Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems

Moyang Wang, Tuan Ta, Lin Cheng, and Christopher Batten
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{mw828,qtt2,lc873,cbatten}@cornell.edu

***Abstract*—Manycore processors, with tens to hundreds of tiny cores but no hardware-based cache coherence, can offer tremendous peak throughput on highly parallel programs while being complexity and energy efficient. Manycore processors can be combined with a few high-performance big cores for executing operating systems, legacy code, and serial regions. These systems use heterogeneous cache coherence (HCC) with hardware-based cache coherence between big cores and software-centric cache coherence between tiny cores. Unfortunately, programming these heterogeneous cache-coherent systems to enable collaborative execution is challenging, especially when considering dynamic task parallelism. This paper seeks to address this challenge using a combination of light-weight software and hardware techniques. We provide a detailed description of how to implement a work-stealing runtime to enable dynamic task parallelism on heterogeneous cache-coherent systems. We also propose direct task stealing (DTS), a new technique based on user-level interrupts to bypass the memory system and thus improve the performance and energy efficiency of work stealing. Our results demonstrate that executing dynamic task-parallel applications on a 64-core system (4 big, 60 tiny) with complexity-effective HCC and DTS can achieve: 7× speedup over a single big core; 1.4× speedup over an area-equivalent eight big-core system with hardware-based cache coherence; and 21% better performance and similar energy efficiency compared to a 64-core system (4 big, 60 tiny) with full-system hardware-based cache coherence.**

I. Introduction

Parallelism and specialization are currently the two major techniques used to turn the increasing number of transistors provided by Moore's law into performance. While hardware specialization has demonstrated its strength in certain domains (e.g., GPUs for data parallelism and accelerators for deep learning), general multi-threaded applications are still better suited for multi-core processors. Hardware architects have relied on parallelism to improve the performance of processors for several decades, and the trend of increasing processor core count is likely to continue. There has been a growing interest in using a *manycore* approach which integrates tens or hundreds of relatively simple cores into a system to further increase hardware parallelism. Examples of manycore processors include the 72-core Intel Knights Landing [65], 64-core Tilera TILE64 [7], and 25-core Piton [47]. The manycore approach has demonstrated its potential in achieving high throughput and energy efficiency per unit area for multi-threaded workloads.

Hardware designers have realized that an unoptimized hardware-based cache coherence protocol (e.g., directory-based MESI and its variants) is difficult to scale due to directory state storage overhead, network latency overhead, as well as design and verification complexity. Designing a performance-, complexity-, and area-scalable hardware-based cache coherence protocol remains an active area of research [6, 14, 23, 25, 45, 48, 74, 75]. Another approach to continue increasing the number of cores in manycore systems is to sidestep hardware-based cache coherence and adopt software-centric cache coherence [17, 35, 58, 69, 70], software-managed scratchpad memory [5, 36], and/or message passing without shared memory [37]. Manycore processors without hardware cache coherence have been fabricated both in industry (e.g., 1024-core Adapteva Epiphany-V [51]) and academia (e.g., 511-core Celerity [20], 1000-core KiloCore [13]). By moving away from hardware-based cache coherence, these processors achieve exceptional theoretical throughput and energy efficiency (due to their massive core count), with relatively simple hardware. However, they have not been widely adopted because of challenges posed to software developers.

Programmers expect to use familiar CPU programming models, especially ones that support dynamic task-based parallelism, such as Intel Cilk Plus [31], Intel Threading Building Blocks (TBB) [32], and OpenMP [4, 52]. These programming models allow parallel tasks to be generated and mapped to hardware dynamically through a software runtime. They can express a wide range of parallel patterns, provide automatic load balancing, and improve portability for legacy code [46]. Unfortunately, manycore processors without hardware-based cache coherence require programmers to explicitly manage data coherence among private caches/memories and adopt a more restricted programming model, such as explicit task partitioning [35], message passing [51], or remote store programming [20]. The difficult programming model is arguably the primary reason why manycore processors without hardware-based cache coherence have not yet been widely accepted.

Furthermore, existing manycore processors without hardware cache coherence are generally used as a discrete co-processor, living in a separate memory space from the main general-purpose processor (i.e., host processor). Enabling efficient collaborative execution between the host and the manycore processor requires significant effort to bridge the gap between their disparate programming models and hide the data offloading latency [66]. Recent work in *heterogeneous cache coherence (HCC)* has demonstrated that hardware-based cache coherence protocols can be seamlessly and efficiently integrated with software-centric cache coherence protocols on chip in a unified memory address space [3]. While HCC solves the issue of data offloading by tightly integrating the host and the manycore, the programming model challenge remains to be addressed.

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00025

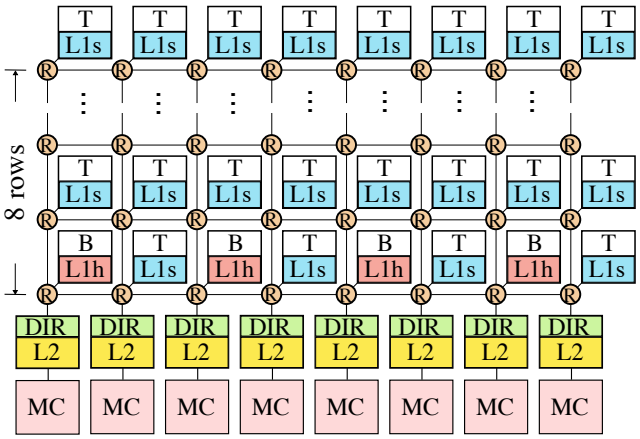

Figure 1. Block Diagram of a big.TINY Manycore System with HCC. T = in-order high efficiency tiny core; B = out-of-order high performance (big) core; R = on-chip interconnect router; L1s = private L1 cache with software-centric cache coherence; L1h = L1 cache with hardware-based cache coherence; DIR = directory; L2 = shared L2 cache bank; MC = main memory controller.

In this work, we attempt to address the software challenges of manycore processors with HCC by offering a TBB/Cilk-like programming model with a work-stealing runtime. In addition, we propose *direct-task stealing* (DTS), a light-weight hardware mechanism based on inter-processor user-level interrupts, to improve the performance and energy efficiency of dynamic task-parallel applications on HCC. We apply our approach to a *big.TINY* manycore architecture with HCC (see Figure 1), which consists of a few high-performance (big) cores with hardware-based cache coherence, and many simple (tiny) cores with software-centric cache coherence. Our approach allows dynamic task-parallel applications written for popular programming frameworks, such as Intel Cilk Plus and Intel TBB, to work collaboratively on both the big cores and the tiny cores without fundamental changes. In Section II, we provide a general background on HCC and work-stealing runtimes. In Section III, we describe in detail how to implement a work-stealing runtime, the core component of dynamic task-parallel frameworks, on HCC. In Section IV, we describe the direct-task stealing technique to address inherent overheads in HCC. We also explain how DTS enables some important optimizations in the work-stealing runtime to improve performance and energy efficiency. In Section V, we use a cycle-level evaluation methodology to demonstrate the potential of our approach. We show that our techniques applied to a 64-core big.TINY system (4 big, 60 tiny) with complexity-effective HCC running 13 TBB/Cilk-like dynamic task-parallel applications can achieve: 7× speedup over a single big core; 1.4× speedup over an area-equivalent eight big-core system with hardware-based cache coherence; and 21% better performance and similar energy efficiency compared to an optimistic big.TINY system (4 big, 60 tiny) with full-system hardware-based cache coherence.

The contributions of this work are: (1) we provide, to the best of our knowledge, the first detailed description on how to implement work-stealing runtimes for HCC; (2) we propose a direct task stealing technique to improve performance and energy efficiency of dynamic task-parallel applications on manycore processors with HCC; and (3) we provide a detailed cycle-level evaluation on our technique.

II. Background

This section provides a background of several cache coherence protocols, HCC, and dynamic task parallelism. We first characterize four representative hardware-based and software-centric coherence protocols. We then describe some existing work on HCC systems. Lastly, we provide a brief overview of dynamic task parallelism exemplified by TBB/Cilk-like programming models.

A. Hardware-Based and Software-Centric Coherence

There are two types of cache coherence protocols: *hardware-based* and *software-centric*. In hardware-based protocols, data coherence among private caches is handled completely by hardware and transparent to software. General-purpose processors usually support hardware-based cache coherence protocols since they are easier to program. In contrast, in software-centric protocols, software is in charge of enforcing the coherence of shared data among private caches. Software-centric cache coherence protocols are relatively simple to implement in hardware but require more effort from software programmers.

We use a similar taxonomy described in previous work [3] to categorize four representative coherence protocols: MESI, DeNovo [69], GPU-WT, and GPU-WB. A cache coherence protocol can be described using three properties: stale invalidation, dirty propagation, and write granularity. The *stale invalidation property* defines how and when stale data in a private cache is invalidated so that a read of the data returns its most up-to-date version. There are two ways to initiate an invalidation of stale data: writer-initiated and reader-initiated. In the first approach, a writer invalidates existing copies of the target data in all private caches prior to writing the data. This approach is used by hardware-based coherence protocols. The other approach is called reader-initiated: a reader invalidates potentially stale data in its private cache before it reads the data. The *dirty propagation property* defines how and when dirty data becomes visible to other private caches. Some coherence protocols track which private cache owns dirty data, so that the owner can propagate the data to readers. This strategy is called ownership dirty propagation. Both MESI and DeNovo take this strategy to propagate dirty data. In contrast, GPU-WB and GPU-WT do not track the ownership of dirty data. Instead, they rely on the writer to write back dirty data. Writes can either be immediately sent to the shared cache (i.e., write-through), or the dirty data can be written back later using a special *flush* instruction. Lastly, the *write granularity property* defines the unit size of data at which writes are performed and the ownership of cache lines is managed.

We summarize the four coherence protocols, MESI, DeNovo, GPU-WT, and GPU-WB, in Table I. These differences represent some fundamental design trade-offs between

TABLE I. CLASSIFICATION OF CACHE COHERENCE PROTOCOLS

Protocol	Who initiates invalidation?	How is dirty data propagated?	Write Granularity
MESI	Writer	Owner, Write-Back	Line
DeNovo	Reader	Owner, Write-Back	Word/Line
GPU-WT	Reader	No-Owner, Write-Through	Word
GPU-WB	Reader	No-Owner, Write-Back	Word

hardware-based and software-centric coherence protocols. In hardware-based protocols, the single-writer multiple-reader (SWMR) invariance [67] is enforced. With the SWMR property, hardware-based coherence protocols are transparent to software, i.e., software can assume there is no cache at all. However, it requires additional hardware complexity, including communication and storage overheads (e.g., extra invalidation traffic, transient coherence states, and directory storage). On the other hand, software-centric protocols (such as DeNovo, GPU-WT, and GPU-WB) suffer less from these overheads: by using the reader-initiated stale invalidation strategy, these protocols do not need to track all readers of a particular cache line, saving both communication traffic in the interconnection network and directory storage. Neither GPU-WT nor GPU-WB requires tracking ownership for writes. DeNovo is a design point between MESI and GPU-WB/GPU-WT: it uses ownership dirty propagation (like MESI) to potentially improve performance of writes and atomic memory operations (AMO); and it uses reader-initiated stale invalidation (like GPU-WB/GPU-WT) to reduce the invalidation overhead. However, software-centric coherence protocols push the complexity into software. Software needs to issue invalidation and/or flush requests at appropriate times to ensure coherence. GPU-WT and GPU-WB may also suffer from slower AMO performance. AMOs have to be handled in a globally shared cache, since private caches do not have ownership.

B. Heterogeneous Cache Coherence

Previous studies have explored different ways to integrate multiple coherence protocols into a heterogeneous cache-coherent system to serve a diversity of coherence requirements. IBM Coherent Accelerator Processor Interface (CAPI) provides a coherent MESI-based proxy interface for accelerators (e.g., FPGAs) to communicate with general-purpose processors through a shared last-level cache and directory [68]. Power et al. proposed a directory-based approach to maintain data coherence between a CPU and GPU at a coarse granularity (i.e., group of cache lines) [53]. Spandex [3] is a recent proposal on providing a general coherence interface implemented in a shared last-level cache to coordinate different coherence protocols, where different protocols interact with the interface through their own translation units.

C. Programming Models for Dynamic Task Parallelism

Task parallelism is a style of parallel programming where the workload is divided into *tasks*, units of computation, that can be executed in parallel. In dynamic task parallelism, tasks and dependencies among tasks are generated at runtime. Tasks are dynamically assigned to available threads. The most common computation model for dynamic task parallelism is the *fork-join* model, which was initially used by MIT Cilk [11] and was later popularized by many modern programming frameworks, including Intel Cilk Plus [31, 42], Intel TBB [32, 56], and others [15, 61]. Fork-join parallelism refers to a way of specifying parallel execution of a program: the program's control flow diverges (*forks*) into two or more flows that can be executed in parallel (but not necessarily); and then these control flows *join* into a single flow after they are finished. A task can fork by creating two or more parallel tasks, which is referred to as *spawning* tasks. The created tasks are called the *child* tasks (or simply *children*); the spawning task becomes the *parent* tasks of its children. The parent task waits until its children completes so that the children can join the parent. This model can serve as a basis to express many complex parallel patterns, including divide-and-conquer, parallel loop, reduction, and nesting [56].

In programming frameworks with the fork-join model, parallel execution is realized by a runtime system using the *work-stealing* algorithm [12]. In work-stealing runtimes, each thread is associated with a *task queue* data structure to store tasks that are available for execution. The task queue is a double-ended queue (*deque*). When a task spawns a child task, it *enqueues* the child on to the task queue of the executing thread. When a thread is available, it first attempts to *dequeue* a task from its own deque in last-in-first-out (LIFO) order from one end. If the thread's own deque is empty, it tries to *steal* a task from another thread's deque in first-in-first-out (FIFO) order from the other end. The thread that steals becomes a *thief*, and the thread whose tasks are stolen becomes a *victim*. When a parent task is waiting for its children to join, the thread executing the parent can steal from other threads. This algorithm thus automatically balances the workload across threads. It leads to better locality and helps establish time and space bounds [12, 24].

III. IMPLEMENTING WORK-STEALING RUNTIMES ON HETEROGENEOUS CACHE COHERENCE

This section gives a detailed description of our approach to implement a work-stealing runtime for HCC. We first show a baseline runtime for hardware-based cache coherence with a TBB/Cilk-like programming model. We then describe our implementation for heterogeneous cache coherence. We conclude this section with a qualitative analysis of the implications of HCC on work-stealing runtime systems.

A. Programming Example

We use the application programming interface (API) of Intel TBB to demonstrate our programming model (see Figure 2). Tasks are described by C++ classes derived from a base class, `task`, which has a virtual `execute()` method. Programmers override the `execute()` method to define the execution body of the task. Scheduling a new task is done by calling the `spawn(task* t)` function. Tasks are synchronized using the `wait()` function. The *reference count* tracks

```
1 class FibTask : public task {
2  public:
3   long* sum;
4   int n;
5
6   FibTask( int _n, long* _sum ) : n(_n), sum(_sum) {}
7
8   void execute() {
9     if ( n < 2 ) {
10      *sum = n;
11      return;
12    }
13    long x, y;
14    FibTask a( n - 1, &x );
15    FibTask b( n - 2, &y );
16    this->reference_count = 2;
17    task::spawn( &a );
18    task::spawn( &b );
19    task::wait( this );
20    *sum = x + y;
21  }
22 }
```

(a) `fib` using `spawn()` and `wait()`

```
1 long fib( int n ) {
2   if ( n < 2 ) return n;
3   long x, y;
4   parallel_invoke(
5     [&] { x = fib( n - 1 ); },
6     [&] { y = fib( n - 2 ); }
7   );
8   return (x + y);
9 }
```

(b) `fib` using `parallel_invoke`

```
1 void vvadd( int a[], int b[], int dst[], int n ) {
2   parallel_for( 0, n, [&]( int i ) {
3     dst[i] = a[i] + b[i];
4   });
5 }
```

(c) `vector-vector add` using `parallel_for`

Figure 2. Task-Based Parallel Programs – A simple example for calculating the Fibonancci number using two different APIs: (a) a low-level API with explicit calls to `spawn` and `wait`; and (b) a high-level API with a generic templated `parallel_invoke` pattern. (c) shows an alternative generic templated `parallel_for` pattern.

the number of unfinished children. When a parent task executes `wait()`, the execution of the parent task stalls until all of its children are finished. In addition to low-level APIs (e.g., `spawn`, `wait`), programmers can use higher-level templated functions that support various parallel patterns. For example, programmers can use `parallel_for` for parallel loops and `parallel_invoke` for divide-and-conquer.

B. Baseline Work-Stealing Runtime

Figure 3(a) shows an implementation, similar to Intel TBB, of the `spawn` and `wait` functions for hardware-based cache coherence. `spawn` pushes a task pointer onto the current thread's task deque, and `wait` causes the current thread to enter a *scheduling loop*. Inside the scheduling loop, a thread first checks if there is any task in its own deque. If so, it dequeues a task, in LIFO order, from its local deque to execute (lines 9–16). If there is none left in the local deque, the current thread becomes a thief and attempts to steal tasks from another thread in FIFO order (lines 19–28). When a task is executed, its parent's reference count is atomically decremented (line 27). When the reference count reaches zero, the parent task exits the scheduling loop and returns from `wait()`. A thread also exits the scheduling loop when the whole program is finished, and the main thread terminates all other threads.

C. Supporting Shared Task Queues on HCC

A task queue is a data structure shared by all threads. On processors with hardware-based cache coherence protocols, a per-deque lock implemented using atomic read-modify-write operations is sufficient for implementing proper synchronization (see lines 2, 4, 9, 11, 21, 23 of Figure 3(a)). On processors with heterogeneous cache coherence, where some private caches use software-centric cache coherence protocols, additional coherence operations are required. Before a thread can access a task queue, in addition to acquiring the lock for mutual exclusion, all clean data in the private cache of the executing thread needs to be invalidated to prevent reading stale data. After a thread finishes accessing a task queue, in addition to releasing the lock, all dirty data needs to be written back (flushed) to the shared cache so that the data becomes visible to other threads. Figure 3(b) shows an implementation of `spawn` and `wait` for HCC protocols. We add an `invalidate` instruction following the lock acquire (e.g., lines 3 and 13), and a `flush` instruction before the lock release (e.g., lines 15 and 29). Note that not all protocols require `invalidate` and `flush`. On private caches with DeNovo, `flush` is not required because it uses an ownership stale-invalidation strategy (see Table I). With DeNovo, `flush` can be treated as no-op. MESI protocol requires neither `flush` or `invalidate`, so both are treated as no-ops.

D. Supporting Task-Stealing on HCC

Adding proper `invalidate` and/or `flush` instructions along with per-deque locks ensures all task queues deques are coherent on HCC systems. However, user-level data needs to be coherent as well. Fortunately, the TBB programming model we use is structured by the DAG consistency model [10]. Informally speaking, the DAG consistency model means data sharing only exists between a parent task and its child task(s). To observe the DAG consistency, there are two requirements that the runtime system must fulfill: (1) child tasks need to see the latest values produced by their parent; and (2) a parent task needs to see all values produced by its children after the `wait()`. There is no synchronization between *sibling* tasks required because sibling tasks from the same parent must be data-race-free with regard to each other. This property allows us to correctly implement work-stealing runtimes on HCC without requiring changes in the user code. When a parent task spawns child tasks, the `flush` after the enqueue (see Figure 3(b), line 5) ensures data written by the parent task is visible to its child tasks, even if the children are stolen and executed by another thread. For requirement (2), the parent task needs to `invalidate` data in its own cache, in case any children are stolen and the parent may have stale data (line 40). Moreover, a thread needs to `invalidate` and `flush` before and after executing a stolen task respectively

```
1  void task::spawn( task* t ) {
2    tq[tid].lock_aq()
3    tq[tid].enq(t)
4    tq[tid].lock_rl()
5  }
6
7  void task::wait( task* p ) {
8    while ( p->rc > 0 ) {
9      tq[tid].lock_aq()
10     task* t = tq[tid].deq()
11     tq[tid].lock_rl()
12
13     if (t) {
14       t->execute()
15       amo_sub( t->p->rc, 1 )
16     }
17
18     else {
19       int vid = choose_victim()
20
21       tq[vid].lock_aq()
22       t = tq[vid].steal()
23       tq[vid].lock_rl()
24
25       if (t) {
26         t->execute()
27         amo_sub( t->p->rc, 1 )
28       }
29     }
30   }
31 }
```

(a) Hardware-Based Cache Coherence

```
1  void task::spawn( task* t ) {
2    tq[tid].lock_aq()
3    cache_invalidate()
4    tq[tid].enq(t)
5    cache_flush()
6    tq[tid].lock_rl()
7  }
8
9  void task::wait( task* p ) {
10   while ( amo_or( p->rc, 0 ) > 0 ) {
11
12     tq[tid].lock_aq()
13     cache_invalidate()
14     task* t = tq[tid].deq()
15     cache_flush()
16     tq[tid].lock_rl()
17
18     if (t) {
19       t->execute()
20       amo_sub( t->p->rc, 1 )
21     }
22
23     else {
24       int vid = choose_victim()
25
26       tq[vid].lock_aq()
27       cache_invalidate()
28       t = tq[vid].steal()
29       cache_flush()
30       tq[vid].lock_rl()
31
32       if (t) {
33         cache_invalidate()
34         t->execute()
35         cache_flush()
36         amo_sub( t->p->rc, 1 )
37       }
38     }
39   }
40   cache_invalidate()
41 }
```

(b) Heterogeneous Cache Coherence

```
1  void task::spawn( task* t ) {
2    uli_disable()
3    tq[tid].enq(t)
4    uli_enable()
5  }
6
7  void task::wait( task* p ) {
8    int rc = p->rc
9    while ( rc > 0 ) {
10
11     uli_disable()
12     task* t = tq[tid].deq()
13     uli_enable()
14
15     if (t) {
16       t->execute()
17       if ( t->p->has_stolen_child )
18         amo_sub( t->p->rc, 1 )
19       else
20         t->p->rc -= 1
21     }
22
23     else {
24       int vid = choose_victim()
25
26       uli_send_req(vid)
27       t = read_stolen_task(tid)
28
29       if (t) {
30         cache_invalidate()
31         t->execute()
32         cache_flush()
33         amo_sub( t->p->rc, 1 )
34       }
35     }
36
37     if ( p->has_stolen_child )
38       rc = amo_or( p->rc, 0 )
39     else
40       rc = p->rc
41   }
42
43   if ( p->has_stolen_child )
44     cache_invalidate()
45 }
46
47 void uli_handler( int thief_id ) {
48   task* t = tq[tid].deq()
49   if (t)
50     t->p->has_stolen_child = 1
51   write_stolen_task( thief_id, t )
52   cache_flush()
53   uli_send_resp( thief_id )
54 }
```

(c) Direct Task-Stealing

Figure 3. Work-Stealing Runtime Implementations – Pseudocode of `spawn` and `wait` functions for: (a) hardware-based cache coherence; (b) heterogeneous cache coherence; and (c) direct-task stealing. `p` = parent task pointer; `t` = current task pointer; `rc` = reference count; `tid` = worker thread id; `lock_aq` = acquire lock; `lock_rl` = release lock; `tq` = array of task queues with one per worker thread; `enq` = enqueue on tail of task queue; `deq` = dequeue from tail of task queue, returns 0 if empty; `choose_victim` = random victim selection; `vid` = victim id; `steal` = dequeue from head of task queue; `amo_or` = atomic fetch-and-or; `amo_sub` = atomic fetch-and-sub; `cache_flush` = flush all dirty data in cache (no-op on MESI, DeNovo, and GPU-WT); `cache_invalidate` = invalidate all clean data in cache (no-op on MESI); `uli_disable` = disable servicing ULI; `uli_enable` = enable servicing ULI; `uli_send_req` = send ULI request message and wait for response, calls `uli_handler` on receiver; `uli_send_resp` = send ULI response message; `read_stolen_task` = read stolen task from per-thread mailbox; `write_stolen_task` = store the stolen task into mailbox.

(lines 33 and 35), because the parent task is executed on a different thread (the victim thread).

E. HCC Performance Impacts

In this section, we qualitatively discuss the performance impacts of HCC on work-stealing runtimes. We defer the quantitative characterization to Section VI.

Reader-initiated invalidation strategy degrades performance in all three software-centric cache coherence protocols (i.e., DeNovo, GPU-WT, and GPU-WB) due to more cache misses. Every `spawn` and `wait` causes the executing thread to invalidate all data in the private cache, causing later reads to experience more cache misses.

Atomic operations may be slower. DeNovo uses the ownership dirty-propagation strategy, so the AMOs can be performed in private caches in the same way as MESI. However, GPU-WT and GPU-WB require AMOs to be performed at the shared cache, increasing the latency per operation.

Flushing is inefficient. DeNovo and GPU-WT do not need `flush` operations. GPU-WB, on the other hand, requires an explicit `flush` at every `spawn`, as well as after executing every stolen task. Writing back an entire private cache to the shared cache may require a significant amount of time and memory traffic, depending on the amount of data being written back.

Fine-grained tasks may exacerbate all of the above problems. Performance impacts discussed above are directly related to the task granularity. Finer-grained applications have many small tasks, and thus require a large number of AMOs, invalidations, and flushes. Therefore, HCC is expected to have more severe performance impacts on applications with fine-grained tasks.

IV. Direct Task Stealing

In this section, we propose a hardware technique called direct task stealing (DTS) to improve the performance of work-stealing runtimes on HCC. DTS leverages the following properties of work-stealing runtimes: **(1) when parallelism is sufficient, the number of steals is relatively small compared to the number of local task enqueues and dequeues [12,24]; and (2) for a steal, synchronization is only required between a victim thread and a thief thread, not among all threads**. As we have mentioned in the previous section, work-stealing may incur significant overhead in terms of performance and memory traffic on HCC. DTS addresses this issue by allowing tasks to be stolen *directly* using user-level interrupts (described below), rather than *indirectly* through shared memory.

A. Implementing Direct Task Stealing

A user-level interrupt (ULI) is a short, light-weight interprocessor interrupt. ULI is included in modern instruction-set architectures (ISA) such as RISC-V [18, 57]. Like regular interrupts, user-level interrupts are handled asynchronously, and can be enabled and disabled on the receiving core by software. When a core has ULI enabled, and receives an ULI, its current executing thread is suspended, and the core jumps to a software-specified ULI handler. The only difference between ULI and regular interrupts is that ULIs are handled completely in user mode. The cost of handling an ULI is similar to regular context switching, minus the overheads associated with privilege mode changing.

Figure 3(c) shows an implementation of a work-stealing runtime with DTS. DTS uses ULI to perform work-stealing: when a thread attempts to steal a task, it sends a ULI to the victim thread (line 26). If the victim has ULI enabled, execution of the victim is redirected to a handler (lines 47–53). In the handler, the victim accesses its *own* task deque, retrieves a task, and sends the task to the thief through shared memory. The victim also sends an ACK message to the thief as a ULI response. With DTS, the victim steals tasks on behalf of the thief. If the victim has ULI disabled, a NACK message is replied to the thief. In the rest of this section, we discuss why DTS can help reduce the overheads of work-stealing runtimes on HCC.

B. Optimizations to Reduce Task Queue Synchronization

DTS reduces the synchronization cost associated with task stealing on HCC. With DTS, task queues are no longer shared data structures. A task queue is only accessed by its owner, either for local accesses, or for steals through ULI. DTS therefore eliminates the need of synchronization (i.e., locks) on task queues. Accesses to task queues are kept mutually exclusive, by requiring a thread to disable ULI when it operates on its task deque (line 11). Work-stealing runtimes on HCC without DTS require every deque access, including ones made to a thread's own task deque, to have a pair of `invalidate` and `flush` (associated with lock acquire and release respectively, as in line 27 and 29 of Figure 3(b)). In work-stealing runtimes with DTS, accessing task queues no longer incurs cache invalidations or flushs since task queues are private. DTS reduces cache misses on HCC caused by cache invalidation and flush, and thus improves performance and reduces memory traffic.

C. Optimizations to Reduce Parent-Child Synchronization

DTS offers an opportunity to further reduce synchronization cost for work stealing on HCC using software optimizations. In work-stealing runtimes without DTS, it is difficult to asses whether a task is actually stolen, since work stealing happens implicitly through shared memory. Without this information, a runtime must conservatively assume all tasks can potentially be stolen, and the runtime must always ensure proper synchronization between parent and child tasks. For example, the reference count in each task always needs to be updated using AMOs (Figure 3(b), line 20), and a parent task always invalidates its cache before returning from `wait` function in case of any of its children has been stolen (Figure 3(b), line 40).

DTS enables the runtime to track whether a task has been stolen and avoid synchronization entirely when a task is not stolen. This is particularly important to cache coherence protocols where flush (e.g., GPU-WB) or AMOs (e.g., GPU-WT and GPU-WB) are expensive. To track whether a task has been stolen, we add an auxiliary variable (`has_stolen_child`) to each task, indicating whether at least one of its child tasks has been stolen (Figure 3(c), line 37). Before sending the stolen task to the thief in the ULI handler, the victim sets `has_stolen_child` to true (line 50). The `has_stolen_child` variable is only accessed by the thread running the parent task, and thus can be modified with a regular load and store instead of an AMO.

According to the DAG-consistency model, if a child task is not stolen (i.e., `has_stolen_child` is false), it is not necessary to make its writes visible to other threads. Because its parent, the only task that needs to read these writes, is executed on the same thread as the child. A `flush` is only required when a task is actually stolen (Figure 3(c), line 51), instead of after each enqueue operation (Figure 3(b), line 5). If there are considerably more local enqueues and dequeues than steals, which is the common case when there is sufficient parallelism, DTS can significantly reduce the number of `flush`. Furthermore, if no child task has been stolen, it is unnecessary to perform an `invalidation` at the end of `wait()`; the parent task cannot possibly read stale data, since all of its child tasks are executed on the same thread as itself (Figure 3(c), line 43). Finally, if no child of a parent task is stolen, it is not necessary to update the parent's reference count using AMOs (line 20 and 40). Instead, the reference

count can be treated as a local variable, because updates to it so far has been performed by the same thread.

In summary, DTS enables software optimizations to leverage properties of the DAG-consistency model in work-stealing runtimes to further reduce the overheads (i.e., invalidation, flush, and AMO) of work-stealing runtimes on HCC.

V. Evaluation Methodology

In this section, we describe our cycle-level performance modeling methodology used to quantitatively evaluate our proposal.

A. Simulated Hardware

We model manycore systems with the gem5 cycle-level simulator [9]. We implement HCC protocols described in Section II using the Ruby memory modeling infrastructure. We use Garnet 2.0 [2] to model on-chip interconnection networks (OCN). Our modeled systems have per-core private L1 caches and a shared banked L2 cache. All caches have a 64B cache line size.

We use the shared L2 cache to integrate different cache coherence protocols, similar to Spandex [3]. The L2 cache supports different coherence request types required by the four cache coherence protocols we study in this work. The L2 cache in the baseline MESI protocol is inclusive of L1 caches. The L2 cache in HCC protocols is inclusive of MESI private L1 caches only. The directory is embedded in the L2 cache and has a precise sharer list for all MESI private L1 caches. There is no additional directory latency.

We implement the inter-processor ULI in our simulated manycore systems as specified by the RISC-V ISA [57]. We model the ULI network as a mesh network with two virtual channels (one for request and one for response to prevent deadlock) using Ruby `SimpleNetwork`. We assume the ULI network has a 1-cycle channel latency and a 1-cycle router latency. We also model the buffering effect in the ULI network. Each ULI message is single-word long. Only one ULI is allowed to be received by a core at any given time. We enhance each core with a simple hardware unit for sending and receiving ULI. A ULI is sent by writing to a dedicated control register. The hardware unit on each core has one buffer for requests and one buffer for responses. When the buffer is full, the receiver sends a NACK to senders.

We configure a 64-core big.TINY system with four out-of-order high-performance big cores, and 60 in-order tiny cores. A tiny core has a private L1 cache capacity of 4KB (i.e., 1/16 the capacity of a big core's L1 cache). The big and tiny cores are connected by an 8×8 on chip mesh network. Each column of the mesh is connected to a L2 cache bank and a DRAM controller. Figure 1 shows the schematic diagram of our simulated systems. Table II summarizes its key parameters.

We study the following big.TINY configurations: *big.TINY/MESI* is a system where both big and tiny cores are equipped with MESI hardware-based cache coherence; *big.TINY/HCC-dnv* is a big.TINY system with HCC, where big cores use MESI and tiny cores use DeNovo (DeNovoSync [69] variant); similarly, *big.TINY/HCC-gwt* and *big.TINY/HCC-gwb* are HCC configurations with GPU-WT and GPU-WB on the tiny cores, respectively. In addition, we implement DTS proposed in Section IV on each HCC configuration. We refer to those configurations with DTS as *big.TINY/HCC-DTS-dnv*, *big.TINY/HCC-DTS-gwt*, and *big.TINY/HCC-DTS-gwb*.

Table II. Simulator Configuration

Tiny Core	RISC-V ISA (RV64GC), single-issue, in-order, single-cycle execute for non-memory inst. L1 cache: 1-cycle, 2-way, 4KB L1I and 4KB L1D, software-centric coherence;
Big Core	RISC-V ISA (RV64GC), 4-way out-of-order, 16-entry LSQ, 128 Physical Reg. 128-entry ROB. L1 cache: 1-cycle, 2-way, 64KB L1I and 64KB L1D, hardware-based coherence;
L2 Cache	Shared, 8-way, 8 banks, 512KB per bank, one bank per mesh column, support heterogeneous cache coherence;
OCN	8×8 mesh topology, XY routing, 16B per flit, 1-cycle channel latency, 1-cycle router latency;
Main Memory	8 DRAM controllers per chip, one per mesh column. 16GB/s total bandwidth.

As a comparison to our big.TINY configurations, we also study a traditional multicore architecture with only big cores. *O3×1*, *O3×4*, and *O3×8* are configurations with one, four, and eight big cores, respectively. We use CACTI [50] to model the area of L1 caches. Our results show that a big core's 64KB L1 cache is 14.9× as large as a tiny core's 4KB L1 cache. Based on total L1 capacity and the CACTI results, we estimate that *O3×8* has similar area to our 64-core big.TINY configurations (4 big cores and 60 tiny cores).

Future manycore processors are likely to have hundreds to thousands of cores. To overcome the challenge of simulation speed with large systems, we use a *weak scaling* approach to only simulate a piece of the envisioned large-scale manycore systems. We scale down the core count and the memory bandwidth to the 64-core system described in Table II. We also choose moderate input dataset sizes with moderate parallelism (see Table III). We attempt to make our results representative of future manycore systems with more cores, more memory bandwidth, and running proportionally larger inputs. To validate our proposal in large systems, we also select a subset of the application kernels and evaluate them using larger datasets on a bigger 256-core big.TINY system.

B. Work-Stealing Runtime Systems

We implement three variations of a C++ library-based work-stealing runtime described in Section III. We have compared the performance of our baseline runtime system for hardware-based cache coherence (Figure 3(a)) to Intel Cilk Plus and Intel TBB using applications we study in this paper running natively on an 18-core Xeon E7-8867 v4 processor, with dataset sizes appropriate for native execution. Our results show that our baseline work-stealing runtime has similar performance to Intel TBB and Intel Cilk Plus.

TABLE III. SIMULATED APPLICATION KERNELS

					Cilkview				Speedup over Serial IO				Speedup over b.T/MESI					
												b.T/	b.T/HCC			b.T/HCC-DTS		
Name	**Input**	**GS**	**PM**	**DInst**	**Work**	**Span**	**Para**	**IPT**	**O3×1**	**O3×4**	**O3×8**	**MESI**	**dnv**	**gwt**	**gwb**	**dnv**	**gwt**	**gwb**
cilk5-cs	3000000	4096	ss	456M	524M	0.9M	612.1	31.9K	1.65	4.92	9.78	18.70	1.01	1.01	1.02	1.01	0.99	1.01
cilk5-lu	128	1	ss	155M	170M	4.8M	35.5	6.5K	2.48	9.46	17.24	23.93	0.91	0.37	1.00	0.85	0.34	1.06
cilk5-mm	256	32	ss	124M	184M	0.4M	449.3	8.7K	11.38	11.76	22.04	41.23	1.00	0.89	0.94	0.98	0.93	1.11
cilk5-mt	8000	256	ss	322M	416M	0.5M	829.3	135K	5.71	19.70	39.94	57.43	0.71	1.05	0.69	0.72	1.04	0.70
cilk5-nq	10	3	pf	100M	180M	0.7M	274.9	0.4K	1.57	3.87	7.03	2.93	1.01	1.18	1.09	0.56	1.52	1.76
ligra-bc	rMat_100K	32	pf	80M	129M	1.1M	117.9	0.4K	2.05	6.29	13.06	11.48	0.96	0.96	1.01	1.01	1.25	1.60
ligra-bf	rMat_200K	32	pf	151M	252M	1.2M	203.3	0.4K	1.80	5.36	11.25	12.80	0.97	0.95	0.98	0.89	1.10	1.32
ligra-bfs	rMat_800K	32	pf	236M	351M	0.9M	402.6	0.5K	2.23	6.23	12.70	15.63	1.02	1.05	1.10	1.08	1.23	1.52
ligra-bfsbv	rMat_500K	32	pf	152M	201M	0.7M	277.5	0.5K	1.91	6.17	12.25	14.42	1.01	0.98	0.98	1.00	1.06	1.18
ligra-cc	rMat_500K	32	pf	226M	278M	0.7M	383.5	0.6K	3.00	9.11	20.66	24.12	0.82	0.94	0.99	0.91	1.08	1.24
ligra-mis	rMat_100K	32	pf	183M	243M	1.3M	177.7	0.5K	2.43	7.70	15.61	19.01	0.88	0.89	0.92	0.97	1.07	1.35
ligra-radii	rMat_200K	32	pf	364M	437M	1.4M	311.4	0.7K	2.80	8.17	17.89	25.94	0.83	0.81	0.85	1.00	1.03	1.17
ligra-tc	rMat_200K	32	pf	286M	342M	1.0M	334.9	3.5K	1.49	4.99	10.89	23.21	1.01	0.86	0.98	1.07	0.92	1.05
geomean									2.56	7.26	14.70	16.94	0.93	0.89	0.96	0.91	1.00	1.21

Input = input dataset; GS = task granularity; PM = parallelization methods: pf = `parallel_for` and ss = recursive spawn-and-sync; DInsts = dynamic instruction count in millions; Work = total number of x86 instructions; Span = number of x86 instructions on the critical path; Para = logical parallelism, defined as work divided by span; IPT = average number of instructions per task; Work, span, and IPT are analyzed by Cilkview; b.T = big.TINY; HCC = heterogeneous cache coherence; dnv = DeNovo; gwt = GPU-WT; gwb = GPU-WB.

C. Benchmarks

We port 13 dynamic task-parallel applications from Cilk v5.4.6 [24] and Ligra [62] to use our work-stealing runtime systems (see Table III). We select applications with varied parallelization methods: applications from Cilk mainly use recursive spawn-and-sync parallelization (i.e., `parallel_invoke`); applications from Ligra mainly use loop-level parallelization (i.e., `parallel_for`). Ligra applications also exhibit non-determinism, as they typically use fine-grained synchronization such as compare-and-swap in their code. *cilk5-cs* performs parallel mergesort algorithm; `lu` calculates LU matrix decomposition; *cilk5-mm* is blocked matrix multiplication; *cilk5-mt* is matrix transpose; *cilk5-nq* uses backtracking to solve the N-queen problem; *ligra-bc* calculates betweeness centrality of a graph; *ligra-bf* uses Bellman-Ford algorithm to calculate the single-source shortest path in a graph; *ligra-bfs* performs bread-first search on graphs; *ligra-bfsbv* is a bit-vector optimized version of bread-first search; *ligra-cc* computes connected components in graphs; *ligra-mis* solves the maximum independent set problem; *ligra-radii* computes the radius of a given graph; *ligra-tc* counts the number of triangles in a graph. A more detailed description for these benchmarks can be found in previous work [24, 62].

D. Task Granularity

Task granularity (i.e., the size of the smallest task) is an important property of task-parallel applications. Programmers can control task granularity by dividing the work into more (fine-grained) or less (coarse-grained) tasks. Task granularity presents a fundamental trade-off: fine-grained tasks increase logical parallelism, but incur higher runtime overheads than coarse-grained tasks. We use a hybrid simulation-native approach to choose the task granularity for each application. We sweep the granularity and use Cilkview [28] to analyze the logical parallelism. We evaluate the speedup over serial code for each granularity on a simulated manycore processor with 64 tiny cores. We select suitable granularity for each application to make sure it achieves the best or close to the best speedup over serial execution (see instruction-per-task (IPT) in Table III). As an example, Figure 4 shows the speed up and the logical parallelism of *ligra-tc* with different granularity. It demonstrates that both a too big and a too small granularity lead to sub-optimal performance: the former due to lack of parallelism, and the latter due to runtime overheads. A smaller granularity penalizes HCC configurations more heavily, and the benefits of DTS technique would be more pronounced. Our choice of task granularity aims to optimize the performance of our baseline, not the relative benefits of our proposed DTS technique.

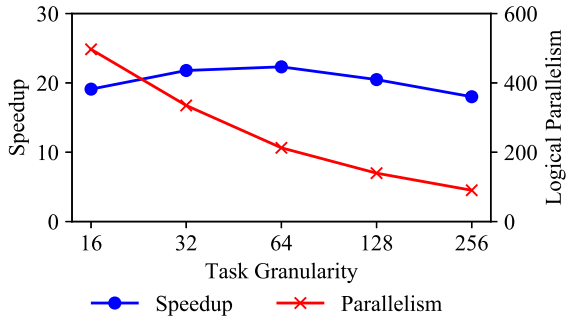

Figure 4. Speedup and Logical Parallelism of *ligra-tc* Running on a 64-core System. Task Granularity = the number of triangles processed by each task.

VI. RESULTS

Table III summarizes the speedup of the simulated configurations. Figure 5 illustrates the speedup of each big.TINY HCC configuration relative to *big.TINY/MESI*. Figure 6

shows the hit rate of L1 data caches. Figure 7 presents the execution time breakdown of the tiny cores. Figure 8 shows the total memory traffic (in bytes) on the on-chip network.

A. Baseline Runtime on big.TINY/MESI

On 11 out of 13 applications, *big.TINY/MESI* has better performance than *O3×8*. The baseline work-stealing runtime enables collaborative execution and load balancing between the big and tiny cores in *big.TINY/MESI*. *cilk5-nq* performs worse on *big.TINY/MESI* than *O3×8* because the runtime overheads outweigh the parallel speedup (as discussed in Section V-D). Overall, our *big.TINY/MESI* vs. *O3×8* results demonstrate the effectiveness of unlocking more parallelism using a big.TINY system compared to an area-equivalent traditional multi-core configuration *O3×8*.

B. Work-Stealing Runtime on HCC

We now discuss our work-stealing runtime on HCC (shown in Figure 3(b)) by analyzing the performance and energy of *big.TINY/HCC-dnv*, *big.TINY/HCC-gwt*, and *big.TINY/HCC-gwb*.

Compared with *big.TINY/MESI*, *big.TINY/HCC-dnv* has decreased L1 hit rate due to its reader-initiated invalidation strategy, as shown in Figure 6. This decrease in L1 hit rate causes a slight increase in memory traffic, as shown in the *cpu_req* and *data_resp* categories in Figure 8. The impact of these negative effects on performance is modest on most of the applications, except for *cilk5-mt*. *cilk5-mt* has a significant performance degradation due to additional write misses caused by invalidation. This effect can be seen in the increased data store latency and write-back traffic (see Figure 8).

big.TINY/HCC-gwt is a write-through and no write-allocate protocol. In GPU-WT, a write miss does not refill the cache. Therefore, *big.TINY/HCC-gwt* is unable to exploit temporal locality in writes, resulting in significantly lower L1 hit rate compared to both *big.TINY/MESI* and *big.TINY/HCC-dnv*. The network traffic of *big.TINY/HCC-gwt* is also significantly higher than others, especially in the *wb_req* category. The reason is every write (regardless of hit or miss) updates the shared cache (write-through). The latency for AMOs and network traffic are also increased (shown in Figure 7 and Figure 8 respectively). *big.TINY/HCC-gwt* has slightly worse performance and significantly more network traffic compared to *big.TINY/MESI* and *big.TINY/HCC-dnv* in all applications except *cilk5-lu*, where it performs significantly worse.

big.TINY/HCC-gwb has similar performance to *big.TINY/HCC-gwt* when dealing with AMOs. However, the write-back policy allows *big.TINY/HCC-gwb* to better exploit temporal locality. On all applications except *cilk5-mt*, *big.TINY/HCC-gwb* has less memory traffic, higher L1 hit rate, and better performance than *big.TINY/HCC-gwt*. *big.TINY/HCC-gwb* is less efficient in memory traffic compared *big.TINY/HCC-dnv* due to its lack of ownership tracking: every private cache needs to propagate dirty data through the shared cache.

In summary, our baseline work-stealing runtime on HCC has moderately worse performance than the *big.TINY/MESI* configuration on almost all applications. These results demonstrate that HCC can effectively reduce hardware complexity with a small performance and energy penalty.

TABLE IV. CACHE INVALIDATION, FLUSH, AND HIT RATE

	InvDec (%)			FlsDec (%)	HitRateInc (%)		
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

Comparisons of big.TINY/HCC-DTS with big.TINY/HCC. InvDec = % decrease in cache line invalidations. FlsDec = % decrease in cache line flushes. HitRateInc = % increase in L1 D$ hit rate.

C. Evaluation of HCC with DTS

In Section IV, we motivate DTS by observing that synchronization is only needed when a task is stolen. DTS avoids using cache invalidations and/or flushes unless a steal actually happens. We compare the results of HCC configurations without DTS (*big.TINY/HCC-**) with those with DTS (*big.TINY/HCC-DTS-**). We profile the number of invalidated and flushed cache lines for each configuration. We summarize the reduction in the number of invalidations and flushes in Table IV. We also calculate the increase in L1 hit rate of *big.TINY/HCC-DTS-** configurations compared with corresponding *big.TINY/HCC-** (HCC without DTS) configurations.

In all three HCC protocols across all benchmarks, *big.TINY/HCC-DTS-** have significantly lower number of cache invalidations. *ligra-bf* and *ligra-bfsbv* show a reduction of 30–50%. *ligra-tc* has a reduction of 10–20%. The rest of the benchmarks each has more than 90% reduction. The number of flushes is also reduced on *big.TINY/HCC-DTS-gwb*. 10 of 13 benchmarks have a reduction of more than 90%. In *ligra-tc*, *ligra-bfsbf*, and *ligra-bf*, DTS achieves less flush reduction due to the relatively higher number of steals.

Table IV shows the reduction in invalidations translates to higher L1 hit rate. The effect of increasing L1 hit rate is less significant on *big.TINY/HCC-DTS-dnv* because it has higher L1 hit rate to begin with (due to its ownership-based dirty propagation). The increase in L1 hit rates also leads to reduced network traffic. Figure 8 shows *big.TINY/HCC-DTS-** have reduced network traffic in *cpu_req* and *data_resp*. In *big.TINY/HCC-DTS-gwb*, *wb_req* traffic is also significantly reduced, due to the reduction in flushes. However, DTS cannot help reduce *wb_req* traffic in *big.TINY/HCC-DTS-gwt* since each write still causes a write-through to the shared cache.

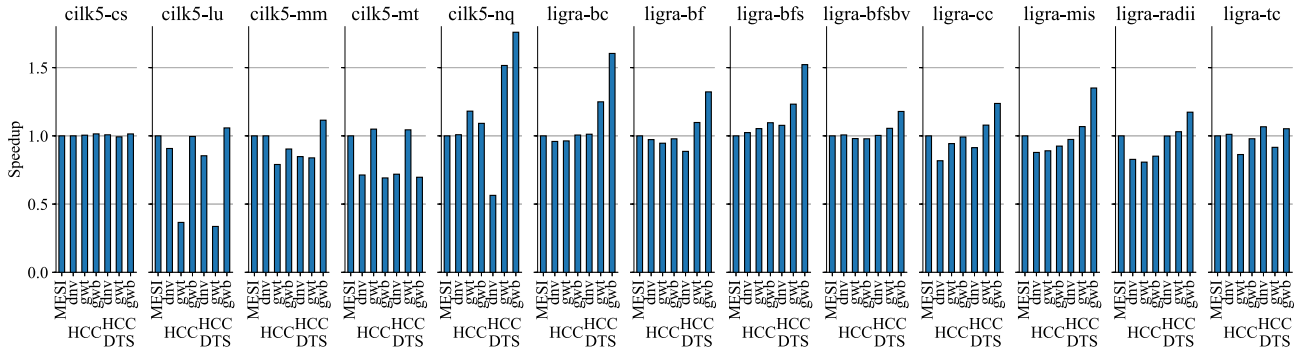

Figure 5. Speedup Over big.TINY/MESI

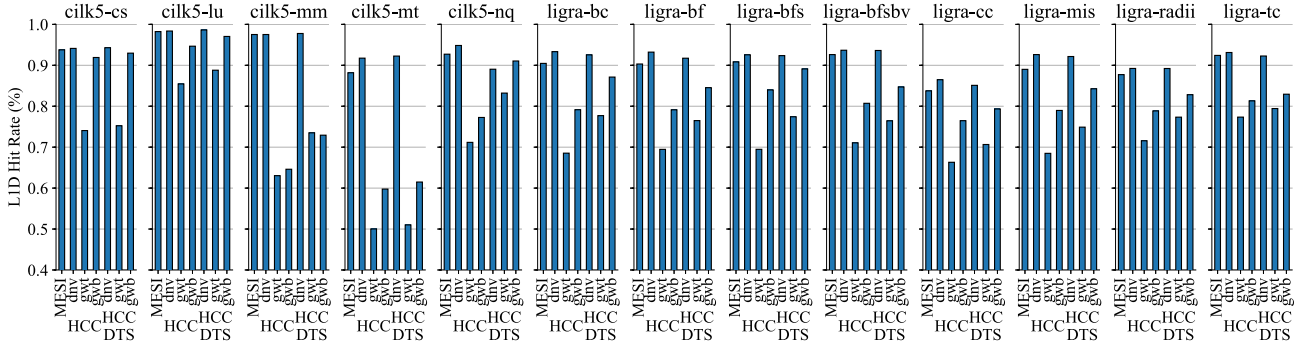

Figure 6. L1 Data (L1D) Cache Hit Rate

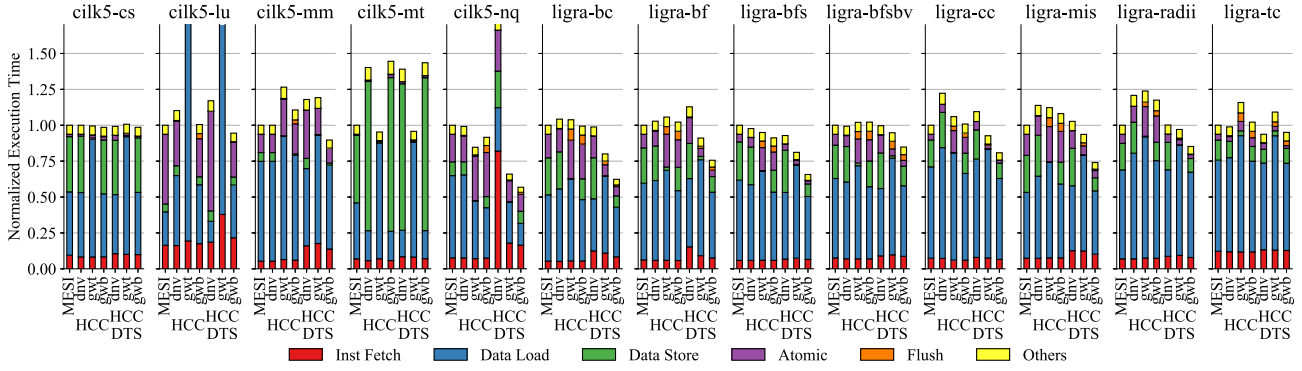

Figure 7. Aggregated Tiny Core Execution Time Breakdown Normalized to big.TINY/MESI

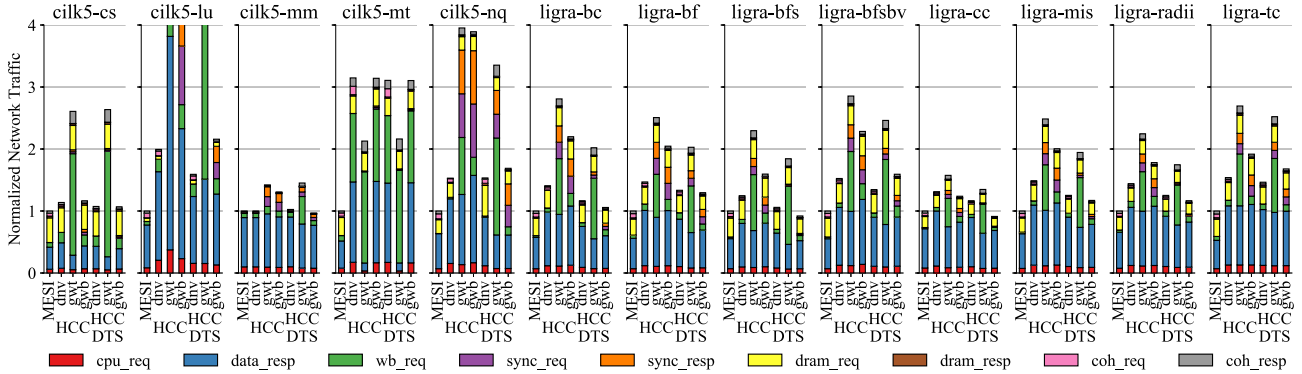

Figure 8. Total On-Chip Network Traffic (in Bytes) Normalized to big.TINY/MESI – *cpu_req* = requests from L1 to L2; *data_resp* = response from L2 to L1; *wb_req* = request for write-back data; *sync_req* = synchronization request; *sync_resp* = synchronization response; *dram_req* = request from L2 to DRAM; *dram_resp* = response from DRAM to L2; *coh_req* = coherence request; *coh_resp* = coherence response

---

MESI = big.TINY/MESI; HCC = configurations with heterogeneous cache coherence; DTS = direct task stealing
dnv = tiny cores use DeNovo protocol; gwt = tiny cores use GPU-WT protocol; gwb = tiny cores use GPU-WB protocol.

TABLE V. RESULTS FOR 256-CORE BIG.TINY SYSTEM

			Speedup		
			vs O3×1	vs b.T/MESI	
Name	Input	DInst	b.T/MESI	HCC-gwb	HCC-DTS-gwb
cilk5-cs	6000000	2.22×	27.7	0.94	1.07
ligra-bc	rMat_1M	8.65×	14.3	0.96	1.61
ligra-bfs	rMat_3M	2.90×	13.5	1.04	1.78
ligra-cc	rMat_1M	1.99×	27.7	0.92	1.26
ligra-tc	rMat_1M	6.05×	18.5	0.69	0.76

Results of using bigger datasets on a 256-core big.TINY manycore processor. The processor consists of four big cores and 252 tiny cores, configured in a 8-row, 32-column mesh. It has 32 L2 banks (16MB total L2 capacity) and 32 DRAM controllers (see Figure 1). Big cores and tiny cores use parameters in Table II. Input = input datasets; DInsts = dynamic instruction count relative to the smaller datasets in Table III; b.T = big.TINY; HCC = heterogeneous cache coherence; DTS = direct task stealing; gwb = GPU-WB.

As we discuss in Section IV, DTS also enables runtime optimizations to avoid using AMOs for the reference count unless a child is stolen. *big.TINY/HCC-DTS-gwt* and *big.TINY/HCC-DTS-gwb* benefit from this optimization since the AMO latency is high in these two protocols. In Figure 8, we can see the reduction of traffic due to less AMO requests. Figure 7 shows the reduced execution time in the `Atomic` category.

Out of the three HCC protocols, *big.TINY/HCC-DTS-gwb* benefits the most from DTS. It can leverage reduction in invalidations, flushes, and AMOs. *big.TINY/HCC-gwt* and *big.TINY/HCC-dnv* benefit from DTS less because they do not need flushes. *big.TINY/HCC-gwt* benefit from reduction in AMOs, but its *wb_req* traffic is unaffected by DTS.

We measure the overhead of DTS in our simulations. In all applications and configurations, we observe that the ULI network has less than 5% network utilization rate, indicating DTS is infrequent. The average latency of the ULI network is around 20 hops (50 cycles). DTS does not incur cache invalidations or flushes on the big cores, where hardware-based coherence is available. It takes a few cycles to interrupt the tiny core, and 10 to 50 cycles to interrupt the big core, since an interrupt needs to wait until all instructions in-flight in the processor pipeline are committed before jumping to the handler. The total number of cycles spent on DTS is less than 1% of the total execution time, and thus DTS has minimal performance overhead.

D. Results on Larger-Scale Systems

To evaluate our techniques on a larger-scale system, we select five application kernels with larger input datasets, and execute these kernels on a 256-core big.TINY system. Compared to the 64-core big.TINY system we use for the rest of the paper, the 256-core big.TINY system has four big cores, 252 tiny cores, 4× the memory bandwidth, and 4× the number of L2 banks. We scale up the input sizes to increase the dynamic instruction count and amount of parallelism, in order to approximately achieve *weak scaling*. For HCC configurations, we select the best performing HCC protocol, GPU-WB. The results are presented in Table V. A big core has 16× the L1 cache capacity as a tiny core (64KB vs. 4KB), and therefore the total L1 cache capacity of the 256-core big.TINY system is equivalent to 20 big cores. The results demonstrate that, even in a larger-scale big.TINY system, HCC with our work-stealing runtime allows dynamic task-parallel applications to achieve similar performance with simpler hardware compared to big.TINY/MESI. Our DTS technique significantly improves performance of our work-stealing runtime on HCC. The relative benefit of DTS is *more* pronounced in the 256-core big.TINY system than in the 64-core system, because stealing overheads without DTS are higher in systems with more cores.

E. Summary

Our overall results show that on big.TINY systems, combining HCC and work-stealing runtimes allows dynamic task-parallel applications with TBB/Cilk-like programming models to work with simpler hardware compared to hardware-based coherence, at the cost of slightly worse performance. The relative performance and energy efficiency of three HCC protocols depend on the characteristics of the application. DTS reduces the number of invalidations on all three HCC protocols. It also reduces the number of flushes on *big.TINY/HCC-gwb*. DTS is able to close the gap and enables HCC systems to match or even exceed the performance of fully hardware-based cache coherent systems. Regarding the geometric mean of all apps, the best performing HCC protocol combined with DTS achieves 21% improvement in performance, and similar amount of network traffic compared to hardware-based cache coherence.

VII. RELATED WORK

There is a large body of work on optimizing hardware-based cache coherence protocols. Coarse-grained sharer list approaches reduce storage overhead for tracking sharers of each cache line, at the expense of unnecessary invalidations and broadcasts [14, 27, 48, 74, 75]. Hierarchical directories attacks the same overhead by adding additional levels of indirection in the directories [45]. Techniques for increasing directory utilization allow for smaller directories, but increase hardware complexity [23,59]. Prior work like SWEL propose removing the sharer list altogether [19, 54]. However, those techniques perform well only if most of the cache lines are not shared. Other optimization techniques, such as dynamic self-invalidation [41], token coherence [44, 55], and destination-set prediction [43], have also been proposed for systems with sequential consistency (SC). There has been work focusing on optimizing banked shared caches, particularly those with non-uniform cache access (NUCA), using static and/or dynamic techniques. Jigsaw [6] addresses the scalability and interference problem in shared caches by allowing software to control data placement in collections of bank partitions. Coherence domain restriction [25] improves scalability of shared caches by restricting data sharing. Whirlpool [49] leverages both static and dynamic information to reduce data movement between shared caches. HCC used in our work primarily addresses scalability of private caches and can be complementary to these NUCA techniques in shared caches.

Techniques to improve hardware-based cache coherence protocols by leveraging relaxed consistency models (e.g., release consistency (RC) [21, 26, 33, 34, 58], entry consistency [8], and scope consistency [30]) have been proposed in the literature as well. An important observation of this prior work is, unlike in the case of SC, cache coherence only needs to be enforced at synchronization points for relaxed consistency models. Some prior work proposed to use self-invalidation at acquires in RC to remove the need of tracking sharers [33, 34, 58]. SARC [33] and VIPS [58] further eliminate the directory all together by leveraging a write-through policy. TSO-CC is another self-invalidation based protocol designed for the total store ordering (TSO) consistency model [22]. To maintain the stronger TSO consistency, however, TSO-CC needs additional hardware logic such as timestamps and epoch tables. Our work-stealing runtime system leverages the DAG-consistency [10] model for heterogeneous coherence.

There have been several software-centric coherence protocols in prior work. DeNovo [17] uses self-invalidation to eliminate sharer-tracking. In addition, by requiring the software to be data-race-free (DRF), DeNovo eliminates transient states. While DeNovo greatly simplifies the hardware, the DRF requirement limits its software scope. To address this problem, follow-up work on DeNovo (i.e., DeNovo-ND [70] and DeNovoSync [69]) added support for lock-based nondeterminism and arbitrary synchronization. We used DeNovoSync in our studies. GPU coherence protocols combine write-through and self-invalidation, and achieve the simplest hardware suitable for GPU workloads [63, 64]. Our GPU-WT protocol is similar to the ones described in the literature. GPU-WB differs from GPU-WT in deferring the write-through until barriers. GPU-WB is studied in manycore systems with a single globally shared task queue [35].

Prior work has proposed efficient heterogeneous coherence protocols for integrated GPU-CPU systems [29, 53], and accelerators [39]. Spandex explores an efficient interface for various hardware-based and software-centric cache coherence protocols [3]. Similar to this prior work, the heterogeneous coherence protocol we described in this paper uses a MESI-based LLC to integrate different cache coherence protocols in private caches.

Past work has looked at improving task-based programming models in various ways such as improving the scheduling algorithms [12, 24, 73], improving the efficiency of software-based task queues [1, 16], and reducing the overhead of memory fences [40]. Carbon [38] improves the performance of fine-grained tasks. ADM [60] uses active messages [72] to improve task scheduling. While both Carbon and ADM provide a task-based model, their programming frameworks are drastically different from widely-used ones like Cilk or TBB. Prior work has also explored improving work-stealing runtime systems for heterogeneous system with different core architectures [71]. We implemented our proposed direct task-stealing (DTS) mechanism with user-level interrupt [18], which is similar to but much simpler than active messages used by ADM.

VIII. Conclusions

This paper has demonstrated how careful hardware/software co-design can enable efficiently exploiting dynamic task parallelism on heterogeneous cache-coherent systems. This work provides a small yet important step towards ushering in an era of complexity-effective big.TINY architectures that combine a few big out-of-order high-performance cores with many tiny energy-efficient cores, while at the same time supporting highly productive programming frameworks. Our vision is of a future where programmers use traditional dynamic task parallel programming frameworks to improve their performance across a few big cores, and then seamlessly without effort these applications can see significant performance improvements (our results suggest up to 2–3× comparing *big.TINY/HCC-DTS-gwb* to *O3×4*) by simply allowing collaborative execution across big and tiny cores using work stealing.

Acknowledgments

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. The authors acknowledge and thank I-Ting Angelina Lee for useful discussions on work-stealing runtimes in general and insightful feedback on how to implement such runtimes on HCC. The authors also thank Ragav Kumar and Ryan Cunningham for their help in developing task-parallel applications. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

References

[1] U. A. Acar, A. Chargeéraud, and M. Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Feb 2013.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

[3] J. Alsop, M. Sinclair, and S. V. Adve. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[4] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 20(3):404–418, Mar 2009.

[5] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, May 2002.

[6] N. Beckmann and D. Sanchez. Jigsaw: Scalable Software-Defined Caches. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2013.

[7] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2008.

[8] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. *The Midway Distributed Shared Memory System*. Digest of Papers. Compcon Spring, 1993.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[10] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 1996.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Aug 1995.

[12] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, Sep 1999.

[13] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):891–902, Apr 2017.

[14] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2005.

[15] P. Charles, C. Grothoff, V. Sarkar, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Oct 2005.

[16] D. Chase and Y. Lev. Dynamic Circular Work-Stealing Deque. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jul 2005.

[17] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2011.

[18] J. Chung and K. Strauss. User-Level Interrupt Mechanism for Multi-Core Architectures. US Patent 8255603, Aug 2012.

[19] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[20] S. Davidson, S. Xie, C. Torng, K. Al-Hawaj, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. G. Dreslinski, C. Batten, and M. B. Taylor. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, 38(2):30–41, Mar/Apr 2018.

[21] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed Consistency and its Effects on the Miss Rate of Parallel Programs. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Aug 1991.

[22] M. Elver and V. Nagarajan. TSO-CC: Consistency Directed Cache Coherence for TSO. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2014.

[23] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A Scalable Directory for Many-Core Systems. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2011.

[24] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, May 1998.

[25] Y. Fu and D. Wentzlaff. Coherence Domain Restriction on Large Scale Systems. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2015.

[26] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, May 1990.

[27] A. Gupta, W. dietrich Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. *Int'l Conf. on Parallel Processing (ICPP)*, Aug 1990.

[28] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview Scalability Analyzer. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 2010.

[29] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2014.

[30] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry consistency. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 1996.

[31] Intel Cilk Plus Language Extension Specification, Version 1.2. Intel Reference Manual, Sep 2013.

[32] Intel Threading Building Blocks. Online Webpage, 2015 (accessed Aug 2015).

[33] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54–65, Sep 2010.

[34] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *Int'l Symp. on Computer Architecture (ISCA)*, May 1992.

[35] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2009.

[36] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve. Stash: Have Your Scratchpad and Cache It Too. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

[37] R. Kumar, T. G. Mattson, G. Pokam, and R. V. D. Wijngaart. The Case for Message Passing on Many-Core Chips. *Multiprocessor System-on-Chip*, pages 115–123, Dec 2011.

[38] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2007.

[39] S. Kumar, A. Shriraman, and N. Vedula. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

[40] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-Based Memory Fences. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 2011.

[41] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory MultiProcessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jul 1995.

[42] C. E. Leiserson. The Cilk++ Concurrency Platform. *Design Automation Conf. (DAC)*, Jul 2009.

[43] M. M. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-memory Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2003.

[44] M. M. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2003.

[45] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Communications of the ACM*, Jul 2012.

[46] M. McCool, A. D. Robinson, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[47] M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, J. Balkind, A. Lavrov, M. Shahrad, S. Payne, and D. Wentzlaff. Piton: A Manycore Processor for Multitenant Clouds. *IEEE Micro*, 37(2):70–80, Mar/Apr 2017.

[48] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2005.

[49] A. Mukkara, N. Beckmann, and D. Sanchez. Whirlpool: Improving Dynamic Cache Management with Static Data Classification. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2016.

[50] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches, 2009.

[51] A. Olofsson. Epiphany-V: A 1024-processor 64-bit RISC System-On-Chip. *CoRR arXiv:1610.01832*, Aug 2016.

[52] OpenMP Application Program Interface, Version 4.0. OpenMP Architecture Review Board, Jul 2013.

[53] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous System Coherence for Integrated CPU-GPU Systems. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2013.

[54] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2010.

[55] A. Raghavan, C. Blundell, and M. M. Martin. Token Tenure: PATCHing Token Counting Using Directory-Based Cache Coherence. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 2008.

[56] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[57] The RISC-V Instruction Set Manual Volume II: Privileged Architecture. Online Webpage, 2019 (accessed Jun 8, 2019).

[58] A. Ros and S. Kaxiras. Complexity-Effective Multicore Coherence. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2012.

[59] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2012.

[60] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2010.

[61] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Jan 2017.

[62] J. Shun and G. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Feb 2013.

[63] M. D. Sinclair, J. Alsop, and S. V. Adve. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[64] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache Coherence for GPU Architectures. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.

[65] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, Mar/Apr 2016.

[66] L. Song, M. Feng, N. Ravi, Y. Yang, and S. Chakradhar. COMP: Compiler Optimizations for Manycore Processors. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[67] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 2011.

[68] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan/Feb 2015.

[69] H. Sung and S. V. Adve. DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2015.

[70] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2013.

[71] C. Torng, M. Wang, and C. Batten. Asymmetry-Aware Work-Stealing Schedulers. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[72] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *Int'l Symp. on Computer Architecture (ISCA)*, May 1992.

[73] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P.-C. Yew. An Adaptive Task Creation Strategy for Work-Stealing Scheduling. *Int'l Symp. on Code Generation and Optimization (CGO)*, Apr 2010.

[74] J. Zebchuk, E. Safi, and A. Moshovos. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2007.

[75] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing Pattern-Based Directory Coherence for Multicore Scalability. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2010.