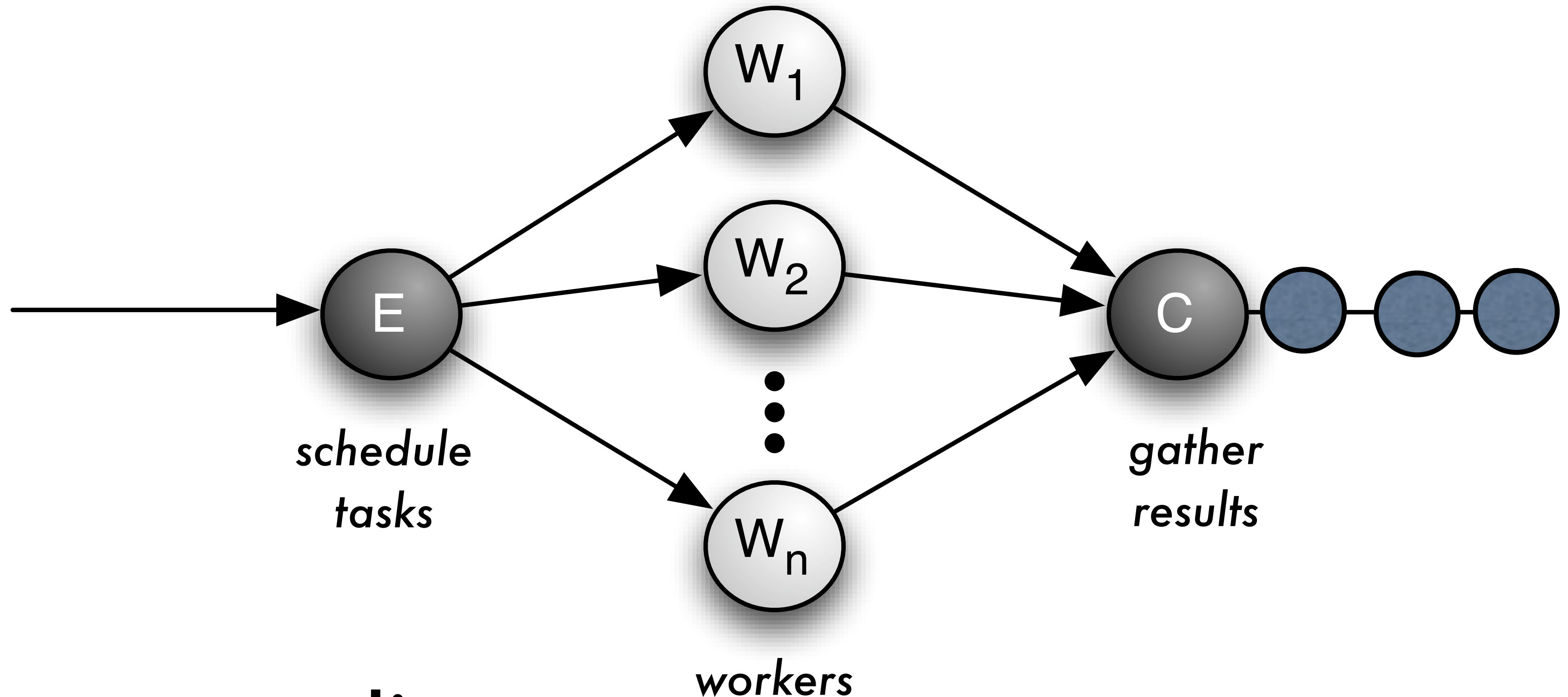
E.g. farm (a.k.a. master-worker)



- Common paradigm *(compute something)*
  - Model foreach and Divide&Conquer
  - Can be used to build data-flow engine
  - Exploit it as a high-order language construct
    - A C++ template factory exploiting highly optimised implementation

E.g. farm (a.k.a. master-worker)

UNIVERSITÀ DEGLI STUDI DI TORINO



- Common paradigm *(compute something)*
  - Model foreach and Divide&Conquer
  - Can be used to build data-flow engine
  - Exploit it as a high-order language construct
    - A C++ template factory exploiting highly optimised implementation

The code

test1.cpp

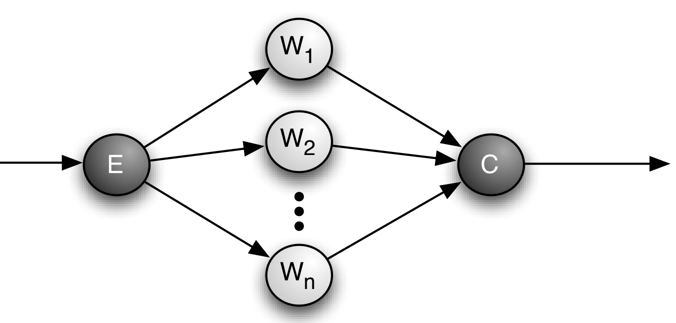
```
using namespace ff;
class Worker: public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        std::cout << "Worker " << ff_node::get_my_id()
                  << " received task " << *t << "\n";
        return task;
    }
};

class Collector: public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        if (*t == -1) return NULL;
        return task;
    }
};

class Emitter: public ff_node {
public:
    Emitter(int max_task):ntask(max_task) {};

    void * svc(void *) {
        int * task = new int(ntask);
        --ntask;
        if (ntask<0) return NULL;
        return task;
    }
private:
    int ntask;
};
```

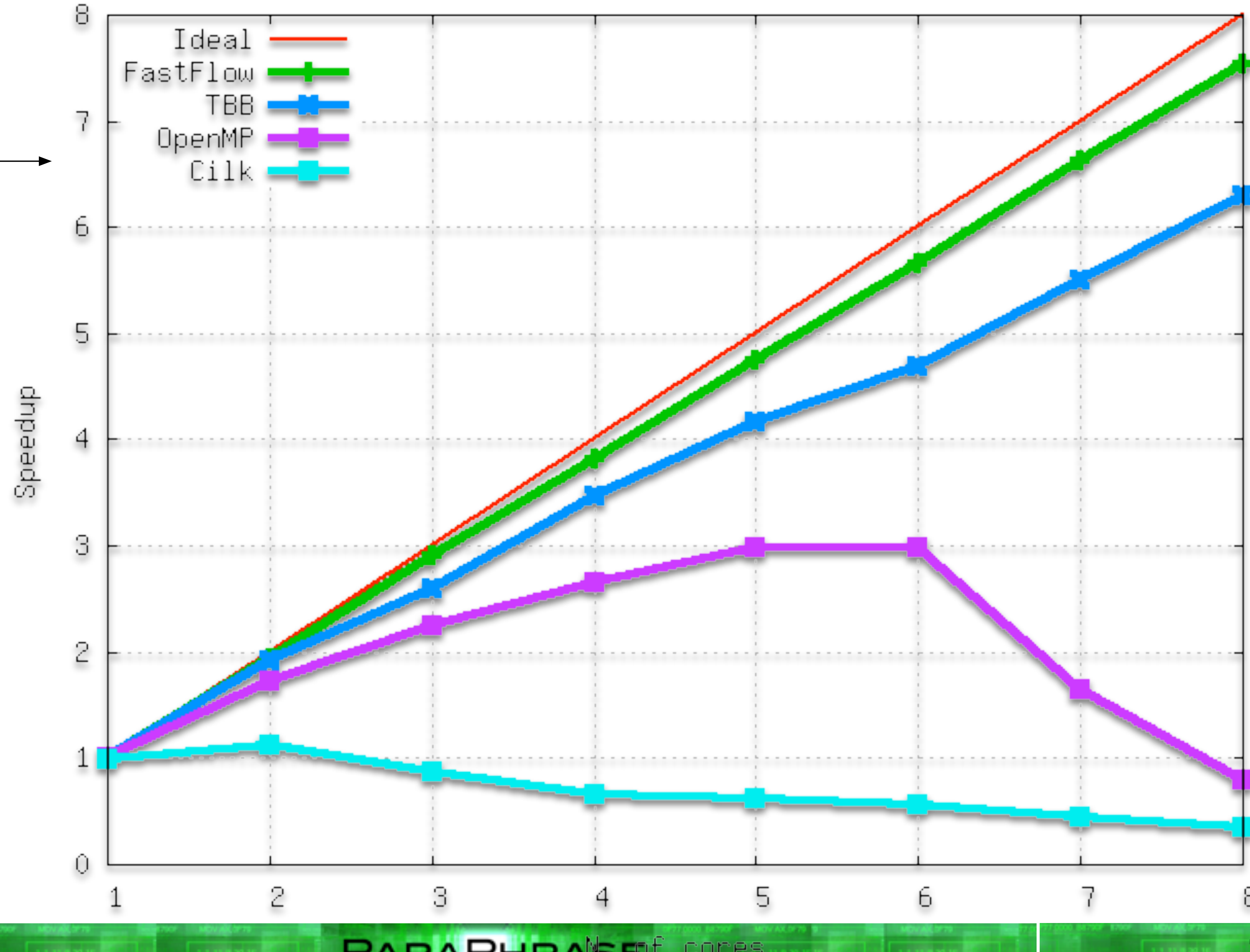
test1.cpp

```
int main(int argc, char * argv[]) {

    if (argc<3) {
        std::cerr << "use: "
                  << argv[0]
                  << " nworkers streamlen\n";
        return -1;
    }

    int nworkers=atoi(argv[1]);
    int streamlen=atoi(argv[2]);

    if (!nworkers || !streamlen) {
        std::cerr << "Wrong parameters values\n";
        return -1;
    }

    ff_farm<> farm; // farm object

    Emitter E(streamlen);
    farm.add_emitter(&E);

    std::vector<ff_node *> w;
    for(int i=0;i<nworkers;++i) w.push_back(new Worker);
    farm.add_workers(w); // add all workers to the farm

    Collector C;
    farm.add_collector(&C);

    if (farm.run_and_wait_end()<0) {
        error("running farm\n");
        return -1;
    }
    std::cerr << "DONE, time= " << farm.ffTime() << " (ms)\n";
    farm.ffStats(std::cerr);

    return 0;
}
```

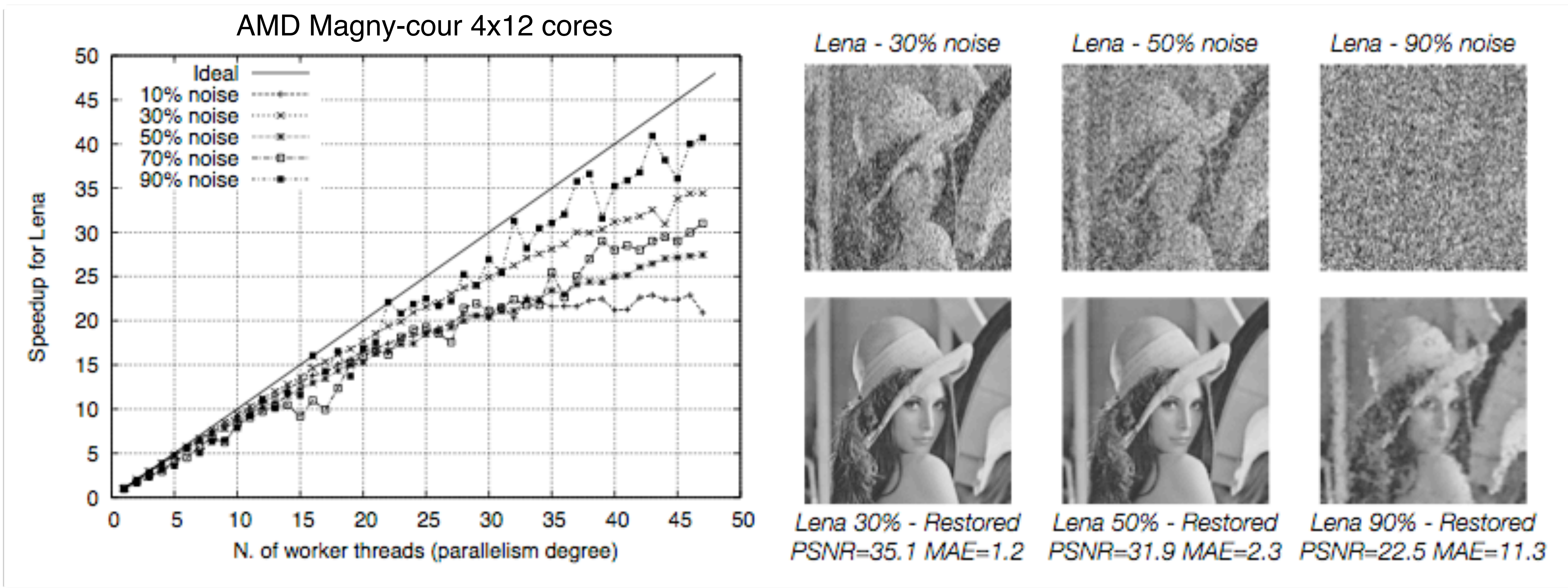
Medium grain (5 μS workload)

UNIVERSITÀ DEGLI STUDI DI TORINO





Edge-preserving denoiser. Live demo!



Lena* with 90% of noise is restored in 4 seconds
Next best result in literature is about 180 seconds

Lena Lombardy. Standard test image, Playboy magazine centerfold (top). Nov. 1972

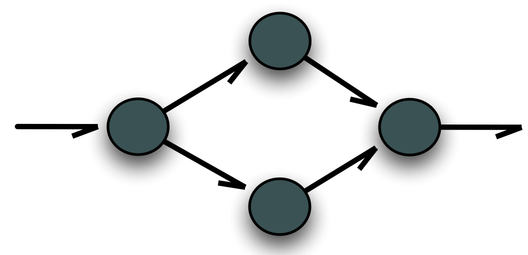
Pattern composition

UNIVERSITÀ DEGLI STUDI DI TORINO

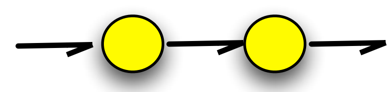
- C++ STL-like implementation
  - used to generatively compile skeletons into streaming networks
  - fully memory barrier free implementation
- High-level pattern compose with ; and { }
  - their implementation as parametric streaming networks (graphs)
  - performance can be optimised as in streaming graphs (network of queues)

Patterns, and they comp. implementation

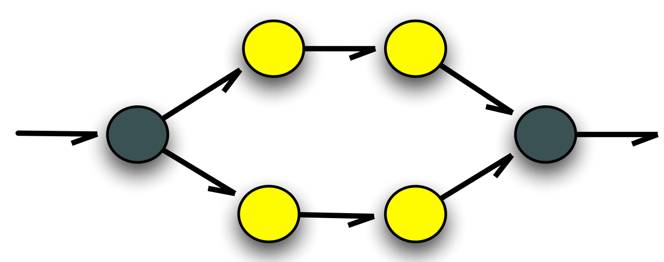
- farm



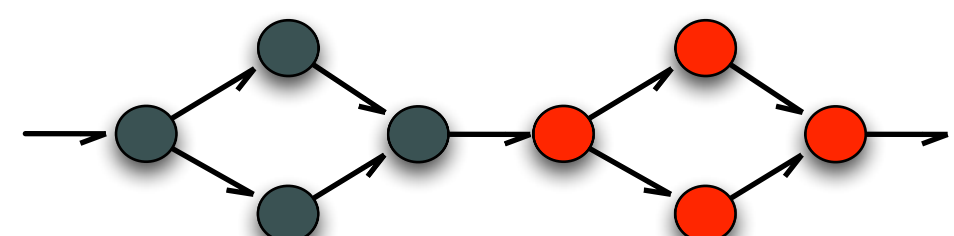
- pipe



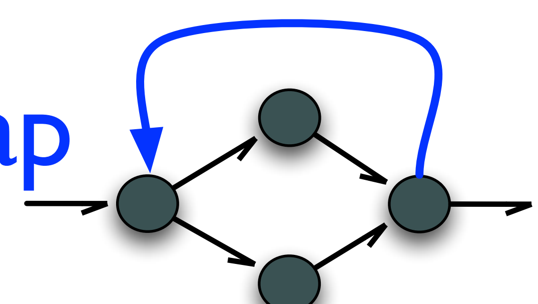
- farm{ pipe }



- farm ; farm



- D&C = farm + wrap



- any variation of them requiring additional synch ...

The challenge for Formal methods. Questions and thoughts in this context (in open order).

Many open problems

UNIVERSITÀ DEGLI STUDI DI TORINO

- 1) Mechanisms e concurrency theory
  - new queues and data containers, new allocation techniques, ...
  - cc-NUMA: mapping tools; smart-network support (RDMA)
- 2) Formal Quantitative
  - performance analysis, optimisation, ...
- 3) Formal Qualitative
  - correctness, protocol proofs, ...
- 4) Design and tools
  - language evolution, compiler evolution, new features, meta-programming technique evolution, staged compilation, adaptive support

From patterns to metal

UNIVERSITÀ DEGLI STUDI DI TORINO

- Graphs can be used as compilation (intermediate) layer
- Is this good or a bad news?
  - Graphs well understood
    - Thread pinning, thread affinity, addresses locality, concurrent code optimisation, ..., can be modelled as graph
    - Traditional tool of formal method community
  - Everything concerning "graph" is complex by its very nature
    - At least for myself

Simple?

UNIVERSITÀ DEGLI STUDI DI TORINO

SHR Inference rules…in one slide

Parallel

$$\frac{\Gamma \vdash G_1 \xrightarrow{\Lambda} \Phi \vdash G_2 \qquad \Gamma' \vdash G_1' \xrightarrow{\Lambda'} \Phi' \vdash G_2' \qquad (\Gamma \cup \Phi) \cap (\Gamma' \cup \Phi') = \emptyset}{\Gamma, \Gamma' \vdash G_1 | G_1' \xrightarrow{\Lambda \cup \Lambda'} \Phi, \Phi' \vdash G_2 | G_2'}$$

The system can do whatever disjoint subsystems do

Restrict

$$\frac{\Gamma, x \vdash G_1 \xrightarrow{\Lambda} \Gamma, x \vdash G_2 \qquad \Lambda(x) = \epsilon \vee \Lambda(x) = \tau}{\Gamma \vdash \nu x\ G_1 \xrightarrow{\Lambda \setminus \{x\}} \Gamma \vdash \nu x\ G_2}$$

The system can do any transition not requiring any synchronisations on restricted node

Merge

$$\frac{\Gamma, x, y \vdash G_1 \xrightarrow{\Lambda} \Phi \vdash G_2}{\Gamma[x/y] \vdash G_1[x/y] \xrightarrow{\Lambda, \{x, \tau, \}} \Phi[x/y] \vdash \nu U\ G_2[x/y]\rho}$$

x and y can be fused provided that they perform *compatible* synchronisation actions

E. Tuosto and myself.
Towards a Formal Semantics for Autonomic Components. Sensoria and CoreGRID FP6

Programming model

- Producer-Consumer and mutal exclusion have a different pragmatics: cooperation vs competition
- FastFlow advocates Producer-Consumer
  - Synchronisation via message-passing, data exchange via both message-passing and shared memory
  - Allow mutual exclusion on business code under the full responsibility of the programmer because this is not efficient at fine grain
    - They are additional bi-directional arrows in the graph
- IBM BlueGene/Q (forthcoming) has hardware transactional memory
  - LL/SC with versions
  - Efficient

A simple proof system for lock-free concurrency

Luís Caires, Carla Ferreira, and **António Ravara**

Dep. of Informatics, Faculty of Sciences and Technology
New University of Lisbon

June 20, 2011

Operational Semantics

A transition relation

Transition system - memory manipulation

$$\frac{[\![E]\!]_{(s,h)} = v}{(s,h,x = E) \longrightarrow ((s[x \mapsto v], h, \texttt{skip})}\,(assignment)$$

$$\frac{[\![E]\!]_{(s,h)} = v_1 \wedge h(v_1) = (v,\_)}{(s,h,x = *E) \longrightarrow ((s[x \mapsto v], h, \texttt{skip})}\,(load)$$

$$\frac{[\![E_2]\!]_{(s,h)} = v_2 \ \wedge\ [\![E_1]\!]_{(s,h)} = v_1 \ \wedge\ h(v_1) = (\_, b)}{(s,h,*E_1 = E_2) \longrightarrow (s, h[v1 \mapsto (v_2, b == 1\,?\,2 : b)], \texttt{skip})}\,(store)$$

New programming primitives

Non-blocking Primitives

CompareAndSwap (CAS)

$\texttt{CAS}(loc, v_{old}, v_{new})$ atomically exchange the value in the memory location *loc* with value v_{new}, provided the current value in *loc* is v_{old}.

LoadLink (LL) / StoreConditional (SC)

The pair of primitives have a transaction-like semantics:

- $\texttt{LL}(loc)$ returns the address of the memory location *loc*

  May be interpreted as the transaction start
- $\texttt{SC}(loc, v)$ tries to store v in the memory location *loc*:

  if the location has not been written by any concurrent thread since LL, v is written atomically and the instruction returns true; otherwise, it returns false, not writing the value

  May be interpreted as the transaction commit, returning either "success" or "abort"

Operational Semantics

A transition relation

Transition system - transaction-like primitives

(*load-link*)

$$\frac{h(s(x)) = (v,b)}{(s,h,y = \texttt{LL}(x)) \longrightarrow (s[y \mapsto v], h[s(x) \mapsto (v, b == 0\,?\,1 : b)], \texttt{skip})}$$

(*store-conditional$_T$*)

$$\frac{h(s(x)) = (\_,1) \ \wedge\ h(s(y)) = (v,\_)}{(s,h,z = \texttt{SC}(x,y)) \longrightarrow (s[z \mapsto 1], h[s(x) \mapsto (v,0)], \texttt{skip})}$$

(*store-conditional$_F$*)

$$\frac{h(s(x)) = (\_,b) \ \wedge\ b \neq 1 \ \wedge\ h(s(y)) = (v,\_)}{(s,h,z = \texttt{SC}(x,y)) \longrightarrow (s[z \mapsto 0], h[s(x) \mapsto s(y)], \texttt{skip})}$$

UNIVERSITÀ DEGLI STUDI
DI TORINO

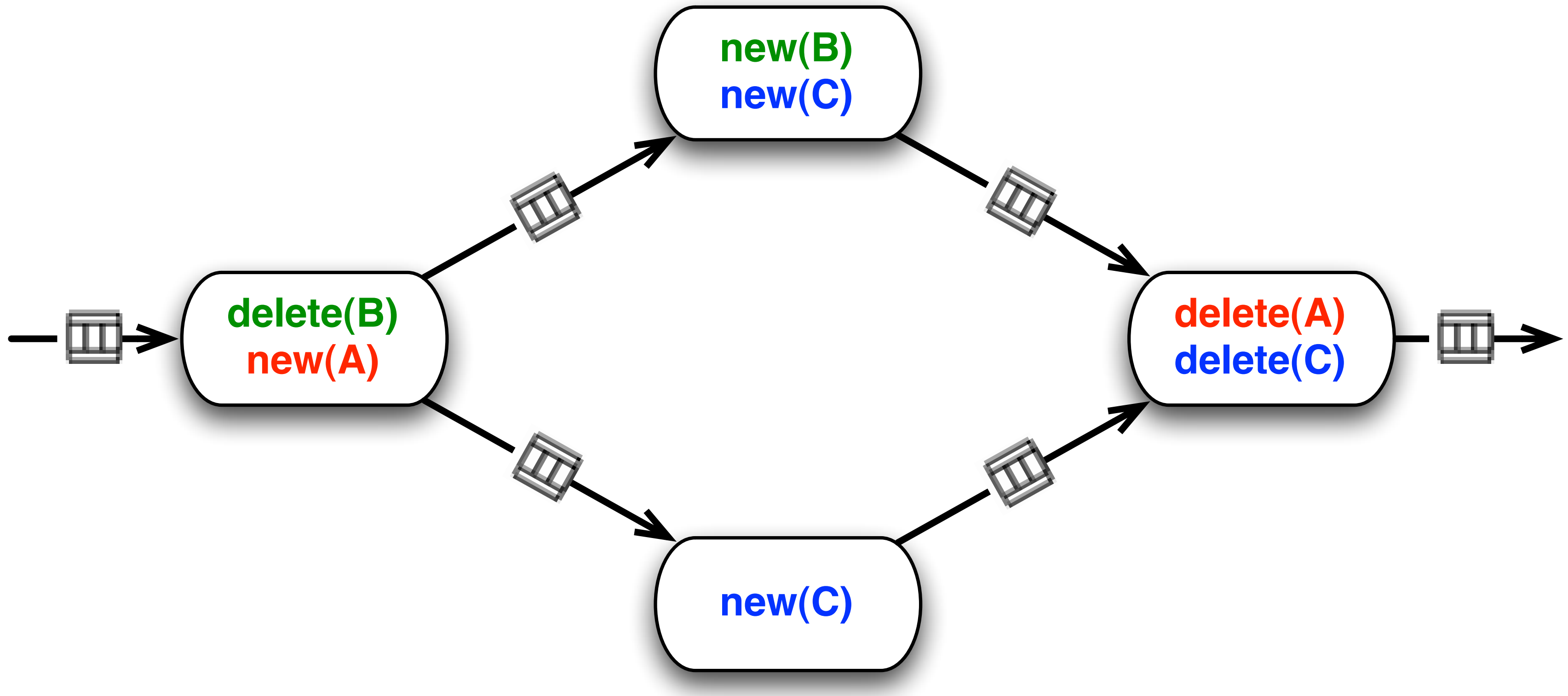
A More Efficient and Type-Safe Version of FastFlow ⋆

Zalán Szűgyi and Norbert Pataki

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
`lupin@ludens.elte.hu`, `patakino@elte.hu`

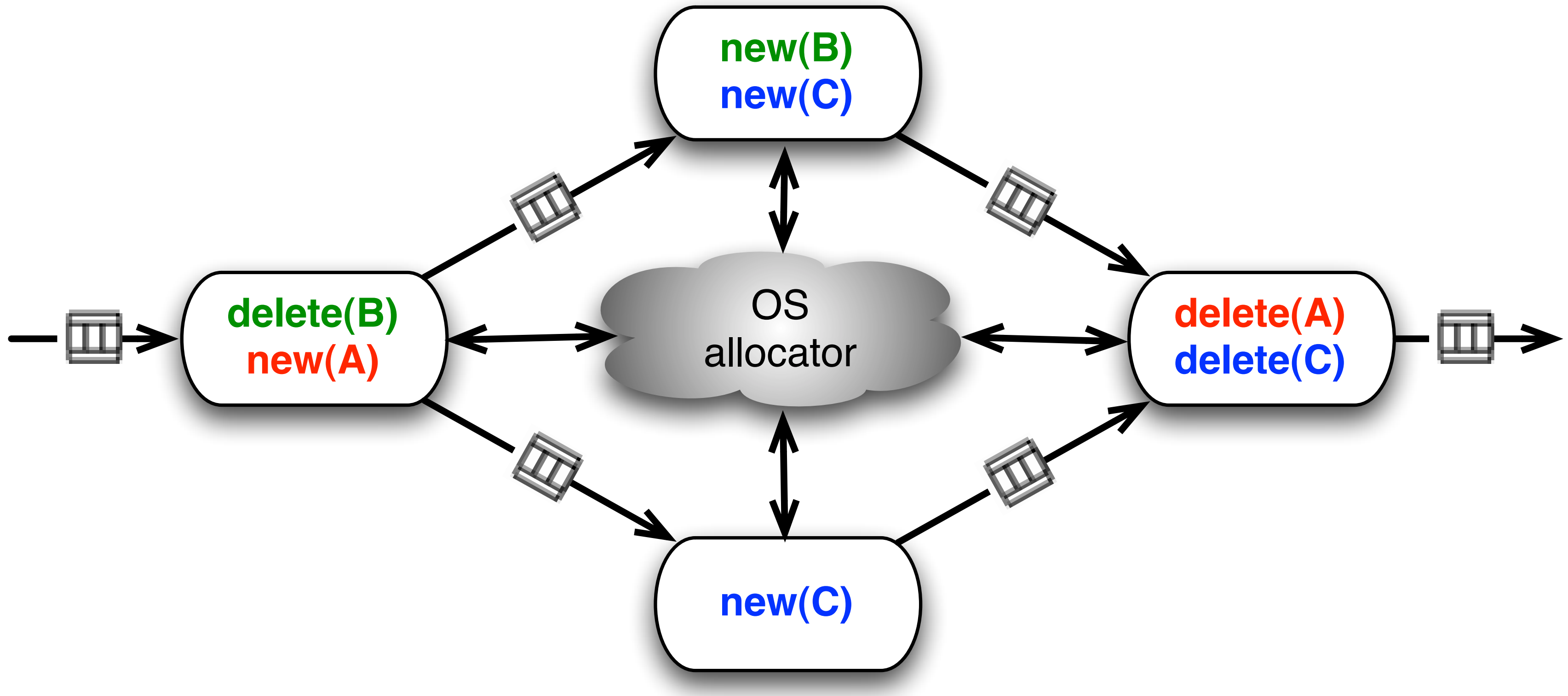
Abstract. Nowadays, one of the most important challenges in programming is the efficient usage of multicore processors. Many new programming languages and libraries support multicore programming.
FastFlow is one of the most promising multicore C++ libraries. Unfortunately, a design problem occurs in the library. One of the most important methods is pure virtual function in a base class. This method supports the communication between different threads. Although, it cannot be template function because of the virtuality, hence, the threads pass and take argument as a `void*` pointer. The base class is not template neither. This is not typesafe approach. We make the library more efficient and safer with the help of generative technologies.

Example: FF-allocator



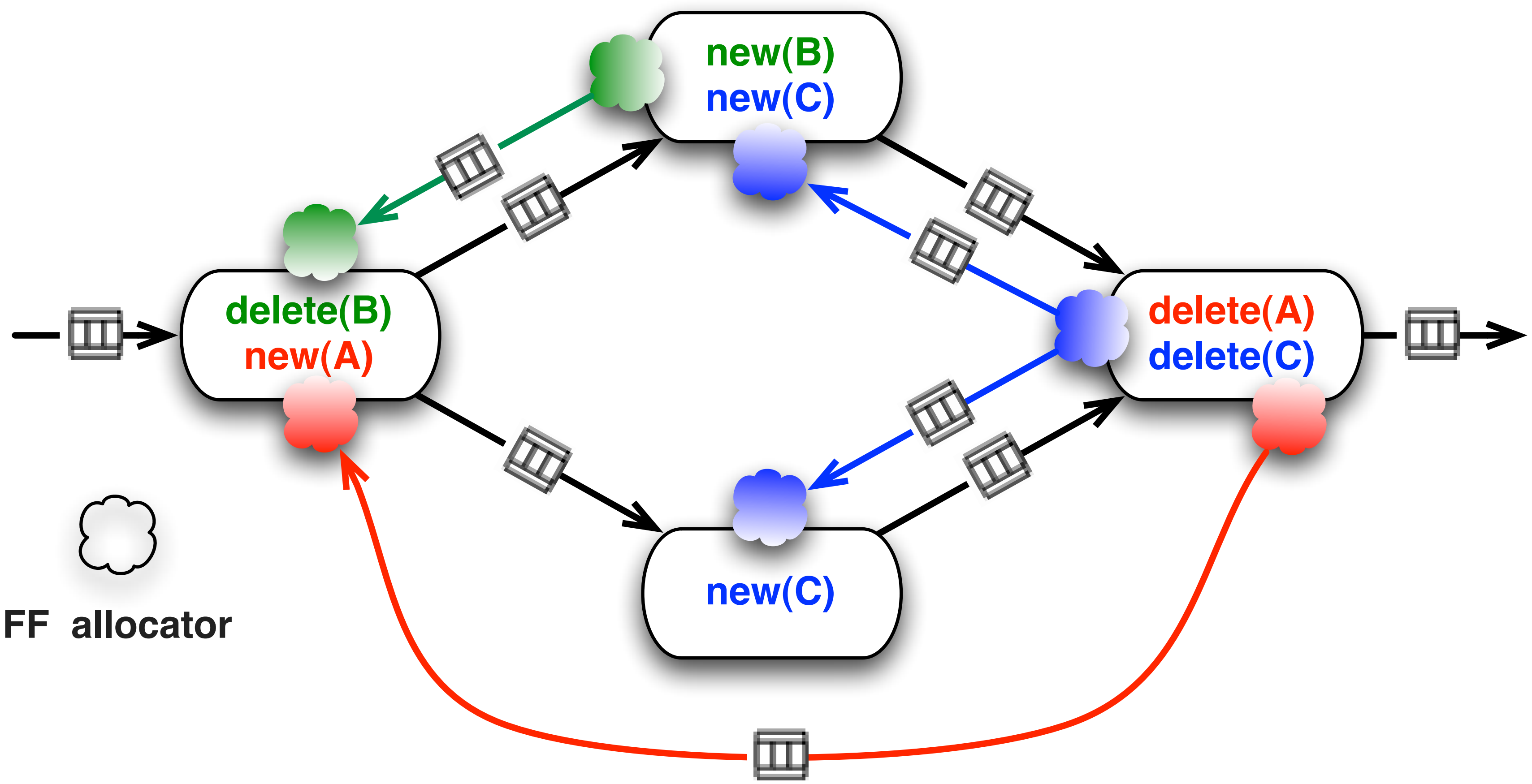
Example: FF-allocator

UNIVERSITÀ DEGLI STUDI DI TORINO



Example: FF-allocator

UNIVERSITÀ DEGLI STUDI DI TORINO



The graph is now cyclic (with bound queues)
➠ Deadlock

- faster than posix, hoard, TBB
  - unpublished, but available on sourceforge

ISSUEs for exascale era

UNIVERSITÀ DEGLI STUDI DI TORINO

- A computer language is not a computing model.
  A library is not a computing model.
  - Data communication happen via both shared-memory and messages. Synchronisations are realised via message-passing (FIFO queues).
  - Synchronisation are local (no barriers) and determined by high-level algorithmic patterns. Data races are identified and solved at design time.
- Increasing scalability requires to decrease concurrency grain. Programming systems should be designed to support fast data movement and enforce locality.
  - FastFlow: inter-core communication latency ~7-10 ns on core2 2Ghz. Better than other approaches at fine grain.

ISSUEs for exascale era

UNIVERSITÀ DEGLI STUDI
DI TORINO

- How to describe concurrency exploitation at large scale?
  - Parametric patterns. QoS/performance as first-class concept. Couple data with flow-of-control beyond OO (how?)
- How we promote scalability "by design" and performance portability?
  - Development tools. Mapping/affinity, should be automatically managed. Graphs very expressive. Graph-to-graph mapping encode semantic-preserving transformation (i.e. optimisation).
- Functional-style coding?
  - Why not. Nicely translated into dataflow. Nicely maps into streaming.
  - Empirically the only way I found to write SSE/AVX.