Software Managed Coherency on SCC

Sai Luo, Xiaocheng Zhou, Tiger Hu Chen,
Shoumeng Yan, Ying Gao
Intel Labs China

Wei Liu, Brian Lewis, Bratin Saha
Programming Systems Lab

Intel Labs Single-chip Cloud Computer Symposium
March 16, 2010

Revive an old topic: cache coherence ?

- Software-managed coherence was a popular topic
  - Been around at least a couple of decades
  - Mostly targeting multiprocessors or clusters of workstations
- World is changing
  - Many cores on a single die
  - Much higher bandwidth and lower latency
  - Running out of power budget
- World is *not* changing
  - Legacy code written in shared memory programming model
  - Coherent memory requirement from ISVs

What is the right trade-off: HW vs. SW?

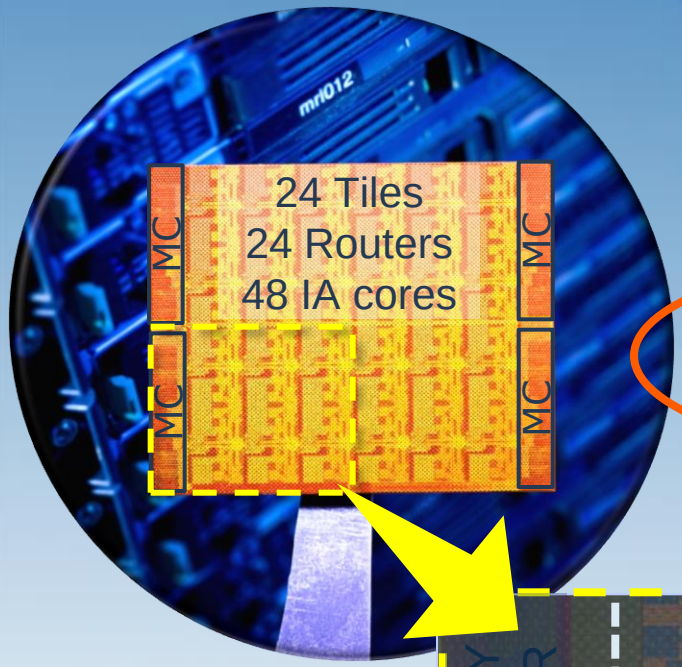
Why Software-Managed Coherency?

(Why not hardware)

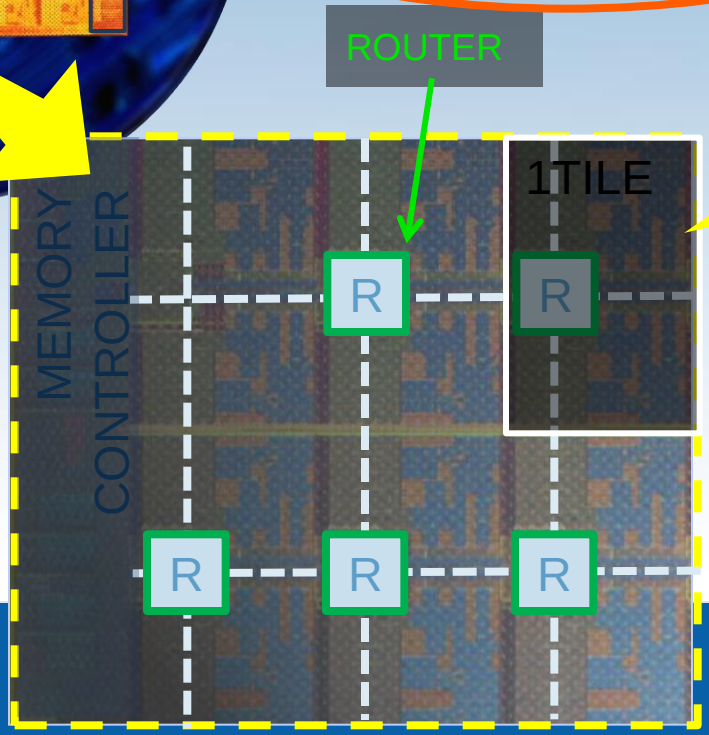
- No or minimal hardware!
  - Limited power budget on many-core
  - High complexity and validation effort to support hardware cache coherence protocol
- Flexibility: Dynamic reconfigurable coherency domains
  - Multiple applications running in separate coherency domains
  - Good match to SCC
  - Enable more optimizations: load balancing etc.
- Emerging applications
  - Most data are RO-shared, few are RW-shared
  - Coarse-grained synchronization: Map-Reduce, BSP, etc

SW-managed coherency can achieve comparable performance

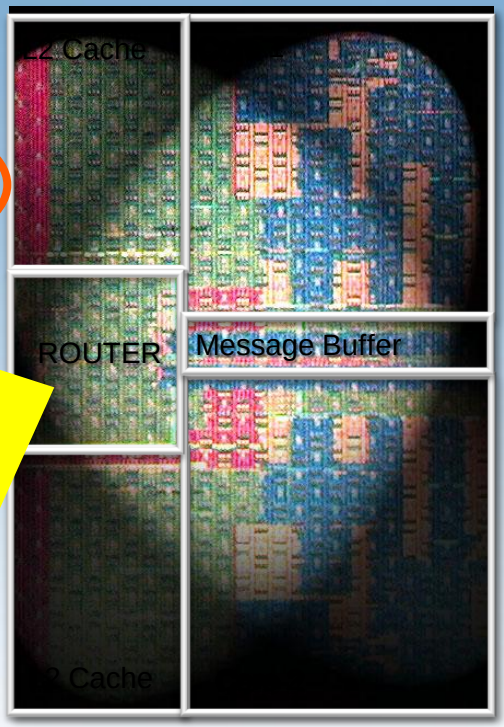
SCC architecture, a brief overview



- 45nm Hi-K metal-gate silicon
- 48 IA cores
- 6x4 2D mesh network
- 4 DDR3 memory controllers
- On-die message buffers
- **No hardware cache coherency**



Dual-core Tile



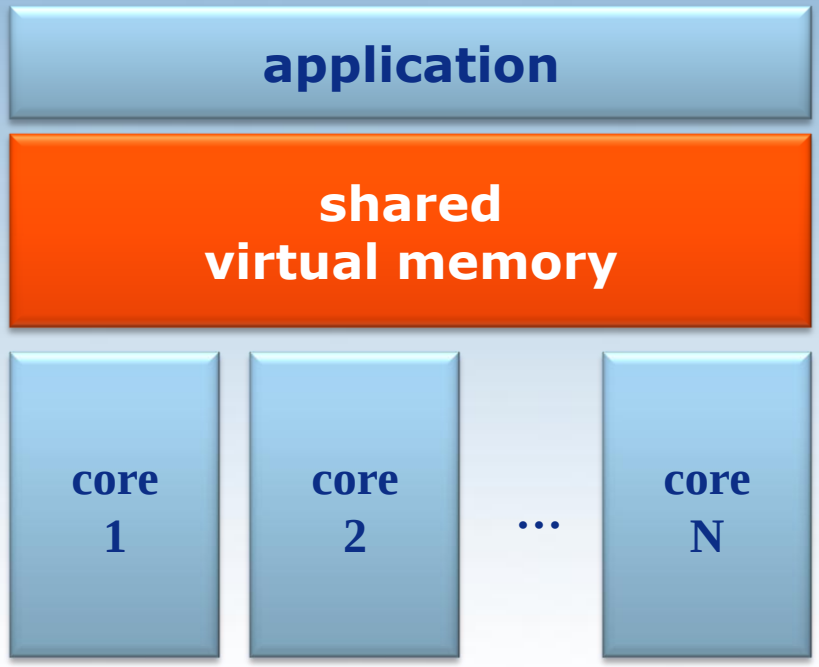
Outline

- Motivation
- Overview of SW managed coherence
- Implementation and Optimizations
- Our results
- Challenges for future research

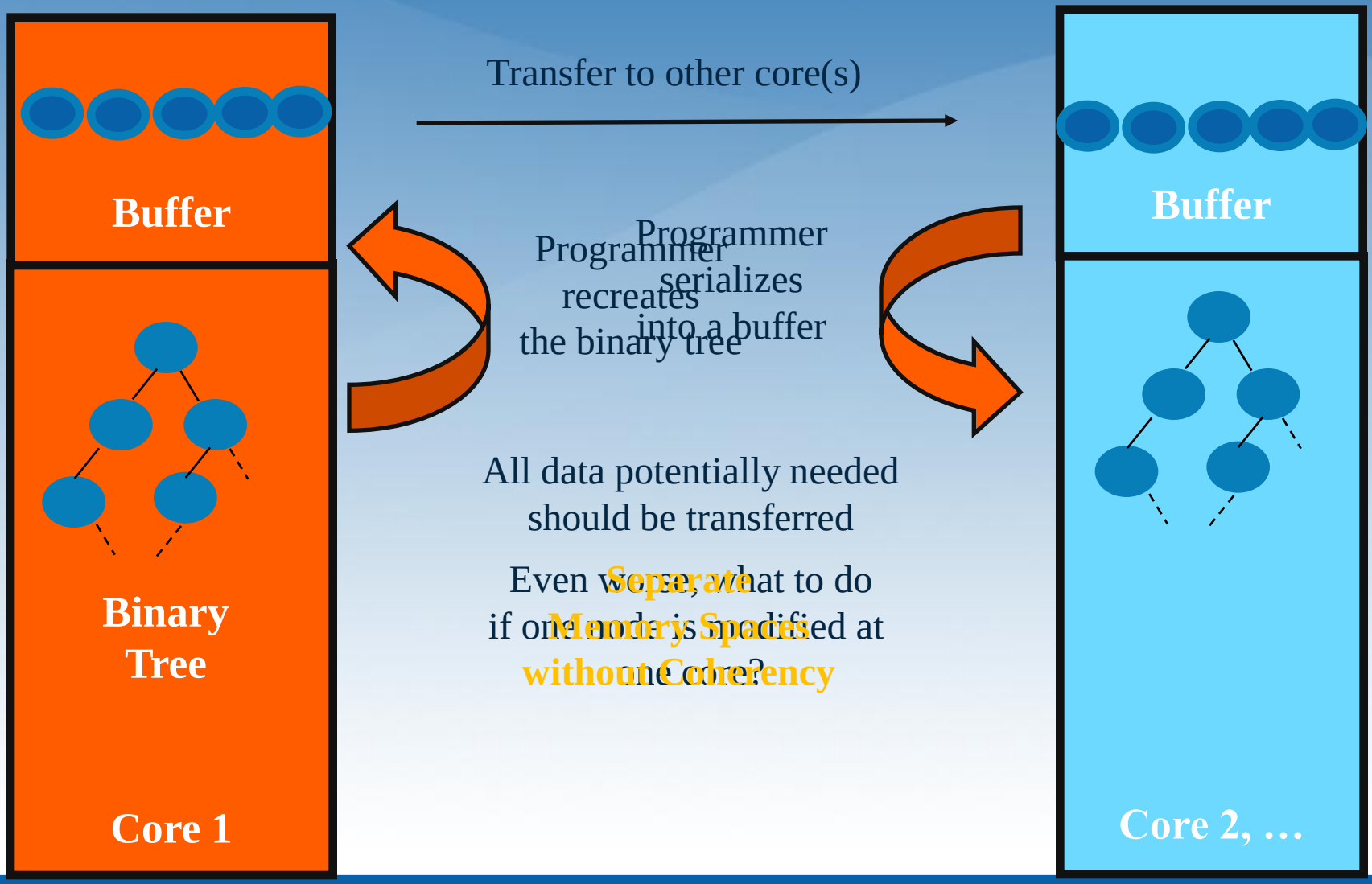
Overview

- Shared virtual memory can be used to support coherency
  - Similar to DSM
  - A single shared memory view to applications
  - Seamlessly sharing data structure and pointers among multiple cores
- No special HW support is needed.

Cores in SCC have separate address spaces



Why Shared Virtual Memory?



All data potentially needed should be transferred

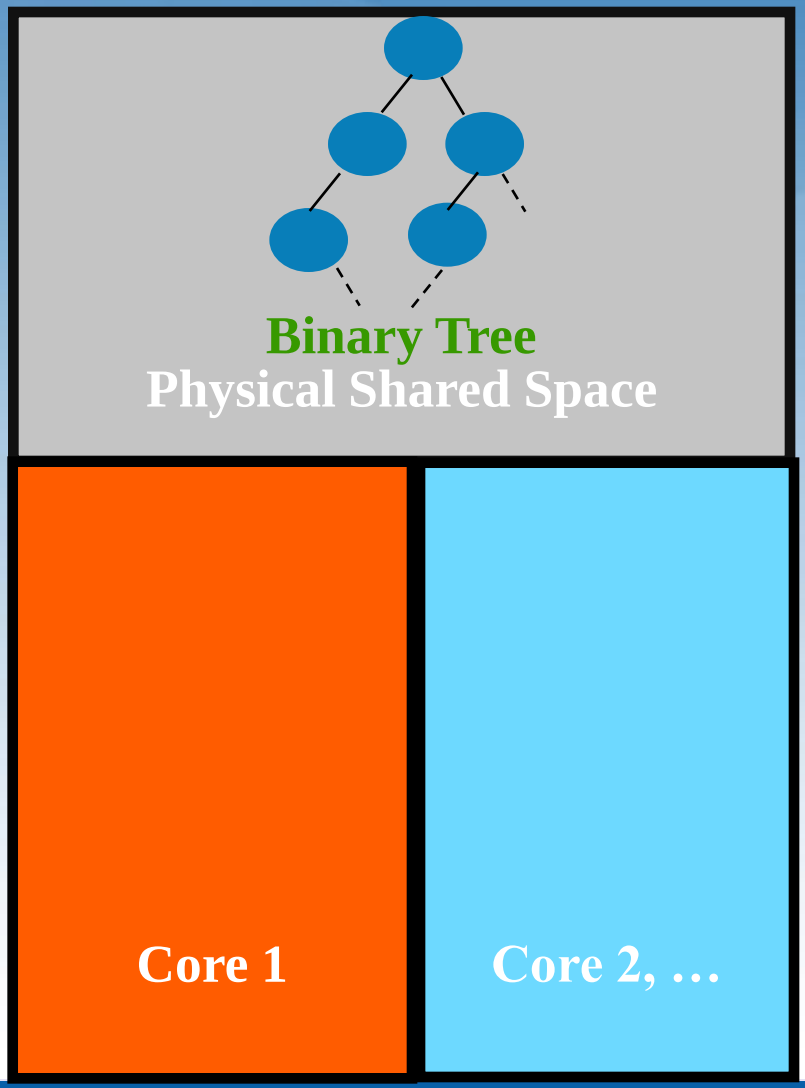
Even worse, what to do if one node is modified at one core?

Separate Memory Spaces without Coherency

82

Why Shared Virtual Memory? (Cont.)

Or



Explicit data management goes away

Only data really needed are accessed

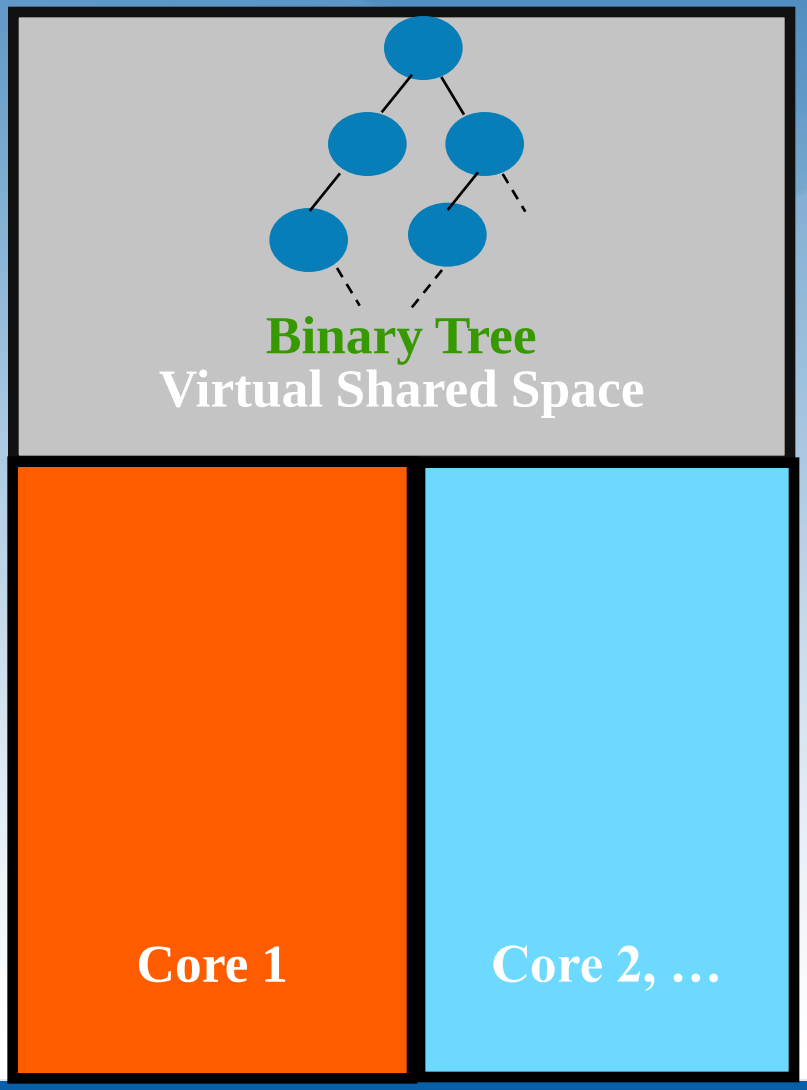
But:
SCC has no hardware cache coherency,
So the shared space must not be cached

It is a performance hit

83

Why Shared Virtual Memory? (Cont.)

How about



Shared data are allocated in the shared virtual address space

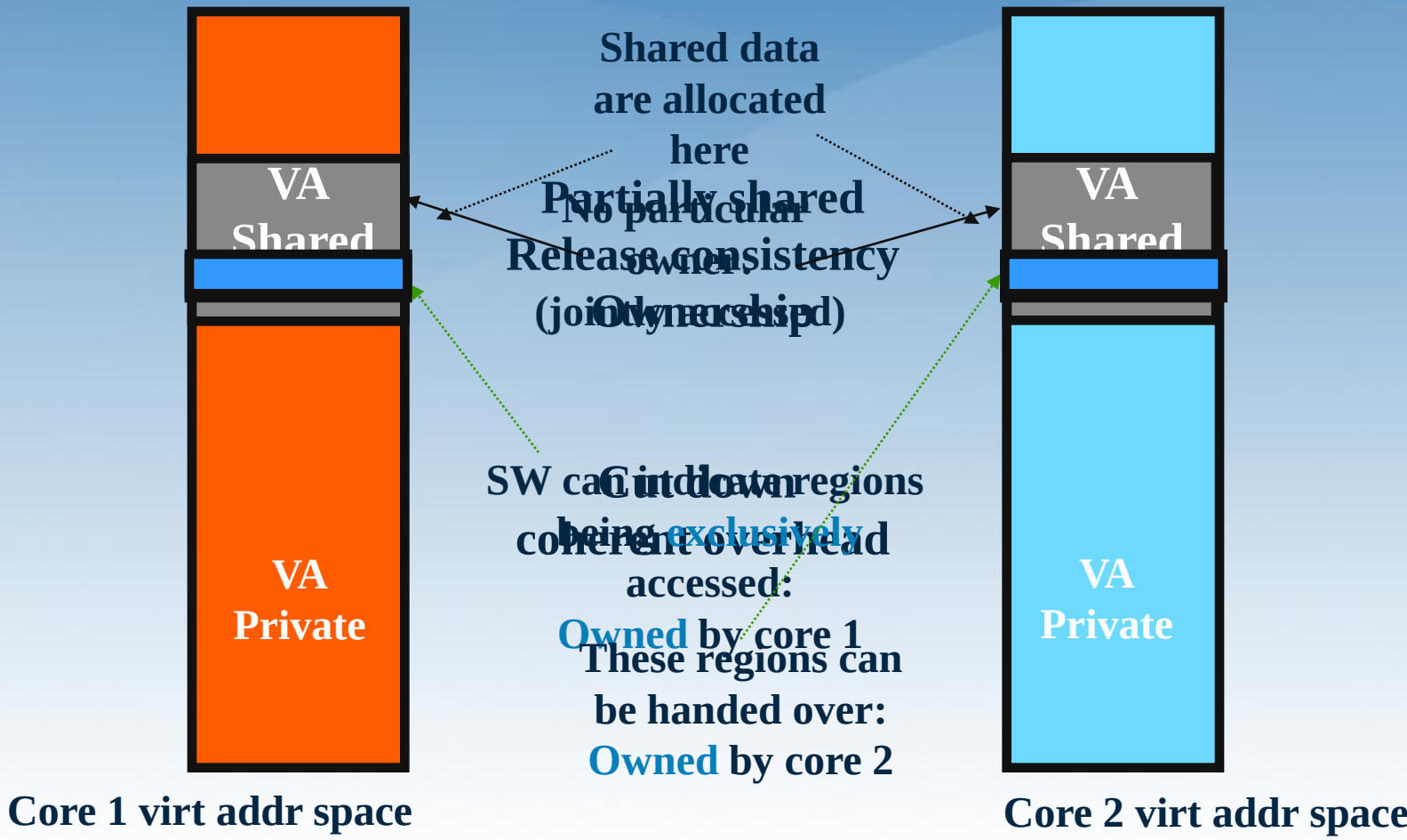
They are cacheable (higher performance)

Data coherency are managed by software

Uses don't care about where the data locate and how many copies exist

84

Shared Virtual Memory Model



85

Language and Compiler Support

- New "shared" type qualifier
  - *shared* int a; //a shared variable
  - *shared* int* pa; //a pointer to a shared int
  - *shared* int* *shared* pb; //a shared pointer to a shared int

- Static checking rules enforced by the compiler

  - No sharing between stack variables
    - foo() {*shared* int c;}

  - Shared pointer can't point to private data
    - int* *shared* pc;
  - And more on pointer assignment and casting etc.

Runtime Support

- Partial sharing on page-level
  - Only those actually shared are subjected to consistency maintenance
- Release consistency model
  - Consistency only guaranteed at the sync points (release, acquire)
    - Significantly reduce coherence traffic
  - Many applications already follow RC model
    - E.g. sync points: pthread_create, mutex, barrier, …
    - Release/Acquire can be inserted automatically at these points
- Ownership rights
  - No coherence traffic until ownership changed
  - They are treated as hints (i.e. optimization opportunities)
    - Fault on touch: fault if touch something owned by others
    - Promote on touch: promote to "jointly accessible"

Object Collision Detection Example: Share Memory Approach

```
typedef shared struct Ball Ball;
struct Ball {
    Vector position, velocity;
    int area_id;
    Ball* next; // balls in the same area
};

Ball* areas[N];

void collision(Ball* all) {
    // do collision detection
    // and compute the new position/velocity
    ……
}
```

```
void simulate()
{
    for(i=0; i<N; i++)
        thd[i] = spawn(collision, areas[i]);
    for(i=0; i<N; i++)
        join(thd[i]);
    update_area_array();
}
```

- It's just like writing a pthread program
- Implicit sync points at spawn, join, the beginning and ending of collision()

Example: Message Passing Approach

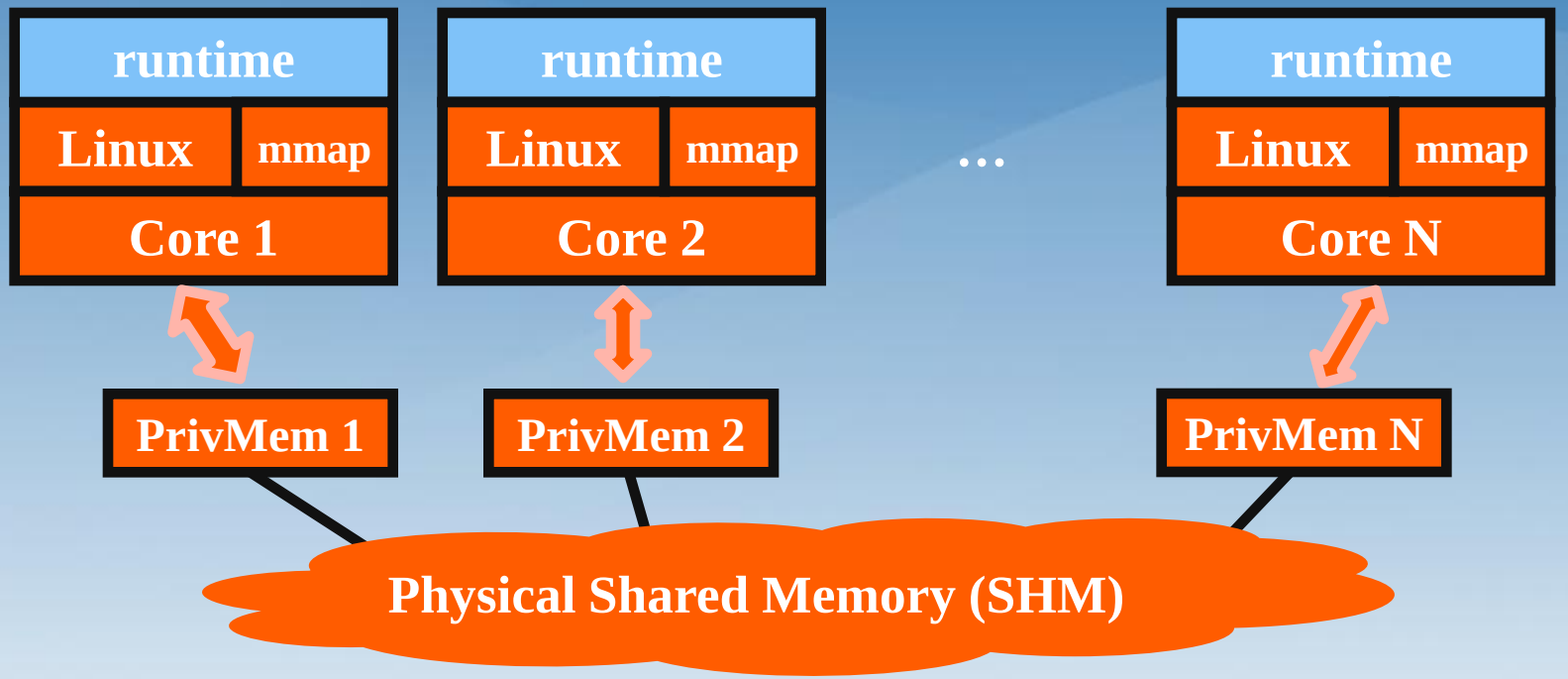
```
typedef struct Ball Ball;
struct Ball {
  Vector position, velocity;
  int area_id;
  Ball* next; // balls in the same area
};
Ball* areas[N];
int get_area_id(Ball* b) { …}
void collision(int id)
{
  // receive the data objects
  // and recreate the structure
  for(i=0; i<N; i++) {
    areas[i] = NULL;
    while(recv(id, buf)) {
      b = malloc(sizeof(Ball));
      *b = *buf;
      b->next = areas[i];
      areas[i] = b;
    }
  }
```

```
  // do collision detection
  // and compute the new pos/vel
  ……

  // send back new data
  // and free the local objects
  for(b=all; b; b=next) {
    new_id = get_area_id(b);
    send(new_id, b);
    next = b->next; free(b);
  }
}

void simulate()
{
  // spawn
  for(i=0; i<N; i++)
    thd[i] = spawn(collision, i);
  // send  data to the individual threads
  // and destroy the objects
```

```
  for(i=0; i<N; i++) {
    for(b=areas[i];b;b=next) {
      for(j=0; j<N; j++) send(j, b);
      next = b->next; free(b);
    }
  }

  // gather data back
  // and recreate the link list
  for(i=0; i<N; i++) {
    areas[i] = NULL;
    while(recv(id, buf)) {
      b = malloc(sizeof(Ball));
      *b = *buf;
      b->next = areas[i];
      areas[i] = b;
    }
  }
  // join
  for(i=0; i<N; i++)
    join(thd[i]);
}
```

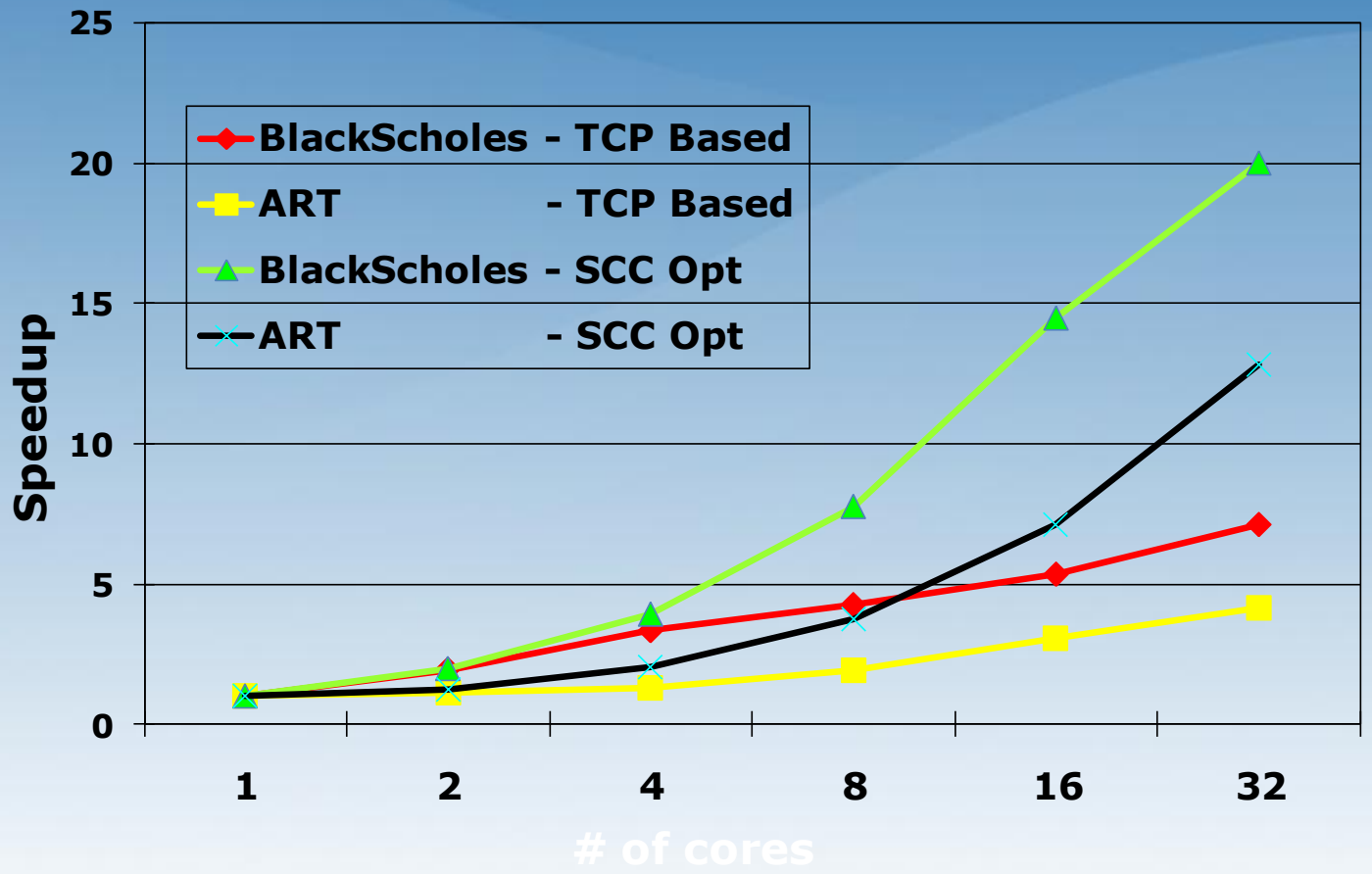
- Lots of code are spent in data serialization and reconstruction.
- Is error-prone and might dead-lock.
- All data are sent even not used.

Optimized implementation for SCC



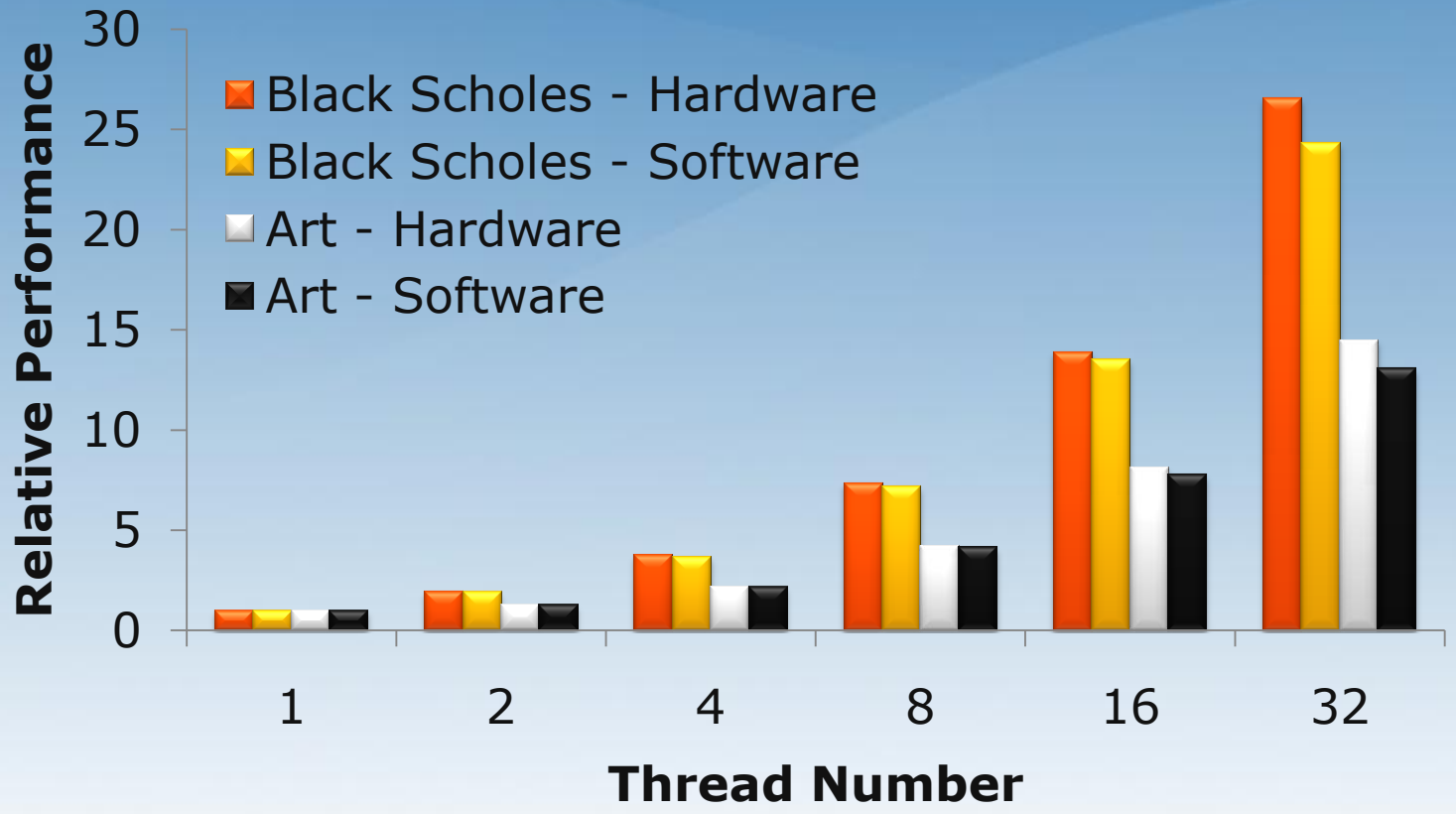
- Leverage shared memory (SHM) support in SCC
- Golden copy is saved at SHM, needn't communicate with any other nodes
- Do memcpy between cacheable private memory & uncacheable SHM

Scalability of both implementations on SCC



- Significantly improved scalability, up to 20X on 32 cores.
- More optimizations (WIP)

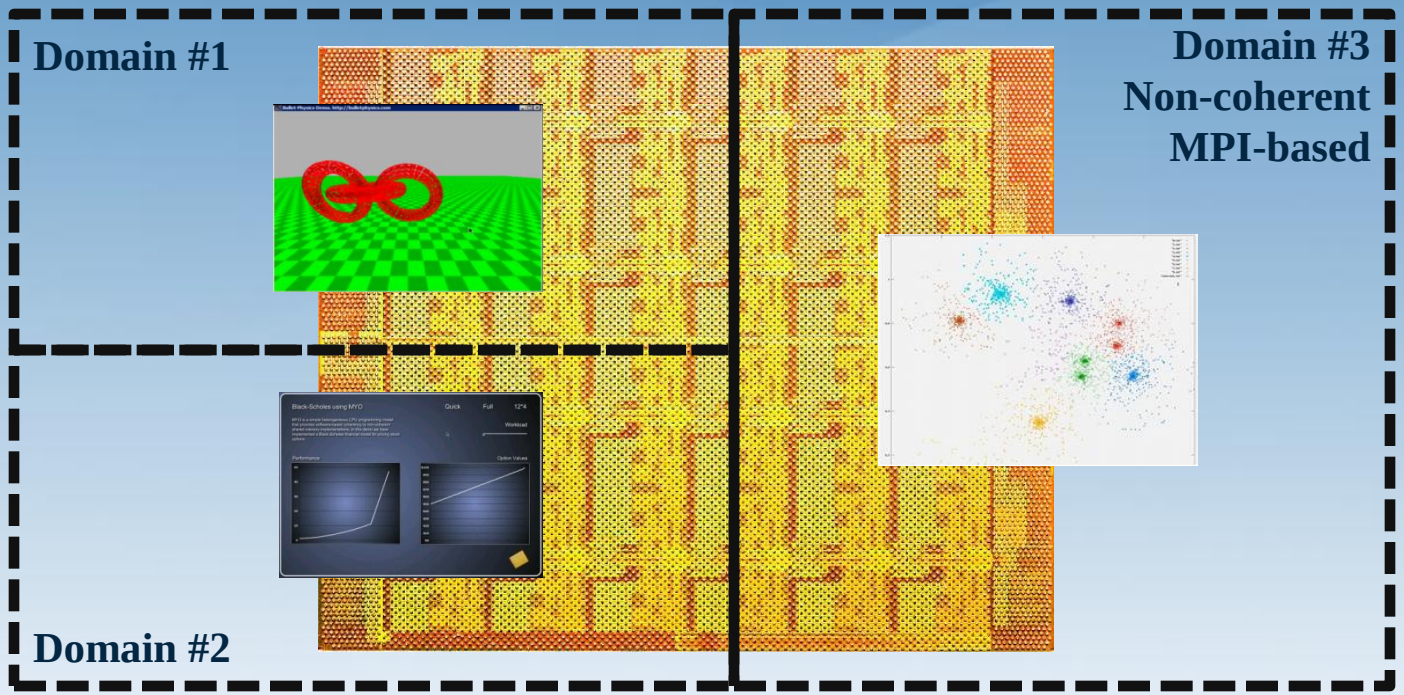
SW managed coherence vs. HW coherence
on 32way SMP server (process per core)



- Software managed coherency is as efficient as hardware cache coherency

Emerging usage models

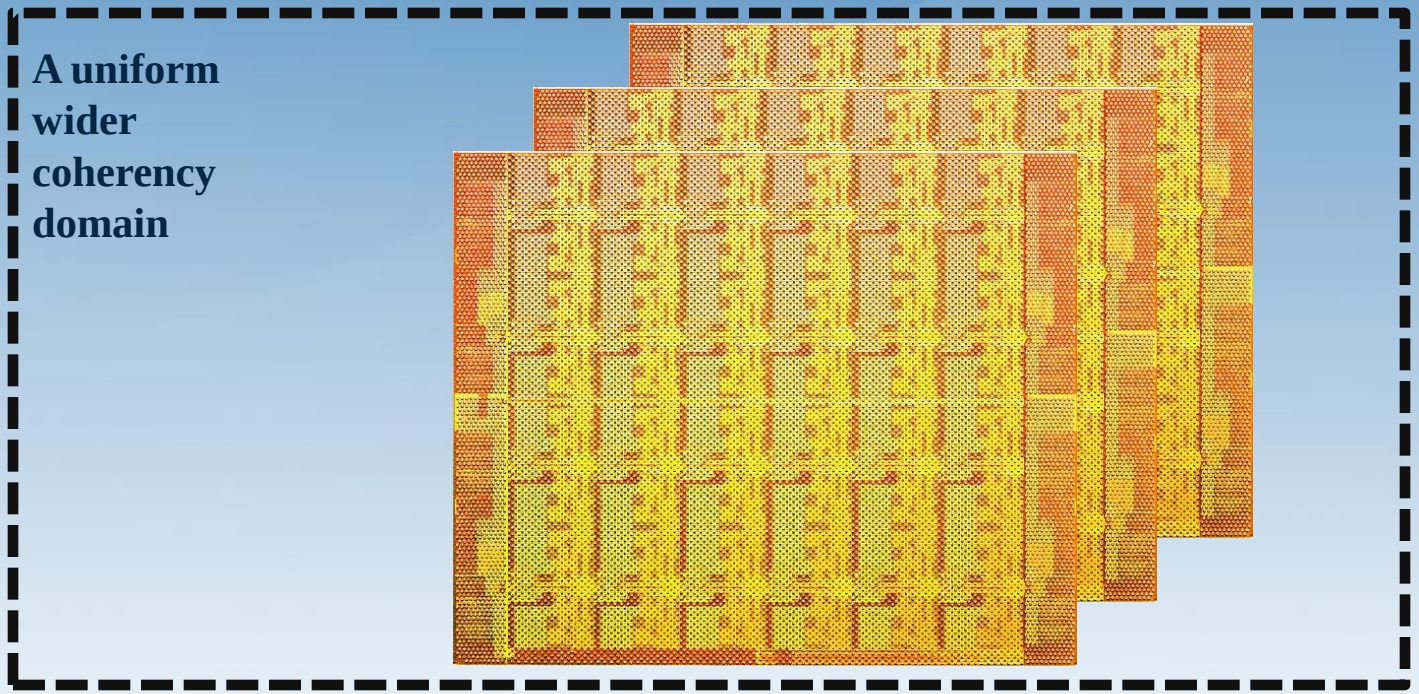
- Separated coherency domains



  - Whole system partitioned into multiple coherency domains
  - Dynamic reconfigurable
  - Mixed mode: share memory in one domain with MPI in others

Another usage models

- Multiple SCC chips



  - When an application is massively parallel, more SCC chips can be connected together to form a uniform wider coherency domain

Summary

- We believe software managed coherency on non-coherent many-core is the future trend
- A prototyped partially shared virtual memory system demonstrates it can be:
  - Easy to program
  - Comparable performance vs. hardware coherence
  - Adaptive to future advanced usage models
- Also opens new research opportunities

95

Challenges for future research

- This revived "software managed coherency" topic opens many "cold cases"
- What are the right software optimizations?
  - Prefetching, locality, affinity, consistency model
  - And more…
- What is the right hardware support?
- How do emerging workloads adapt to this?

Please contact us if you are interested in this topic.