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**

**Small Core Count**
- Cavium ThunderX
- Tilera TILE64
- Intel Xeon Phi

48 Cores

Hardware-Based Cache Coherence

**Large Core Count**
- NVIDIA GV100 GPU
- Celerity
- KiloCore
- Adapteva Epiphany

72 SM
511 Cores
1000 Cores
1024 Cores

Software-Centric Cache Coherence / No Coherence
```c
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);
}
```

- 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

- 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**

- 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.

Motivation • Background • Implementing Work-Stealing on HCC • DTS • Evaluation
EFFICIENTLY SUPPORTING DYNAMIC TASK PARALLELISM ON HCC

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

<table>
<thead>
<tr>
<th></th>
<th>Stale Data Invalidation</th>
<th>Dirty Data Propagation</th>
<th>Write Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI</td>
<td>Writer</td>
<td>Owner, Write-Back</td>
<td>Cache Line</td>
</tr>
<tr>
<td>DeNovo</td>
<td>Reader</td>
<td>Owner, Write-Back</td>
<td>Flexible</td>
</tr>
<tr>
<td>GPU-WT</td>
<td>Reader</td>
<td>No-Owner, Write-Through</td>
<td>Word</td>
</tr>
<tr>
<td>GPU-WB</td>
<td>Reader</td>
<td>No-Owner, Write-Back</td>
<td>Word</td>
</tr>
</tbody>
</table>


**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

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

```c
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

```cpp
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;
    }
}
```
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 );
        }
    }
}

Check local task queue
WORK-STEALING RUNTIMES

```c
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);
            }
        }
    }
}
```

Check local task queue

Execute dequeued task

Motivation • Background • Implementing Work-Stealing on HCC • DTS • Evaluation
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 );
            }
        }
    }
}

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**

<table>
<thead>
<tr>
<th>Task Queues</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Task A" /></td>
</tr>
<tr>
<td>Core 0</td>
</tr>
</tbody>
</table>

**Motivation**

**Background**

Implementing Work-Stealing on HCC

**DTS**

**Evaluation**

```cpp
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 );
            }
        }
    }
}
```
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 );
            }
        }
    }
}
```cpp
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[vid].lock_acquire();
            t = task_queue[vid].steal();
            task_queue[tid].lock_release();
            if (t) {
                t->execute();
                amo_sub( t->parent->ref_count, 1 );
            }
        }
    }
}
```
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 );
            }
        }
    }
}
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 );
            }
        }
    }
}
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);
            }
        }
    }
}
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**

```cpp
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);
            }
        }
    }
}
```
EFFICIENTLY SUPPORTING DYNAMIC TASK PARALLELISM ON HCC

- **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.
**Work-Stealing Runtimes on Software-Centric 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

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 ON SOFTWARE-CENTRIC CACHE COHERENCE

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

```cpp
void task::wait( task* p ) {
  while ( p->ref_count > 0 ) {
    cache Invalidate();
    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[vid].lock_acquire();
      t = task_queue[vid].steal();
      task_queue[tid].lock_release();
      if (t) {
        t->execute();
        amo_sub(t->parent->ref_count, 1 );
      }
    }
  }
}
```
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
• Supporting shared queues:
  - Lock-acquire -> invalidation
  - Lock-release -> cache flush

• Stolen task on HCC:
  - Invalidate before execution
  - Flush after execution

• Ensure parent-child synchronization

```cpp
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 ON SOFTWARE-CENTRIC CACHE COHERENCE

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

- Lock-acquire -> invalidation
- Lock-release -> 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.

- 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.

- 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.
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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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
EFFICIENTLY SUPPORTING DYNAMIC TASK PARALLELISM ON HCC

- 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.
• 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.
What we want to achieve

```cpp
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();
    }
}
```
What we want to achieve

- No Inv/Flush when accessing the local task queue

```cpp
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 );
            }
        }
    }
}
```
• What we want to achieve
  - No Inv/Flush when accessing the local task queue
  - No invalidation if children not stolen

```cpp
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 );
    }
  }
}
```
• 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

```cpp
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 );
     }
   }
 }
}
```
- 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

```cpp
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 );
      }
    }
  }
}
```
**USER-LEVEL INTERRUPT (ULI)**

- 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].

![Diagram of a big.TINY architecture](image)

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].


**USER-LEVEL INTERRUPT (ULI)**

- 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].

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

---


**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].

---


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

**Motivation**

**Background**

**Implementing Work-Stealing on HCC**

**DTS**

**Evaluation**

---

**Task Queues**

(Private)

**Shared LLC**

**Send Interrupt**

ULI with id

Victim

Thief

Parent Task

Child Task

Work In-Progress

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

```cpp
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_flush();
}
```
DTS achieves:
- Access task queues without locking

```cpp
void task::wait( task* p ) {
    uli_disable();
    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();
}
```
WORK-STEALING RUNTIMES WITH DTS

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

```cpp
void task::wait( task* p ) {
  uli_disable();
  while ( p->ref_count > 0 ) {
    task* t = task_queue[tid].dequeue();
    uli_enable();
    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
  - No AMO unless the parent has a child stolen
  - No invalidation unless a child is stolen

```cpp
void task::wait( task* p ) {
    uli_disable();
    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;
        }
    }
    uli_enable();
}
```

- **Motivation**
- **Background**
- **Implementing Work-Stealing on HCC**
- **DTS**
- **Evaluation**
EFFICIENTLY SUPPORTING DYNAMIC TASK PARALLELISM ON HCC

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

- 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

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

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

- 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
We port 13 dynamic task-parallel applications from Cilk

### C. Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>O3×1</th>
<th>O3×4</th>
<th>O3×8</th>
<th>MESI</th>
</tr>
</thead>
<tbody>
<tr>
<td>cilk5-cs</td>
<td>30000000</td>
<td>1.65</td>
<td>4.92</td>
<td>9.78</td>
<td>18.70</td>
</tr>
<tr>
<td>cilk5-lu</td>
<td>128</td>
<td>2.48</td>
<td>9.46</td>
<td>17.24</td>
<td>23.93</td>
</tr>
<tr>
<td>cilk5-mm</td>
<td>256</td>
<td>11.38</td>
<td>11.76</td>
<td>22.04</td>
<td>41.23</td>
</tr>
<tr>
<td>cilk5-mt</td>
<td>8000</td>
<td>5.71</td>
<td>19.70</td>
<td>39.94</td>
<td>57.43</td>
</tr>
<tr>
<td>cilk5-nq</td>
<td>10</td>
<td>1.57</td>
<td>3.87</td>
<td>7.03</td>
<td>2.93</td>
</tr>
<tr>
<td>ligra-bc</td>
<td>rMat_100K</td>
<td>2.05</td>
<td>6.29</td>
<td>13.06</td>
<td>11.48</td>
</tr>
<tr>
<td>ligra-bf</td>
<td>rMat_200K</td>
<td>1.80</td>
<td>5.36</td>
<td>11.25</td>
<td>12.80</td>
</tr>
<tr>
<td>ligra-bfs</td>
<td>rMat_800K</td>
<td>2.23</td>
<td>6.23</td>
<td>12.70</td>
<td>15.63</td>
</tr>
<tr>
<td>ligra-bfsbv</td>
<td>rMat_500K</td>
<td>1.91</td>
<td>6.17</td>
<td>12.25</td>
<td>14.42</td>
</tr>
<tr>
<td>ligra-cc</td>
<td>rMat_500K</td>
<td>3.00</td>
<td>9.11</td>
<td>20.66</td>
<td>24.12</td>
</tr>
<tr>
<td>ligra-mis</td>
<td>rMat_100K</td>
<td>2.43</td>
<td>7.70</td>
<td>15.61</td>
<td>19.01</td>
</tr>
<tr>
<td>ligra-radii</td>
<td>rMat_200K</td>
<td>2.80</td>
<td>8.17</td>
<td>17.89</td>
<td>25.94</td>
</tr>
<tr>
<td>ligra-tc</td>
<td>rMat_200K</td>
<td>1.49</td>
<td>4.99</td>
<td>10.89</td>
<td>23.21</td>
</tr>
<tr>
<td>geomean</td>
<td></td>
<td>2.56</td>
<td>7.26</td>
<td>14.70</td>
<td>16.94</td>
</tr>
</tbody>
</table>

- 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
Motivation • Background • Implementing Work-Stealing on HCC • DTS • Evaluation

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
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

<table>
<thead>
<tr>
<th>App</th>
<th>Invalidation Decrease (%)</th>
<th>Flush Decrease (%)</th>
<th>Hit Rate Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dnv</td>
<td>gwt</td>
<td>gwb</td>
</tr>
<tr>
<td>cilk5-cs</td>
<td>99.42</td>
<td>99.28</td>
<td>99.50</td>
</tr>
<tr>
<td>cilk5-lu</td>
<td>98.83</td>
<td>99.78</td>
<td>99.53</td>
</tr>
<tr>
<td>cilk5-mm</td>
<td>99.22</td>
<td>99.67</td>
<td>99.62</td>
</tr>
<tr>
<td>cilk5-mt</td>
<td>99.88</td>
<td>99.73</td>
<td>99.93</td>
</tr>
<tr>
<td>cilk5-nq</td>
<td>97.74</td>
<td>97.88</td>
<td>98.32</td>
</tr>
<tr>
<td>ligra-bc</td>
<td>94.89</td>
<td>97.04</td>
<td>97.33</td>
</tr>
<tr>
<td>ligra-bf</td>
<td>29.02</td>
<td>38.14</td>
<td>40.24</td>
</tr>
<tr>
<td>ligra-bfs</td>
<td>94.18</td>
<td>95.85</td>
<td>95.90</td>
</tr>
<tr>
<td>ligra-bfsb</td>
<td>39.31</td>
<td>47.36</td>
<td>50.74</td>
</tr>
<tr>
<td>ligra-cc</td>
<td>98.03</td>
<td>98.17</td>
<td>98.16</td>
</tr>
<tr>
<td>ligra-mis</td>
<td>97.35</td>
<td>98.28</td>
<td>98.36</td>
</tr>
<tr>
<td>ligra-radii</td>
<td>95.97</td>
<td>98.17</td>
<td>98.19</td>
</tr>
<tr>
<td>ligra-tc</td>
<td>10.83</td>
<td>15.99</td>
<td>17.02</td>
</tr>
</tbody>
</table>
DTS reduces the number of cache invalidations

- DTS reduces the number of cache flushes

<table>
<thead>
<tr>
<th>App</th>
<th>Invalidation Decrease (%)</th>
<th>Flush Decrease (%)</th>
<th>Hit Rate Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dnv gwt gwb</td>
<td>gwb</td>
<td>dnv gwt gwb</td>
</tr>
<tr>
<td>cilk5-cs</td>
<td>99.42 99.28 99.50</td>
<td>98.86</td>
<td>1.80 2.45 1.30</td>
</tr>
<tr>
<td>cilk5-lu</td>
<td>98.83 99.78 99.53</td>
<td>98.40</td>
<td>1.12 7.12 2.94</td>
</tr>
<tr>
<td>cilk5-mm</td>
<td>99.22 99.67 99.62</td>
<td>99.12</td>
<td>30.03 42.19 36.80</td>
</tr>
<tr>
<td>cilk5-ct</td>
<td>99.88 99.73 99.93</td>
<td>99.82</td>
<td>12.45 2.70 6.56</td>
</tr>
<tr>
<td>cilk5-mm</td>
<td>97.74 97.88 98.32</td>
<td>95.84</td>
<td>16.84 28.87 27.04</td>
</tr>
<tr>
<td>ligra-bc</td>
<td>94.89 97.04 97.33</td>
<td>93.80</td>
<td>7.64 21.43 14.99</td>
</tr>
<tr>
<td>ligra-bf</td>
<td>29.02 38.14 40.24</td>
<td>21.63</td>
<td>7.22 17.14 11.17</td>
</tr>
<tr>
<td>ligra-bsf</td>
<td>94.18 95.85 95.90</td>
<td>91.23</td>
<td>3.48 15.76 8.00</td>
</tr>
<tr>
<td>ligra-bfsbv</td>
<td>39.31 47.36 50.74</td>
<td>29.46</td>
<td>3.10 12.65 7.56</td>
</tr>
<tr>
<td>ligra-cc</td>
<td>98.03 98.17 98.16</td>
<td>95.89</td>
<td>3.11 11.11 6.17</td>
</tr>
<tr>
<td>ligra-mis</td>
<td>97.35 98.28 98.36</td>
<td>96.16</td>
<td>5.62 16.29 11.10</td>
</tr>
<tr>
<td>ligra-radii</td>
<td>95.97 98.17 98.19</td>
<td>95.75</td>
<td>3.62 11.00 7.03</td>
</tr>
<tr>
<td>ligra-tc</td>
<td>10.83 15.99 17.02</td>
<td>7.52</td>
<td>1.59 3.55 3.02</td>
</tr>
</tbody>
</table>
DTS reduces the number of cache invalidations
DTS reduces the number of cache flushes
DTS improves L1 hit rate

![Table](image.png)
• DTS reduces the number of cache invalidations
• DTS reduces the number of cache flushes
• DTS improves L1 hit rate
• DTS improves overall performance

<table>
<thead>
<tr>
<th>App</th>
<th>Invalidation Decrease (%)</th>
<th>Flush Decrease (%)</th>
<th>Hit Rate Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dnv</td>
<td>gwt</td>
<td>gwb</td>
</tr>
<tr>
<td>cilk5-cs</td>
<td>99.42</td>
<td>99.28</td>
<td>99.50</td>
</tr>
<tr>
<td>cilk5-lu</td>
<td>98.83</td>
<td>99.78</td>
<td>99.53</td>
</tr>
<tr>
<td>cilk5-mm</td>
<td>99.22</td>
<td>99.67</td>
<td>99.62</td>
</tr>
<tr>
<td>cilk5-mt</td>
<td>99.88</td>
<td>99.73</td>
<td>99.93</td>
</tr>
<tr>
<td>cilk5-nq</td>
<td>97.74</td>
<td>97.88</td>
<td>98.32</td>
</tr>
<tr>
<td>ligra-bc</td>
<td>94.89</td>
<td>97.04</td>
<td>97.33</td>
</tr>
<tr>
<td>ligra-bf</td>
<td>29.02</td>
<td>38.14</td>
<td>40.24</td>
</tr>
<tr>
<td>ligra-bfs</td>
<td>94.18</td>
<td>95.85</td>
<td>95.90</td>
</tr>
<tr>
<td>ligra-bfsbv</td>
<td>39.31</td>
<td>47.36</td>
<td>50.74</td>
</tr>
<tr>
<td>ligra-cc</td>
<td>98.03</td>
<td>98.17</td>
<td>98.16</td>
</tr>
<tr>
<td>ligra-mis</td>
<td>97.35</td>
<td>98.28</td>
<td>98.36</td>
</tr>
<tr>
<td>ligra-radii</td>
<td>95.97</td>
<td>98.17</td>
<td>98.19</td>
</tr>
<tr>
<td>ligra-tc</td>
<td>10.83</td>
<td>15.99</td>
<td>17.02</td>
</tr>
</tbody>
</table>
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
Motivation • Background • Implementing Work-Stealing on HCC • DTS • Evaluation

**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
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

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.