Cornell University
Computer Systems Laboratory

Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems

Moyang Wang, Tuan Ta, Lin Cheng, Christopher Batten

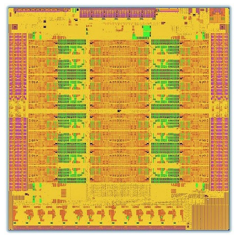
Computer Systems Laboratory
Cornell University

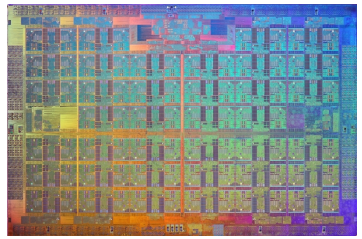
Manycore Processors

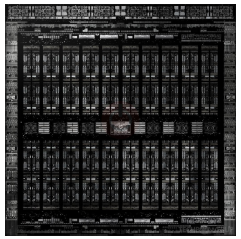
CSL

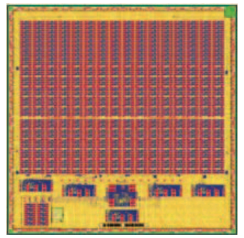
Small Core Count

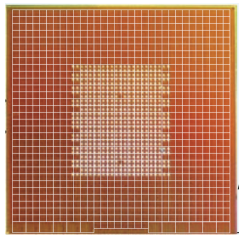
Large Core Count

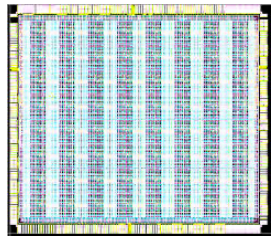












Cavium ThunderX	Tilera TILE64	Intel Xeon Phi	NVIDIA GV100 GPU	Celerity	KiloCore	Adapteva Epiphany
48 Cores	64 Cores	72 Cores	72 SM	511 Cores	1000 Cores	1024 Cores

Hardware-Based Cache Coherence

Software-Centric Cache Coherence / No Coherence

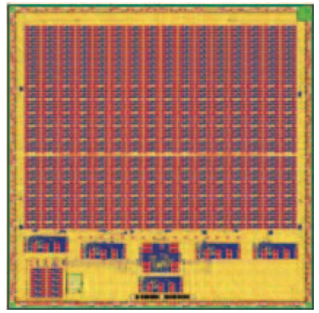
Software Challenge

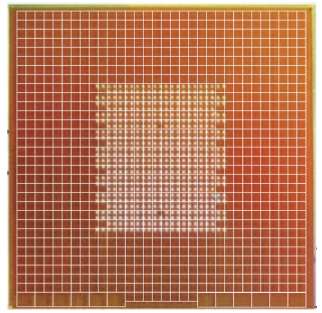
Intel TBB

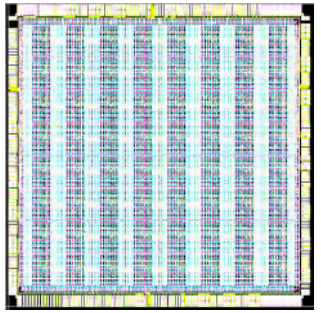
```
int fib( int n ) {
  if ( n < 2 ) return n;
  int x, y;
  tbb::parallel_invoke(
    [&] { x = fib( n - 1 ); },
    [&] { y = fib( n - 2 ); }
  );
  return (x + y);
}
```

Intel Cilk Plus

```
int fib( int n ) {
  if ( n < 2 ) return n;
  int x = cilk_spawn fib( n - 1 );
  int y = fib( n - 2 );
  cilk_sync;
  return (x + y);
}
```







- Programmers expect to use familiar shared-memory programming models on manycore processors
- Even more difficult to allow cooperative execution between host processor and manycore co-processor

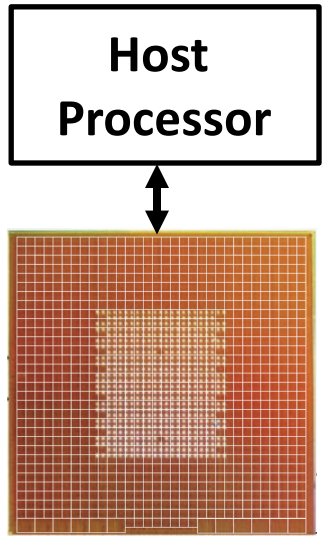
Software Challenge

Intel TBB

```
int fib( int n ) {
  if ( n < 2 ) return n;
  int x, y;
  tbb::parallel_invoke(
    [&] { x = fib( n - 1 ); },
    [&] { y = fib( n - 2 ); }
  );
  return (x + y);
}
```

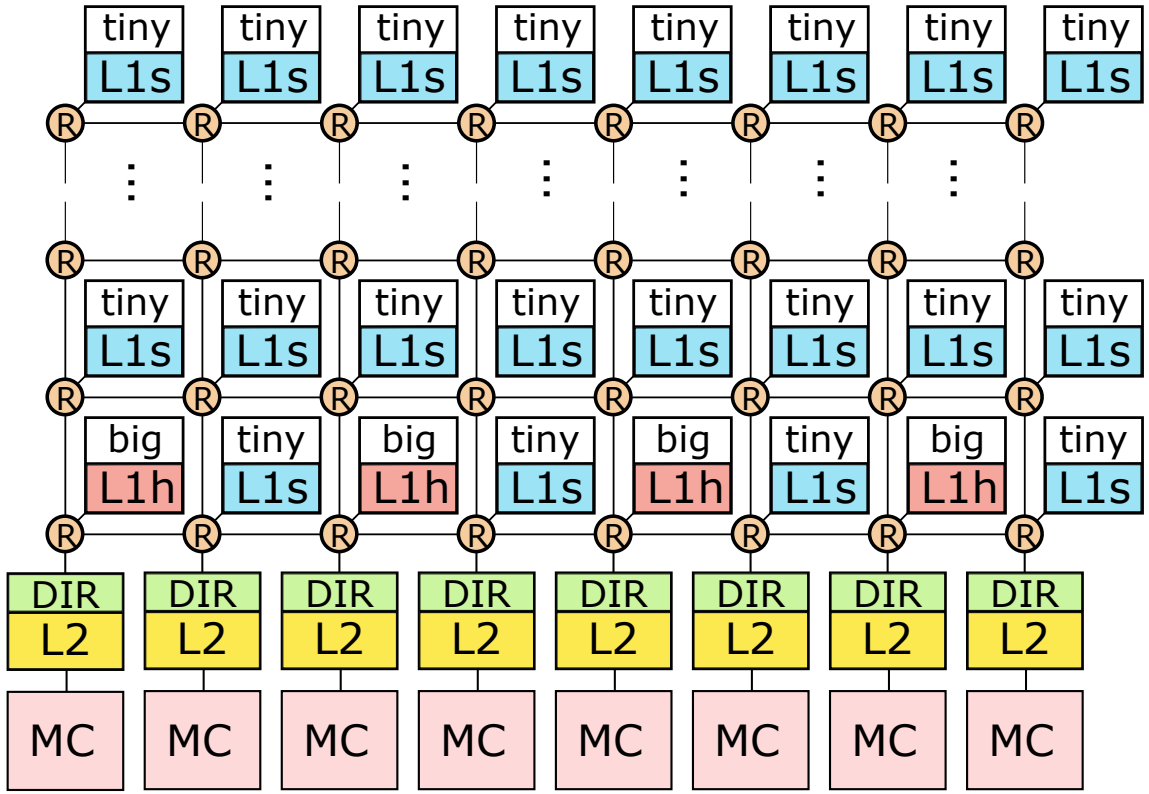
Intel Cilk Plus

```
int fib( int n ) {
  if ( n < 2 ) return n;
  int x = cilk_spawn fib( n - 1 );
  int y = fib( n - 2 );
  cilk_sync;
  return (x + y);
}
```



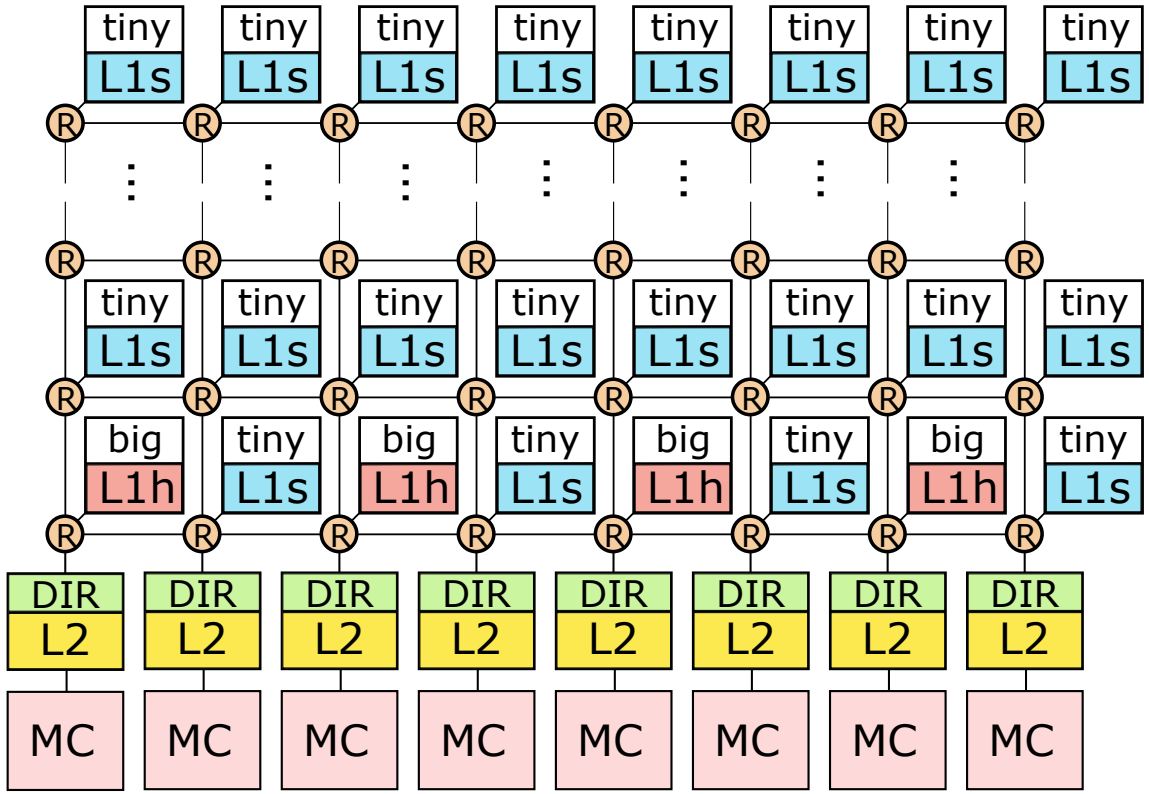
- Programmers expect to use familiar shared-memory programming models on manycore processors
- Even more difficult to allow cooperative execution between host processor and manycore co-processor

CONTRIBUTIONS



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Work-Stealing Runtime for manycore processors with heterogeneous cache coherence (HCC)
  - TBB/Cilk-like programming model
  - Efficient cooperative execution between big and tiny cores
- Direct task stealing (DTS), a lightweight software and hardware technique to improve performance and energy efficiency
- Detailed cycle-level evaluation



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- **Background**
- Implementing Work-Stealing Runtimes on HCC
- Direct Task Stealing
- Evaluation

Heterogeneous Cache Coherence (HCC)

- We study three exemplary software-centric cache coherence protocols:
  - DeNovo [1]
  - GPU Write-Through (GPU-WT)
  - GPU Write-Back (GPU-WB)
- They vary in their strategies to invalidate stale data and propagate dirty data
- Prior work on Spandex [2] has studied how to efficiently integrate different protocols into HCC systems

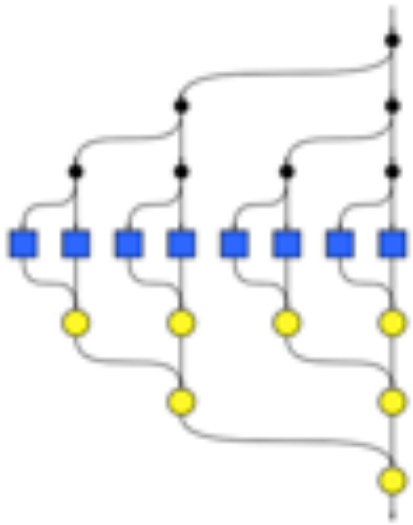
	Stale Data Invalidation	Dirty Data Propagation	Write Granularity
MESI	Writer	Owner, Write-Back	Cache Line
DeNovo	Reader	Owner, Write-Back	Flexible
GPU-WT	Reader	No-Owner, Write-Through	Word
GPU-WB	Reader	No-Owner, Write-Back	Word

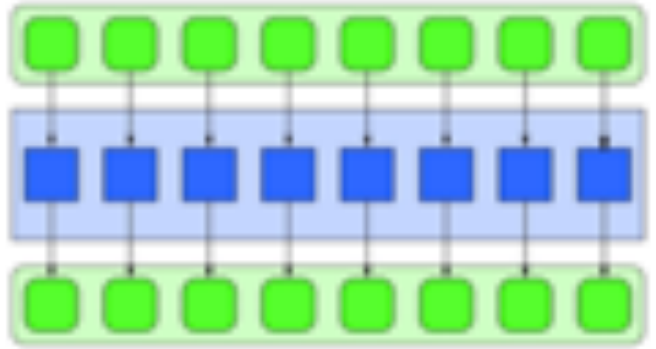
[1] H. Sung and S. V. Adve. DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations. ASPLOS 2015.

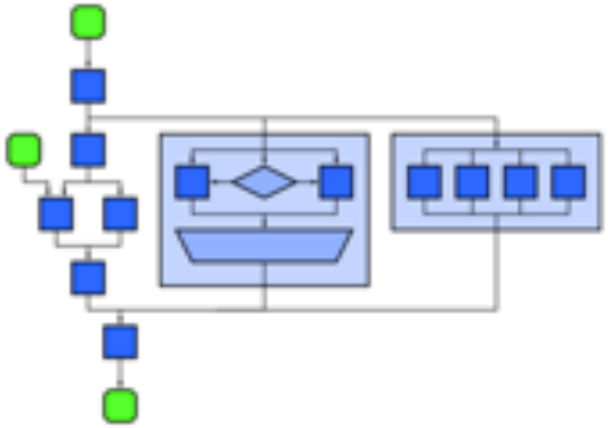
[2] J. Alsop, M. Sinclair, and S. V. Adve. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. ISCA 2018.

Dynamic Task Parallelism

- Tasks are generated dynamically at run-time
- Diverse current and emerging parallel patt
  - Map (for-each)
  - Fork-join
  - Nesting
- Supported by popular frameworks:
  - Intel Threading Building Blocks (TBB)
  - Intel Cilk Plus
  - OpenMP
- Work-stealing runtimes provide automatic load-balancing







Pictures from Robinson et al., *Structured Parallel Programming: Patterns for Efficient Computation*, 2012

Dynamic Task Parallelism

- Tasks are generated dynamically at run-time
- Diverse current and emerging parallel patterns:
  - Map (for-each)
  - Fork-join
  - Nesting
- Supported by popular frameworks:
  - Intel Threading Building Blocks (TBB)
  - Intel Cilk Plus
  - OpenMP
- Work-stealing runtimes provide automatic load-balancing

```
long fib( int n ) {
  if ( n < 2 ) return n;
  long x, y;
  parallel_invoke(
    [&] { x = fib( n - 1 ); },
    [&] { y = fib( n - 2 ); }
  );
  return (x + y);
}
```

```
void vvadd( int a[], int b[], int dst[],
            int n ) {
  parallel_for( 0, n, [&]( int i ) {
    dst[i] = a[i] + b[i];
  });
}
```

Dynamic Task Parallelism

- Tasks are generated dynamically at run-time
- Diverse current and emerging parallel patterns:
  - Map (for-each)
  - Fork-join
  - Nesting
- Supported by popular frameworks:
  - Intel Threading Building Blocks (TBB)
  - Intel Cilk Plus
  - OpenMP
- Work-stealing runtimes provide automatic load-balancing

```
class FibTask : public task {
  int n, *sum;

  void execute() {
    if ( n < 2 ) {
      *sum = n;
      return;
    }
    long x, y;
    FibTask a( n - 1, &x );
    FibTask b( n - 2, &y );
    this->reference_count = 2;
    task::spawn( &a );
    task::spawn( &b );
    task::wait( this );
    *sum = x + y;
  }
}
```

Work-Stealing Runtimes

Check local task queue

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

Work-Stealing Runtimes

Check local task queue

Execute dequeued task

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

Work-Stealing Runtimes

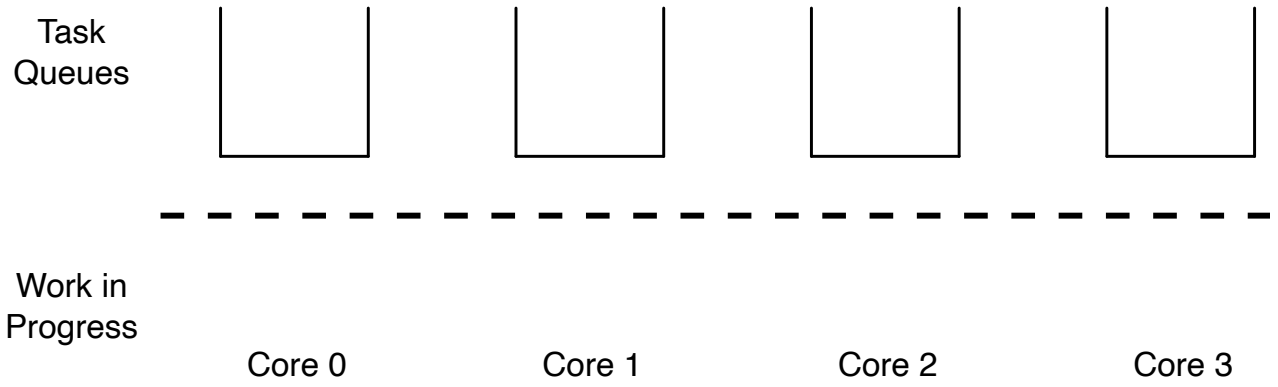
Check local task queue

Execute dequeued task

Steal from another queue

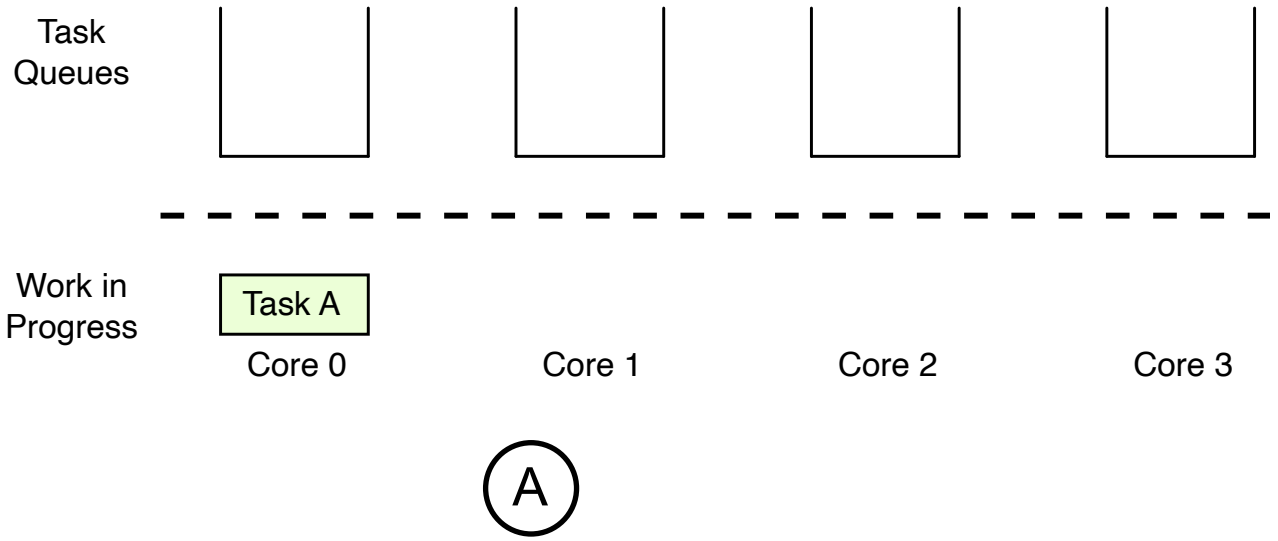
```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

Work-Stealing Runtimes



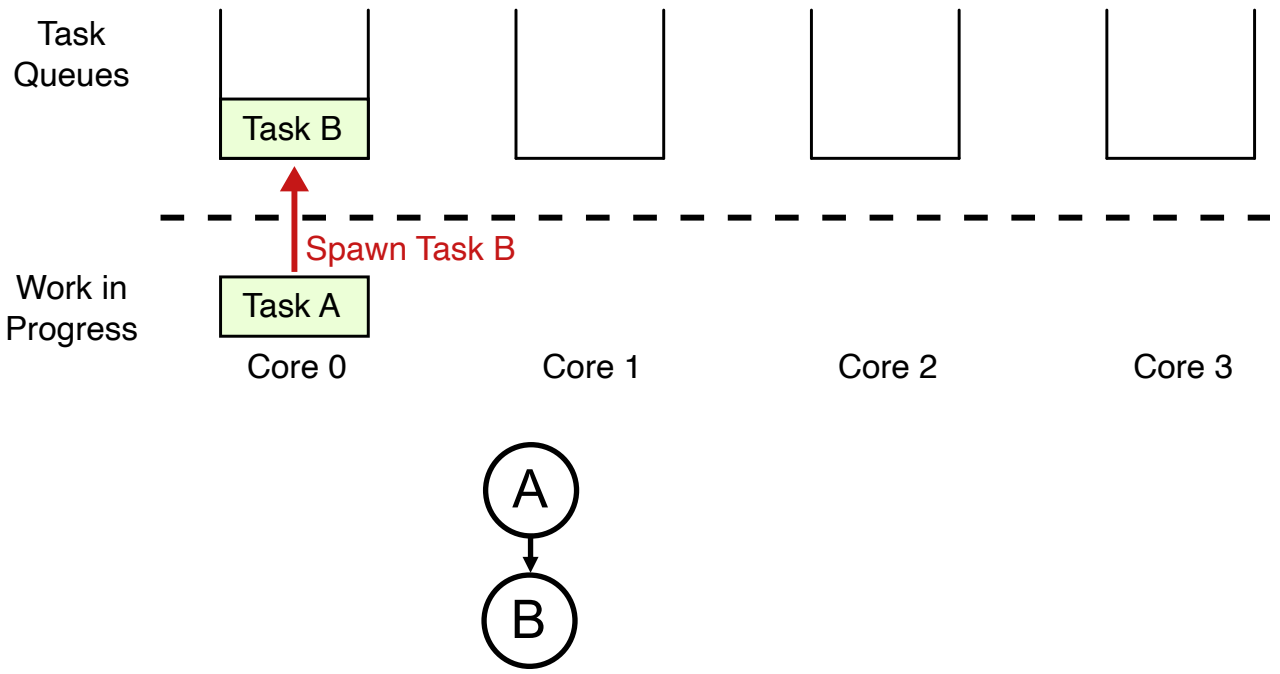

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

Work-Stealing Runtimes



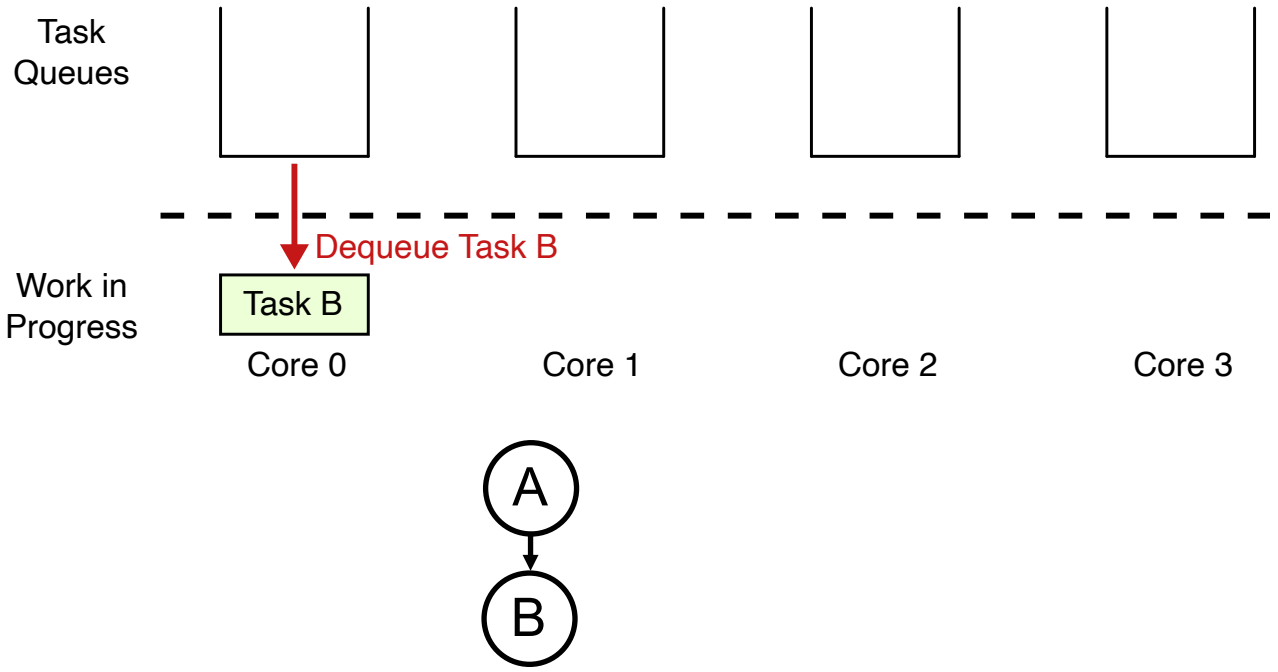

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

Work-Stealing Runtimes



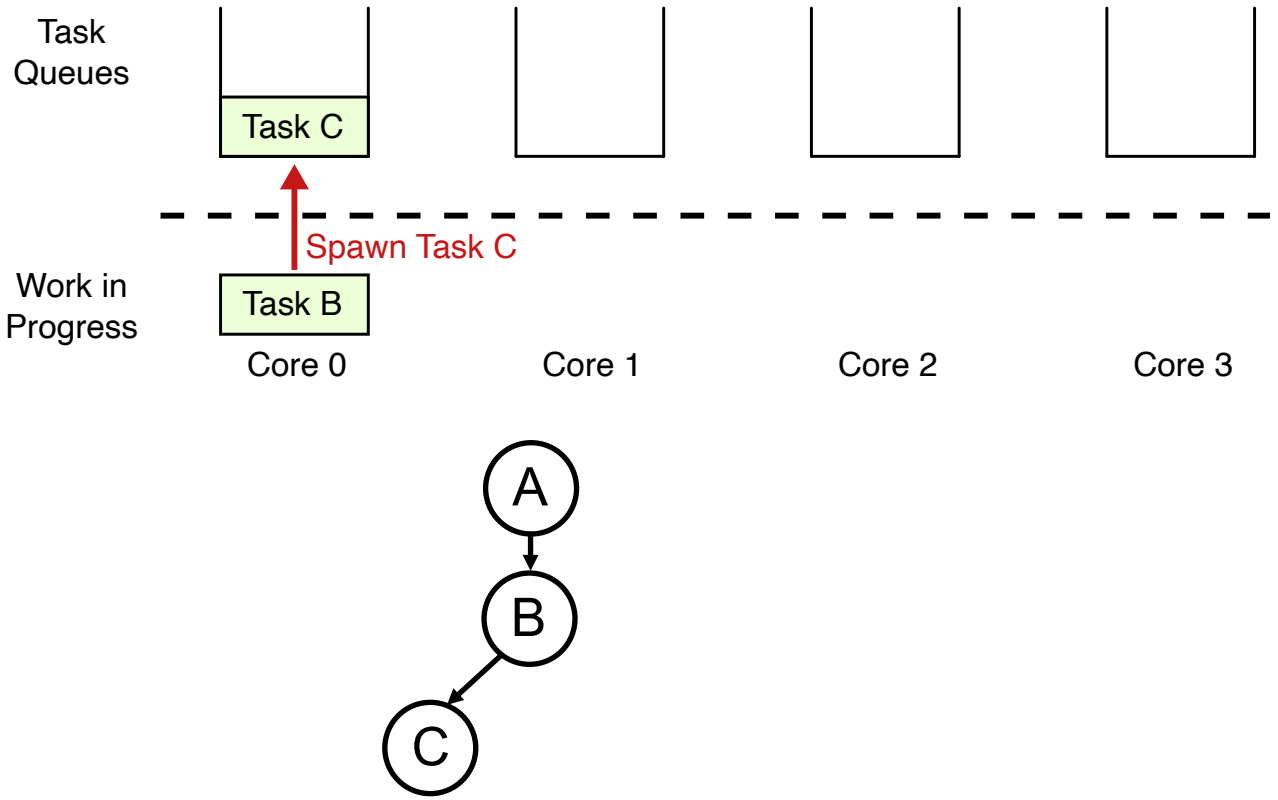

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

Work-Stealing Runtimes



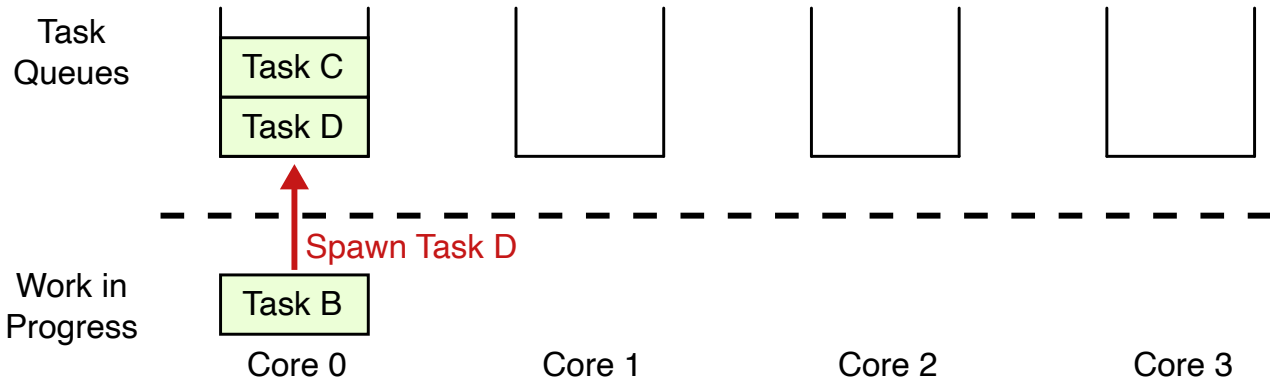

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

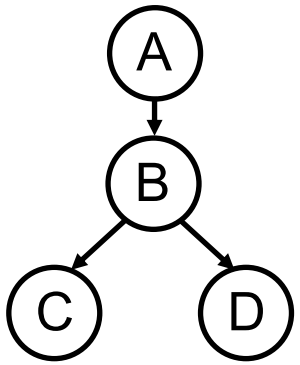
Work-Stealing Runtimes




```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

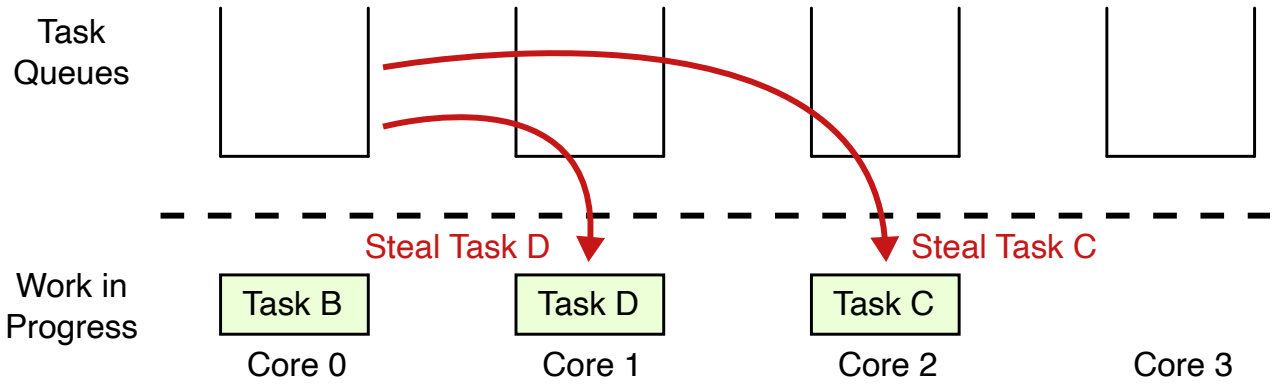
Work-Stealing Runtimes

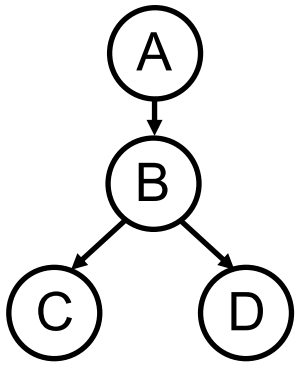





```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

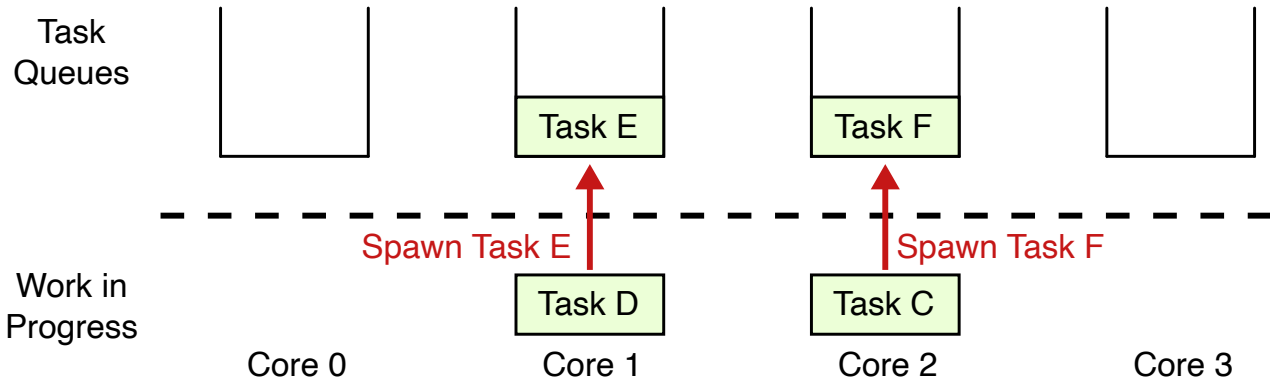
Work-Stealing Runtimes

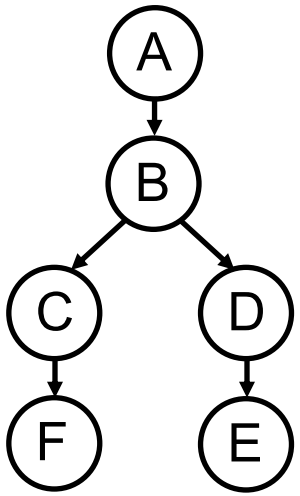





```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

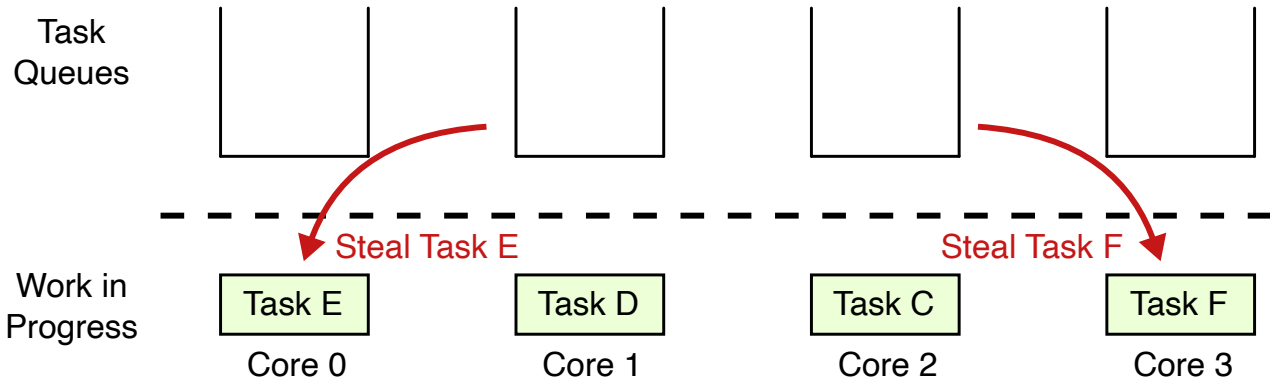
Work-Stealing Runtimes

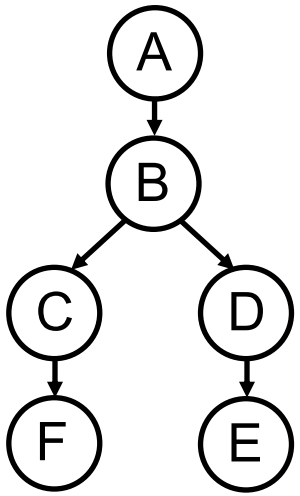




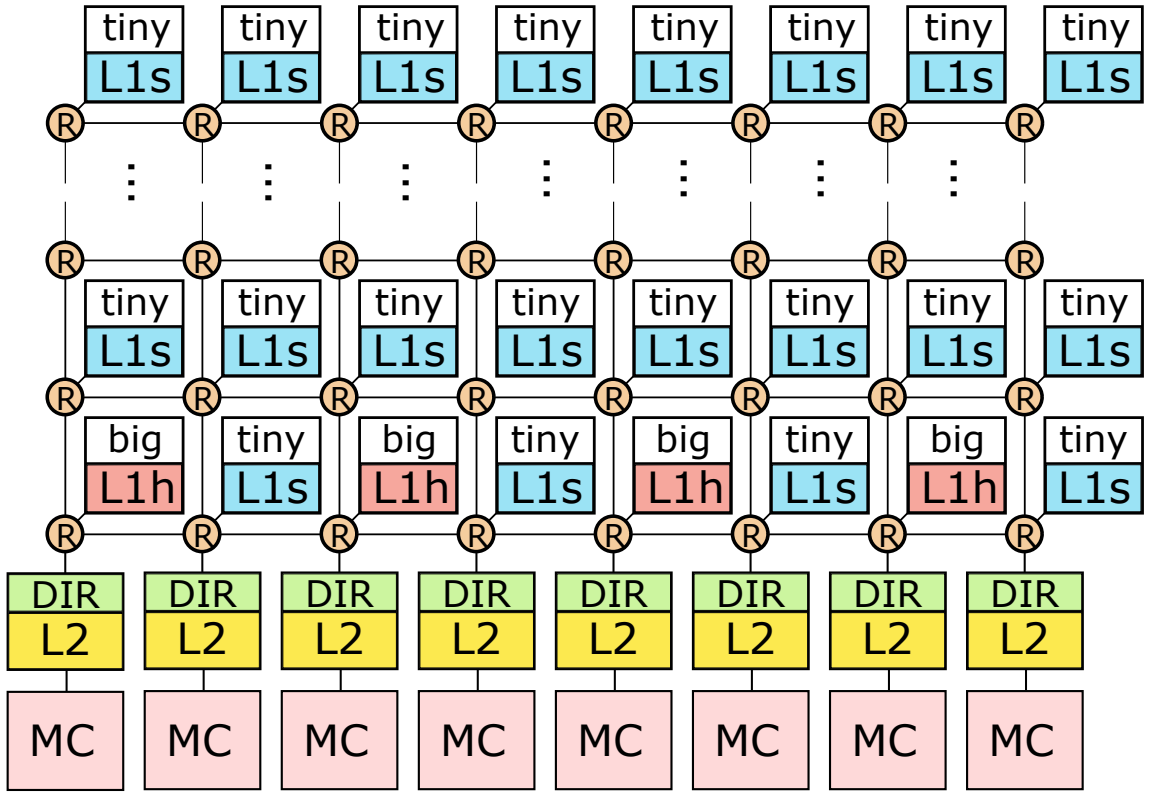

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

Work-Stealing Runtimes



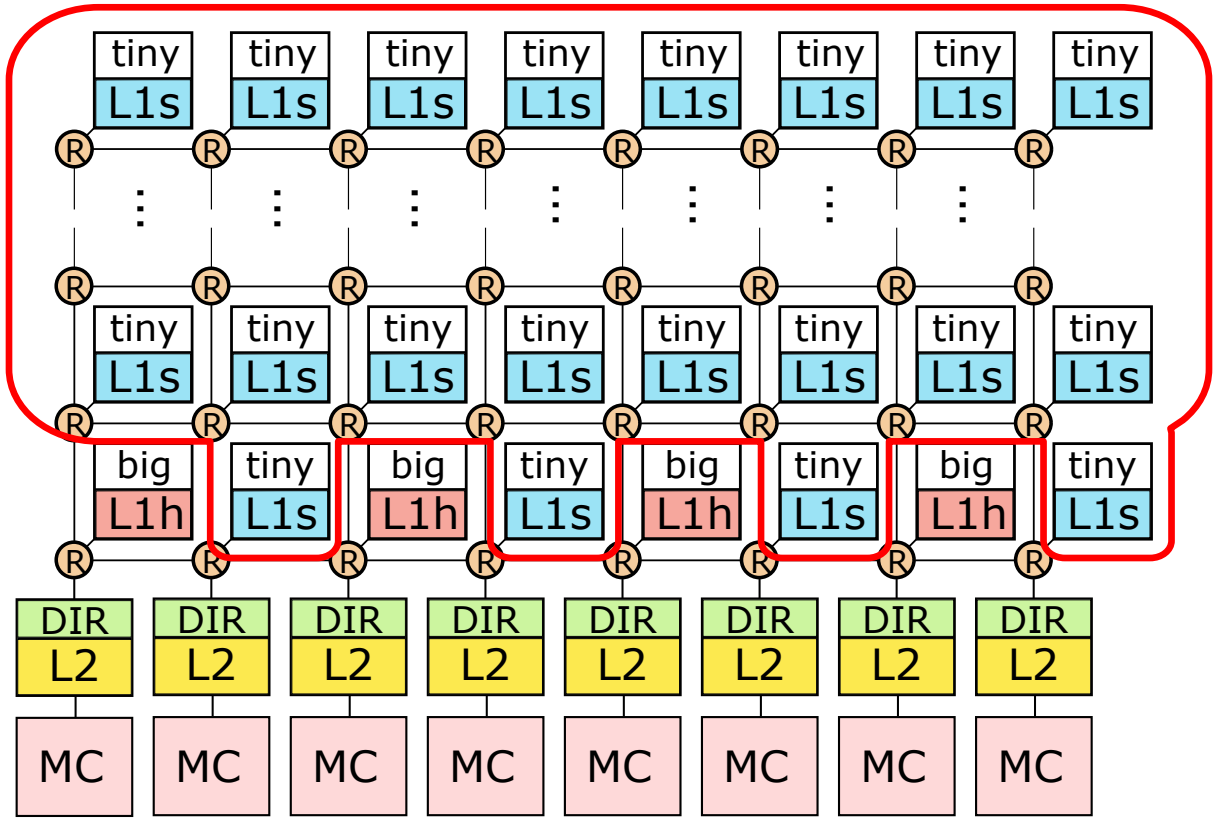



```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Background
- **Implementing Work-Stealing Runtimes on HCC**
- Direct Task Stealing
- Evaluation



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Shared task queues must be coherent
- DAG-Consistency [1]:
  - Child tasks read up-to-date data from parent
  - Parent read up-to-date data from (finished) children

[1] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. SPAA 1996.

Work-Stealing Runtimes on Software-Centric Cache Coherence

- Supporting shared queues:
  - Lock-acquire -> invalidation
  - Lock-release -> cache flush

```
                                    void task::wait( task* p ) {
                                      while ( p->ref_count > 0 ) {
cache_invalidate();                     task_queue[tid].lock_acquire();
                                        task* t = task_queue[tid].dequeue();
        cache_flush();                  task_queue[tid].lock_release();
                                        if (t) {
                                          t->execute();
                                          amo_sub( t->parent->ref_count, 1 );
                                        }
                                        else {
                                          int vid = choose_victim();
cache_invalidate();                       task_queue[tid].lock_acquire();
                                          t = task_queue[vid].steal();
        cache_flush();                    task_queue[tid].lock_release();
                                          if (t) {
                                            t->execute();
                                            amo_sub(t->parent->ref_count, 1 );
                                          }
                                        }
                                      }
                                    }
```

Work-Stealing Runtimes on Software-Centric Cache Coherence

- Supporting shared queues:
  - Lock-acquire -> invalidation
  - Lock-release -> cache flush
- Stolen task on HCC:
  - Invalidate before execution
  - Flush after execution

```
                                    void task::wait( task* p ) {
                                      while ( p->ref_count > 0 ) {
cache_invalidate();                     task_queue[tid].lock_acquire();
                                        task* t = task_queue[tid].dequeue();
        cache_flush();                  task_queue[tid].lock_release();
                                        if (t) {
                                          t->execute();
                                          amo_sub( t->parent->ref_count, 1 );
                                        }
                                        else {
                                          int vid = choose_victim();
cache_invalidate();                       task_queue[tid].lock_acquire();
                                          t = task_queue[vid].steal();
        cache_flush();                    task_queue[tid].lock_release();
                                          if (t) {
cache_invalidate();                         t->execute();
                                            amo_sub(t->parent->ref_count, 1 );
        cache_flush();                    }
                                        }
                                      }
                                    }
```

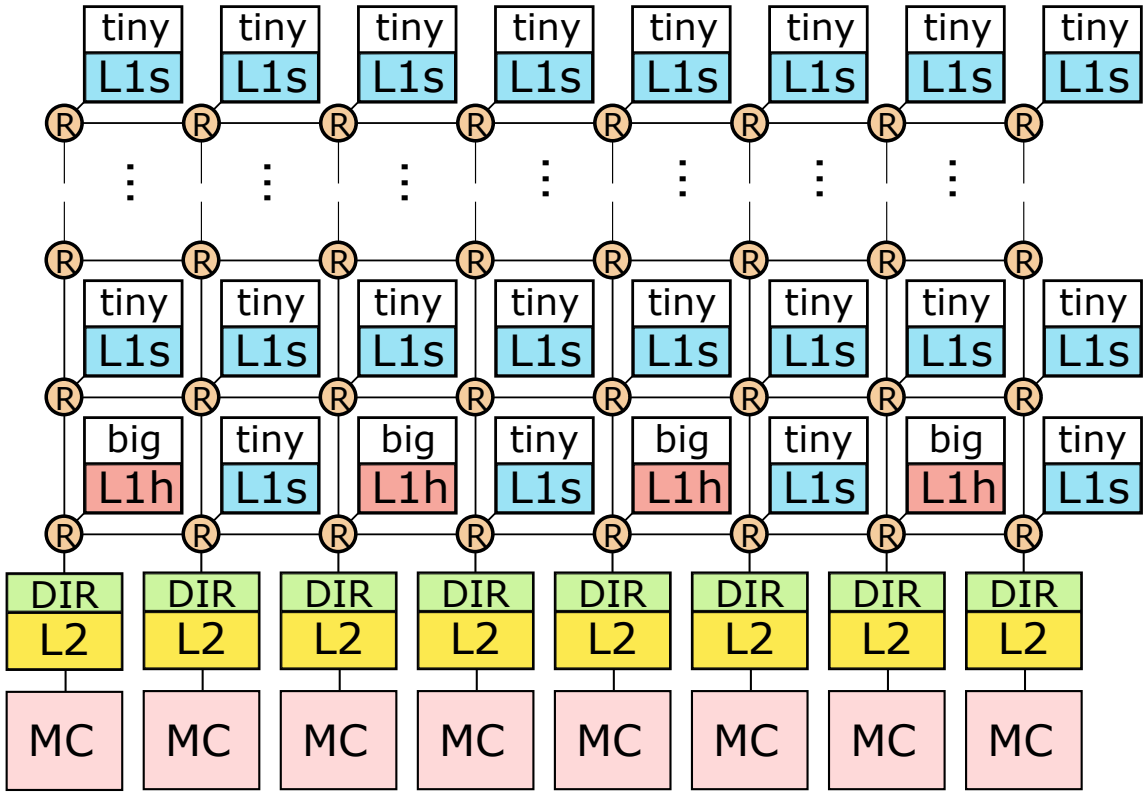
- Supporting shared queues:
  - Lock-acquire -> invalidation
  - Lock-release -> cache flush
- Stolen task on HCC:
  - Invalidate before execution
  - Flush after execution
- Ensure parent-child synchronization


```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

- Supporting shared queues:
  - Lock-acquire -> invalidation
  - Lock-release -> cache flush
- Stolen task on HCC:
  - Invalidate before execution
  - Flush after execution
- Ensure parent-child synchronization
- No-op when invalidation or flush is not required

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    cache_invalidate();
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    cache_flush();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      cache_invalidate();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      cache_flush();
      task_queue[tid].lock_release();
      cache_invalidate();
      if (t) {
        t->execute();
        cache_flush();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
  cache_invalidate();
}
```

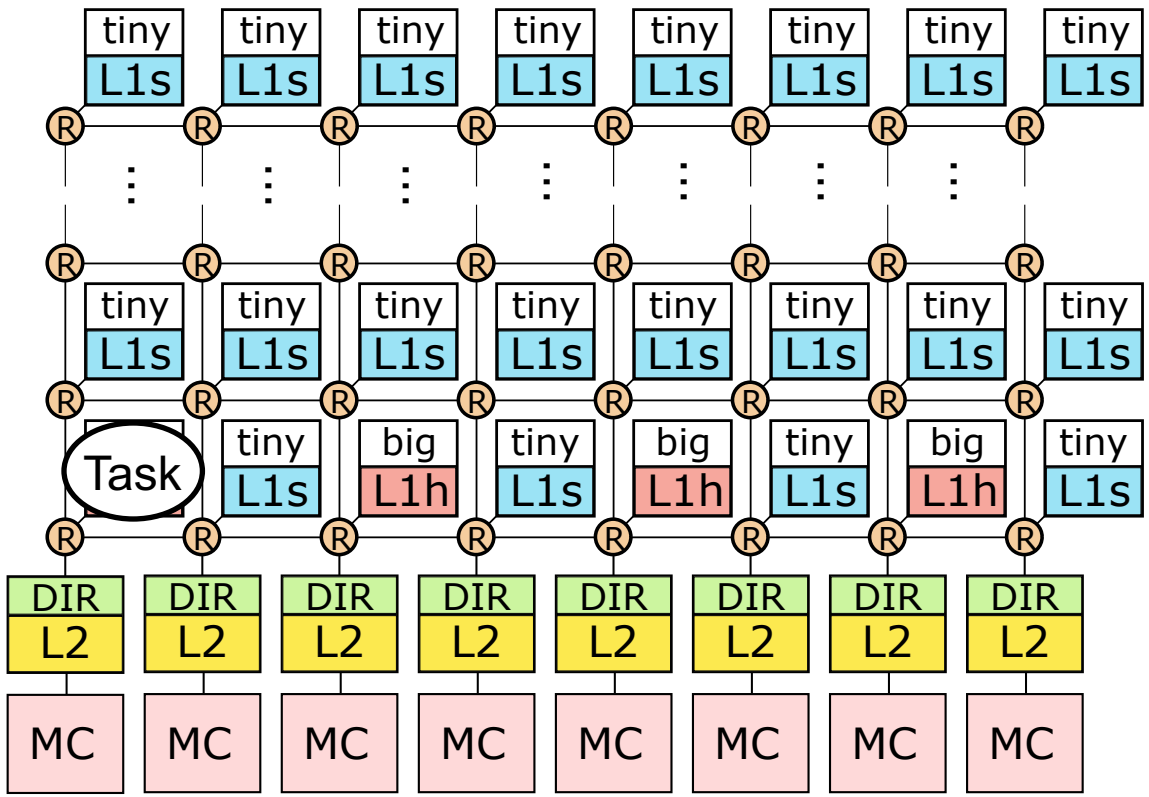
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

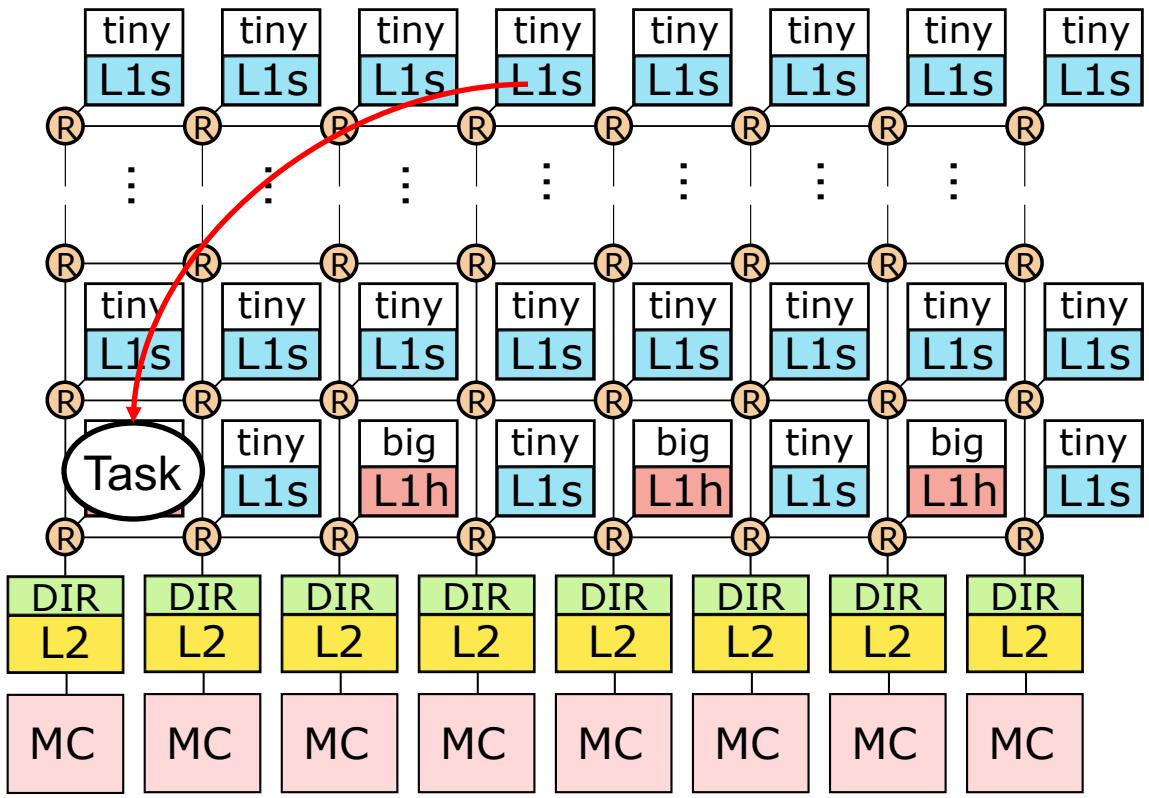
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

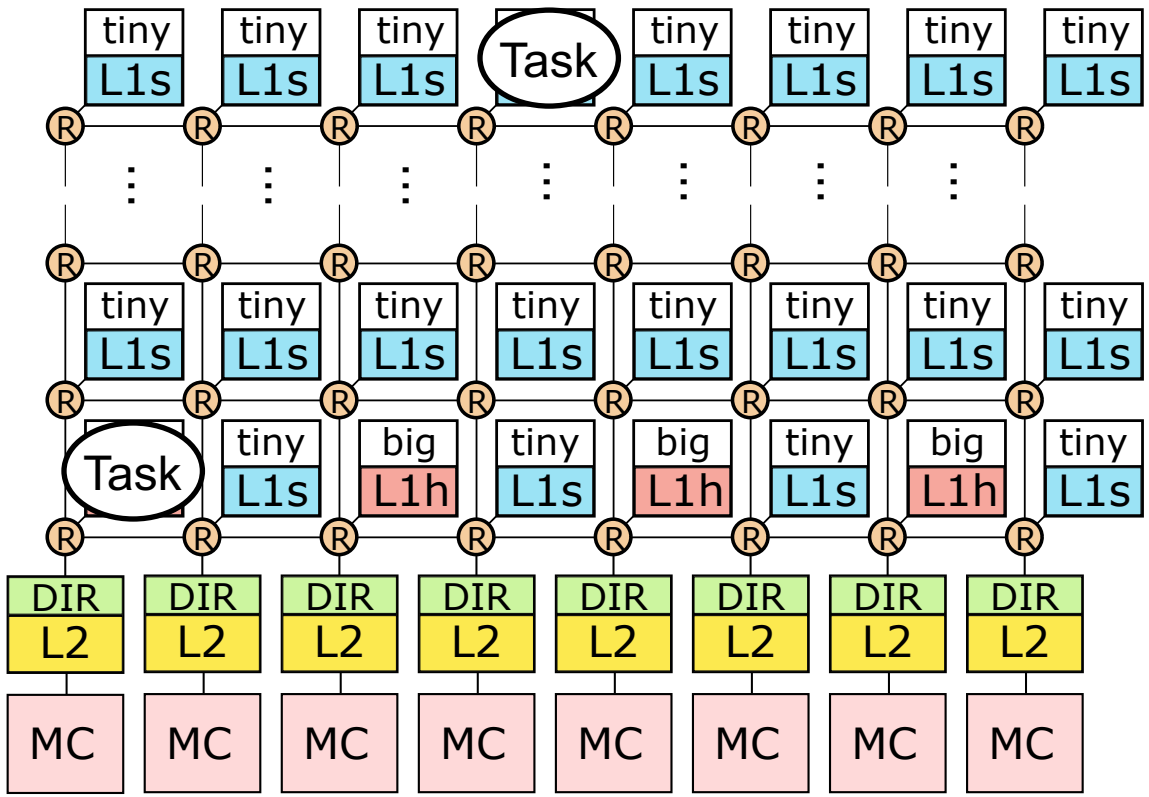
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

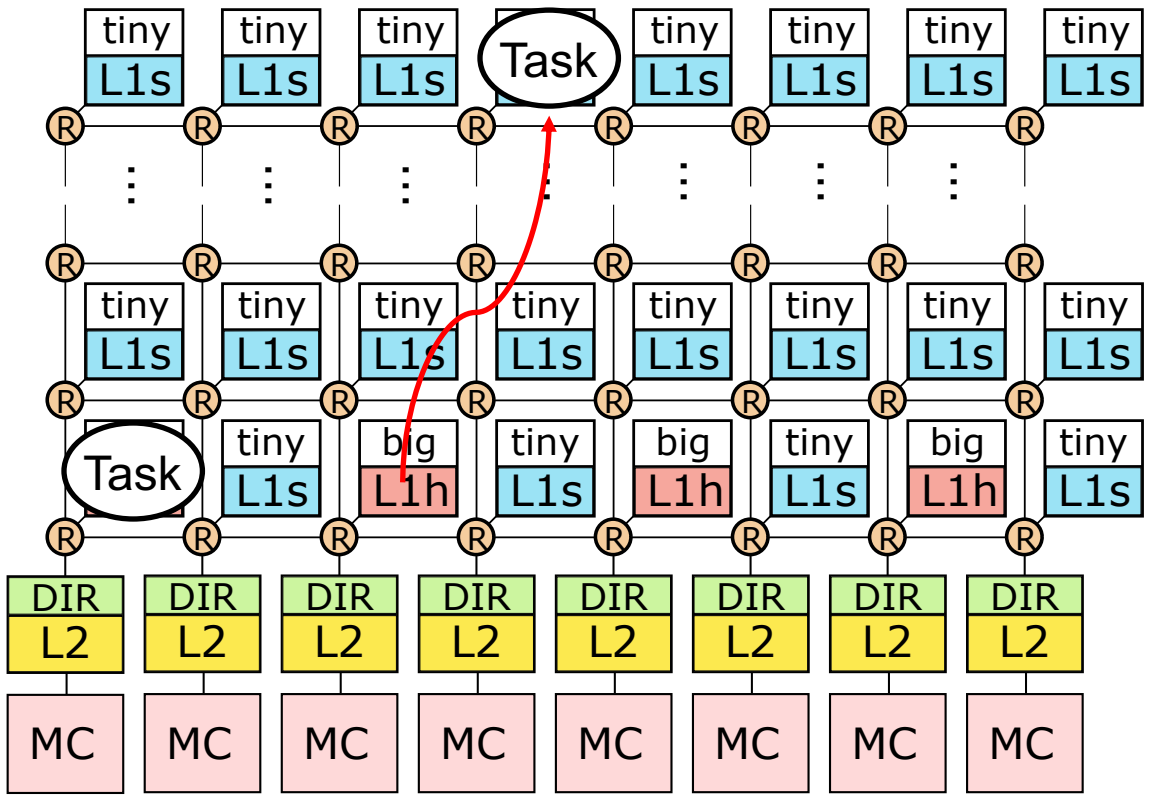
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

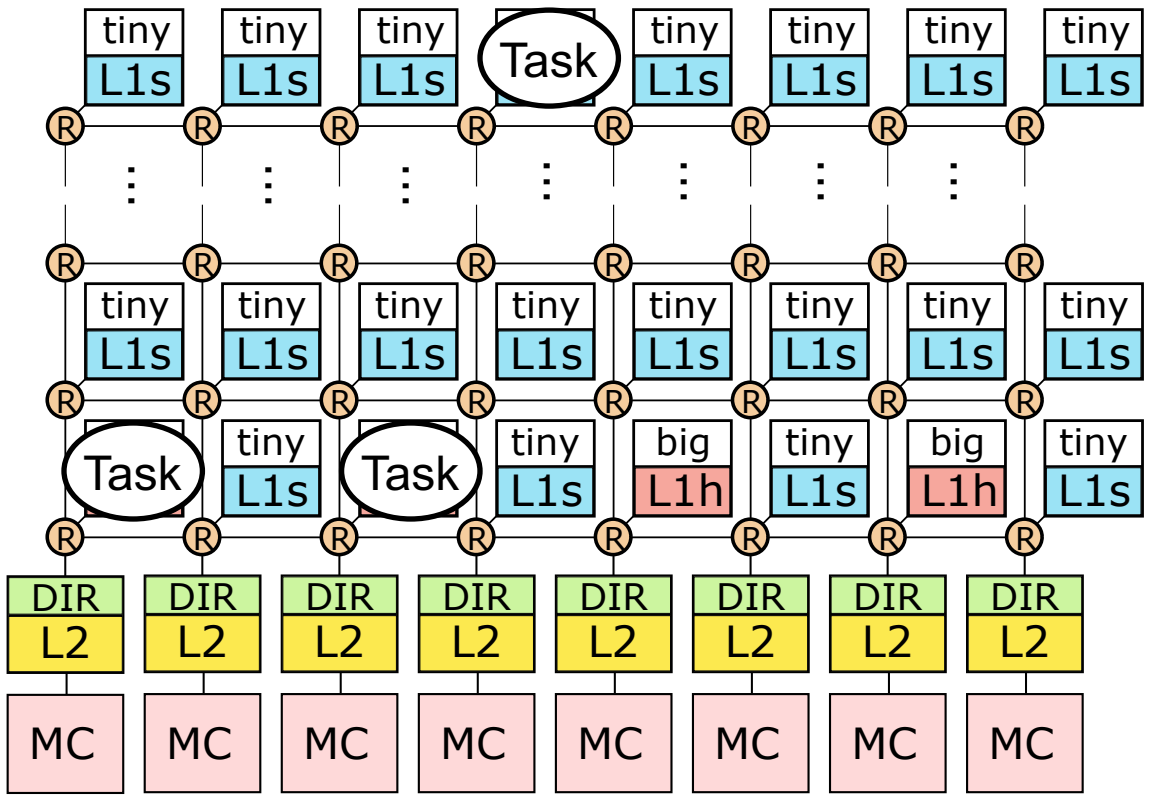
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

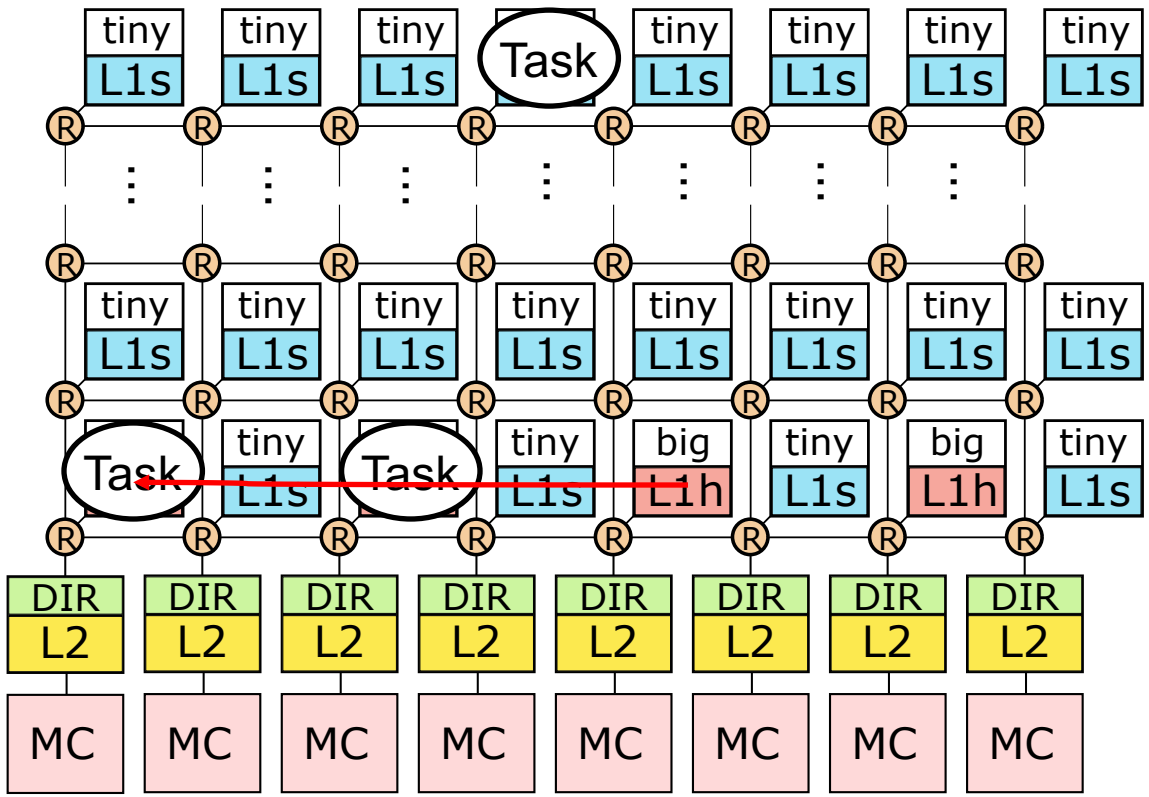
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

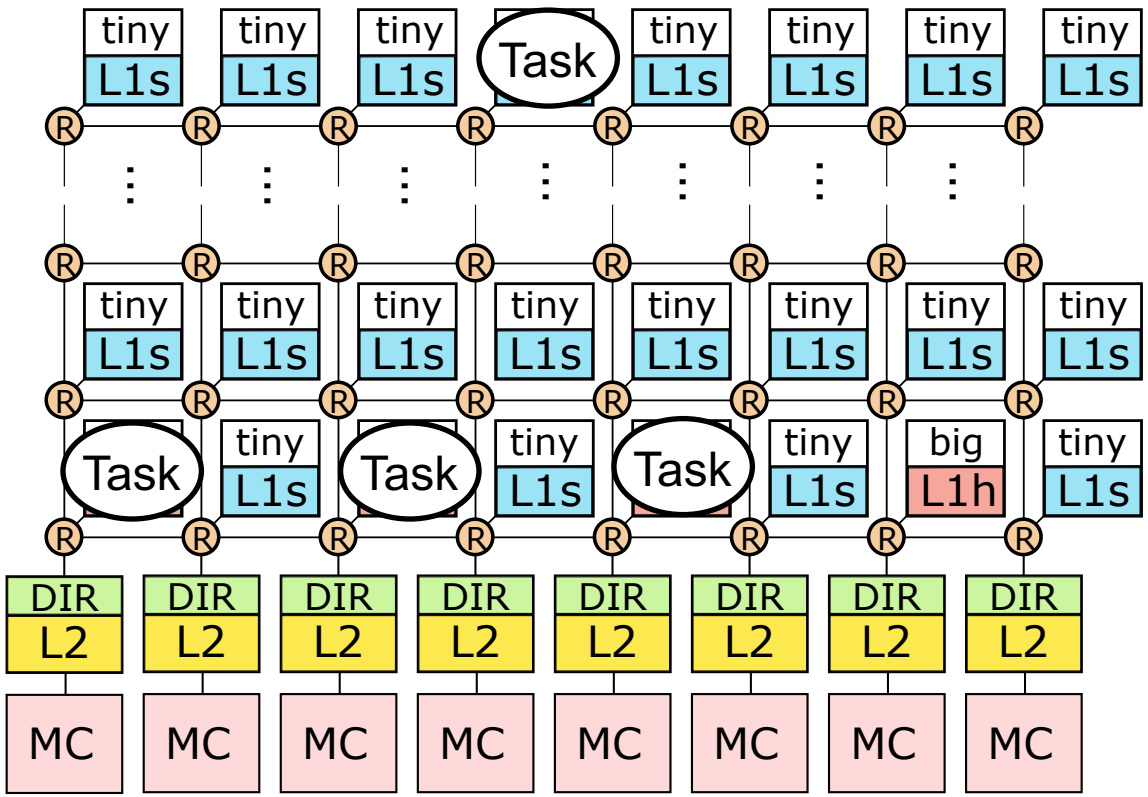
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

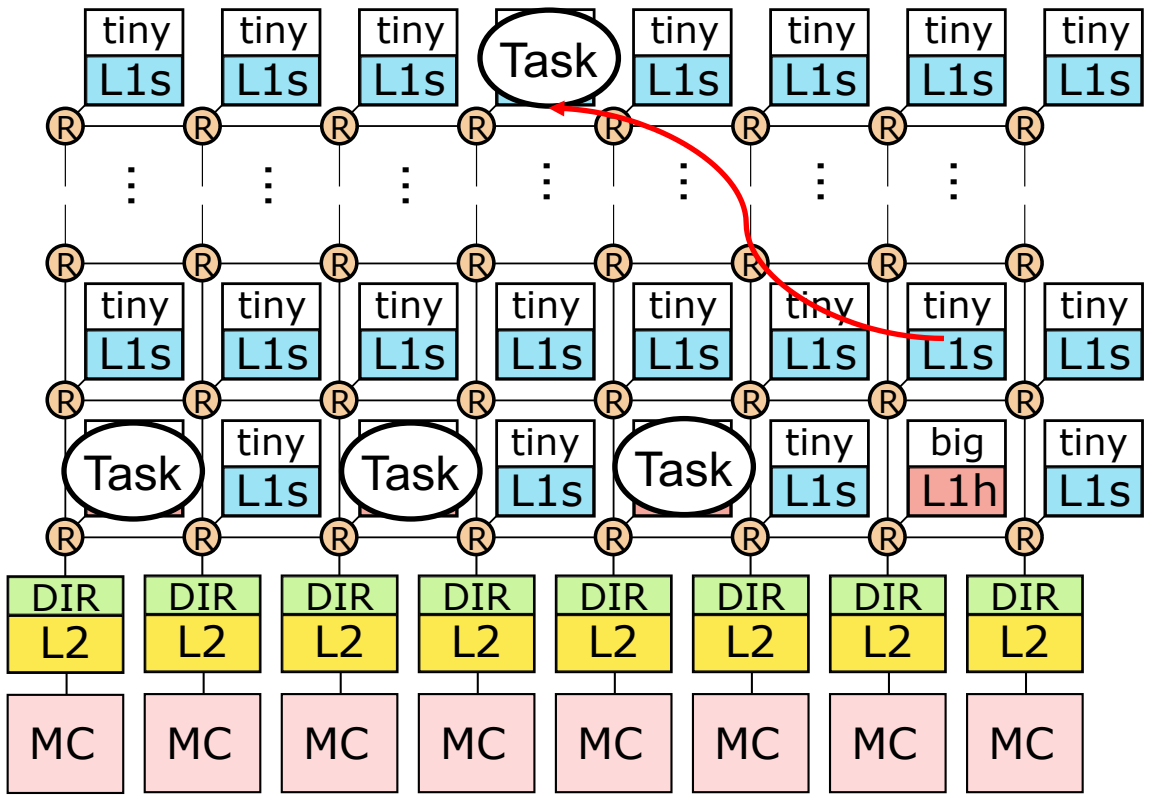
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

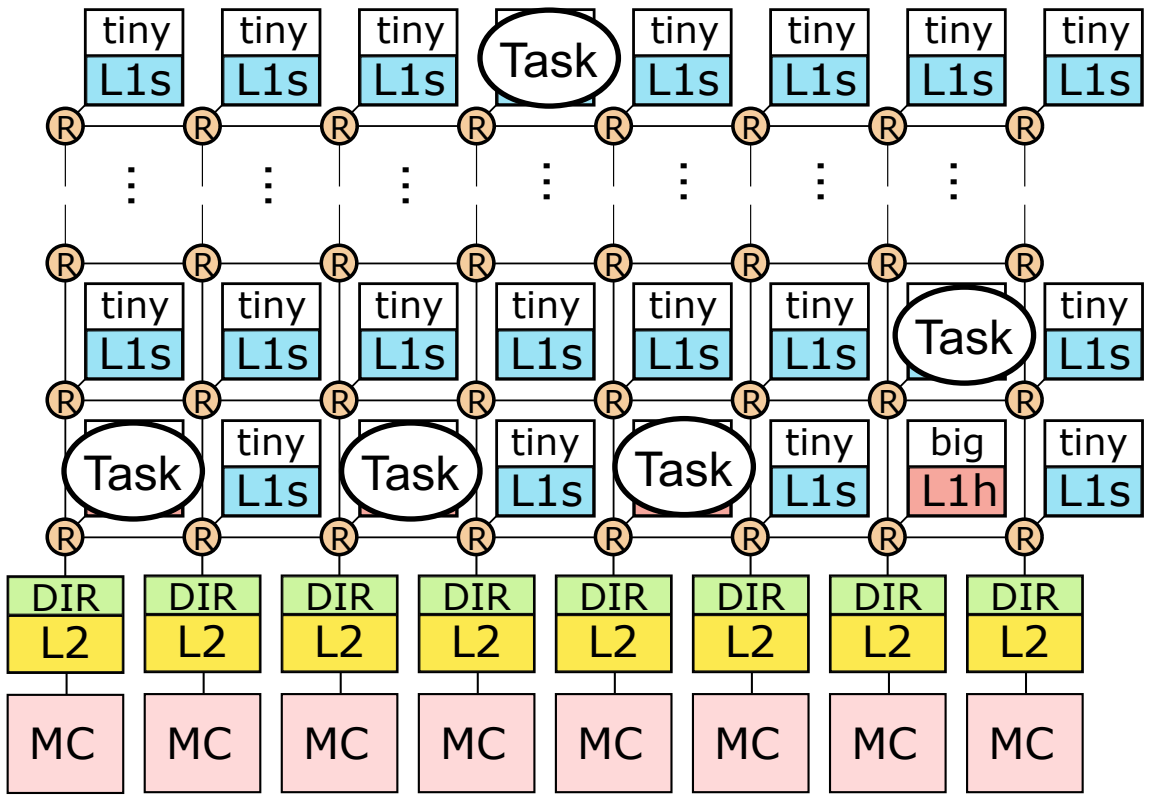
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

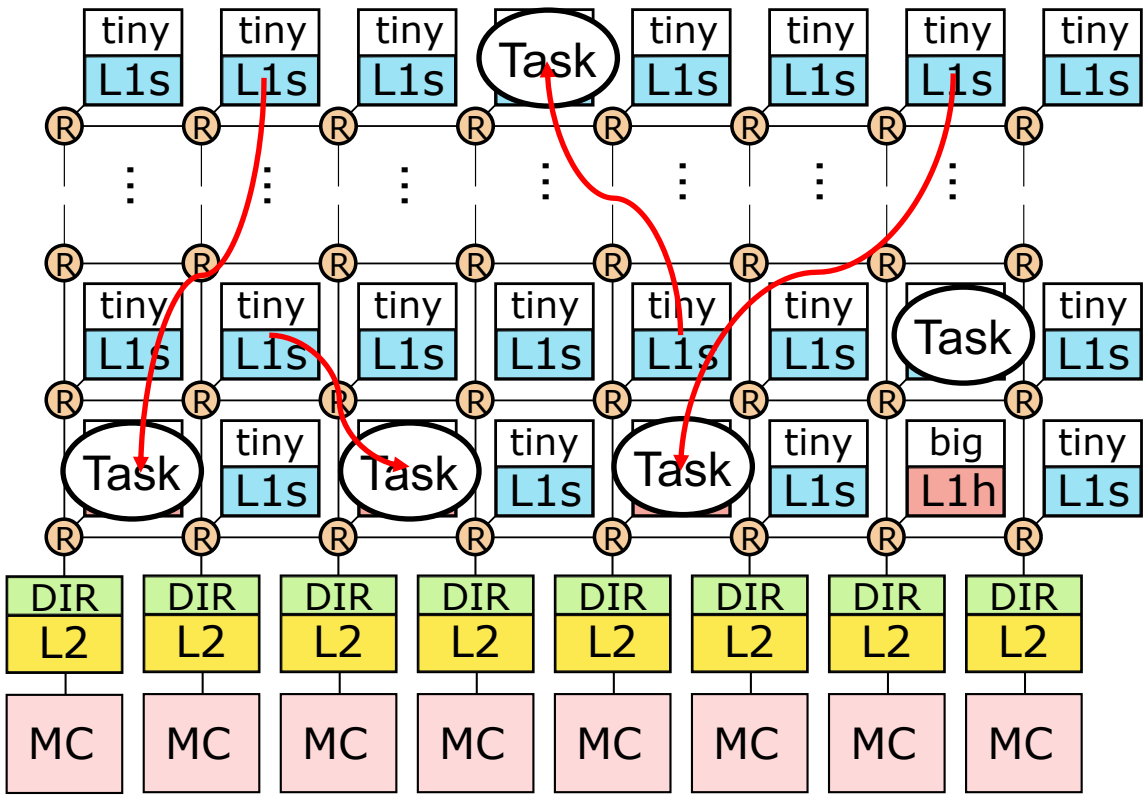
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

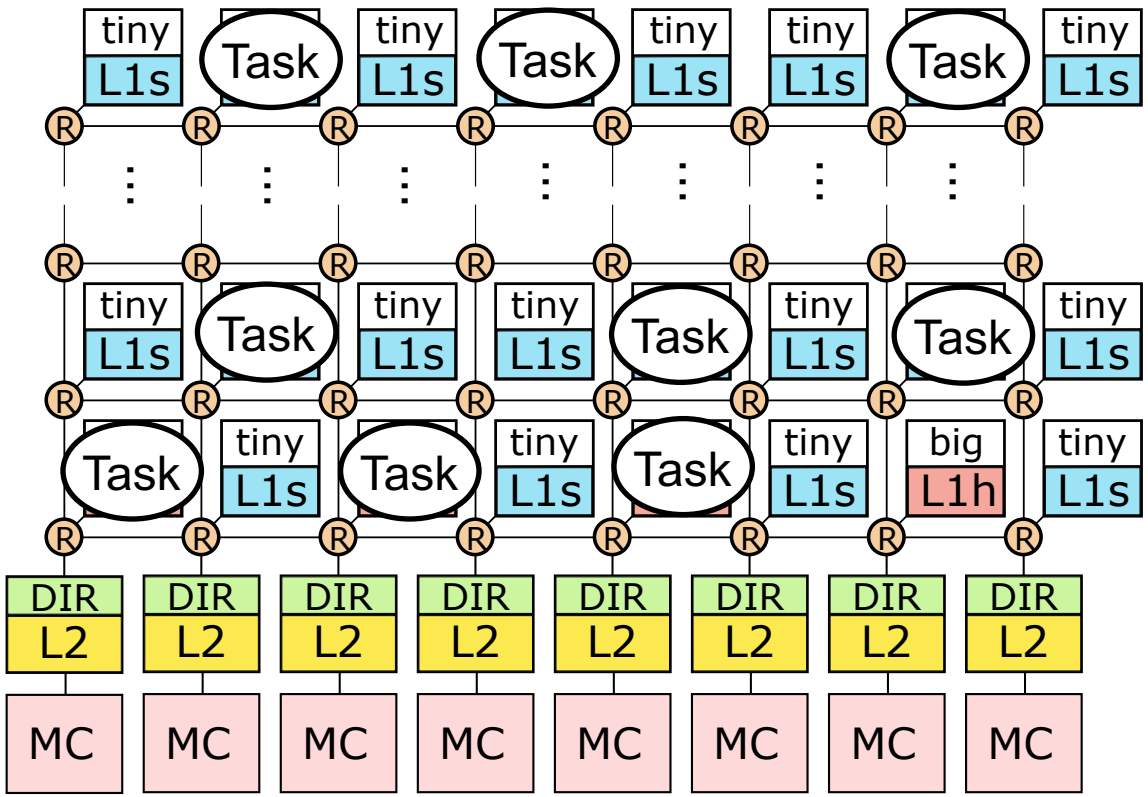
Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

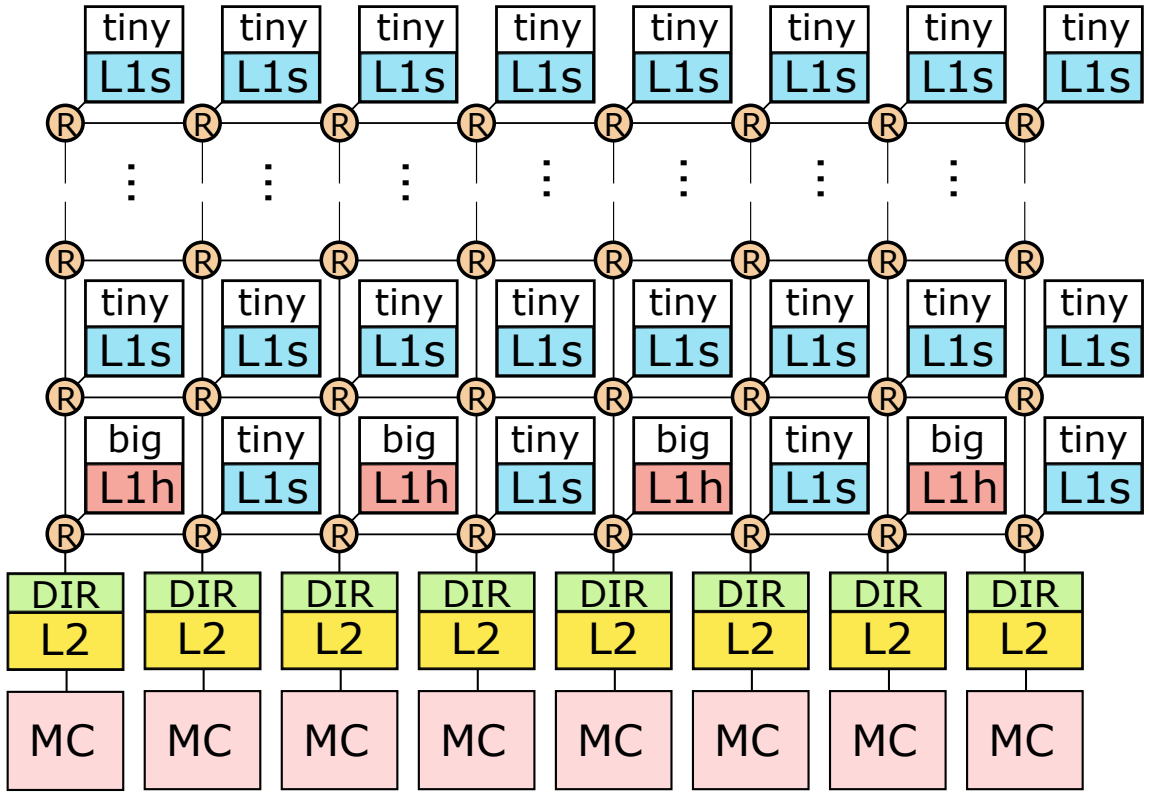
- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core

Cooperative Execution



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Same runtime loop runs on both big and tiny cores
- Invalidations and flushes are no-ops on big cores with MESI
- Enables seamless work-stealing between big cores and tiny core



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Background
- Implementing Work-Stealing Runtimes on HCC
- **Direct Task Stealing**
- Evaluation

The Overheads of Work-Stealing Runtimes on HCC

- Invalidation and/or flush on **all** accesses to task queues
- Only need to maintain data consistency between parent and child.
- In work-stealing runtimes, steals are relatively rare, but every task **can** be stolen.
- Hard to know whether child tasks are stolen
- Cost of AMOs.

```
                                    void task::wait( task* p ) {
                                      while ( p->ref_count > 0 ) {
cache_invalidate();                     task_queue[tid].lock_acquire();
                                        task* t = task_queue[tid].dequeue();
cache_flush();                          task_queue[tid].lock_release();
                                        if (t) {
                                          t->execute();
                                          amo_sub( t->parent->ref_count, 1 );
                                        }
                                        else {
                                          int vid = choose_victim();
cache_invalidate();                       task_queue[tid].lock_acquire();
                                          t = task_queue[vid].steal();
cache_flush();                            task_queue[tid].lock_release();
                                          if (t) {
cache_invalidate();                         t->execute();
                                            amo_sub(t->parent->ref_count, 1 );
cache_flush();                            }
                                        }
                                    }
cache_invalidate();                 }
```

Improving Work-Stealing Runtimes on HCC

- What we want to achieve

```
                                    void task::wait( task* p ) {
                                      while ( p->ref_count > 0 ) {
cache_invalidate();                     task_queue[tid].lock_acquire();
                                        task* t = task_queue[tid].dequeue();
cache_flush();                          task_queue[tid].lock_release();
                                        if (t) {
                                          t->execute();
                                          amo_sub( t->parent->ref_count, 1 );
                                        }
                                        else {
                                          int vid = choose_victim();
cache_invalidate();                       task_queue[tid].lock_acquire();
                                          t = task_queue[vid].steal();
cache_flush();                            task_queue[tid].lock_release();
                                          if (t) {
cache_invalidate();                         t->execute();
                                            amo_sub(t->parent->ref_count, 1 );
cache_flush();                            }
                                        }
                                      }
cache_invalidate();                   }
```

- What we want to achieve
  - No Inv/Flush when accessing the local task queue

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

cache_invalidate();

cache_flush();

cache_invalidate();

Improving Work-Stealing Runtimes on HCC

- What we want to achieve
  - No Inv/Flush when accessing the local task queue
  - No invalidation if children not stolen

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

cache_invalidate();

cache_flush();

Improving Work-Stealing Runtimes on HCC

- What we want to achieve
  - No Inv/Flush when accessing the local task queue
  - No invalidation if children not stolen
  - No AMO if child not stolen

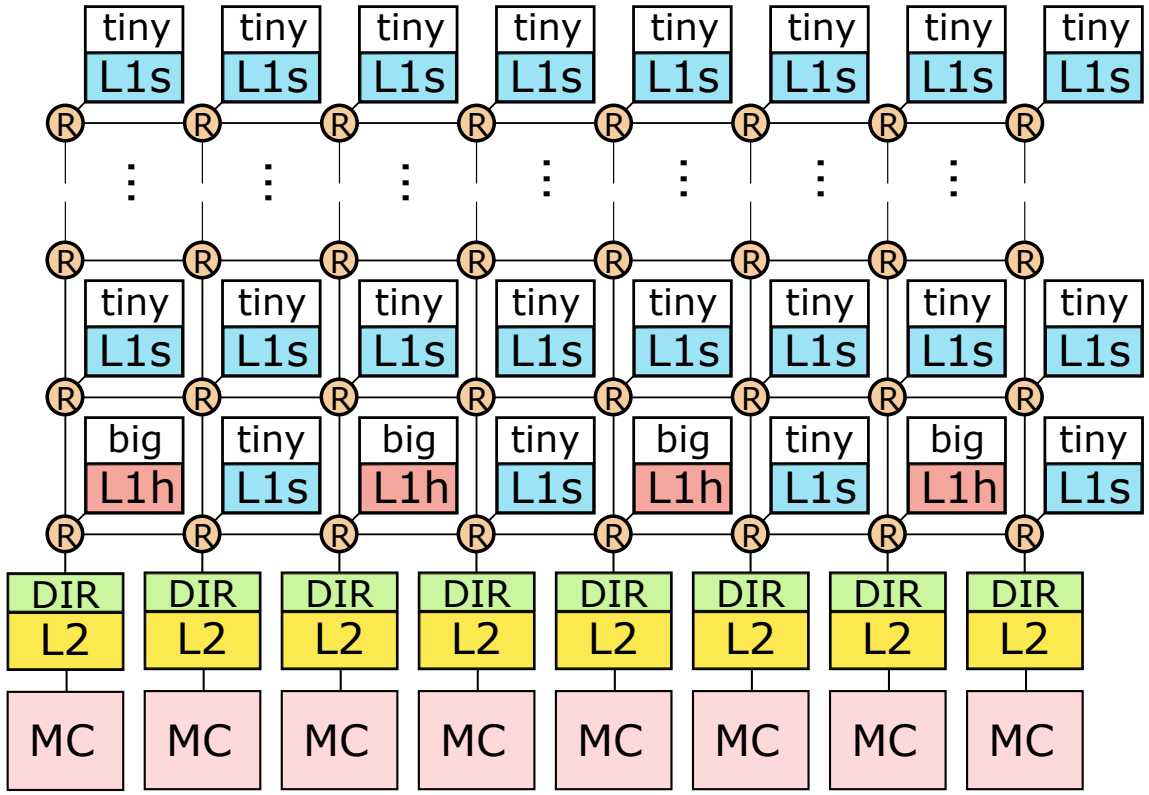
```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        cache_invalidate();
        t->execute();
        cache_flush();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```

Improving Work-Stealing Runtimes on HCC

- What we want to achieve
  - No Inv/Flush when accessing the local task queue
  - No invalidation if children not stolen
  - No AMO if child not stolen
- Our technique: direct task stealing (DTS) instead of indirect task stealing through shared task queues

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task_queue[tid].lock_acquire();
    task* t = task_queue[tid].dequeue();
    task_queue[tid].lock_release();
    if (t) {
      t->execute();
      amo_sub( t->parent->ref_count, 1 );
    }
    else {
      int vid = choose_victim();
      task_queue[tid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        cache_invalidate();
        amo_sub(t->parent->ref_count, 1 );
        cache_flush();
      }
    }
  }
}
```

User-Level Interrupt (ULI)



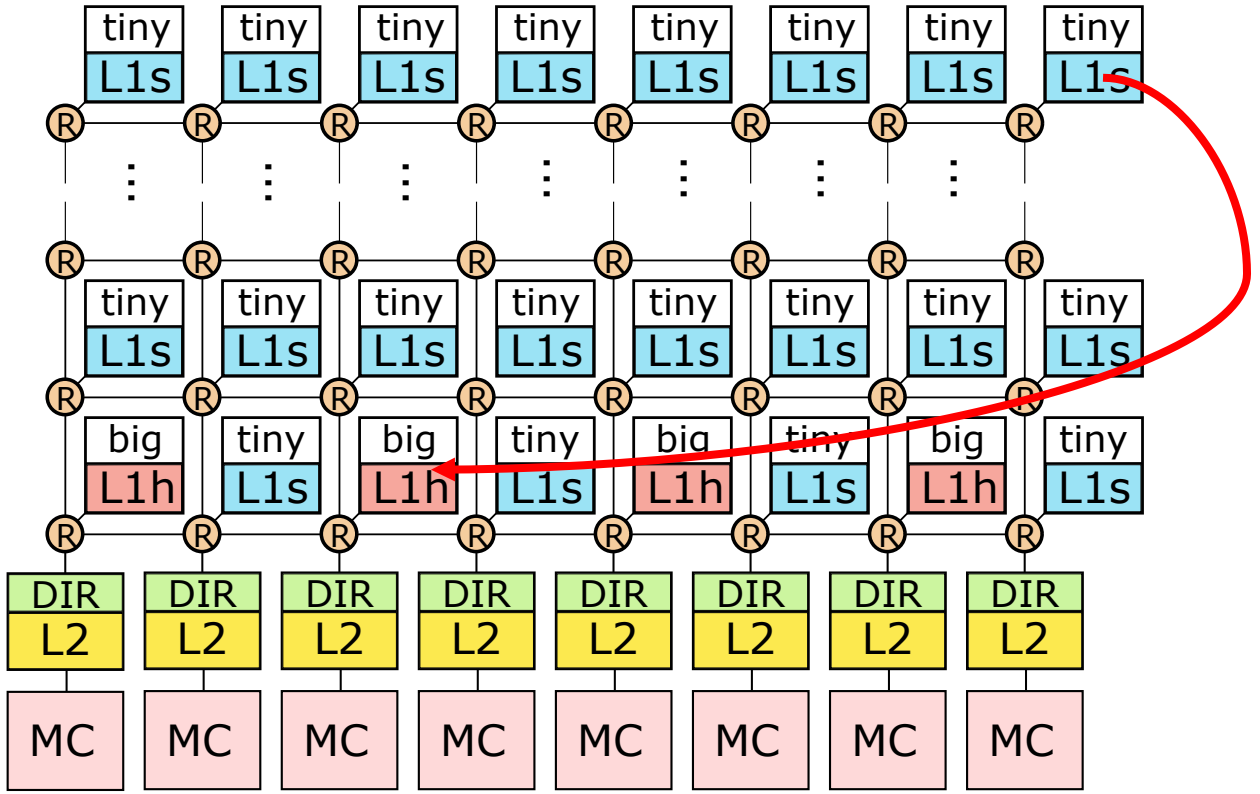
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- DTS is based on lightweight inter-processor user-level interrupt.
- Included in recent ISAs (e.g. RISC-V).
- Similar to active messages [1] and ADM [2].

[1] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. ISCA 1992.

[2] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. ASPLOS 2010.

User-Level Interrupt (ULI)



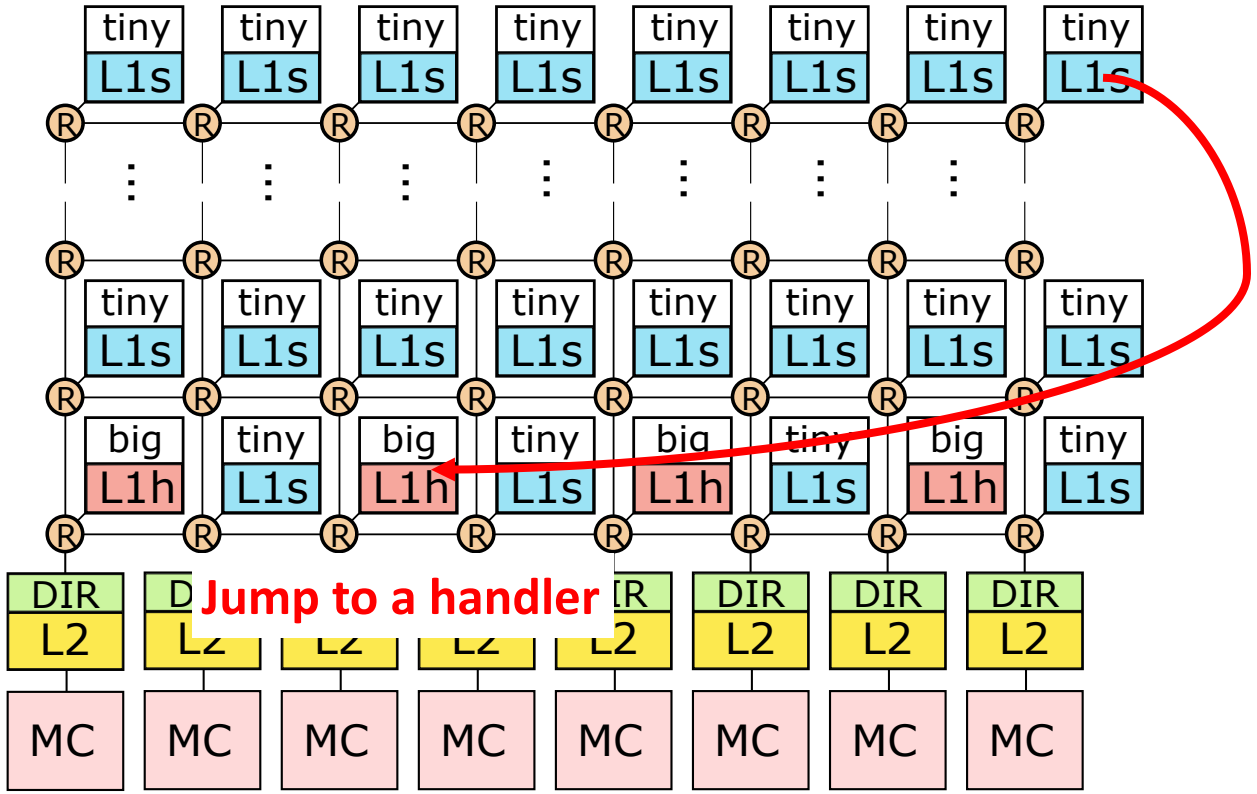
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- DTS is based on lightweight inter-processor user-level interrupt.
- Included in recent ISAs (e.g. RISC-V).
- Similar to active messages [1] and ADM [2].

[1] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. ISCA 1992.

[2] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. ASPLOS 2010.

User-Level Interrupt (ULI)



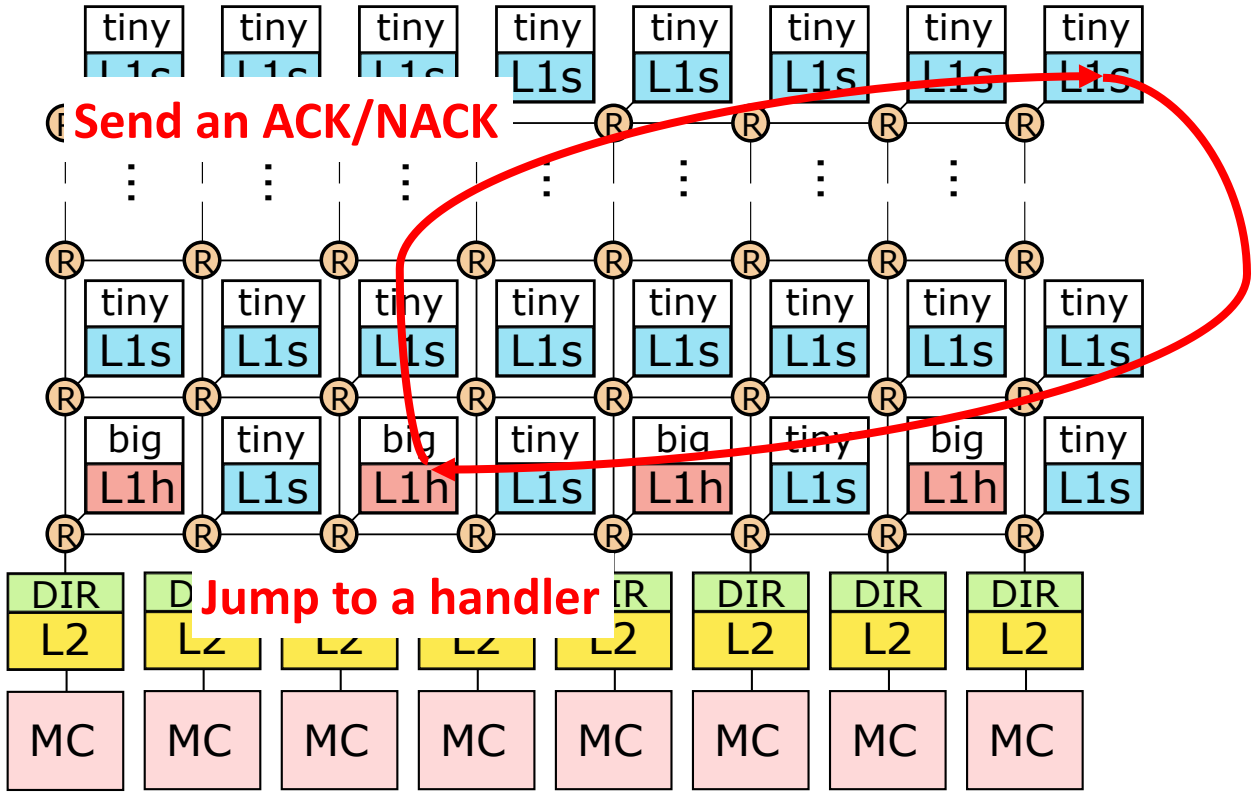
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- DTS is based on lightweight inter-processor user-level interrupt.
- Included in recent ISAs (e.g. RISC-V).
- Similar to active messages [1] and ADM [2].

[1] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. ISCA 1992.

[2] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. ASPLOS 2010.

User-Level Interrupt (ULI)



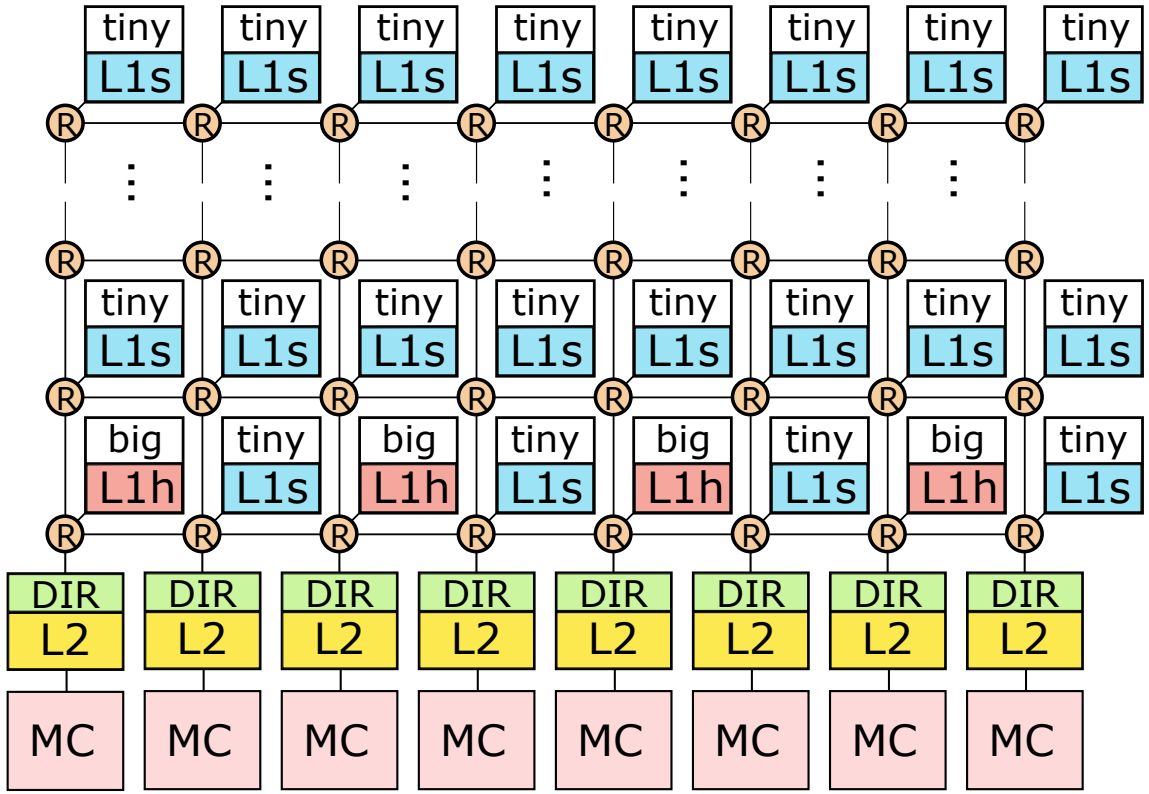
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- DTS is based on lightweight inter-processor user-level interrupt.
- Included in recent ISAs (e.g. RISC-V).
- Similar to active messages [1] and ADM [2].

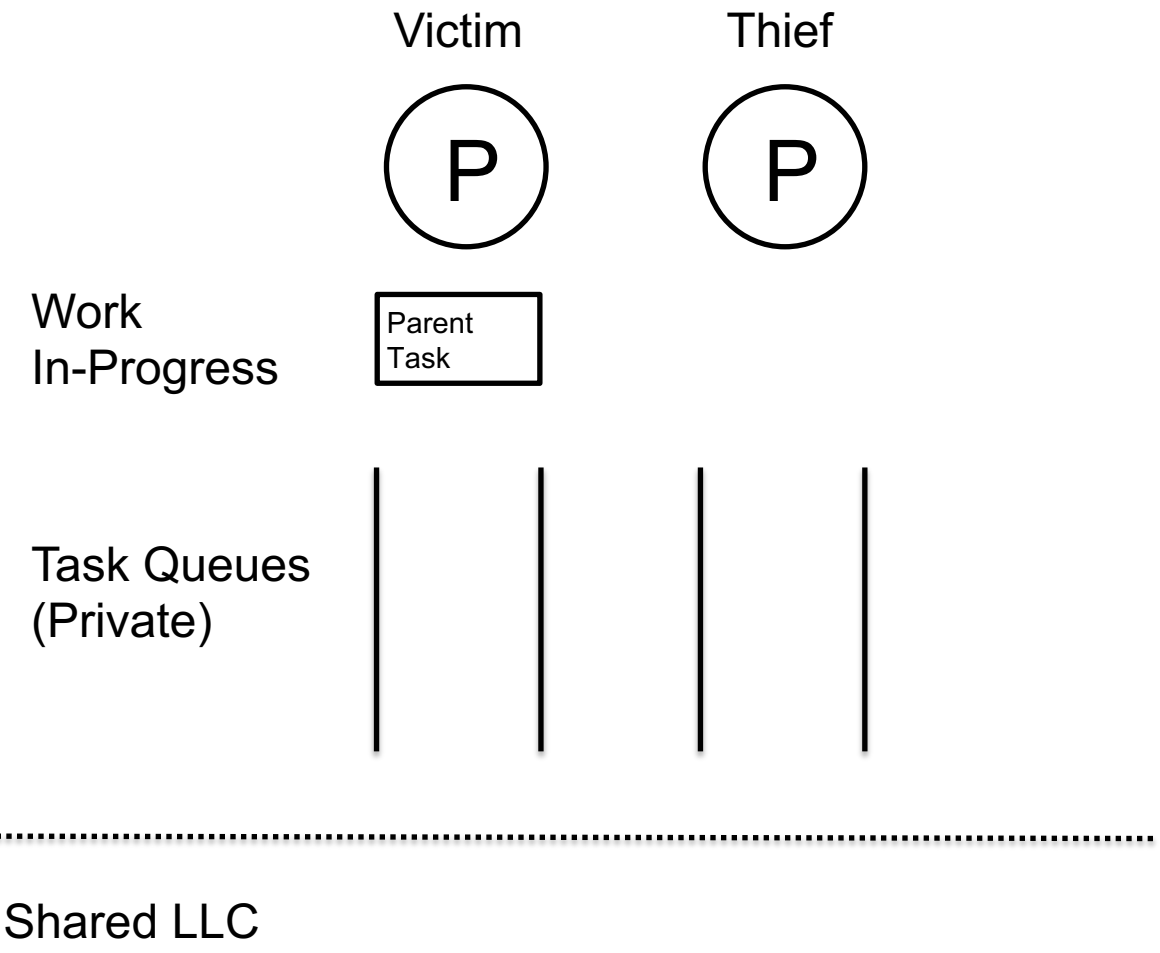
[1] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. ISCA 1992.

[2] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. ASPLOS 2010.

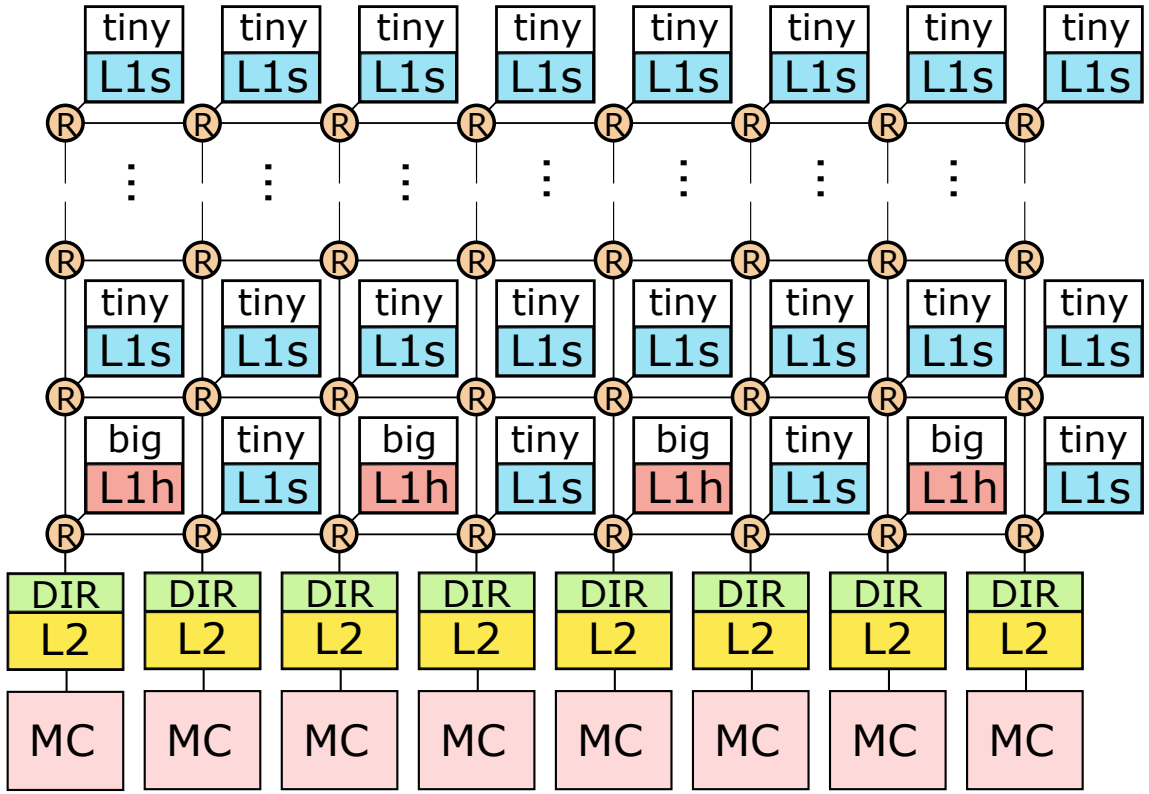
Implementing Direct-Task Stealing with ULI



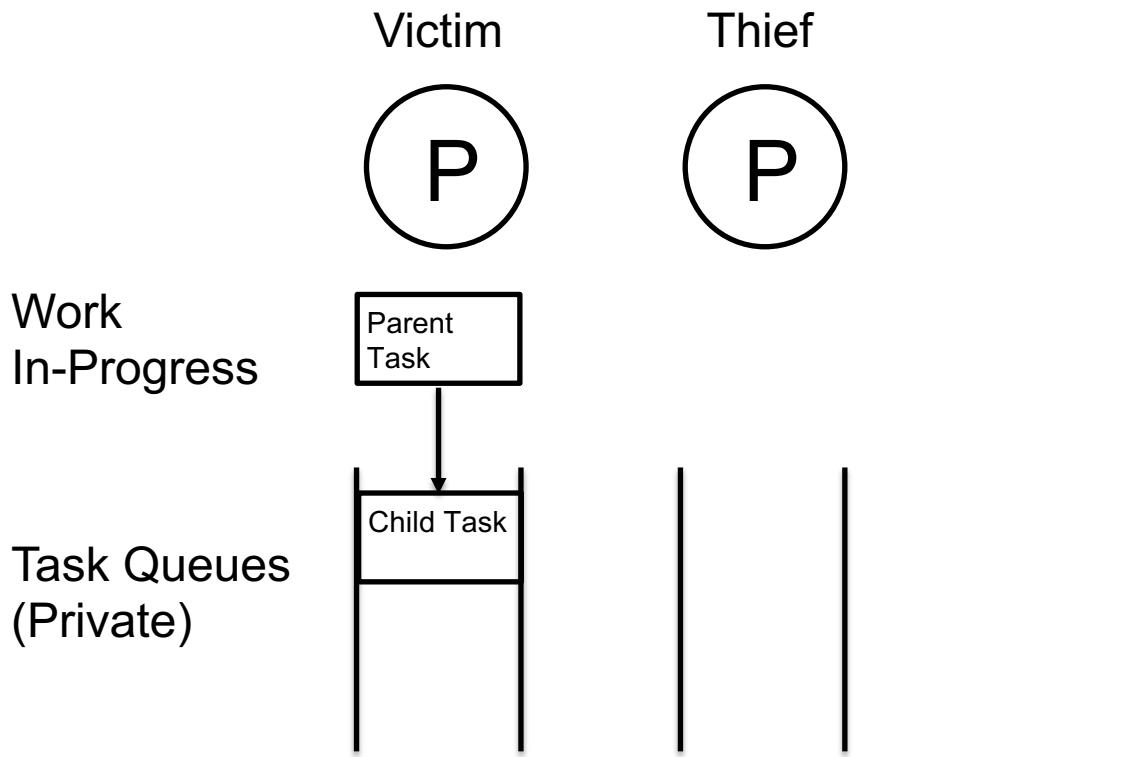
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence



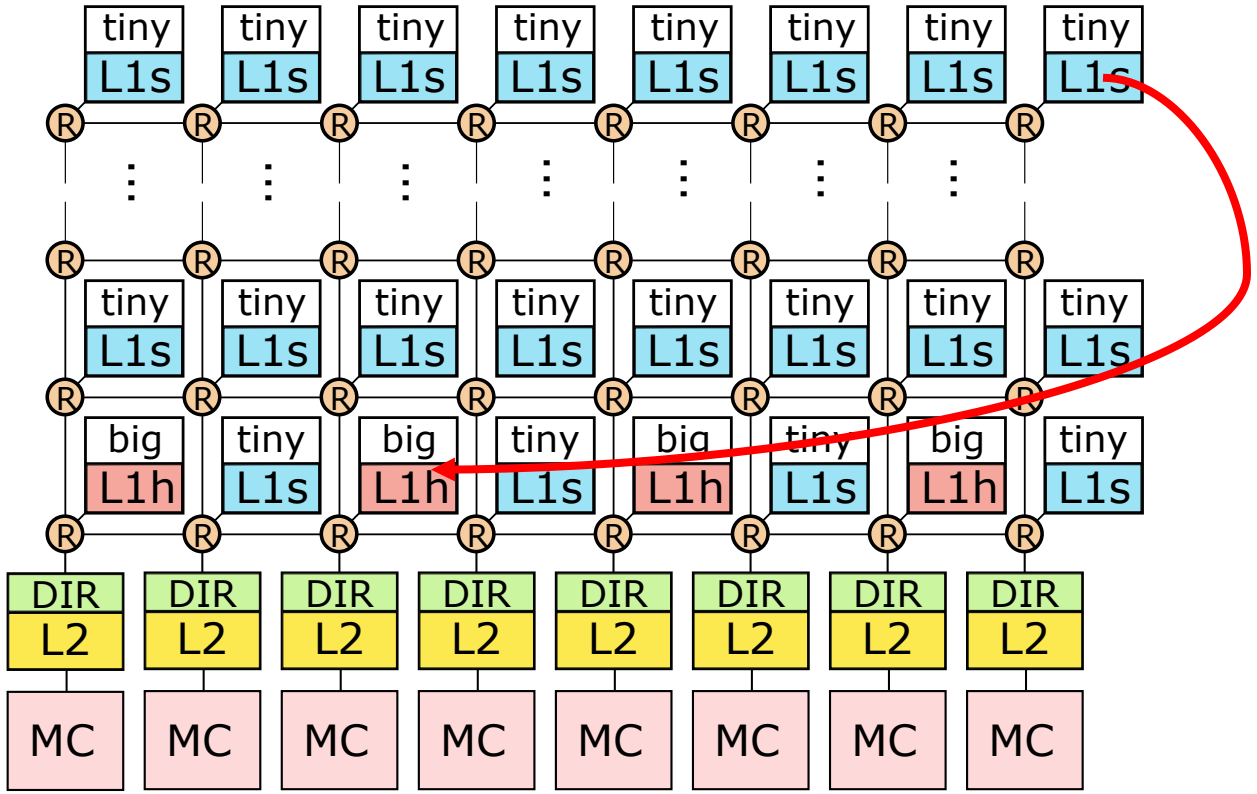
Implementing Direct-Task Stealing with ULI



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

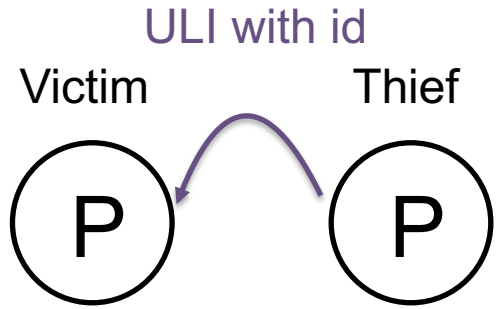


Implementing Direct-Task Stealing with ULI

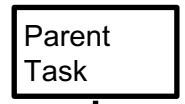


A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

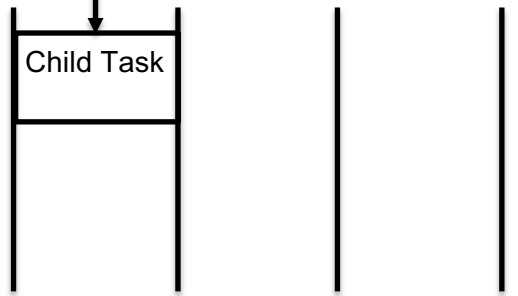
Send Interrupt



Work In-Progress

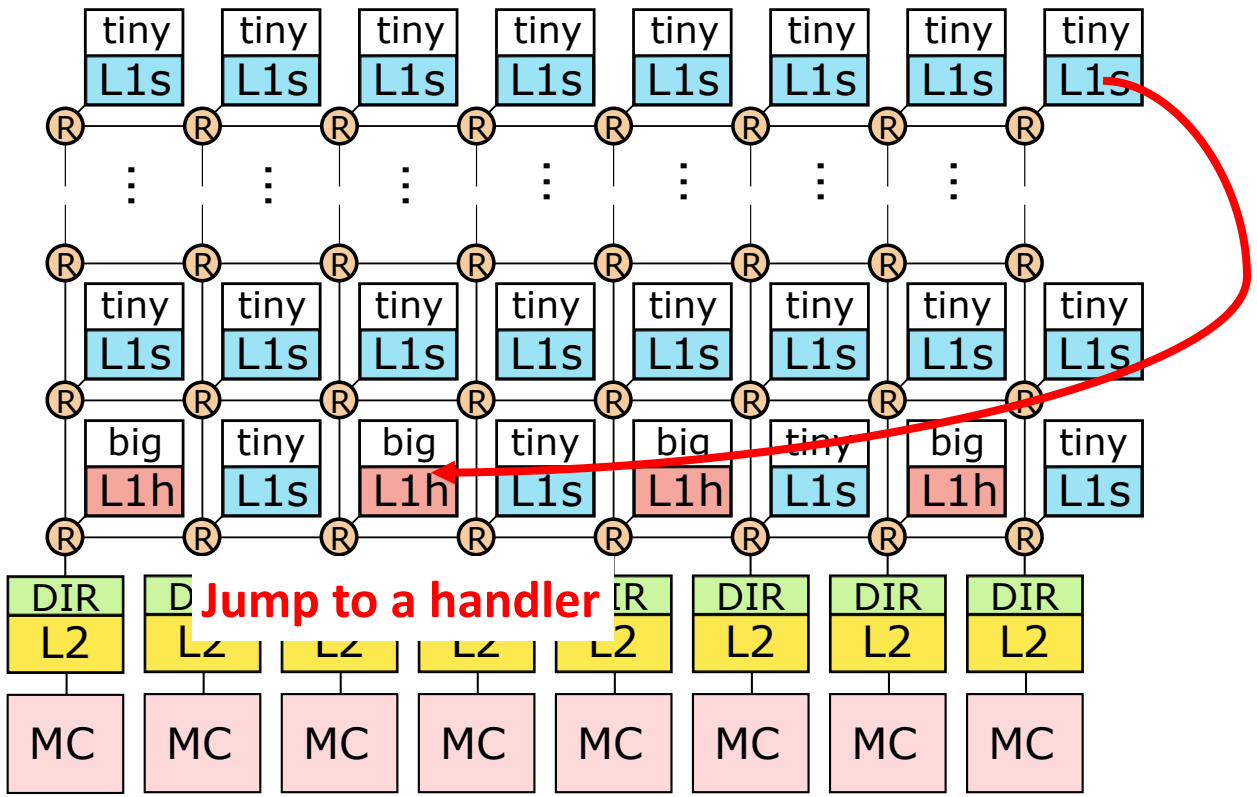


Task Queues (Private)

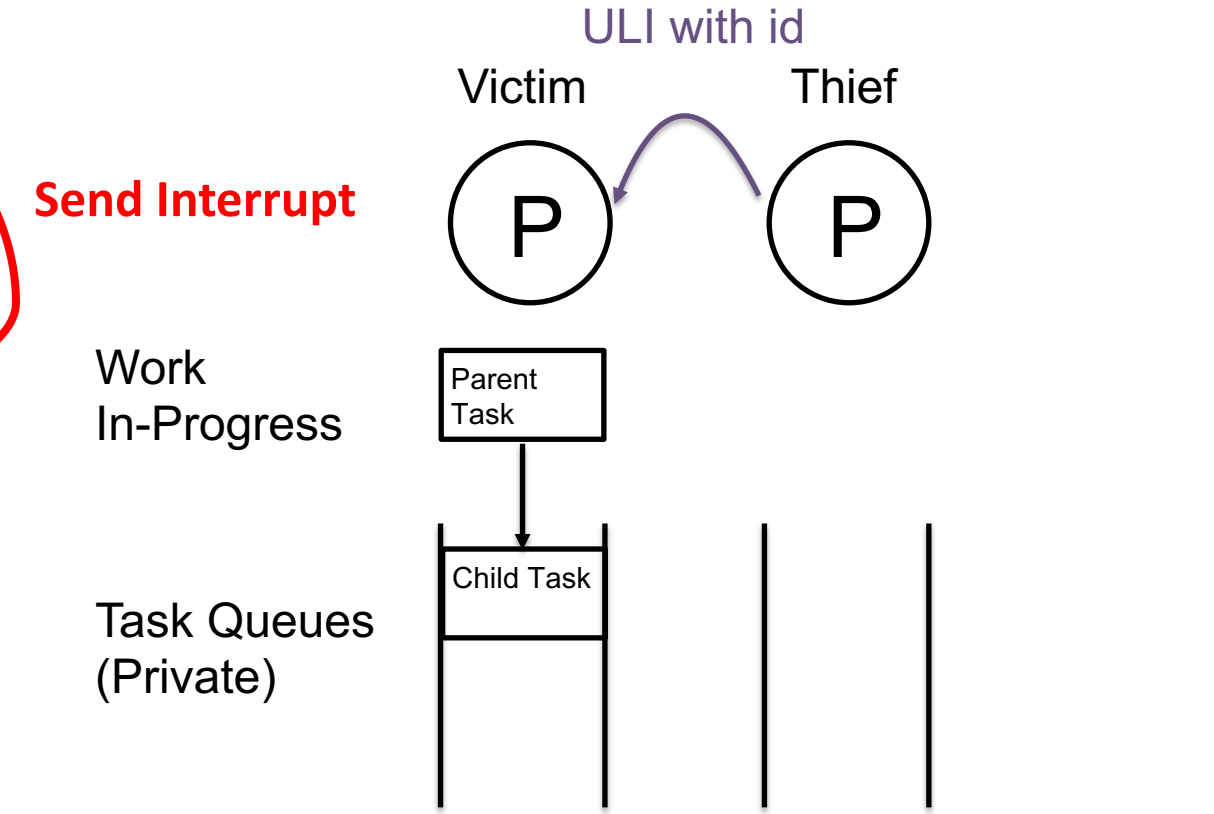


Shared LLC

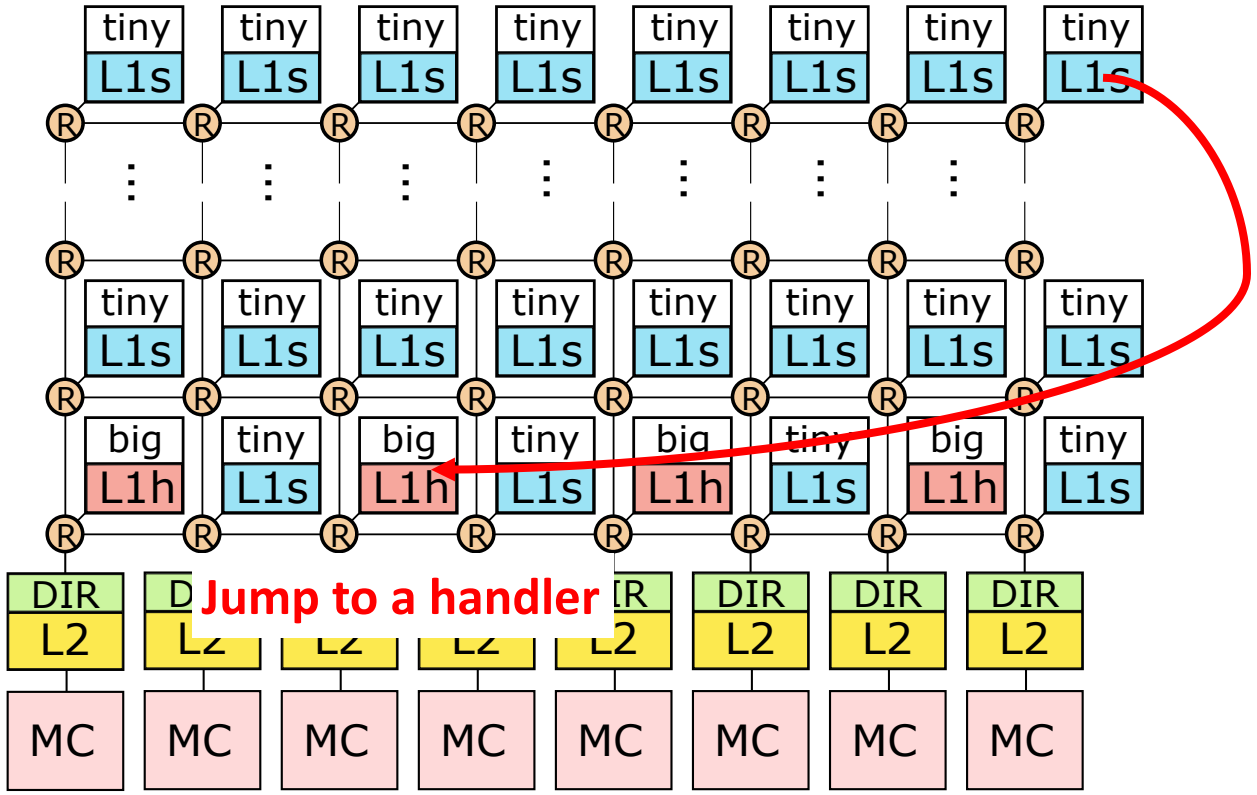
Implementing Direct-Task Stealing with ULI



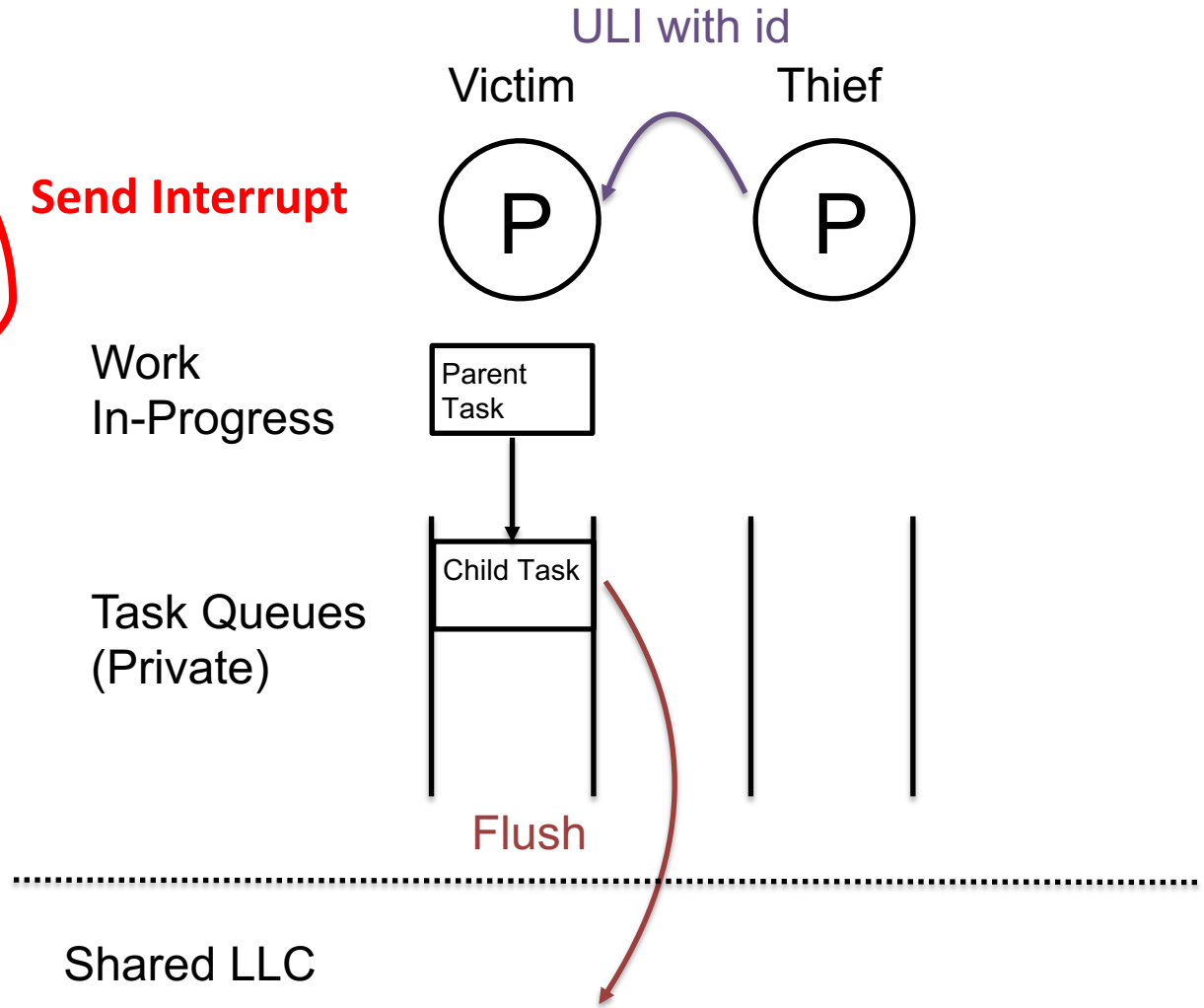
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence



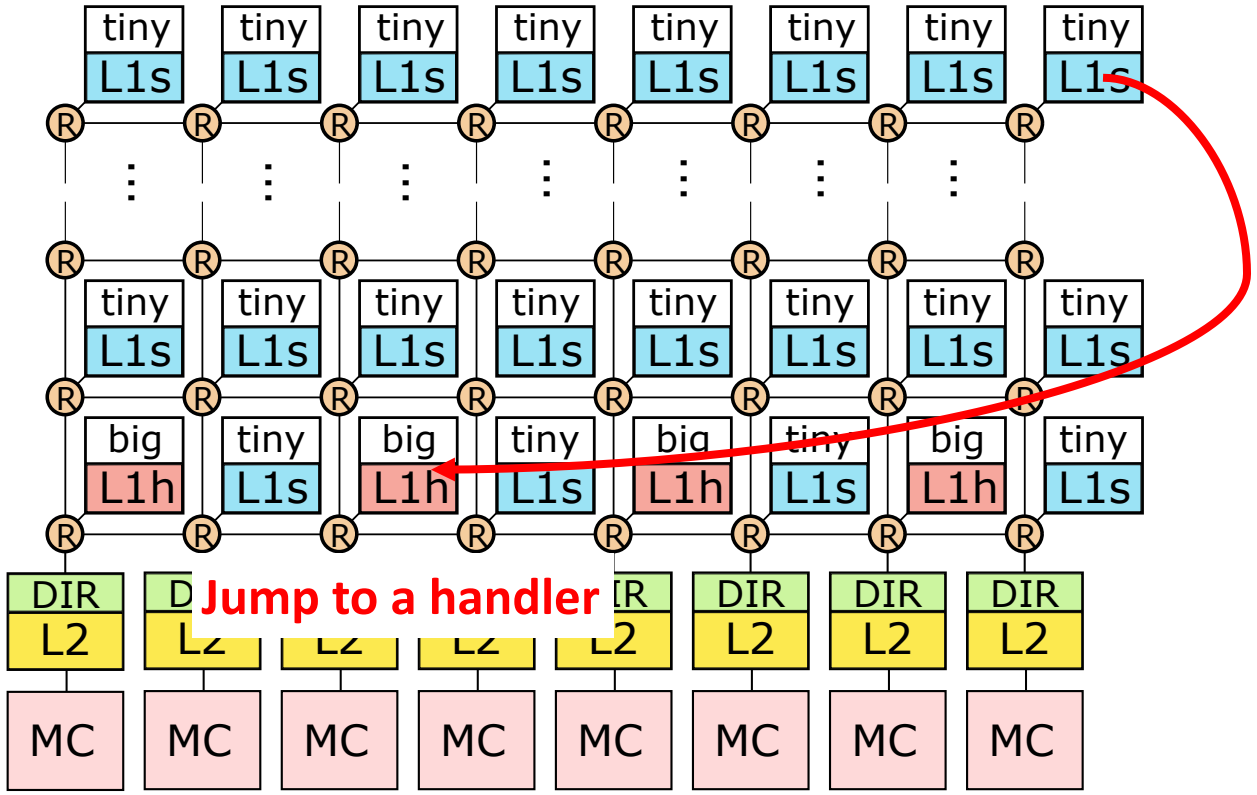
Implementing Direct-Task Stealing with ULI



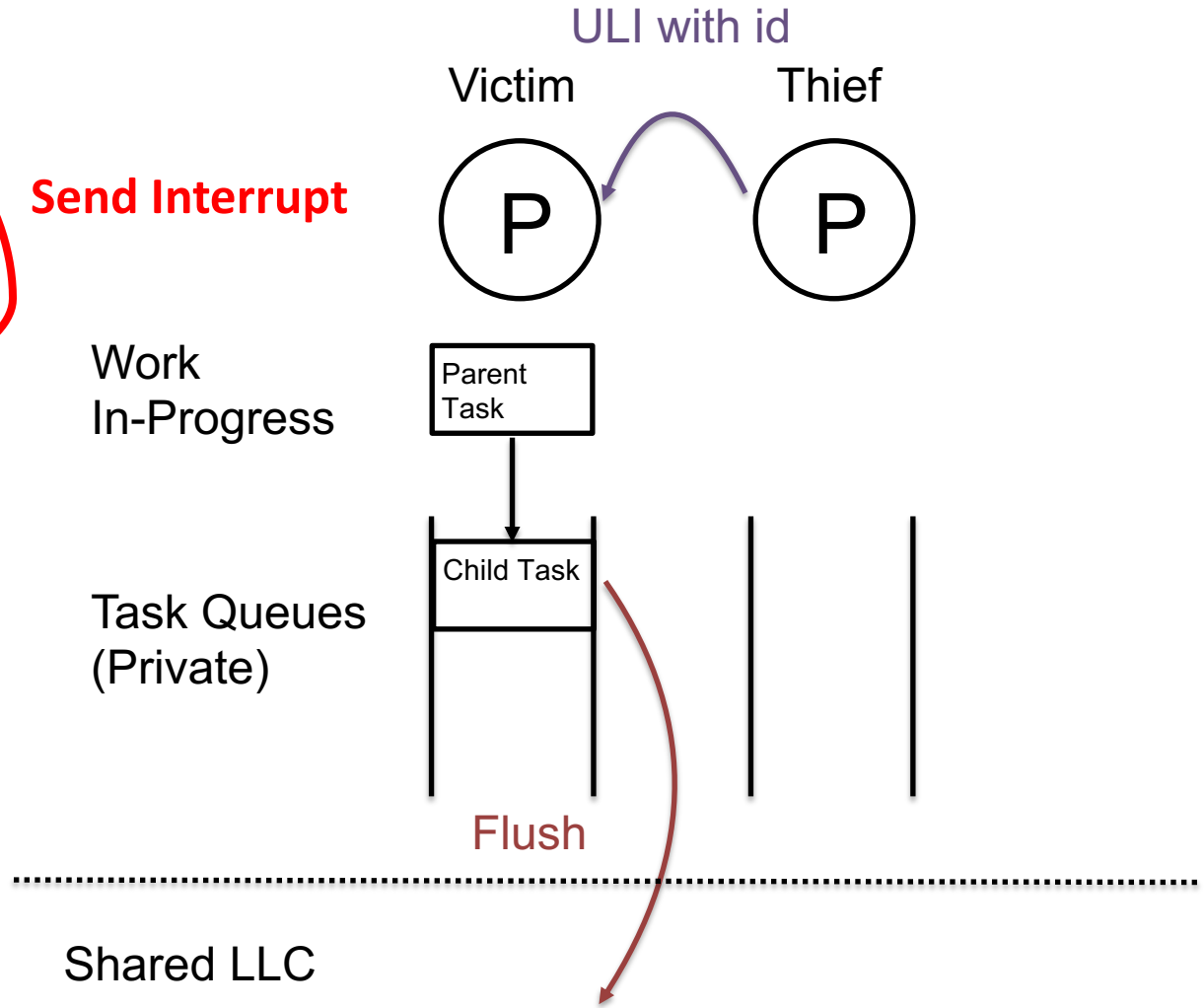
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence



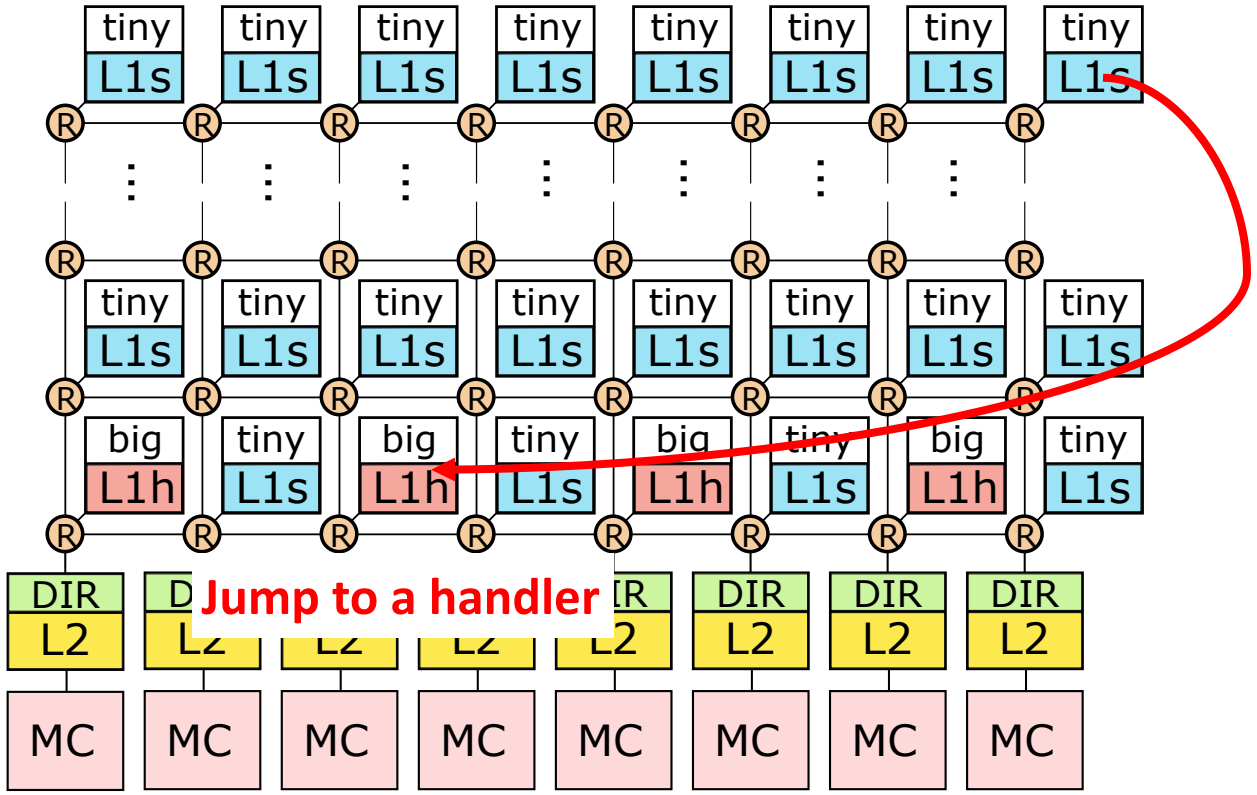
Implementing Direct-Task Stealing with ULI



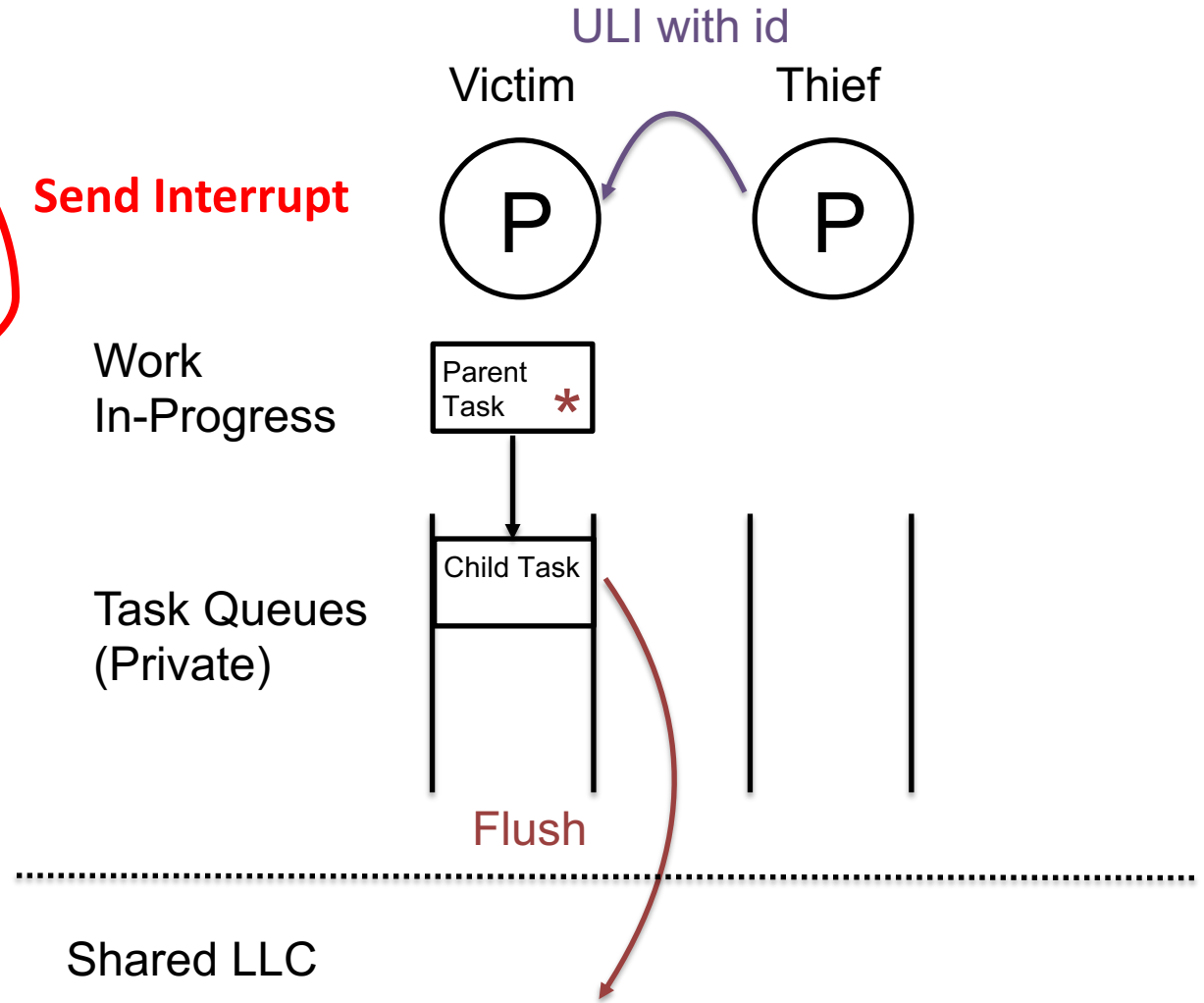
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence



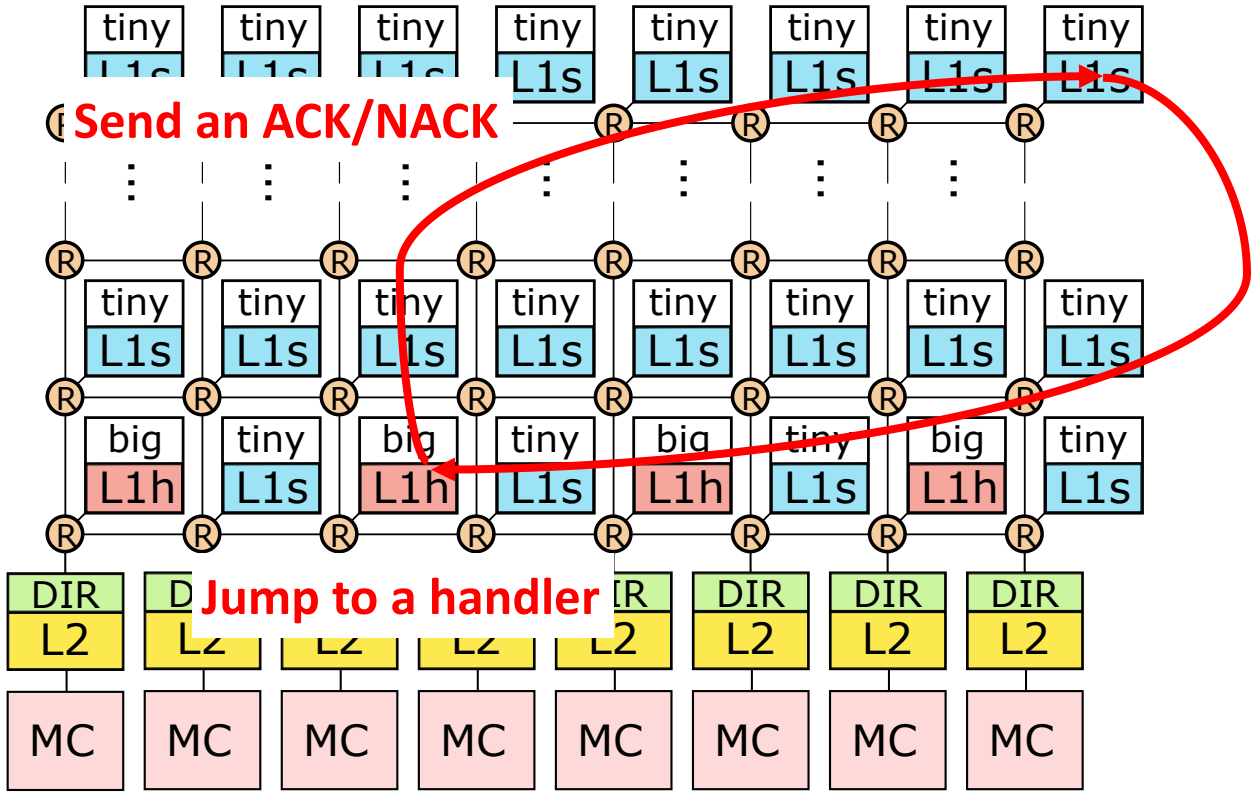
Implementing Direct-Task Stealing with ULI



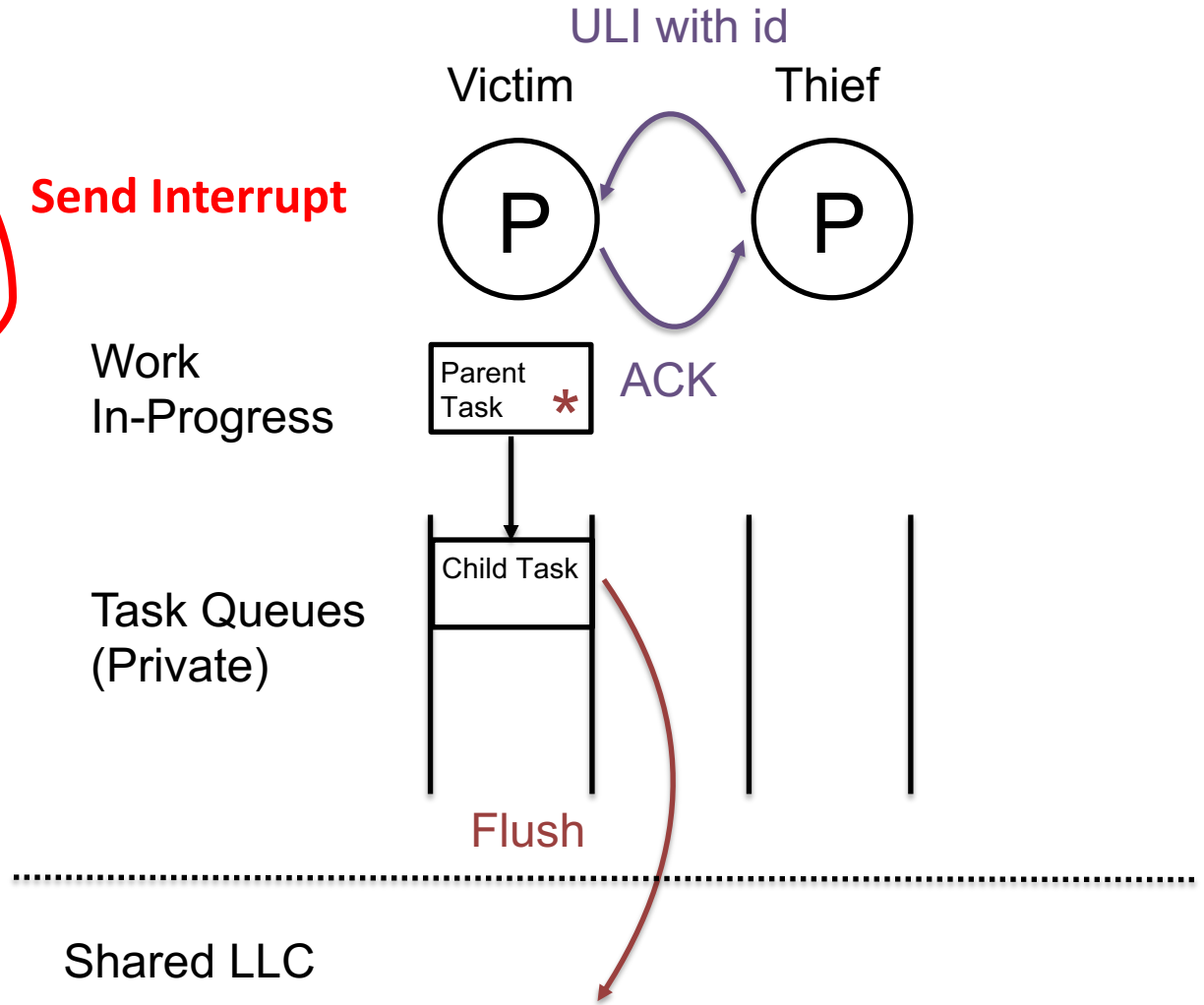
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence



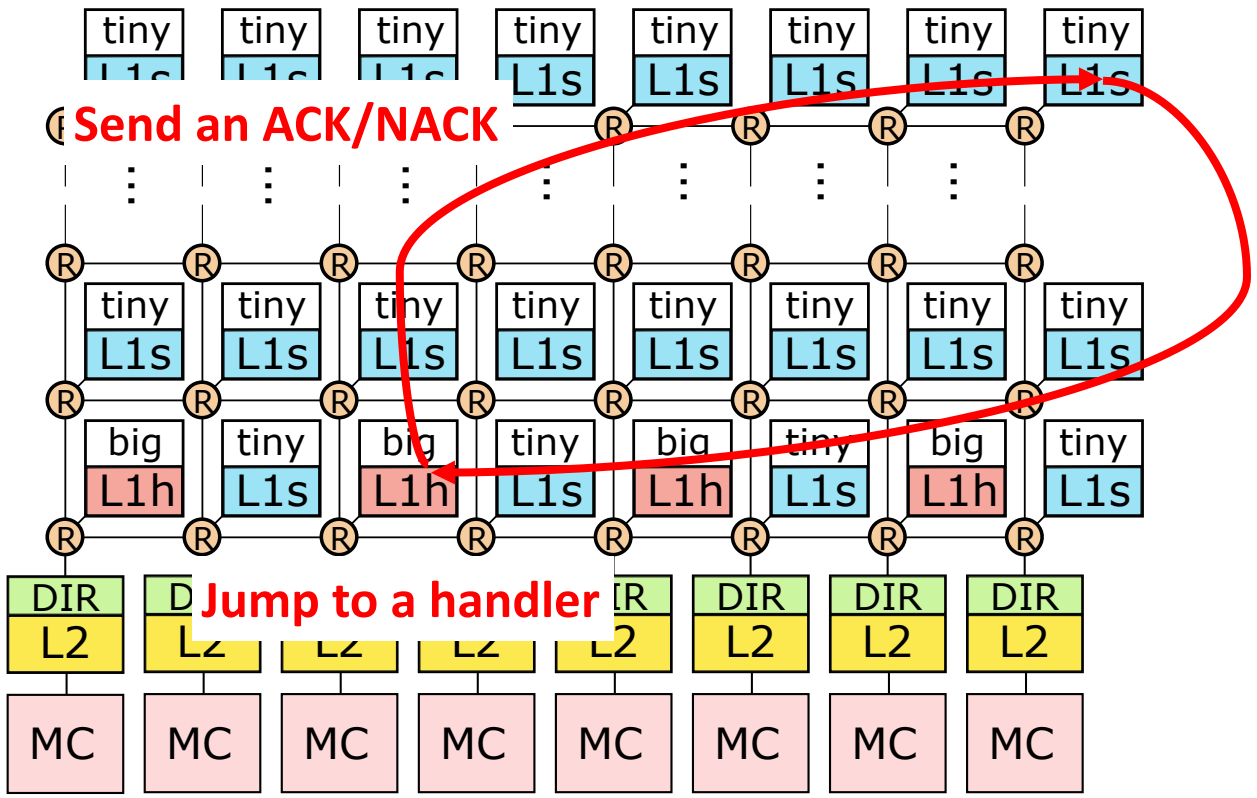
Implementing Direct-Task Stealing with ULI



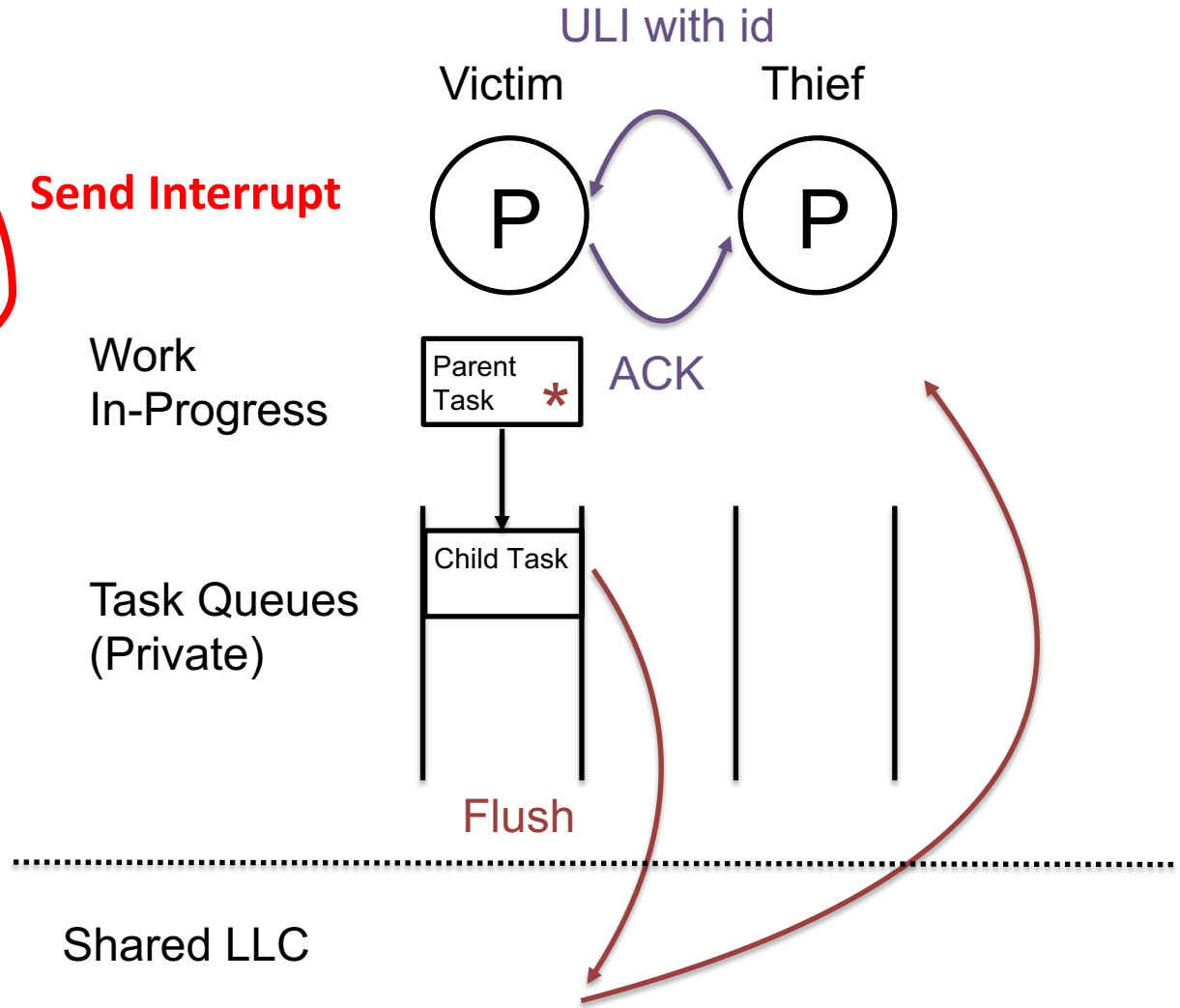
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence



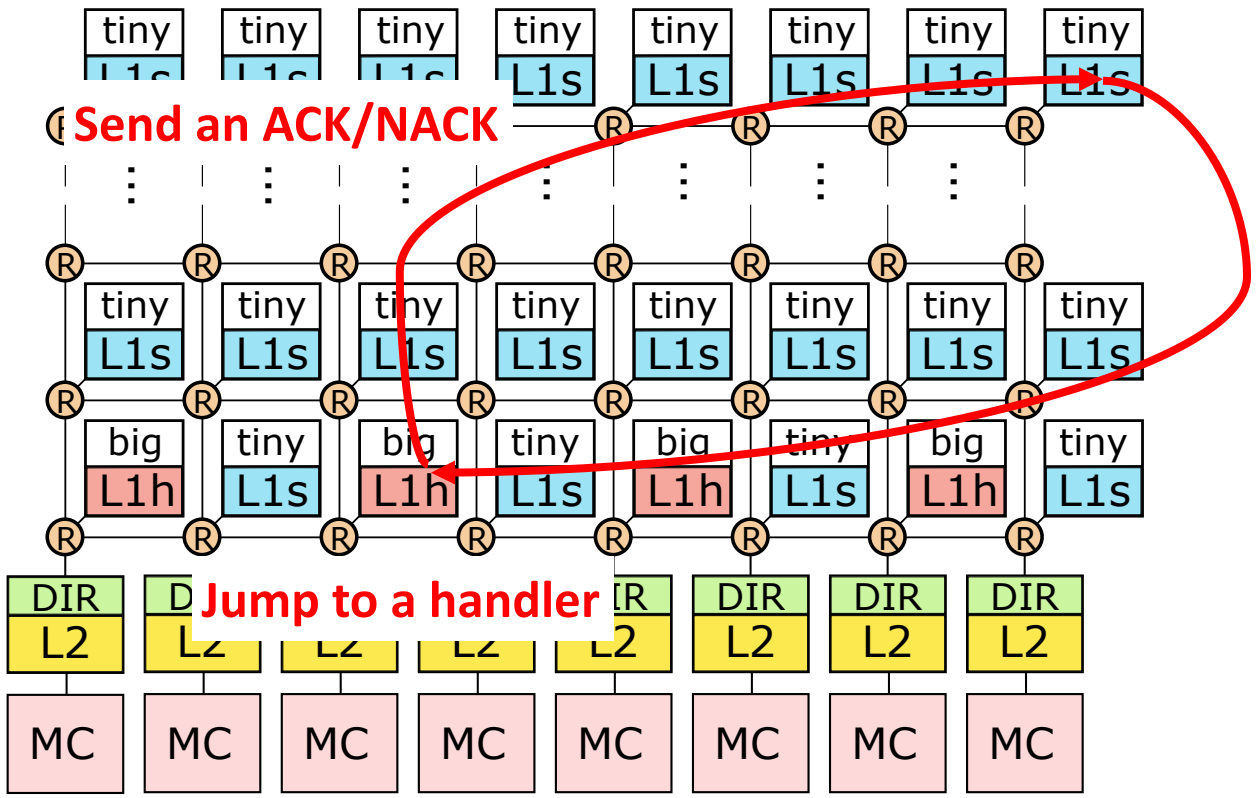
Implementing Direct-Task Stealing with ULI



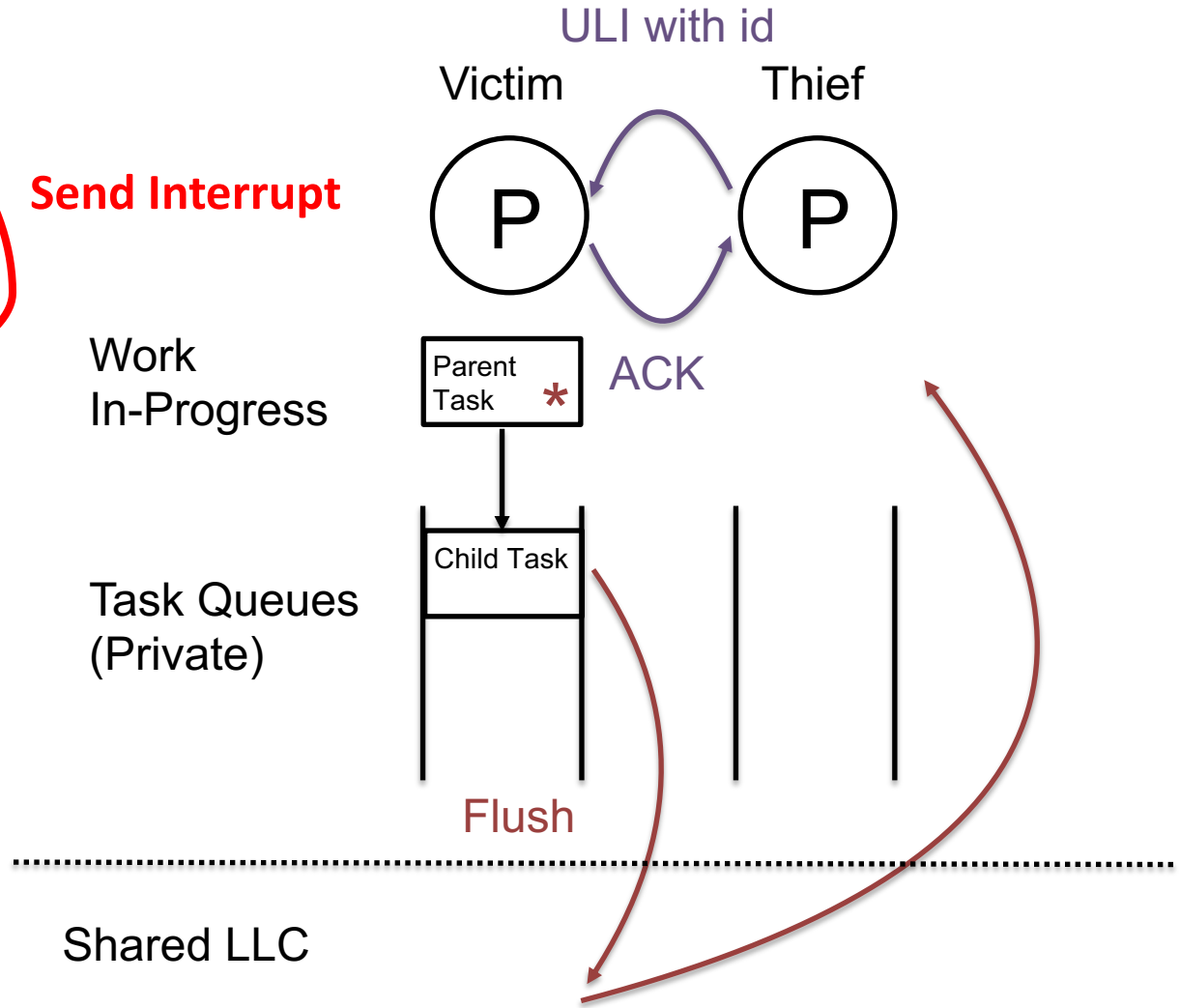
A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence



Implementing Direct-Task Stealing with ULI



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence



Work-Stealing Runtimes with DTS

- DTS achieves:

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task* t = task_queue[tid].dequeue();
    if (t) {
      t->execute();
      if (t->parent->child_stolen)
        amo_sub( t->parent->ref_count, 1 );
      else
        t->parent->ref_count -= 1;
    }
    else {
      t = steal_using_dts();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
  if ( p->has_stolen_child )
    cache_invalidate();
}
```

cache_invalidate();

cache_flush();

Work-Stealing Runtimes with DTS

- DTS achieves:
  - Access task queues without locking

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task* t = task_queue[tid].dequeue();
    if (t) {
      t->execute();
      if (t->parent->child_stolen)
        amo_sub( t->parent->ref_count, 1 );
      else
        t->parent->ref_count -= 1;
    }
    else {
      t = steal_using_dts();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
  if ( p->has_stolen_child )
    cache_invalidate();
}
```

uli_disable();

uli_enable();

cache_invalidate();

cache_flush();

Work-Stealing Runtimes with DTS

- DTS achieves:
  - Access task queues without locking
  - No AMO unless the parent has a child stolen

```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task* t = task_queue[tid].dequeue();
    if (t) {
      t->execute();
      if (t->parent->child_stolen)
        amo_sub( t->parent->ref_count, 1 );
      else
        t->parent->ref_count -= 1;
    }
    else {
      t = steal_using_dts();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
  if ( p->has_stolen_child )
    cache_invalidate();
}
```

uli_disable();

uli_enable();

cache_invalidate();

cache_flush();

Work-Stealing Runtimes with DTS

- DTS achieves:
  - Access task queues without locking
  - No AMO unless the parent has a child stolen
  - No invalidation unless a child is stolen

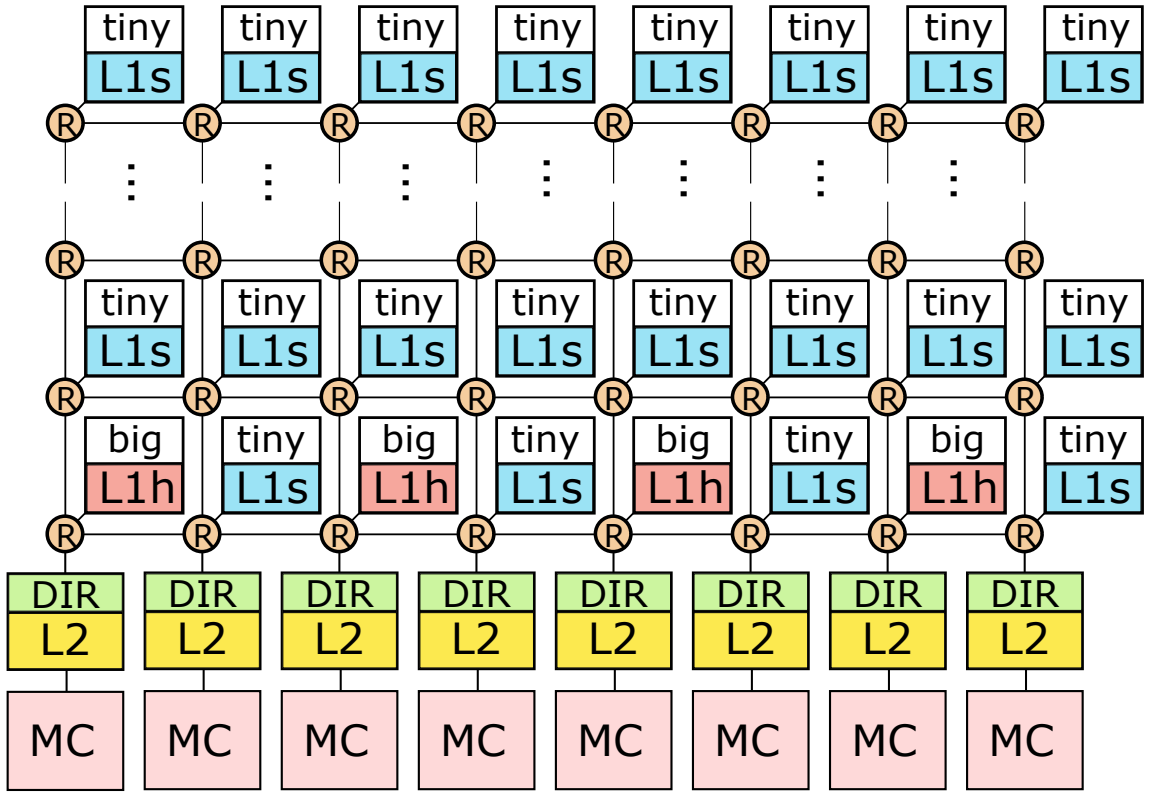
```
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    task* t = task_queue[tid].dequeue();
    if (t) {
      t->execute();
      if (t->parent->child_stolen)
        amo_sub( t->parent->ref_count, 1 );
      else
        t->parent->ref_count -= 1;
    }
    else {
      t = steal_using_dts();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
  if ( p->has_stolen_child )
    cache_invalidate();
}
```

uli_disable();

uli_enable();

cache_invalidate();

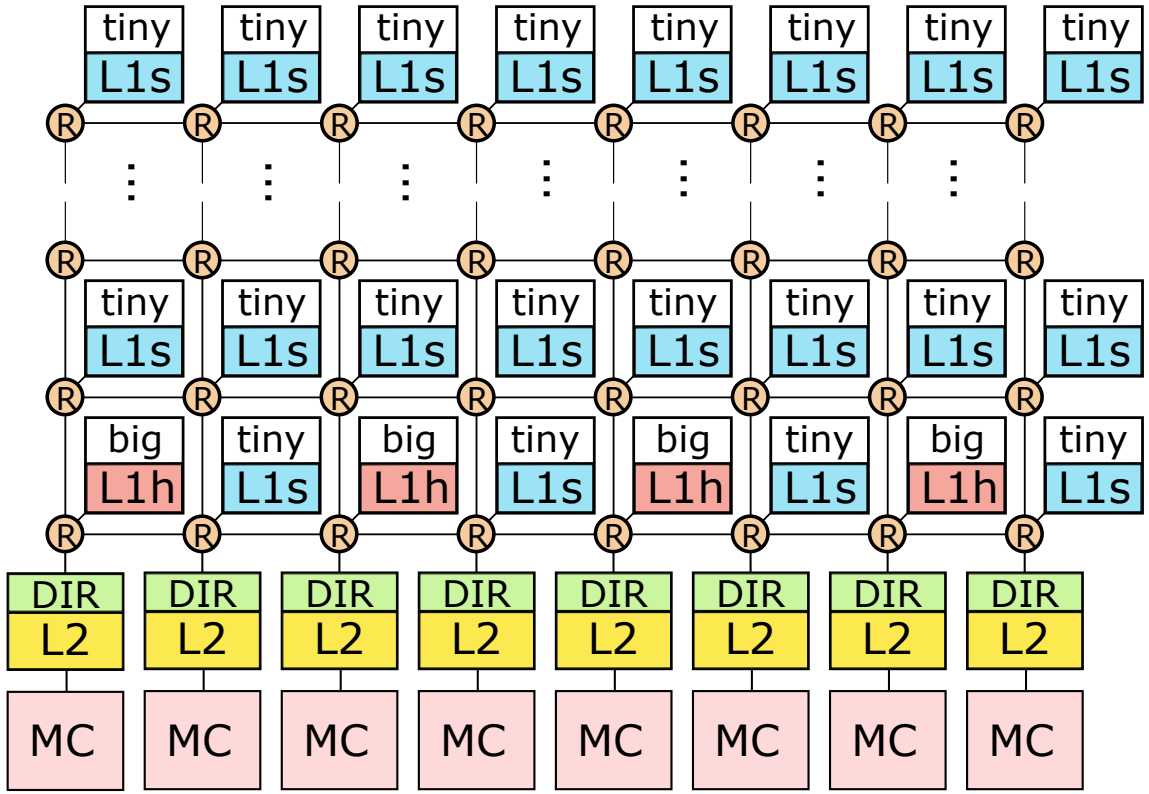
cache_flush();



A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- Background
- Implementing Work-Stealing Runtimes on HCC
- Direct Task Stealing
- **Evaluation**

Evaluation Methodology

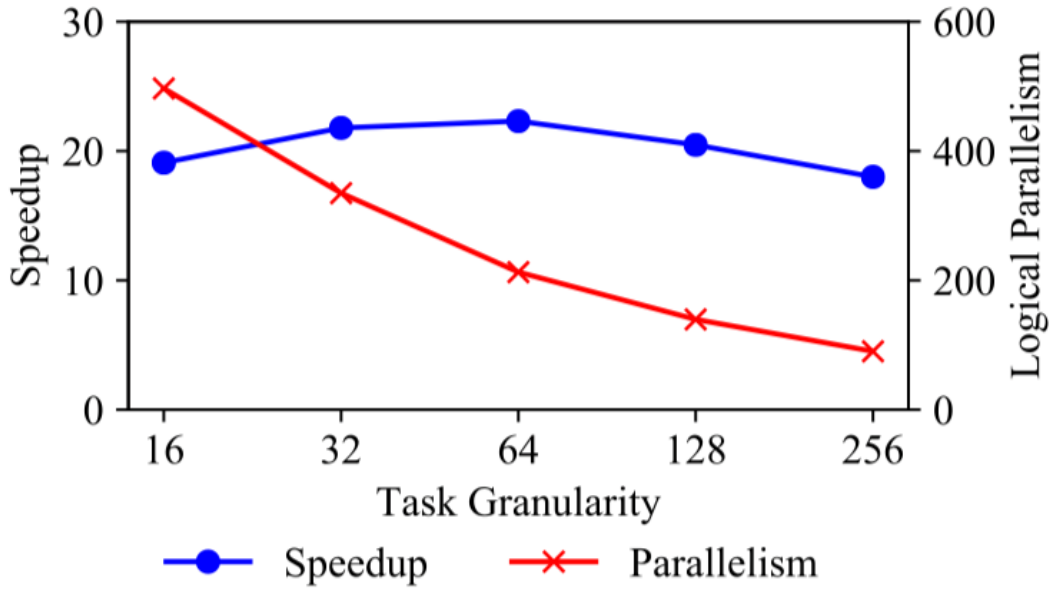


A big.TINY architecture combines a few big OOO cores with many tiny IO cores on a single die using heterogeneous cache coherence

- gem5 (Ruby and Garnet2.0) cycle-Level simulator
  - 4 big core: OOO, 64KB L1D cache
  - 60 tiny core: in-order, 4KB L1D cache
- Total cache capacity: 16 tiny cores = 1 big core
- Baselines:
  - O3x8: eight big cores
  - big.TINY/MESI
- big.TINY with HCC:
  - big.TINY/HCC
  - big.TINY/HCC-DTS

Evaluation Methodology

- 13 dynamic task-parallel application kernels from Cilk-5 and Ligra benchmark suites
- Optimize task granularity for the big.TINY/MESI baseline
- We use moderate input data sizes and moderate parallelism on a 64-core system to be representative of larger systems running larger input sizes (weak scaling)
- See paper for 256-core case study to validate our weak-scaling claim



Performance: big.TINY/MESI vs. O3x8

		Speedup over Serial IO			
Name	Input	O3×1	O3×4	O3×8	b.T/ MESI
cilk5-cs	3000000	1.65	4.92	9.78	18.70
cilk5-lu	128	2.48	9.46	17.24	23.93
cilk5-mm	256	11.38	11.76	22.04	41.23
cilk5-mt	8000	5.71	19.70	39.94	57.43
cilk5-nq	10	1.57	3.87	7.03	2.93
ligra-bc	rMat_100K	2.05	6.29	13.06	11.48
ligra-bf	rMat_200K	1.80	5.36	11.25	12.80
ligra-bfs	rMat_800K	2.23	6.23	12.70	15.63
ligra-bfsbv	rMat_500K	1.91	6.17	12.25	14.42
ligra-cc	rMat_500K	3.00	9.11	20.66	24.12
ligra-mis	rMat_100K	2.43	7.70	15.61	19.01
ligra-radii	rMat_200K	2.80	8.17	17.89	25.94
ligra-tc	rMat_200K	1.49	4.99	10.89	23.21
geomean		2.56	7.26	14.70	16.94

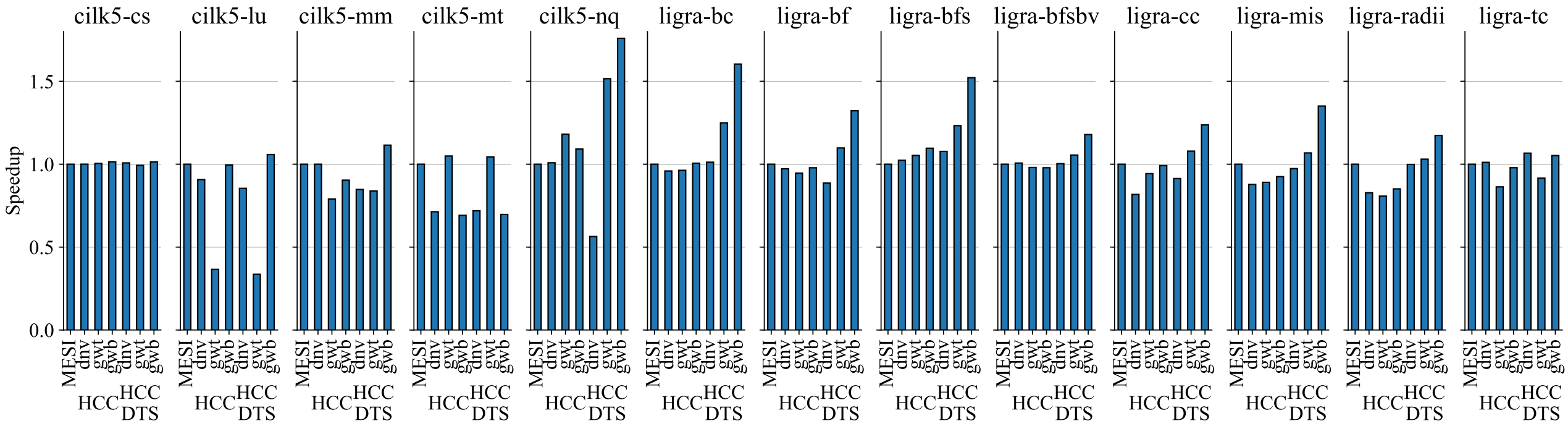
- Work-Stealing runtimes enable cooperative execution between big and tiny cores
- Total cache capacity: 4 big cores + 60 tiny cores = 7.8 big cores
- big.TINY achieves better performance by exploiting parallelism and cooperative execution

Performance: big.TINY/MESI vs. O3x8

		Speedup over Serial IO			
Name	Input	O3×1	O3×4	O3×8	b.T/ MESI
cilk5-cs	3000000	1.65	4.92	9.78	18.70
cilk5-lu	128	2.48	9.46	17.24	23.93
cilk5-mm	256	11.38	11.76	22.04	41.23
cilk5-mt	8000	5.71	19.70	39.94	57.43
cilk5-nq	10	1.57	3.87	7.03	2.93
ligra-bc	rMat_100K	2.05	6.29	13.06	11.48
ligra-bf	rMat_200K	1.80	5.36	11.25	12.80
ligra-bfs	rMat_800K	2.23	6.23	12.70	15.63
ligra-bfsbv	rMat_500K	1.91	6.17	12.25	14.42
ligra-cc	rMat_500K	3.00	9.11	20.66	24.12
ligra-mis	rMat_100K	2.43	7.70	15.61	19.01
ligra-radii	rMat_200K	2.80	8.17	17.89	25.94
ligra-tc	rMat_200K	1.49	4.99	10.89	23.21
geomean		2.56	7.26	14.70	16.94

- Work-Stealing runtimes enable cooperative execution between big and tiny cores
- Total cache capacity: 4 big cores + 60 tiny cores = 7.8 big cores
- big.TINY achieves better performance by exploiting parallelism and cooperative execution

Performance: big.TINY/HCC vs. big.TINY/MESI

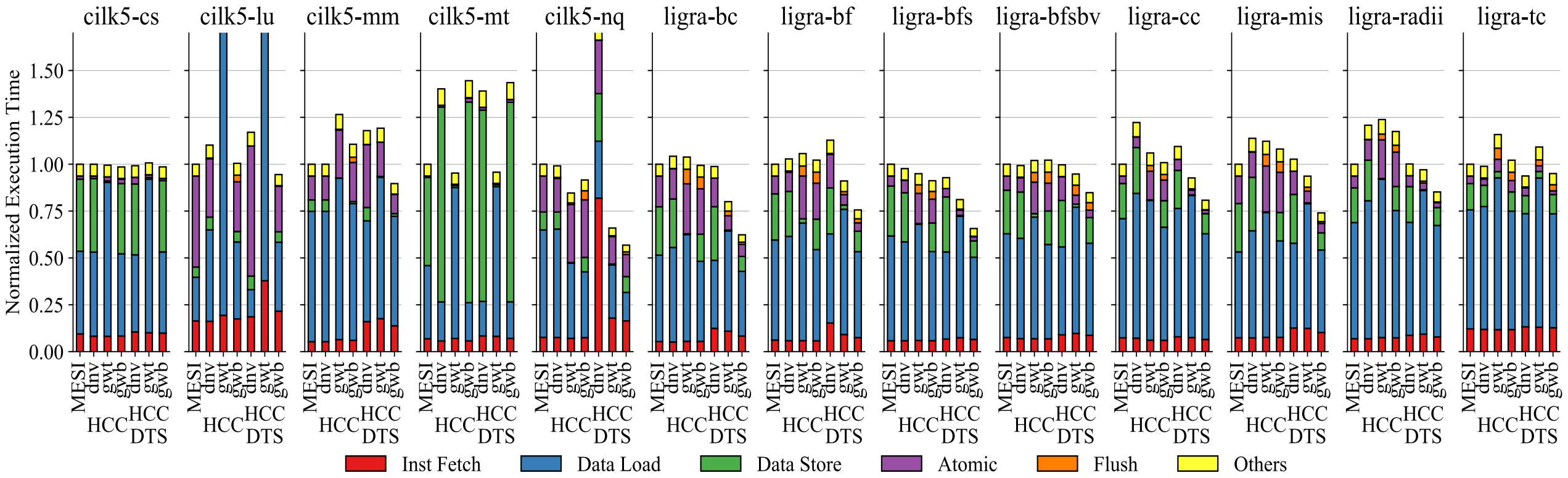


Big cores always use MESI, tiny cores use:

- dnv = DeNovo
- gwt = GPU-WT
- gwb = GPU-WB

- HCC configurations has slightly worse performance than big.TINY/MESI
- DTS improves performance of work-stealing runtimes on HCC

Execution Time Breakdown: big.TINY/HCC vs. big.TINY/MESI

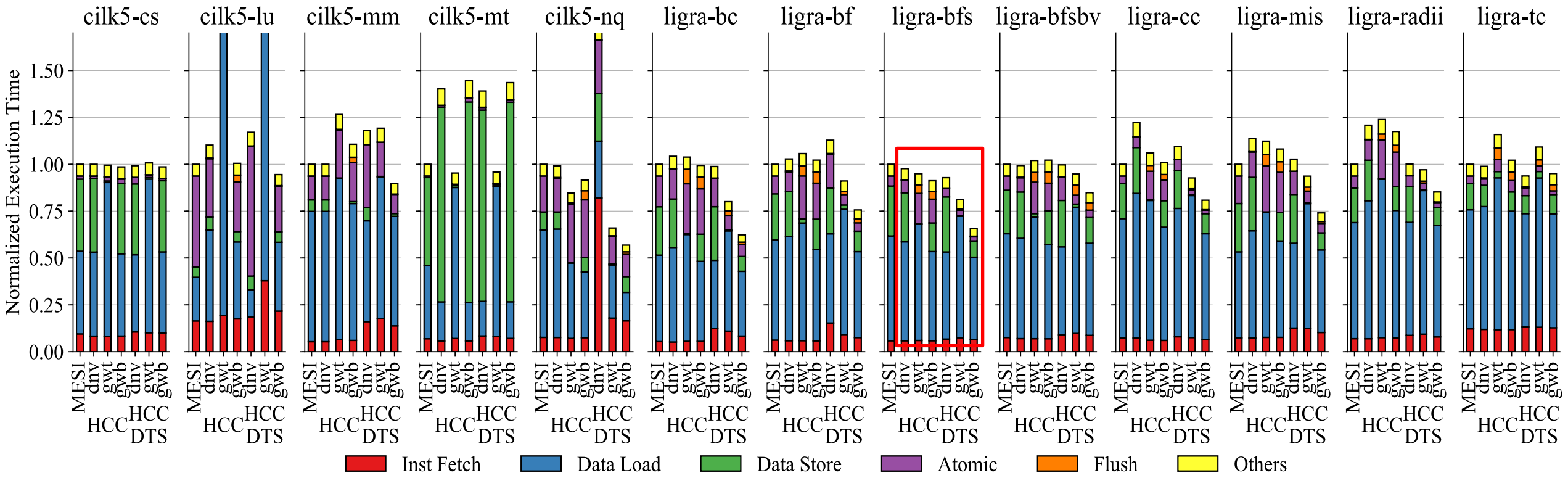


Big cores always use MESI, tiny cores use:

- dnv = DeNovo
- gwt = GPU-WT
- gwb = GPU-WB

- The overhead of HCC comes from data load, data store, and AMO
- DTS mitigates these overheads

Execution Time Breakdown: big.TINY/HCC vs. big.TINY/MESI



Big cores always use MESI, tiny cores use:

- dnv = DeNovo
- gwt = GPU-WT
- gwb = GPU-WB

- The overhead of HCC comes from data load, data store, and AMO
- DTS mitigates these overheads

Effects of DTS

- DTS reduces the number of cache invalidations

App	Invalidation Decrease (%)			Flush Decrease (%)	Hit Rate Increase (%)		
	dnv	gwt	gwb	gwb	dnv	gwt	gwb
cilk5-cs	99.42	99.28	99.50	98.86	1.80	2.45	1.30
cilk5-lu	98.83	99.78	99.53	98.40	1.12	7.12	2.94
cilk5-mm	99.22	99.67	99.62	99.12	30.03	42.19	36.80
cilk5-mt	99.88	99.73	99.93	99.82	12.45	2.70	6.56
cilk5-nq	97.74	97.88	98.32	95.84	16.84	28.87	27.04
ligra-bc	94.89	97.04	97.33	93.80	7.64	21.43	14.99
ligra-bf	29.02	38.14	40.24	21.63	7.22	17.14	11.17
ligra-bfs	94.18	95.85	95.90	91.23	3.48	15.76	8.00
ligra-bfsbv	39.31	47.36	50.74	29.46	3.10	12.65	7.56
ligra-cc	98.03	98.17	98.16	95.89	3.11	11.11	6.17
ligra-mis	97.35	98.28	98.36	96.16	5.62	16.29	11.10
ligra-radii	95.97	98.17	98.19	95.75	3.62	11.00	7.03
ligra-tc	10.83	15.99	17.02	7.52	1.59	3.55	3.02

Effects of DTS

- DTS reduces the number of cache invalidations
- DTS reduces the number of cache flushes

	Invalidation Decrease (%)			Flush Decrease (%)	Hit Rate Increase (%)		
App	dnv	gwt	gwb	gwb	dnv	gwt	gwb
cilk5-cs	99.42	99.28	99.50	98.86	1.80	2.45	1.30
cilk5-lu	98.83	99.78	99.53	98.40	1.12	7.12	2.94
cilk5-mm	99.22	99.67	99.62	99.12	30.03	42.19	36.80
cilk5-mt	99.88	99.73	99.93	99.82	12.45	2.70	6.56
cilk5-nq	97.74	97.88	98.32	95.84	16.84	28.87	27.04
ligra-bc	94.89	97.04	97.33	93.80	7.64	21.43	14.99
ligra-bf	29.02	38.14	40.24	21.63	7.22	17.14	11.17
ligra-bfs	94.18	95.85	95.90	91.23	3.48	15.76	8.00
ligra-bfsbv	39.31	47.36	50.74	29.46	3.10	12.65	7.56
ligra-cc	98.03	98.17	98.16	95.89	3.11	11.11	6.17
ligra-mis	97.35	98.28	98.36	96.16	5.62	16.29	11.10
ligra-radii	95.97	98.17	98.19	95.75	3.62	11.00	7.03
ligra-tc	10.83	15.99	17.02	7.52	1.59	3.55	3.02

Effects of DTS

- DTS reduces the number of cache invalidations
- DTS reduces the number of cache flushes
- DTS improves L1 hit rate

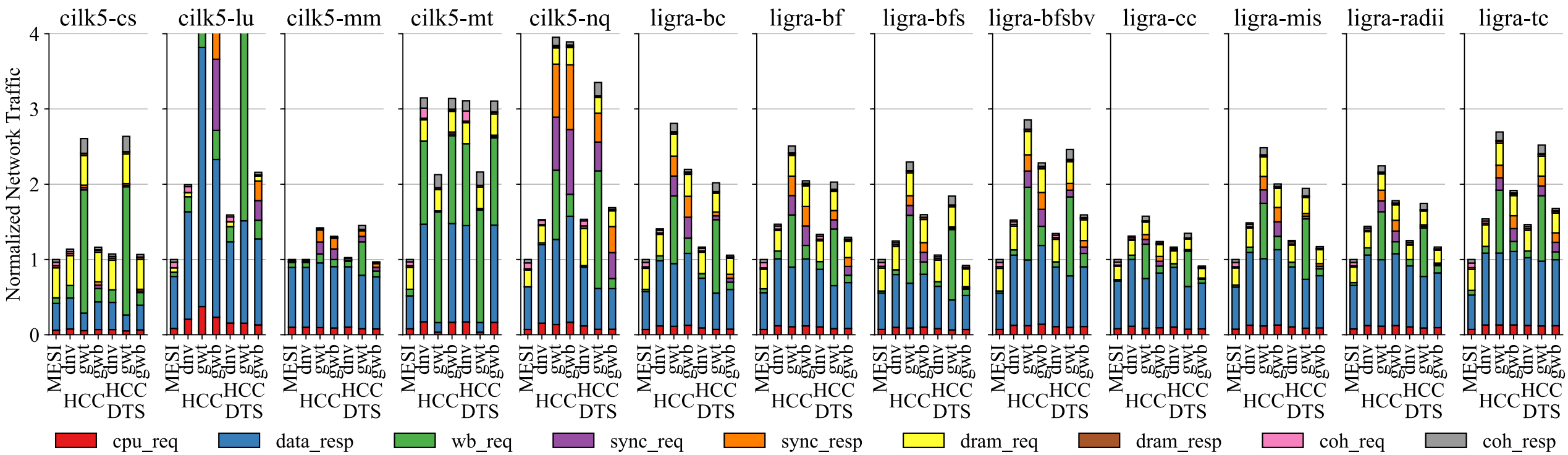
	Invalidation Decrease (%)			Flush Decrease (%)	Hit Rate Increase (%)		
App	dnv	gwt	gwb	gwb	dnv	gwt	gwb
cilk5-cs	99.42	99.28	99.50	98.86	1.80	2.45	1.30
cilk5-lu	98.83	99.78	99.53	98.40	1.12	7.12	2.94
cilk5-mm	99.22	99.67	99.62	99.12	30.03	42.19	36.80
cilk5-mt	99.88	99.73	99.93	99.82	12.45	2.70	6.56
cilk5-nq	97.74	97.88	98.32	95.84	16.84	28.87	27.04
ligra-bc	94.89	97.04	97.33	93.80	7.64	21.43	14.99
ligra-bf	29.02	38.14	40.24	21.63	7.22	17.14	11.17
ligra-bfs	94.18	95.85	95.90	91.23	3.48	15.76	8.00
ligra-bfsbv	39.31	47.36	50.74	29.46	3.10	12.65	7.56
ligra-cc	98.03	98.17	98.16	95.89	3.11	11.11	6.17
ligra-mis	97.35	98.28	98.36	96.16	5.62	16.29	11.10
ligra-radii	95.97	98.17	98.19	95.75	3.62	11.00	7.03
ligra-tc	10.83	15.99	17.02	7.52	1.59	3.55	3.02

Effects of DTS

- DTS reduces the number of cache invalidations
- DTS reduces the number of cache flushes
- DTS improves L1 hit rate
- DTS improves overall performance

	Invalidation Decrease (%)			Flush Decrease (%)	Hit Rate Increase (%)		
App	dnv	gwt	gwb	gwb	dnv	gwt	gwb
cilk5-cs	99.42	99.28	99.50	98.86	1.80	2.45	1.30
cilk5-lu	98.83	99.78	99.53	98.40	1.12	7.12	2.94
cilk5-mm	99.22	99.67	99.62	99.12	30.03	42.19	36.80
cilk5-mt	99.88	99.73	99.93	99.82	12.45	2.70	6.56
cilk5-nq	97.74	97.88	98.32	95.84	16.84	28.87	27.04
ligra-bc	94.89	97.04	97.33	93.80	7.64	21.43	14.99
ligra-bf	29.02	38.14	40.24	21.63	7.22	17.14	11.17
ligra-bfs	94.18	95.85	95.90	91.23	3.48	15.76	8.00
ligra-bfsbv	39.31	47.36	50.74	29.46	3.10	12.65	7.56
ligra-cc	98.03	98.17	98.16	95.89	3.11	11.11	6.17
ligra-mis	97.35	98.28	98.36	96.16	5.62	16.29	11.10
ligra-radii	95.97	98.17	98.19	95.75	3.62	11.00	7.03
ligra-tc	10.83	15.99	17.02	7.52	1.59	3.55	3.02

NOC TRAFFIC: BIG.TINY/HCC VS. BIG.TINY/MESI

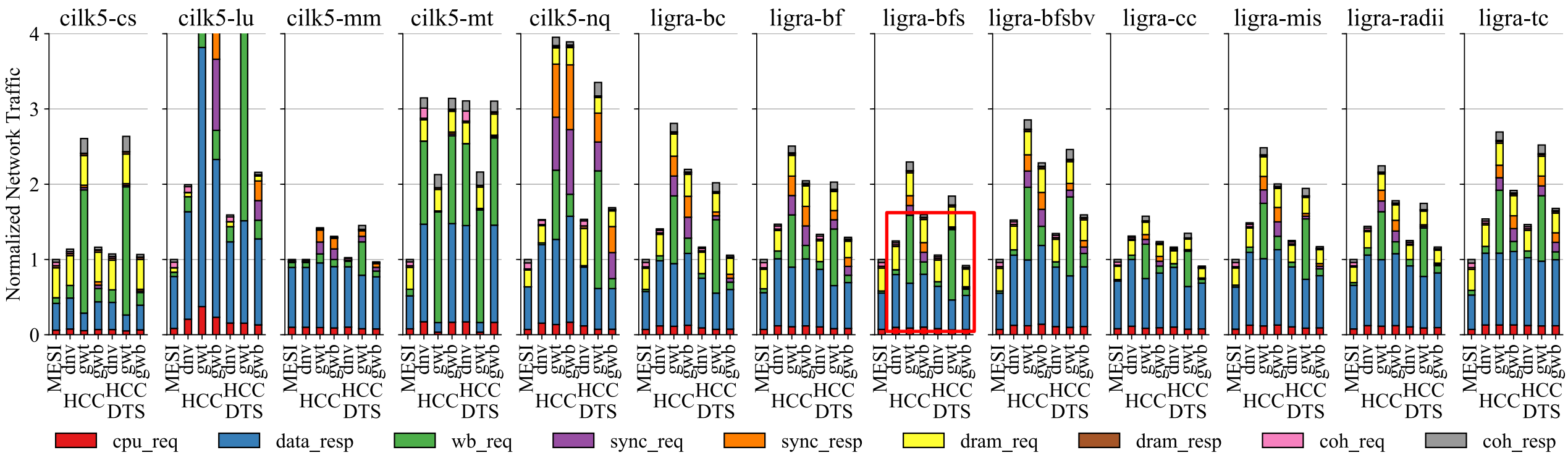


Big cores always use MESI, tiny cores use:

- dnv = DeNovo
- gwt = GPU-WT
- gwb = GPU-WB

- HCC configurations increase network traffic due to invalidations and flushes
- DTS can reduce network traffic, therefore reduce energy
- HCC+DTS achieves similar energy with big.TINY/MESI

NOC TRAFFIC: BIG.TINY/HCC VS. BIG.TINY/MESI

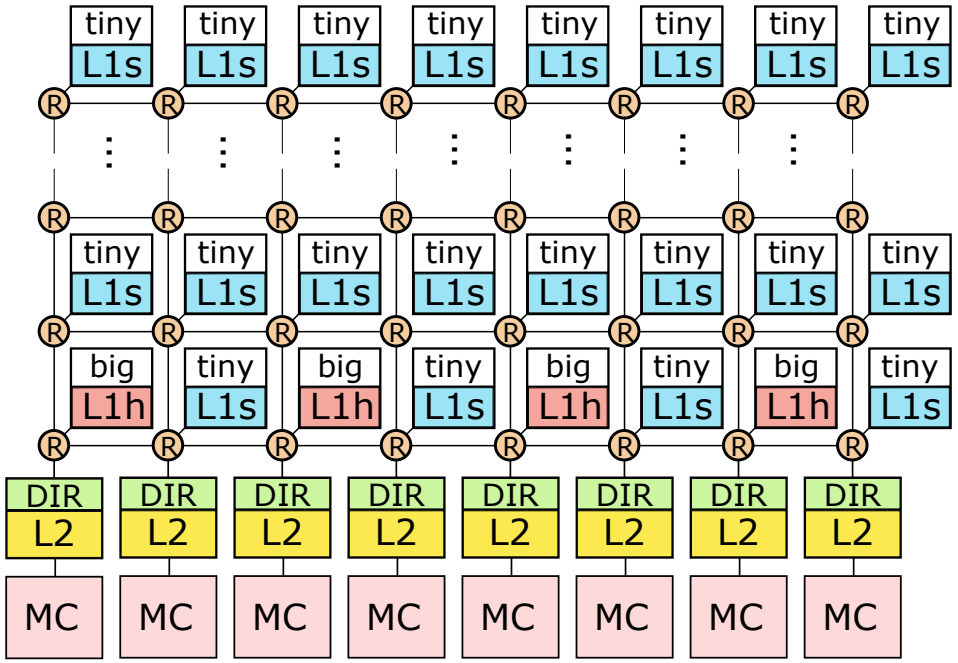


Big cores always use MESI, tiny cores use:

- dnv = DeNovo
- gwt = GPU-WT
- gwb = GPU-WB

- HCC configurations increase network traffic due to invalidations and flushes
- DTS can reduce network traffic, therefore reduce energy
- HCC+DTS achieves similar energy with big.TINY/MESI

Take-Away Points



This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program cosponsored by DARPA, and equipment donations from Intel.

- We present a work-stealing runtime for HCC systems:
  - Provides a Cilk/TBB-like programming model
  - Enables cooperative execution between big and tiny cores
- DTS improves performance and energy efficiency
- Using DTS, HCC systems achieve better performance and similar energy efficiency compared to full-system hardware-based cache coherence