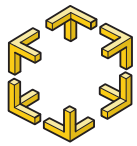
DESIGN CHALLENGES FOR SCALABLE CONCURRENT DATA STRUCTURES

for Many-Core Processors

DIMACS
March 15th, 2011



Distributed Computing and Systems
Chalmers university of technology

Philippas Tsigas

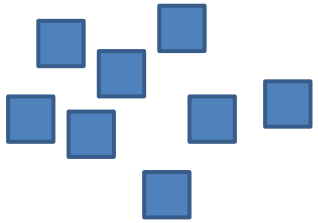
Data Structures In Manycore Sys.

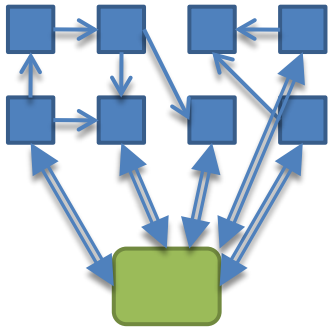
Synchronization

Decomposition

Load Balancing

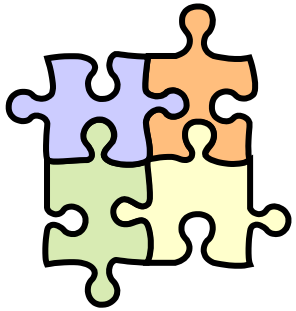
Inter Task Communication

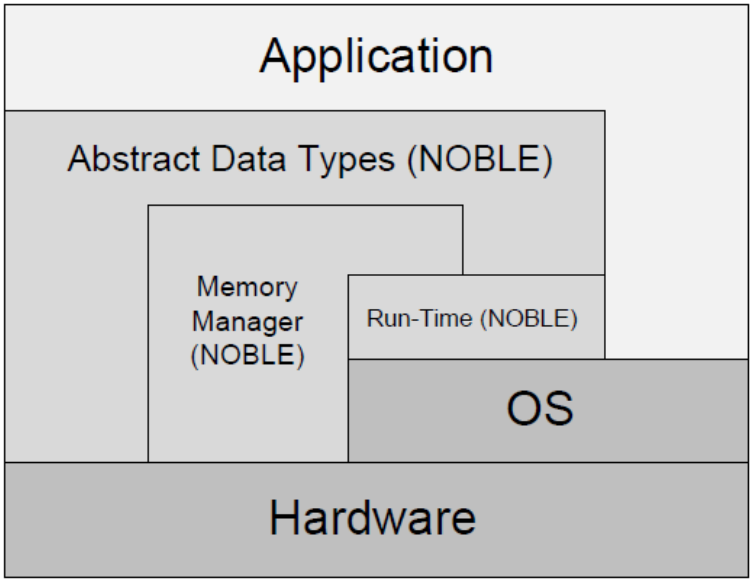




Concurrent Data Structures: Our NOBLE Library project

- Fundamental shared data structures
  - Stacks
  - Queues
  - Deques
  - Priority Queues
  - Dictionaries
  - Linked Lists
  - Snapshots
- Memory Management
  - Memory allocation
  - Memory reclamation (garbage collection)
- Atomic primitives
  - Single-word and Multi-word transactions.





Many-core?

- No clear definition, but at least **more than 10 cores**
- Some say thousands

Dual-, Quad-, Hexa-, Octo-, ~~Dekaexi-? Trianta dyo-?~~ Many-core!

- The most commonly available many-core platforms are the medium to high end graphics processors
- Have up to 30 multiprocessors and available at a low-cost
- **CUDA** and **OpenCL** have made them easily accessible

Framework for parallel computing

Computing platform developed by NVIDIA

A Basic Comparison

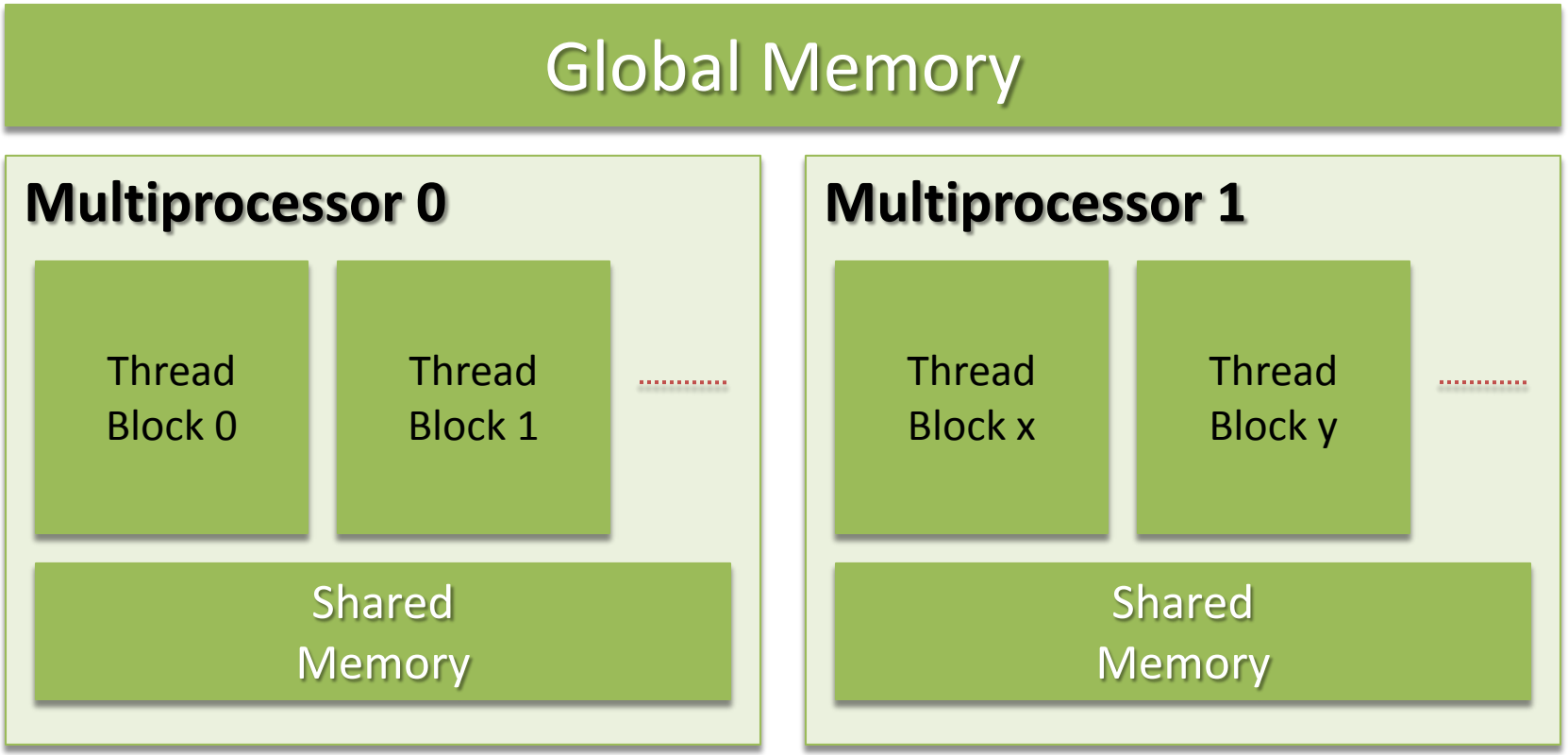
Normal processors

- Large cache
- Few threads

Graphics processors

- Small/No cache
- SIMD
- Wide and fast memory bus with memory operations that can be coalesced
- Thousands of threads masks memory latency

CUDA System Model



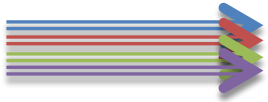
CUDA System Model

- Atomic primitives
  - None -> For global memory -> For shared memory
- Threads per multiprocessor
  - 768 -> 1024 -> 1536
- Shared memory
  - 16KB -> 48KB
- SIMD width
  - 8 words -> 32 words

Locks are not supported

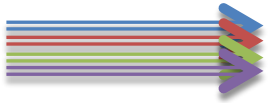
- Not in CUDA, not in OpenCL
  - **Fairness** of hardware scheduler **unknown**
  - Thread block holding a lock might be swapped out **indefinitely**, for example

No Fairness Guarantees



…

…



while(atomicCAS(&lock,0,1));

ctr++;

lock = 0;

Thread holding lock is never scheduled!

Lock-free Data Structures

- Mutual exclusion (Semaphores, mutexes, spin-locks, disabling interrupts: Protects critical sections)
  - Locks limits concurrency, priority inversion
  - Busy waiting – repeated checks to see if lock has been released or not
  - Convoying – processes stack up before locks
  - **Blocking Locks are not composable**

    All code that accesses a piece of shared state must know and obey the locking convention, regardless of who wrote the code or where it resides.

- Lock-freedom is a **progress guarantee**
- In practice it means that
  - A fast process doesn't have to wait for a slow or dead process
  - No deadlocks
- Shown to **scale better** than blocking approaches

Definition

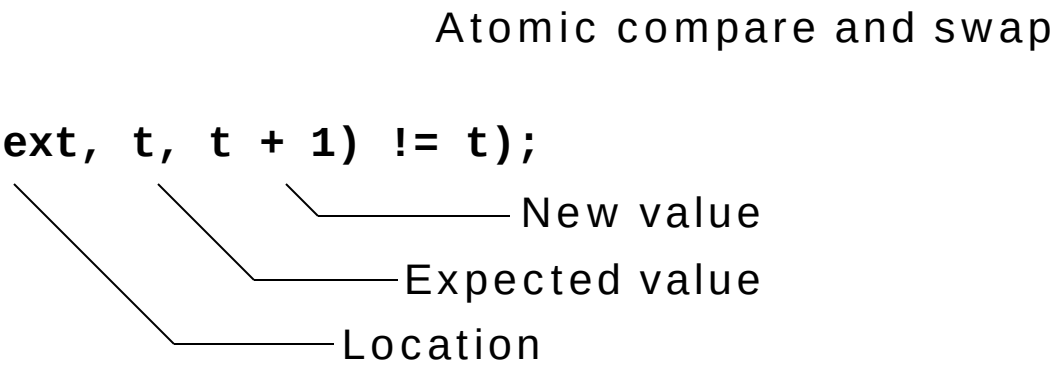
For all possible executions, **at least one** concurrent operation will **succeed** in a **finite** number of its own steps

A Lock-free Implementation of a Counter

- In this case a non-blocking design is easy:

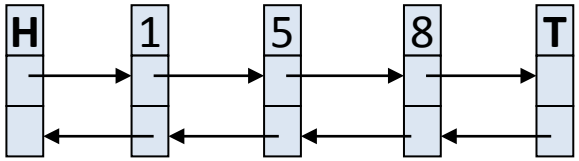
```
class Counter {
  int next = 0;

  int getNumber () {
    int t;
    do {
      t = next;
    } while (CAS (&next, t, t + 1) != t);
    return t;
  }
}
```

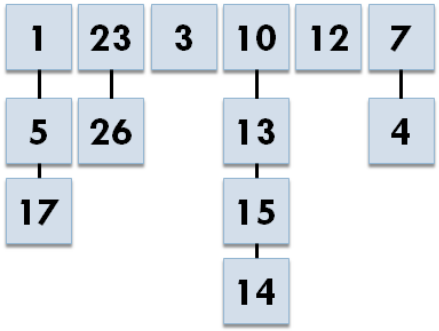


Lock Free Concurrent Data Structures

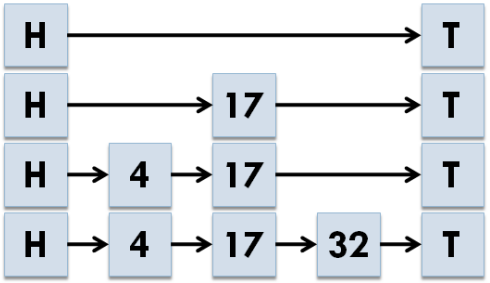
Doubly Linked Lists



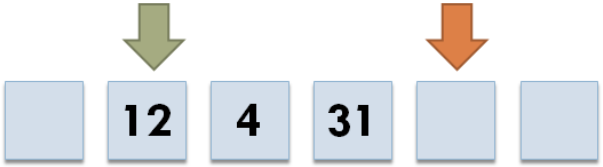
Hashtables, Dictionaries



Skiplists



Queues, Priority, Deques



...

LF DS in Normal Processors: Joint work with D. Cederman, A. Gidenstamn, Ph. Ha, M. Papatriantafilou, H. Sundell, Y. Zhang

Graphics Processors: Joint work D. Cederman, Ph. Ha, O. Anshus

A Basic Comparison

Normal processors

- Large cache
- Few threads
- **Atomics (CAS, …)**

Graphics processors

- Small/No cache
- SIMD
- Wide and fast memory bus with memory operations that can be coalesced
- Thousands of threads masks memory latency
- **No atomics -> …-> expensive ones**

Lock-free Data Structures Without Atomics?

Emulating CAS from Coalesced Memory Access

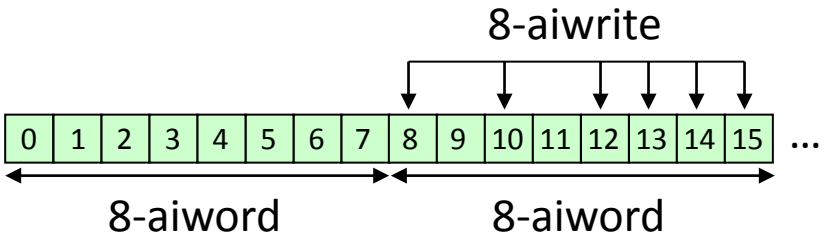
Coalesced Global Memory Accesses

- **The simultaneous global memory accesses by each thread of a half-warp during the execution of a single read or write instruction will be *coalesced* into a single access if:**
  - The size of the memory element accessed by each thread is either 4, 8, or 16 bytes
  - The elements form a contiguous block of memory
  - The N^{th} element is accessed by the N^{th} thread in the half-warp
  - The address of the first element is aligned to 16 times the element's size

- **Coalescing happens even if some threads do not access memory (divergent warp)**

Aligned-inconsecutive word (aiword)

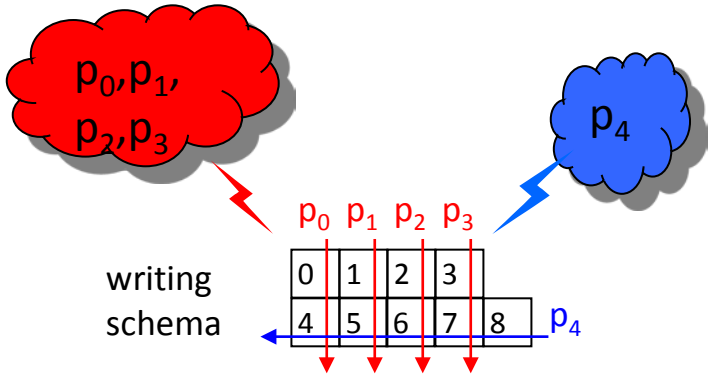
- Memory is aligned to *m*-unit words, *m* is a constant.
  - *m*-aiword for short
- A read/write operation accesses an arbitrary non-empty subset of the *m* units of an aiword.
  - *m*-aiwrite = *m*-aiword assignment.
- Alignment restriction
  - *m*-aiwords must start at addresses that are multiples of *m*.
- Ex: 8-aiwrite

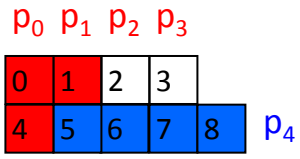


m-aiword's consensus no. $\geq \left\lfloor \frac{m+1}{2} \right\rfloor$

- Idea:
  - Construct a *binary* consensus object for $N=\left\lfloor \frac{m+1}{2} \right\rfloor$ processes in which *(N-1)* processes propose the same value.
  - Construct a *multivalued* consensus object for *N* processes using the binary consensus object.
- Ex: 9-aiword

Binary consensus (BC) for 4+1 processes

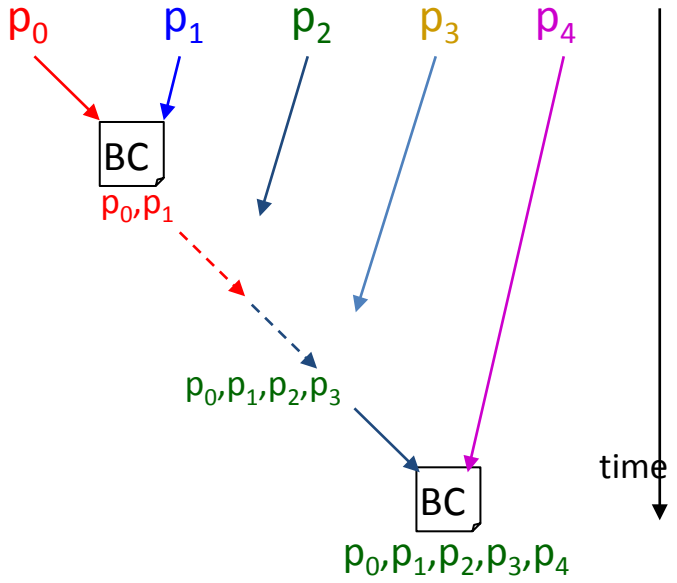




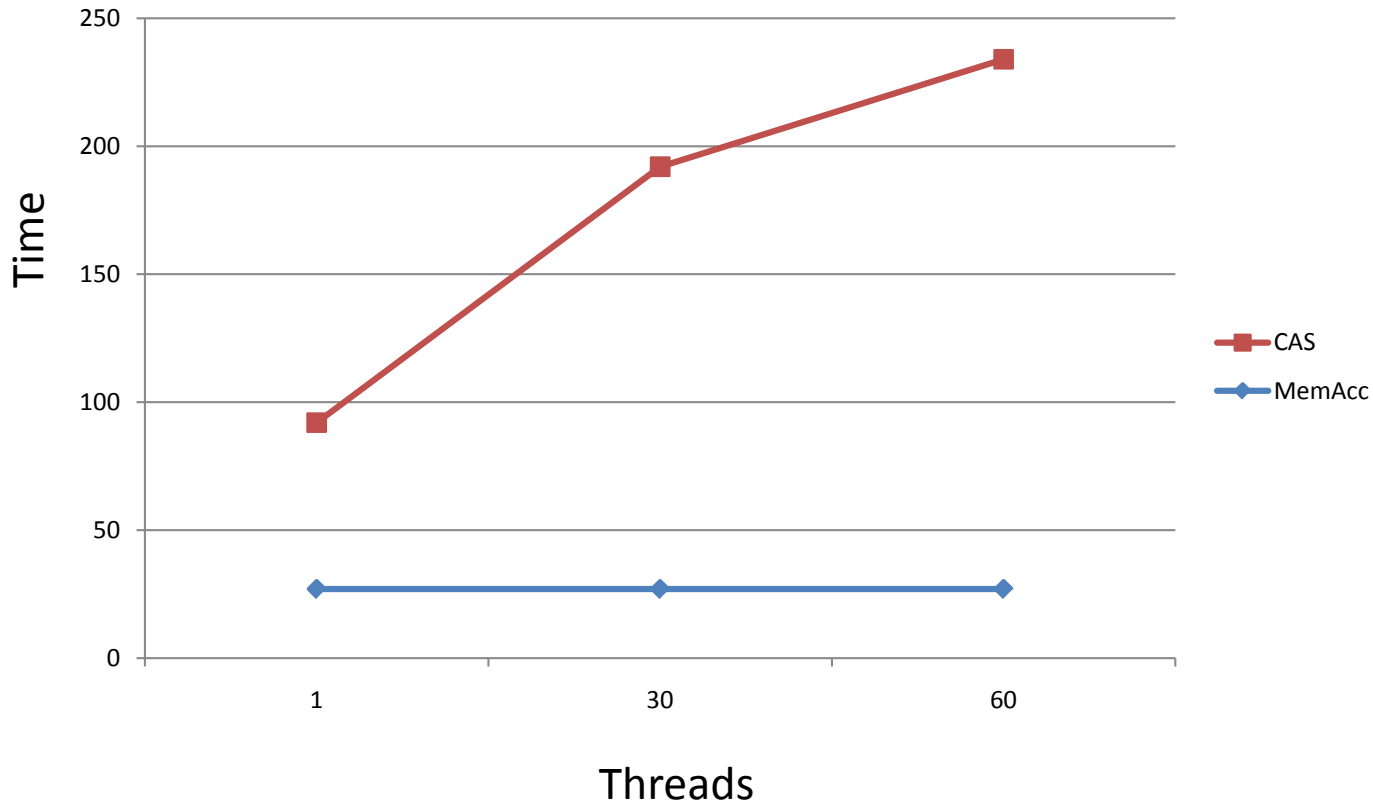
[0,4,8] ⇒ $p_4 \rightarrow p_0$
[1,5,8] ⇒ $p_1 \rightarrow p_4$
[2,6,8] ⇒ $p_4 \rightarrow p_2$
[3,7,8] ⇒ $p_4 \rightarrow p_3$

⇒ red wrote first

Consensus for 5 processes



Hardware Primitives are Significant for Concurrent Data Structures



Concurrent Data Structures Need Scalable Strong Synchronization Primitives

Desired Features

- Scalable
- Universal
  - powerful enough to support any kind of synchronization (like CAS, LL/SC)
- Feasible
  - Easy to implement in hardware
- Easy-to-use in Algorithmic Design

Non-blocking Full/Empty Bit

Joined work with Phuong Ha and Otto Anshus

Non-blocking Full/Empty Bit

- Combinable operations
- Universal
- Feasible
  - Slight modification of a primitive that has been implemented in hardware
- Easy-to-use

A variant of the original FEB that always returns a value instead of waiting for a conditional flag

Test-Flag-and-Set

```
TFAS( x, v) {
    (o, flag_o) ← (x, flag_x);
    if flag_x = false then
        (x, flag_x) ← (v, true);
    end if
   return (o, flag_o);
}
```

Original FEB: Store-if-Clear-and-Set

```
SICAS(x,v) {
    Wait for flag_x to be false;
    (x, flag_x) ← (v, true);
}
```

A variant of the original FEB that always returns a value instead of waiting for a conditional flag

Test-Flag-and-Set

```
TFAS( x, v) {
    (o, flag_o) ← (x, flag_x);
    if flag_x = false then
        (x, flag_x) ← (v, true);
    end if
   return (o, flag_o);
}
```

Load

```
Load( x) {
   return (x, flag_x);
}
```

Store-And-Clear

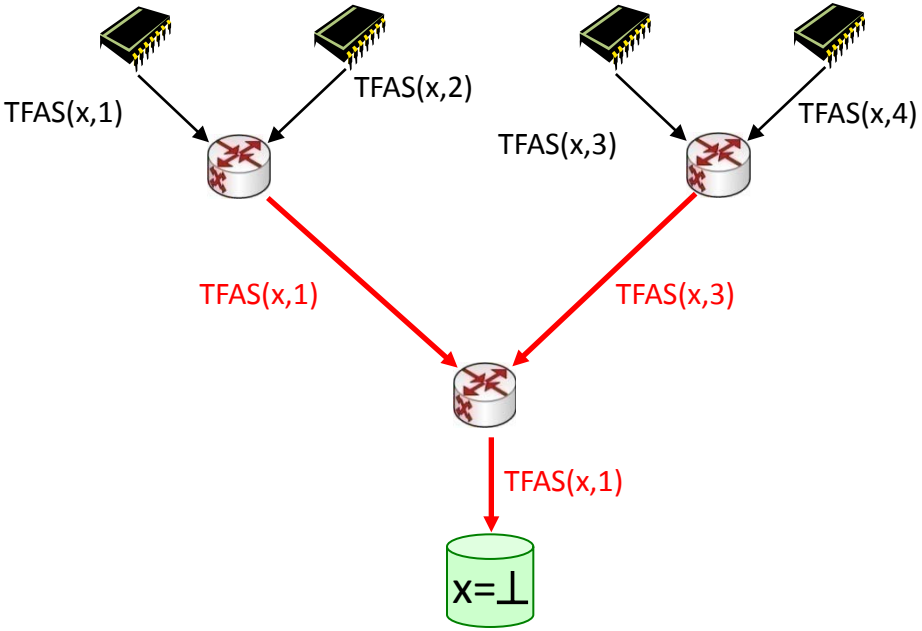
```
SAC( x, v) {
     (o, flag_o) ← (x, flag_x);
     (x, flag_x) ← (v, false);
    return (o, flag_o);
}
```

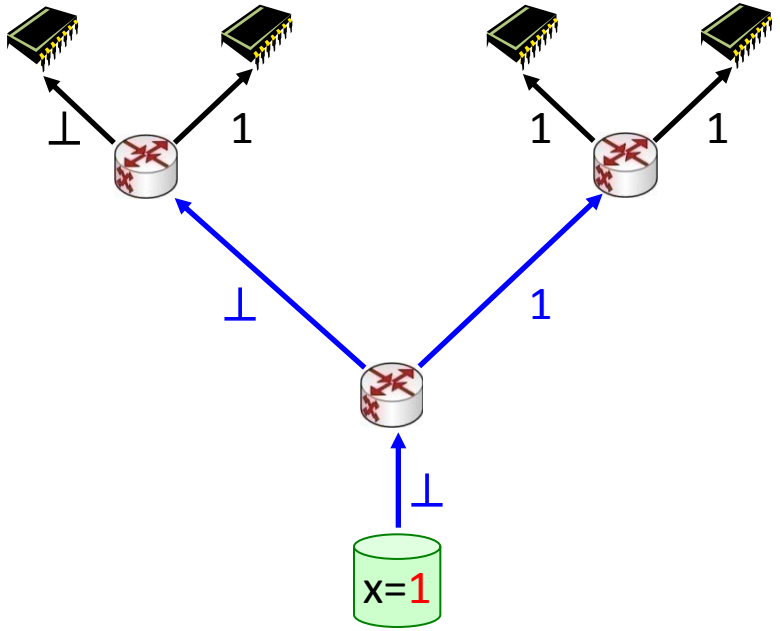
Store-And-Set

```
SAS( x, v) {
     (o, flag_o) ← (x, flag_x);
     (x, flag_x) ← (v, true);
    return (o, flag_o);
}
```

Combinability

- Key idea: Combinability
  - ⇒ eliminates contention & reduce load
  - Ex: TFAS





Note: CAS or LL/SC is not combinable

Conclusions

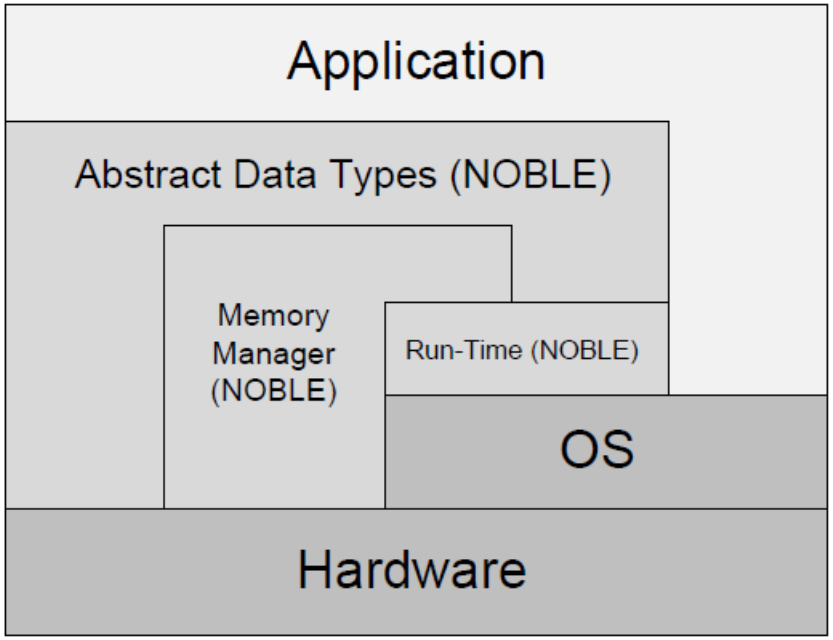
New algorithmic techniques that come from the introduction of new hardware features.

Core algorithmic design did not change when going from GP CPU to GPU.

Optimistic concurrency control works in manycore systems. Hard to derive worst case guarantees.

Scheduler part of the reference model?

Need to start a discussion with the architects about the abstractions/primitives that we want/need.


Application
Abstract Data Types (NOBLE)
Memory
Manager
(NOBLE)
Run-Time (NOBLE)
OS
Hardware

PEPPHER: PERFORMANCE PORTABILITY AND PROGRAMMABILITY FOR HETEROGENEOUS MANY-CORE ARCHITECTURES

This project is part of the portfolio of the
G.3 - Embedded Systems and Control Unit
Information Society and Media Directorate-General
European Commission

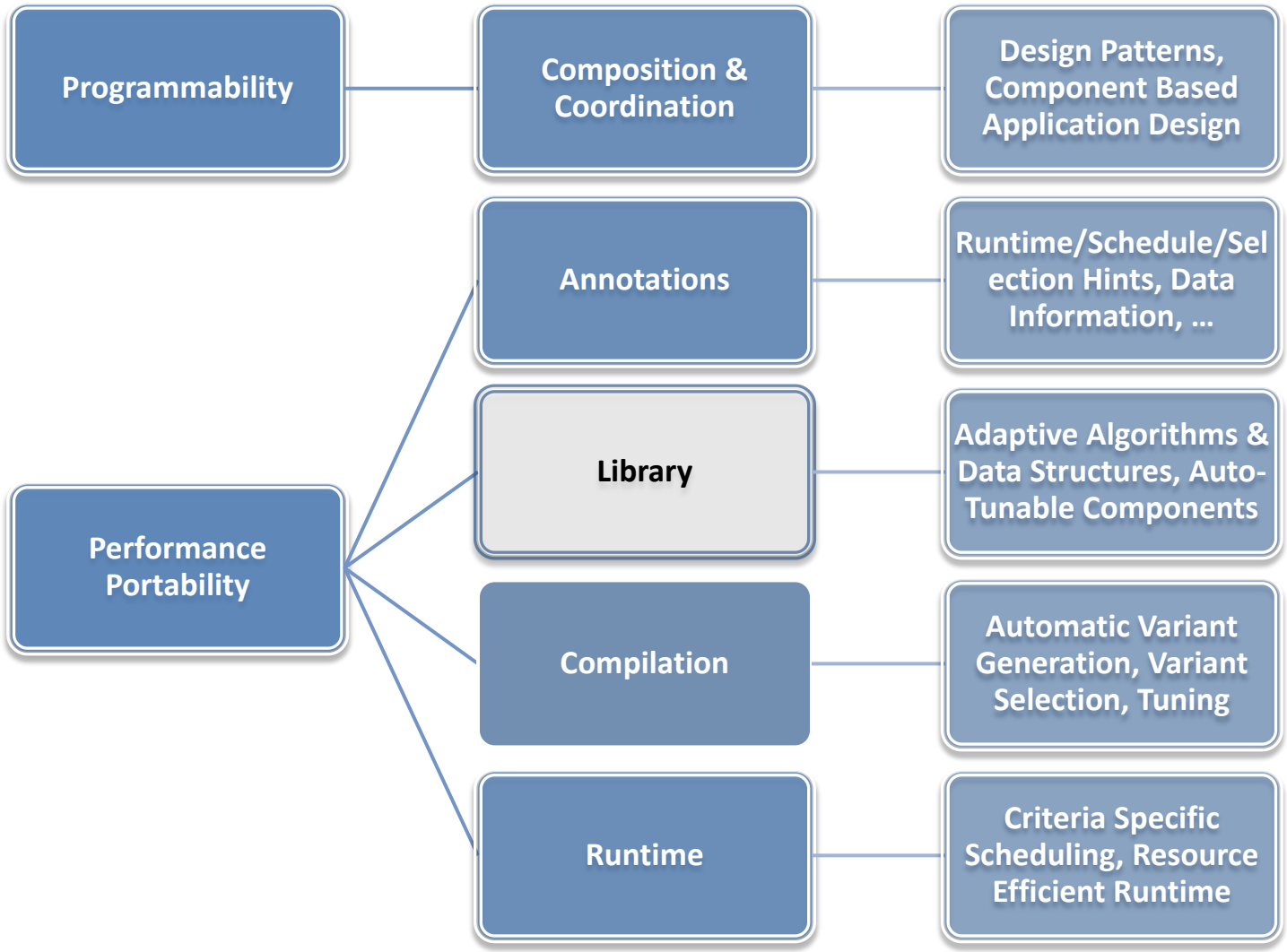
Contract Number: 248481
Total Cost [€]: 3.44 million
Starting Date: 2010-01-01
Duration: 36 months

Project Consortium

- University of Vienna (Coordinator), Austria
  - Siegfried Benkner, Sabri Pllana and Jesper Larsson Träff
- Chalmers University, Sweden
  - Philippas Tsigas
- Codeplay Software Ltd., UK
  - Andrew Richards
- INRIA, France
  - Raymond Namyst
- Intel GmbH, Germany
  - Herbert Cornelius
- Linköping University, Sweden
  - Christoph Kessler
- Movidius Ltd. Ireland
  - David Moloney
- Karlsruhe Institute of Technology, Germany
  - Peter Sanders

Programmability and Performance Portability

PEPPHER



Thank You!