Beyond Static Parallel Loops: Supporting Dynamic Task Parallelism on Manycore Architectures with Software-Managed Scratchpad Memories

Lin Cheng*
lc873@cornell.edu
Cornell University
USA

Max Ruttenberg*
mrutt@washington.cs.edu
University of Washington
USA

Dai Cheol Jung
dcjung@uw.edu
University of Washington
USA

Dustin Richmond
drichmond@ucsc.edu
University of California, Santa Cruz
USA

Michael Taylor
profmbt@cs.washington.edu
University of Washington
USA

Mark Oskin
oskin@cs.washington.edu
University of Washington
USA

Christopher Batten
cbatten@cornell.edu
Cornell University
USA

ABSTRACT

Manycore architectures integrate hundreds of cores on a single chip by using simple cores and simple memory systems usually based on software-managed scratchpad memories (SPMs). However, such architectures are notoriously challenging to program, since the programmers need to manually manage all aspects of data movement and synchronization for both correctness and performance. We argue that this manycore programmability challenge is one of the key barriers to achieving the promise of manycore architectures. At the same time, the dynamic task parallel programming model is enjoying considerable success in addressing the programmability challenge of multi-core processors with tens of complex cores and hardware cache coherence.

Conventional wisdom suggests a work-stealing runtime, which forms the core of most dynamic task parallel programming models, is ill-suited for manycore architectures. In this work, we demonstrate that a work-stealing runtime is not just feasible on manycore architectures with SPMs, but such a runtime can also significantly improve the performance of irregular workloads when executing on these architectures. We also explore three optimizations that allow the runtime to leverage unused SPM space for further performance benefit. Our dynamic task parallel programming framework achieves 1.2–28.5× speedup on workloads that benefit from our techniques, and only induces minimal overhead for workloads that do not.

*Both authors contributed equally to this research.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582020

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; • **Computing methodologies** → **Parallel computing methodologies**; **Parallel algorithms**.

KEYWORDS

Manycore architecture, parallel programming, load-balancing, scratchpad memory, fine-grained threading

ACM Reference Format:
Lin Cheng, Max Ruttenberg, Dai Cheol Jung, Dustin Richmond, Michael Taylor, Mark Oskin, and Christopher Batten. 2023. Beyond Static Parallel Loops: Supporting Dynamic Task Parallelism on Manycore Architectures with Software-Managed Scratchpad Memories. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3582016.3582020

1 INTRODUCTION

Scratchpad memories (SPMs) provide key advantages in single-chip parallel architectures. Most crucially, they improve the efficiency and scaling of the memory system by removing the need for a coherence protocol and associated network traffic. When used effectively, SPMs can yield critical performance and energy savings by reducing data movement, improving synchronization times, and eliminating overheads that can arise from false sharing. As a result, academic and industry chip-makers have increasingly favored these software-managed fast memories over L1 caches as core counts scale from the tens to hundreds and thousands [2, 11, 12, 18, 41], a trend illustrated in Figure 1.

Replacing the traditional L1 caches in favor of SPMs comes at a cost to software productivity. Manycore architectures (i.e., those with more than a hundred cores) that rely heavily on SPMs are notoriously challenging to program. Such systems usually require programmers to write applications in low-level C environments

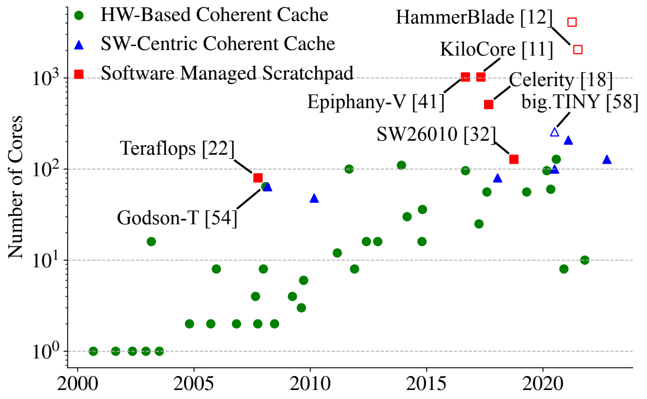

Figure 1: On Chip Memory Hierarchy in Manycore Architectures – SPM is needed for manycore architectures to reach very high core counts. Filled marker = real chip; unfilled marker = proposal/simulator only. Data is in part from CPU DB [17].

and/or directly in assembly. This places the burden on the programmer to explicitly manage data coherence among private memories and adopt a more restricted programming model (e.g., explicit task partitioning [29], message passing [41], and remote store programming [18]). The cumbersome programming environment coupled with the need for software optimizations to realize the performance promised by hardware is a critical barrier to widespread adoption of most manycore architectures with software-managed SPMs.

One common method to facilitate programming on such architectures is by providing domain-specific frameworks. This approach has had success in application spaces such as graph processing [12] and deep learning [15]. These frameworks express domain-specific workloads effectively and achieve high performance. However, not every domain is covered. Extending and re-purposing these frameworks for another domain requires non-trivial effort by programmers. General-purpose parallel programming frameworks provide more flexibility than domain-specific ones. However, most such frameworks (e.g., OpenCL [30]) usually adopt a single-program-multiple-data (SPMD) programming model, in which native support for dynamic work scheduling and load balancing is highly limited, if provided at all.

In this work, we take inspiration from the success of the dynamic task parallel programming model in the multi-core era, and attempt to address the programmability challenge of manycore architectures with software-managed SPMs by offering a dynamic task parallel programming framework that is similar to those that are common on multi-core systems (e.g., Intel Cilk Plus [24], Intel Threading Building Blocks (TBB) [25], and OpenMP [5, 42]). These programming frameworks 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 [38].

We demonstrate our ideas by implementing the proposed dynamic task parallel programming framework on an open source manycore. Our approach allows dynamic task parallel applications written for traditional hardware-based cache coherence multi-cores to work on manycore architectures with only minimal changes to the software. In Section 2, we provide a general background on our target open-source manycore architecture, work-stealing runtimes, and the manycore architecture programmability challenge. In Section 3, we describe in detail how to implement a work-stealing runtime, which is the core component of dynamic task parallel frameworks, on manycore architectures with software-managed SPMs. In Section 4, we discuss three optimizations for enabling the runtime to leverage SPMs and achieve high performance. In Section 5 and Section 6, we use a cycle-accurate RTL evaluation methodology to demonstrate the potential of our approach with four categories of workloads: *static-balanced*, *static-unbalanced*, *dynamic-balanced*, and *dynamic-unbalanced*. While conventional wisdom believes implementing a work-stealing runtime is either not viable or not beneficial on systems that do not have caches [58, 61], our evaluation demonstrates that our proposed task parallel programming framework can achieve 1.2×–28.5× speedup for workloads that benefit from our techniques, and only induce minimal overhead for workloads that do not.

The contributions of this work are: (1) we provide, to the best of our knowledge, the first work that describes the implementation of a work-stealing runtime on manycore architectures with software-managed SPMs; (2) we summarize three optimizations which enable the runtime to leverage scratchpad memories to achieve high performance; and (3) we provide a detailed cycle-accurate evaluation using a silicon-validated RTL design of an open source manycore architecture.

2 BACKGROUND

In this section, we first introduce the target manycore architecture. We then give a brief introduction on dynamic task parallelism and the programmability challenge of manycore architectures.

2.1 Target Manycore Architecture

While manycore architectures have a broad software and hardware design space, they usually share a set of common features. These features include simple cores, software-managed memory systems, mesh-based on-chip networks, and simple low-level programming interfaces. In this section, we provide a brief introduction on the HammerBlade architecture which is representative of modern manycore architectures [1, 11, 12, 18, 41, 48, 49].

The HammerBlade manycore architecture is a configurable-sized array of scalar RISC-V cores supporting the floating-point and AMO extensions. Each core owns a 4 KB region of low-latency SPM. Cores communicate with a load/store interface over a 2-D mesh-with-ruching on chip network (OCN) [26, 44]. Fig. 2 presents an architectural diagram of a small-scale (i.e., 128-core) HammerBlade system. There are four levels of the memory hierarchy: a core-local scratchpad; inter-core scratchpad(s); a banked, last-level cache (LLC); and DRAM. The core-local SPM, remote SPMs, caches, and other network locations are mapped to non-intersecting regions of a core's address space. Consequently, the architecture exposes a partitioned global address space (PGAS) programming model.

Memory operations can complete out of order when accessing remote SPM due to HammerBlade's relaxed-consistency model. This introduces complexity during synchronization. Software can enforce memory ordering explicitly with fences.

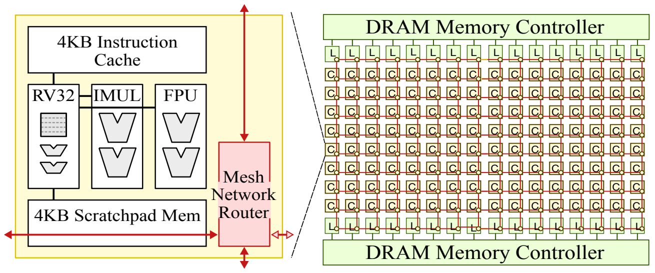

Figure 2: HammerBlade Manycore Architecture Hardware – A version of HammerBlade with 128 cores (C) and 32 last-level cache (L) banks interconnected via mesh-based on-chip network; each core is a RISC-V RV32IMAF processor (RV32) with instruction cache and 4 KB SPM.

2.2 Programming Models for Dynamic Task Parallelism

Task parallelism is a style of parallel programming where the workload is divided into *tasks* (i.e., units of computation that can execute in parallel). Dynamic task parallelism is a subset of task parallelism in which tasks and dependencies among tasks are generated at runtime. Dynamically generated tasks are assigned to available worker threads based on a certain scheduling algorithm. The most common computation model for dynamic task parallelism is the *fork-join* model. It was popularized by MIT Cilk [8] and then adopted by various parallel programming frameworks [13, 24, 25, 31, 47, 51]. In a task parallel programming framework that adopts the fork-join model, the process in which a task forks two or more parallel tasks is also referred to as *spawning* tasks. The newly created tasks are called the *child* tasks and the original task is called the *parent*. The parent task can continue until it reaches the point where the *join* (also commonly referred to as *wait*) primitive is called. The parent task blocks until all of its child tasks have finished. The fork-join model has the following properties: (1) a task can only wait for its children to join (e.g. no waiting on locks); and (2) a task cannot complete until all of its children complete and join it. This set of properties is called *fully-strict* in Cilk literature [9, 20].

Work-stealing is likely the most widely-adopted scheduling algorithm for task parallel programming frameworks [10]. In a typical work-stealing runtime, each thread is associated with a *task queue* to store tasks that are ready for execution. The task queue is usually implemented with 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 becomes idle, either because a parent task is waiting for its child tasks to return or the thread has no active task running, it attempts to *dequeue* a task from its own task queue from the tail (i.e., in last-in-first-out (LIFO) order). If the task queue is empty, the thread then attempts to *steal* a task from the head of the task queue of another thread (i.e., in first-in-first-out (FIFO) order). The stealing thread becomes a *thief*, and the thread whose tasks are stolen becomes a *victim*. Stealing in FIFO order allows the thief to steal a task that is located higher in the task graph, which typically contains more work. The stealing mechanism automatically balances the workload across threads, leads to better locality, and helps establish time and space bounds [10, 20].

2.3 Manycore Architecture Programmability Challenge

Manycore architectures that have high core counts (i.e., more than a hundred cores) and adopt software-manage scratchpad memories have been proposed and fabricated by both academia and industry [2, 11, 12, 18, 41]. While the hardware has gained most of the attention, the software stack of such architectures is less explored. As is the case with similar architectures, programming HammerBlade without loss of domain generality requires using a low-level C runtime environment. This demands that the programmer have both an extensive domain knowledge for their application and for the underlying hardware. Concerns such as data placement, synchronization, and load-balancing are left entirely to the programmer. Having to use a low-level C runtime environment prevents easily reusing existing code written for multi-cores and requires most applications to be completely rewritten for such manycore architectures.

Prior works propose leveraging a domain-specific framework approach to address the manycore programmability challenge (e.g., machine learning frameworks based on adapting PyTorch [15] and graph processing frameworks based on porting GraphIt [12]). A domain-specific framework approach has three main drawbacks: (1) programmers need to rewrite their applications to use the constructs provided by the framework; (2) the framework is designed for a specific domain, meaning it is difficult to express computation from other domains; and (3) there is no easy way for a programmer who has little knowledge about the underlying manycore hardware to extend the framework.

3 SUPPORTING DYNAMIC TASK PARALLELISM ON MANYCORE ARCHITECTURES

In this work, we propose resolving the manycore architecture programmability challenge by implementing a TBB/Cilk-like dynamic task parallel programming framework on such systems. Compared to the typical low-level C runtimes provided by these architectures which usually adopt the SPMD programming model, the proposed framework supports parallel patterns beyond simple static parallel loops, allows parallel patterns to be arbitrarily nested, and provides dynamic load balancing. Compared to prior work on resolving the programmability challenge through domain-specific frameworks, our framework is general-purpose. Furthermore, it provides an interface with which programmers that have used Cilk/TBB or OpenMP are familiar, making it possible to port legacy code to manycore architectures.

The core component of the proposed TBB/Cilk-like dynamic task parallel programming framework is a work-stealing runtime. While how to implement work-stealing runtimes on systems with hardware-based coherence [8], software-centric coherence [36, 53, 58], and distributed memory [19, 45, 50] has been studied extensively in the literature, conventional wisdom claims that implementing such a runtime is either not viable or not beneficial on systems with software-managed scratchpad memories [58, 61].

In this section, we first demonstrate our programming model using running examples. We give details on how we implement a low-level API for spawning and synchronizing with new tasks.

```
template <typename Func>
class FibTask : public Task {
public:
  FibTask( int n_, int* sum_,
           Task* parent_) :
    n( n_ ), sum( sum_ ),
    parent( parent_ );
  Task* execute() {
    if ( n < 2 ) {
      *sum = n;
      return;
    }

    int x, y;
    FibTask a( n - 1, &x, this );
    FibTask b( n - 2, &y, this );
    this->set_ready_count( 1 );

    task::spawn(b);
    a.execute();

    task::wait();
    *sum = x + y;
    return nullptr;
  }
private:
  int n;
  int* sum;
  Task* parent;
};
```

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

```
class Task {
public:
  Task();
  virtual Task* execute();
  void set_ready_count(
    int ready_count );
private:
  int ready_count;
};
```

(b) `Task` base class

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

(c) `fib` using `parallel_invoke`

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

(d) `vvadd` using `parallel_for`

```
void sum( int a[], int n ) {
  int ident = 0;
  parallel_reduce(0, n, ident,
    [&](int i) {
      return a[i];
    },
    [](int x, int y) {
      return x + y;
    });
}
```

(e) `sum` using `parallel_reduce`

Figure 3: Task-Based Parallel Programs – Examples for calculating the Fibonacci number using (a) a low-level API with explicit calls to `spawn()` and `wait()`, the implementations of which are shown in Figure 4; and (c) a high-level API with templated `parallel_invoke()` pattern. (b) shows the `Task` based class from which the `FibTask` class inherits in (a). (d) and (e) show alternative templated patterns `parallel_for()` and `parallel_reduce()` respectively.

We also give a description of a higher-level API for expressing common parallel programming patterns. Lastly, We describe a naive implementation of a work-stealing runtime on the HammerBlade manycore, before discussing key optimizations in Section 4.

3.1 Running Example

We use an application programming interface (API) similar to Intel TBB to illustrate our programming model (see Figure 3). Each task is described by a C++ class derived from the `Task` base class (Figure 3 (b)) which contains an `execute()` method and a metadata variable `ready_count`, also known as the *reference counter*. This metadata tracks a task's unfinished children. After a task finishes execution, it checks if it has a parent task. If so, the child will decrement the `ready_count` variable of its parent task to signal its completion. A task in `wait` will be blocked until its `ready_count` reaches 0 (i.e., all children have completed their execution). This mechanism enforces the ordering between parent and children: a task cannot complete until all of its children complete and join it (see Section 2.2). Programmers override the virtual `execute()` function to hold the logic of the concrete task. In this example (Figure 3 (a)), after creating two child tasks `a` and `b`, one for `fib(n-1)` and one for `fib(n-2)`, the parent task (i.e., `fib(n)`) puts `fib(n-2)` onto the task queue and executes `fib(n-1)` locally, before calling `wait()`, which blocks its execution until task `fib(n-2)` returns. The parent task then calculates `fib(n)` by adding the partial results from both tasks and returns.

Besides the low-level APIs, our framework also provides templated functions that support various parallel patterns. This includes `parallel_invoke()` for divide-and-conquer (Figure 3 (c)), `parallel_for()` for parallel loops (Figure 3 (d)), and `parallel_reduce()` for parallel reduction (Figure 3 (e)).

3.2 A Naive Work-Stealing Runtime

The key challenge of implementing a work-stealing runtime on a system like HammerBlade is to cope with the lack of data coherence mechanisms. Typical work-stealing runtimes are built upon various shared data structures (e.g., task queues and reference counters). Where to allocate them and how to keep them coherent is critical to both correctness and performance. While possible if carefully implemented, programmers usually avoid keeping copies of shared data in software-managed scratchpads. Instead, they tend to allocate them in the last shared level of the memory hierarchy. While doing so causes longer memory latency when accessing this shared data, allocating it in SPM would require software to keep it coherent, introducing significant software complexity. By allocating all data in the shared memory space, we can easily implement a naive work-stealing runtime that runs on the HammerBlade manycore architecture. Namely, the runtime does not utilize the scratchpads at all: all data lives in the DRAM address space (recall that HammerBlade adopts a PGAS memory model, and DRAM has an address space that is separated from the scratchpads).

Figure 4 (a) shows an implementation of the `spawn()` and `wait()` functions for this naive work-stealing runtime. `spawn` enqueues a task pointer onto the current thread's task queue, and `wait` puts the current thread into a *scheduling loop*. Within the scheduling loop, a thread first checks if all of its child tasks have returned (i.e., `ready_count` has a non-zero value). If so, the thread exits from the scheduling loop and resumes the execution of the parent task (line

8). Otherwise, the thread first attempts to pop a task from the end of its own task queue (i.e., LIFO order, lines 9–15). If there is no task left in the local queue, the current thread becomes a thief and attempts to steal tasks from the queue of another thread, a victim. Tasks are stolen from the victim's head (i.e., FIFO order, lines 17–24). The victim is selected randomly (line 17). When a task is executed, its parent's reference counter is atomically decremented (lines 14 and 23).

Readers familiar with Intel TBB-like work-stealing runtimes may notice that this implementation is similar to the implementation on traditional hardware coherent multi-cores. On hardware coherent multi-cores, hardware cache coherence protocols keep multiple copies of shared data coherent. On HammerBlade, as all data is allocated in DRAM, there is exactly one copy of every shared data. All cores access the same copy. Note that the atomics used for reference counter decrement have release semantics associated. This is to ensure that writes by child tasks complete before the parent task can exit from the scheduling loop (i.e., reference counter reaches 0).

4 SCRATCHPAD ENHANCED RUNTIME

Prior work has shown that leveraging the scratchpad memory is critical to achieving peak performance on manycore architectures [15]. However, SPMs are often underutilized due to the high demand they put on programmers, in addition to the fact that not every workload is able to benefit from leveraging them (e.g., streaming workloads that do not have any reuse of input data). The naive work-stealing runtime we introduced in Section 3 allocates all data, including both the stack and runtime data structures, such as the task queues, in DRAM. While this naive implementation yields a functionally correct work-stealing runtime, it is likely to have sub-optimal performance due to high memory latency and contention at the LLC for applications that have frequent stack operations, task queue operations, or both. Instead of leaving the SPMs unused, we introduce three optimizations which enable work-stealing runtimes to efficiently leverage scratchpads if they are not claimed by programmers. To the best of our knowledge, this is the first work that describes the implementation of a work-stealing runtime that automatically utilizes SPMs on manycore architectures.

Before the runtime can safely claim scratchpad space for its own, it has to know how much scratchpad space is reserved by programmers for user code. Reserving scratchpad space on Mosaic is realized through two APIs: (1) `spm_reserve()` and (2) `spm_malloc()`. `spm_reserve()` sets the maximum amount of SPM a core will use throughout execution. Programmers cannot reserve more space than what is available in the hardware (i.e., 4 KB). `spm_malloc()` returns a pointer to a chunk of memory allocated in the scratchpad. If the total amount of memory allocated/requested through `spm_malloc()` is larger than the amount set by `spm_reserve()`, it reports a failure by returning a null pointer. Our work-stealing runtime allocates a buffer at the top of the scratchpad as requested by the user, and automatically uses the scratchpad space that is not claimed by the user for both the stack and the task queue. By default, our runtime reserves 512 B of a threads's SPM for the task queue and leaves the remaining 3.5 KB for application tasks. However, we also provide APIs to allow experienced programmers to fine-tune the runtime usage of the scratchpad. For example, the programmer can instruct the runtime to only scratchpad allocate the stack but not the task queue.

4.1 Scratchpad-Allocated Stack

Allocating the stack in SPM has been mentioned and explored by various prior work in the literature [15]. However, there are two main concerns on doing the same in the context of a work-stealing runtime: (1) user data can become shared variables when tasks are stolen; and (2) the stack can easily overflow the size of the scratchpad (e.g., recursively called runtime functions such as `wait()` and divide-and-conquer algorithms with deep recursion depth).

Data in the user code (e.g., `y` in line 14 of Figure 3 (a)) includes potential shared variables that can be accessed by more than one core if the corresponding task `b` in line 16 is stolen. However, this is not an issue for manycore architectures which adopt the PGAS memory model (e.g., HammerBlade). The PGAS memory model allows every core to read and write any other core's scratchpad (see Section 2.1), and it enables us to keep unique copies of shared data in a core's SPM. For example, assume `y` mentioned above is allocated in `core_0`'s scratchpad, and the corresponding task (i.e., `b`) is stolen by `core_1`. When `core_1` accesses `y` through the address taken at line 16 while creating the task, it performs a direct remote scratchpad access. The `y` in the scratchpad of the parent task's core remains as the only copy of `y`. The fully-strict properties of dynamic task parallelism (see Section 2.2) guarantees that reads and writes by `core_0` and `core_1` to `y` will not result in any data-race.

Manycore architectures like HammerBlade usually have limited per core scratchpad space (e.g., each core in HammerBlade has a 4 KB SPM). Applications running recursive algorithms (e.g., divide-and-conquer) can easily create deep call stacks, which cannot fit in the SPM. When the stack does not fit, ideally we would like to keep the active and more recent frames (i.e., top frames) in scratchpad memory, since these frames are more likely to be accessed than older ones. To achieve this, one can either put the base of the stack in DRAM, and only start allocating in the scratchpad when the stack reaches a certain depth, or one can spill the older stack frames to DRAM when the scratchpad becomes full. However, both approaches have their caveats: starting in DRAM requires determining an ideal switching depth which can vary from workload to workload, while stack spilling cannot be realized without implementing complex hardware/software mechanisms. In this work, we opt for a simpler but less ideal solution: rather than keeping the top frames in scratchpads, we keep the bottom frames. When the stack overflows available SPM space, it automatically goes to DRAM, and we refer to this as *overflowing to DRAM*. While overflowing does happen, it only happens in applications with deep recursion depth. We optimize for the common case in which the stack can fit in scratchpads.

We leveraged a software/hardware co-design approach and extended each core with a light-weight hardware extension that snoops on the stack pointer register. We added two new control and status registers (CSRs). One for storing the DRAM overflow threshold (i.e., lowest address of the stack space in scratchpad), and the other for storing the pointer to the DRAM overflow buffer.

```
void task::spawn( task* t ) {
  tq[tid].lock_aq()
  tq[tid].enq(t)
  tq[tid].lock_rl()
}

void task::wait( task* p ) {
  while ( p->rc > 0 ) {
    tq[tid].lock_aq()
    task* t = tq[tid].deq()
    tq[tid].lock_rl()
    if (t) {
      t->execute()
      amo_sub_lr( t->p->rc, 1 )
    }
    else {
      int vid = choose_victim()
      tq[vid].lock_aq()
      t = tq[vid].steal()
      tq[vid].lock_rl()
      if (t) {
        t->execute()
        amo_sub_lr( t->p->rc, 1 )
      }
    }
  }
}
```

(a) Runtime Data in DRAM

```
void task::spawn( task* t ) {
  spm_lock.lock_aq()
  spm_tq.enq(t)
  spm_lock.lock_rl()
}

void task::wait( task* p ) {
  while ( p->rc > 0 ) {
    spm_lock.lock_aq()
    task* t = spm_tq.deq()
    spm_lock.lock_rl()
    if (t) {
      t->execute()
      amo_sub_lr( t->p->rc, 1 )
    }
    else {
      int vid = choose_victim()
      TaskQ* remote_tq =
        get_remote_ptr(vid, &spm_tq)
      QLock* remote_lock =
        get_remote_ptr(vid, &spm_lock)
      remote_lock->lock_aq()
      t = remote_tq->steal()
      remote_lock->lock_rl()
      if (t) {
        t->execute()
        amo_sub_lr( t->p->rc, 1 )
      }
    }
  }
}
```

(b) Runtime Data in Scratchpad

Figure 4: Work-Stealing Runtime Implementations – Pseudo-code of `spawn` and `wait` functions for: (a) having runtime data in DRAM; and (b) having runtime data in scratchpads. `tq` = array of task queues; `tid` = thread id; `lock_aq` = acquire lock; `lock_lr` = release lock; `rc` = ready count; `deq` = dequeue from the tail of the task queue; `enq` = enqueue to the tail of the task queue; `steal` = dequeue from the head of the task queue; `choose_victim` = random victim selection; `amo_sub_lr` atomic fetch-and-sub with release semantics; `spm_lock` = task queue lock allocated in scratchpad; `spm_tq` = task queue allocated in scratchpad; `get_remote_pointer` = calculate the address of a piece of data in another core's scratchpad.

When a new frame is pushed onto the stack and the stack pointer is modified, we check if the stack is overflowed (i.e., new stack pointer has become smaller than the DRAM overflow threshold). If so, we replace the stack pointer with the pointer to the core's DRAM overflow buffer and allocate the new frame in DRAM. Similar checks and replacements are performed when a frame is popped off the stack. By default, the runtime allocates a 256 KB stack space for each core to enable deep recursion calls that can produce many stack frames. As we have mentioned before, the runtime calculates available stack space using the information given by programmers through `spm_reserve()`. It then allocate a buffer with proper size for each core in DRAM, and writes both the pointer of the DRAM allocated buffer and overflow threshold address to corresponding CSRs.

Although we chose to implement overflowing to DRAM on HammerBlade with a software/hardware co-design approach, the same functionality can be easily implemented in software with modifications to the compiler on a system where making hardware changes is not feasible. Namely, we can take an approach similar to one proposed by [57] which modifies the compiler to generate checkpoints at which an overflow stack region will be swapped-in when the newly created frame would not fit in SPM. A pointer rewrite scheme can redirect the new stack pointer to the DRAM overflow buffer. While this software solution involves adding extra instructions compared to our software/hardware co-design approach, this check is light-weight and the fast path (i.e., frames other than the frame that crosses the boundary) contains only two instructions: a load instruction for loading the overflow threshold address and a conditional jump which compares the stack pointer with the threshold address. The threshold address can and should be allocated in the scratchpad for low overhead access.

4.2 Scratchpad-Allocated Task Queue

A common goal of various parallel programming frameworks is to reduce the overhead of their runtimes. Our framework is not an exception. In the naive runtime implementation, all runtime data structures, including the core local task queues, are allocated in DRAM. Applications with fine-grained tasks tend to induce frequent task queue operations as they generate more tasks than coarse-grained ones. For these applications, being able to manipulate the local task queue efficiently is key to achieving high performance. The local scratchpad has a 2-cycle access latency where the DRAM has an access latency of tens to hundreds of cycles. Therefore, instead of going to DRAM for runtime data, we would like to keep them in the SPMs for faster accesses.

Similar to what we have mentioned in Section 4.1, data coherence is not an issue as we keep only one copy of data and perform remote scratchpad accesses if the data is located in another core's SPM. However, unlike the user data, to which a pointer is passed around dynamically, a core must know before run-time where other cores' task queues are located in order to conduct stealing without first accessing a DRAM allocated centralized data structure, such as the array of pointers to task queues (i.e., `tg[]` in Figure 4 (a)). Having such a DRAM allocated data structure diminishes the benefit of keeping stealing traffic away from DRAM. To achieve this, we reserve, by default, the top 512 B of the scratchpad for the core local task queue. The task queue is allocated at a fixed offset from the scratchpad base pointer across all cores. Therefore, if we have a pointer to the local task queue, we can easily calculate the pointer of the task queue of any other core. Figure 4 (b) shows an implementation of `spawn()` and `wait()` for our runtime which has both the stack and runtime data structures in the SPMs. The first noticeable difference is instead of loading the victim's queue from an array

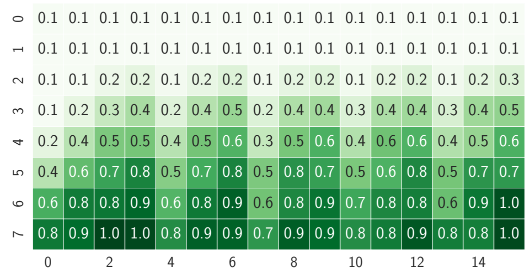

Figure 5: Normalized Remote Scratchpad Load Latency – Remote scratchpad load latency of 128 cores arranged in 16 rows and 8 columns, normalized to the core which has the highest latency.

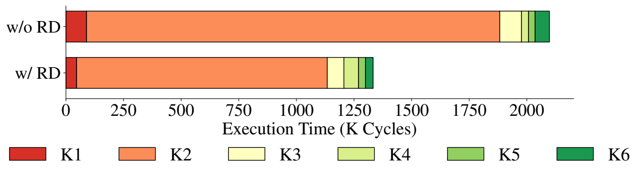

Figure 6: Performance Impact of Read-Only Data Duplication – Execution time of six parallel kernels (K1 to K6) in one iteration of PageRank with and without read-only data duplication optimization.

(line 18 in Figure 4 (a)), we calculate the address of victim's queue using the address of the local queue (lines 18–19 in Figure 4 (b)). We also separate the spin lock protecting the task queue from the queue itself (lines 2–4 in Figure 4 (b)). Doing so allows us to directly calculate the address of the remote spin lock (lines 20–21 in Figure 4 (b)): we do not need the remote scratchpad access for loading the pointer of the lock as in the case where the lock is a member of the task queue.

4.3 Read-Only Data Duplication

After implementing the two optimizations described above, profiling data collected from the one of the apps (i.e., *PageRank*) shows an unexpected pattern. Figure 5 shows a heat map of normalized remote scratchpad access latency measured on each core in the 16×8 mesh. From the plot we can observe a clear pattern: cores that are located farther away from `core_0` (upper left corner) generally have longer remote scratchpad access latency. Note that, the distance in Y-direction has a more significant impact than the distance in X-direction. This is because HammerBlade adopts X-Y routing and when all other cores are accessing `core_0`, the bandwidth in the Y-direction is much scarcer. The difference of latency within the same row is caused by the network topology of the 2-D mesh-with-ruching OCN [26, 44]. Our work-stealing runtime selects victims randomly, and thus we expect cores read and write their peers' scratchpads uniformly and there should not be any hot spots.

A closer look at the profiling data revealed the causes: (1) when we implement the high-level templated functions, such as `parallel_for()`, we keep a pointer to the user defined lambda function in the customized task class; and (2) in the user code, we write the lambda functions using reference capture (&), including for read-only values (e.g., pointers `dst` in line 5 of Figure 3 (d)). On systems with hardware-base or software-centric coherence, this read-only data can be cached and reused. However, in our case, these values are all allocated on the scratchpad of `core_0`, and thus other cores repeatedly load from `core_0`. This traffic to `core_0` causes congestion in the OCN. We resolve this issue by changing both the runtime and user code to duplicate read-only data that is allocated in the scratchpad (e.g., capture `dst` in Figure 3 (d) by value). We show the performance impact of the read-only data duplication optimization on *PageRank* in Figure 6. Each iteration of *PageRank* is composed by six parallel kernels. The proposed optimization is able to reduce execution time of all but one kernel, and achieve an overall speedup of 1.57×. Read-only data duplication applies to the case where the stack is DRAM allocated as well. It helps eliminate the hot spot in LLC in a similar manner as it eliminates the hot spot in `core_0`'s SPM. We enable this optimization for all work-stealing runtime configurations.

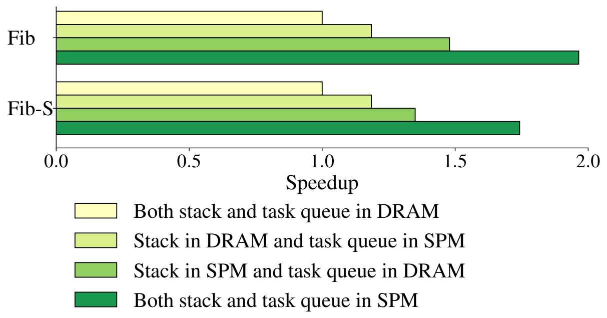

Figure 7: Speedup from Optimizing Data-Placement with SPM in Work-Stealing Runtime – Fib = measured speedups with the proposed SW/HW co-design scheme; Fib-S = estimated speedups with the 2-instruction SW-only scheme. Note that when both the stack and the tasks queues are in DRAM, Fib and Fib-S reduce to a common implementation and thus have identical time-to-completions.

4.4 Micro-Benchmarking

We use *Fib*, a widely adopted micro-benchmark for demonstrating work-stealing runtimes in the literature, to illustrate the benefits of having the runtime leveraging the scratchpads. Figure 3 (c) shows its implementation, and Section 5.1 provides details on the simulated hardware. *Fib* is known for generating many tasks each of which only contains a minimal amount of compute. It yields both frequent stack operations (both runtime function calls and user-defined functor calls) and frequent task queue operations. We evaluate *Fib* on four variants of the runtime: both stack and task queue in DRAM which is the naive implementation we introduced in Section 3, stack in DRAM and task queue in scratchpad, stack in scratchpad and task queue in DRAM, and both stack and task queue in scratchpad. Results are summarized in Figure 7. From the plot we can observe that, as we expected, the naive runtime implementation has the worst performance. As we add optimizations and migrate either the stack or the task queue to scratchpad memories, we observe improved performance due to reduced access latency. Compared with task queue in SPM, stack in SPM shows better performance and it illustrates that having low latency access to the stack is more important for *Fib*. This is caused by: (1) the task queue is protected by a spin lock and the time spent on getting the lock, instead of accessing the task queue itself, dominates the execution time of pushing/popping task queues; and (2) stack operations (e.g., register spilling and saving/restoring saved registers) generate more traffic than task queue operations. Best performance is achieved when both optimizations are applied (i.e., both task queue and stack in SPM).

We also provide a first-order estimation on the impact of implementing the stack overflowing technique with the 2-instruction scheme in software (Section 4.1) by adding an additional 2-cycle delay to each `jal` and `ret` instruction. Results are illustrated in Figure 7 as *Fib-S*. We can observe that both configurations which have stack in SPM achieve less performance improvement for *Fib-s* than for *Fib* due to the overhead added by the extra instructions. However, both variants still perform significantly better than the naive implementation. Note that *Fib* is close to the worst case for the potential software overflowing scheme, as it produces many tasks each with little compute and thus frequent stack frame pushing/popping with short-living task body. In more realistic workloads, we expect the overhead of the potential software overflowing scheme to be much less significant.

5 EVALUATION METHODOLOGY

In this section, we describe our RTL-level cycle-accurate performance modeling methodology. We used this to quantitatively evaluate the proposed work-stealing runtime. We also give a brief introduction on the workloads we used in the evaluation.

5.1 Simulated Hardware

We model the HammerBlade manycore architecture using cycle-accurate RTL simulation. We leverage an RTL simulator to model a silicon-validated small-scale early version of the HammerBlade manycore system running at 1.5 GHz with 16 columns and 8 rows (i.e., 128-cores in total). The RTL of this design has been validated in silicon. The DRAM timing is modeled with the timing-accurate open-source DRAMSim3 simulator [33]. We model a single 1.0 GHz HBM2 channel with a bus width of 64 and a burst length of 4, yielding a theoretical peak bandwidth of 16 GB/s. We model one HBM2 channel because, through experimentation, we found that 128 cores is required to saturate a single channel's bandwidth. Performance counters are implemented with nonsynthesizable SystemVerilog `bind` statements. This allows us to conduct performance analysis without introducing any overhead to the workloads or modifying the digital logic design.

5.2 Runtimes

We conduct evaluation on both a traditional static runtime which supports only statically scheduled parallel loops and the proposed work-stealing runtime. We implement two variants of the static runtime, one variant has stacks allocated in DRAM and the other has stacks allocated in the SPM. We evaluate all four variants of the work-stealing runtime as in Section 4.4.

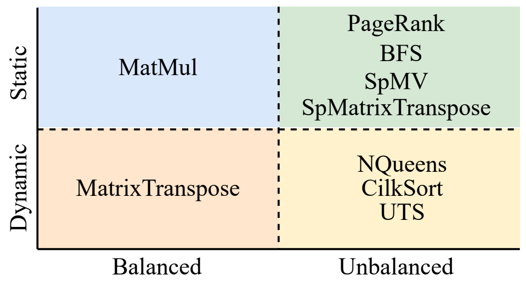

Figure 8: Anatomy of Workloads – we categorize workloads into four categories based on if he workload leverages dynamic parallelism and if the tasks have load imbalance

5.3 Workloads

We use a group of nine workloads to evaluate our proposed parallel programming framework, and the applications are summarized in Table 1. We select workloads with varied parallelization methods. *MatMul*, *SpMV*, and *SpMatrixTranspose* are dense matrix multiplication, sparse matrix dense vector multiplication, and sparse matrix transpose, respectively. All three workloads are implemented in-house and leverage a single parallel loop. *PageRank* and *BFS* implement pull-based PageRank and pull/push hybrid breadth-first search with the the Ligra graph processing framework [52]. Both mainly use a pair of nested parallel loops: The outer loop iterates over vertices in the active vertex set while the inner loop iterates over a particular vertex's neighbors. Both *MatrixTranspose* and *CilkSort* mainly use recursive spawn-and-sync parallelization (i.e, `parallel_invoke()`). *MatrixTranspose* is dense matrix transpose and *CilkSort* performs parallel mergesort. Both do not have static baseline implementations as spawn-and-sync parallelization starts with a single task. Without a dynamic runtime, their execution is serialized on a single core. *NQueens* uses bactracking to solve the N-queens problem. It is parallelized over the potential positions of the next queen to be placed on the board and contains recursive parallel loops. *UTS* is the Unbalanced Tree Search benchmark introduced by Olivier et al. [40], which contains recursive parallel loops to enumerate an unbalanced tree. Among these nine workloads, only *MatMul*, which allocates a 3 KB buffer, utilizes SPM in user code. We characterize these nine workloads into four categories (i.e., *static-balanced*, *static-unbalanced*, *dynamic-balanced*, and *dynamic-unbalanced*) by two metrics: (1) if the workload leverages dynamic parallelism; and (2) if the tasks have load imbalance (see Figure 8).

6 RESULTS

Table 1 summarizes the cycles and dynamic instruction counts of simulated configurations. Figure 9 shows speedup of workloads over a static runtime with stack in SPM. We plot *MatrixTranspose* and *CilkSort* separately in Figure 10, as they do not have static baselines. Comparing the left-most two bars in Figure 9, we can see that in the context of the static runtime, allocating the stack in SPM does not provide significant improvement over allocating the stack in DRAM, except in the case of *NQueens*. Workloads other than *NQueens* do not have frequent stack operations when running with the static runtime, and thus leaving the stack in DRAM does not

Table 1: Simulated Workloads – Cat = workload category; SB = static-balanced; SU = static-unbalanced; DB = dynamic-balanced; DU = dynamic-unbalanced; PM = parallelization methods; pf = `parallel_for`, npf = nested or recursive `parallel_for` and ss = recursive spawn and sync; Input = input dataset; DI = dynamic instruction count in millions; C = simulated cycles in thousands.

				Static Runtime				**Work-Stealing Runtime**							
				DRAM Stack		**SPM Stack**		**DRAM Stack DRAM Queue**		**DRAM Stack SPM Queue**		**SPM Stack DRAM Queue**		**SPM Stack SPM Queue**	
Cat	**Name**	**PM**	**Input**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**
SB	MatMul	pf	256	37	543	37	512	38	527	39	556	39	573	38	509
			512	289	6914	289	6579	293	5049	295	5333	294	5321	297	5260
SU	PageRank	npf	g14k16	11	1586	11	1685	23	1649	24	1451	23	1425	25	1343
			email	11	5679	11	5384	27	1786	29	1638	24	1471	28	1358
			c-58	15	5136	15	5136	32	2257	40	2257	33	2044	38	1961
SU	BFS	npf	g14k16	3	1114	3	1062	22	1149	27	1102	21	914	26	871
			bundle1	6	1988	6	2065	30	1881	40	1892	29	1604	39	1561
			c-58	7	1943	7	1881	27	1852	35	1806	26	1495	33	1440
SU	SpMV	pf	bundle1	4	1483	4	1476	6	1005	7	995	6	1007	8	978
			email	2	4144	2	4129	95	4046	132	3820	87	3657	142	4060
			c-58	3	3442	3	3444	10	1047	14	1012	11	1019	15	1009
SU	SpMatrix Transpose	pf	bundle1	42	50850	42	50718	183	12877	281	13409	189	12911	279	12992
			email	22	47310	22	47343	1112	45864	1569	44351	1112	45456	1622	45391
			c-58	24	16570	24	16655	91	7568	123	7325	89	7222	129	7177
DB	Matrix Transpose	ss	512	–	–	–	–	3	496	3	502	3	416	3	421
			1024	–	–	–	–	8	2238	9	2240	8	2031	8	1969
DU	CilkSort	ss	16384	–	–	–	–	7	304	9	279	6	264	8	253
			131072	–	–	–	–	30	1799	31	1658	29	1305	32	1264
DU	NQueens	npf	8	4	1094	4	513	8	545	9	546	8	140	8	151
			9	19	5371	19	2522	36	2478	37	2508	37	910	37	1026
			10	100	24820	100	11691	177	11089	182	11381	181	6695	181	7367
DU	UTS	npf	small-t1	11	90684	11	90228	53	3266	71	3236	55	3280	71	3156
			small-t3	13	127199	13	126594	468	21028	663	21209	480	20878	680	20770

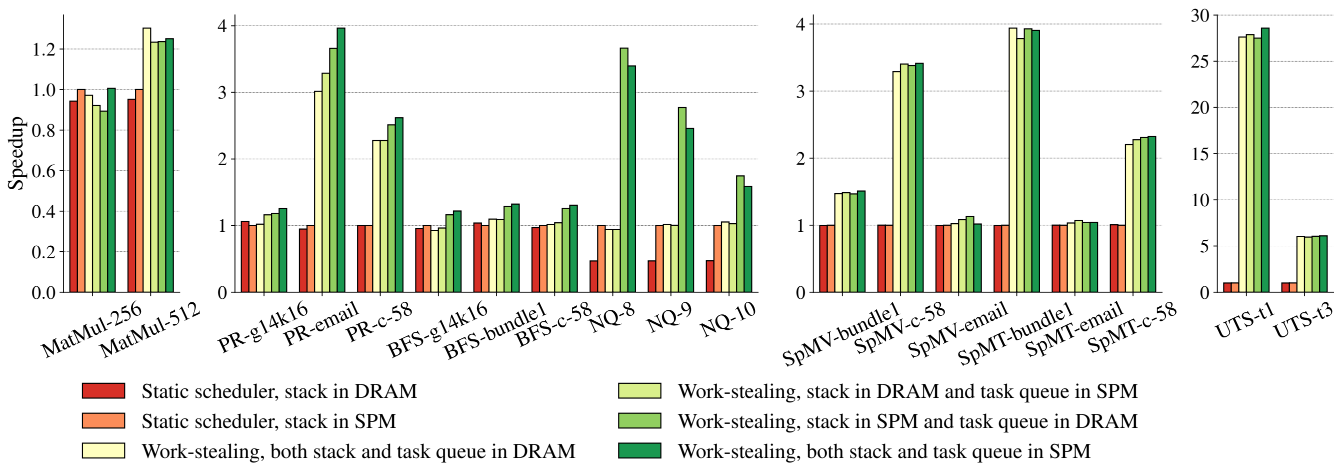

Figure 9: Work-stealing runtime provides a speedup between 1.2 - 28× and a slowdown of no more than 10% – PR = PageRank, NQ = NQueens, SpMT = SpMatrixTranspose. Applying data-placement optimizations to leverage the SPM provides an additional benefit of as much as 25% and compensates for any slowdown observed from work-stealing overhead.

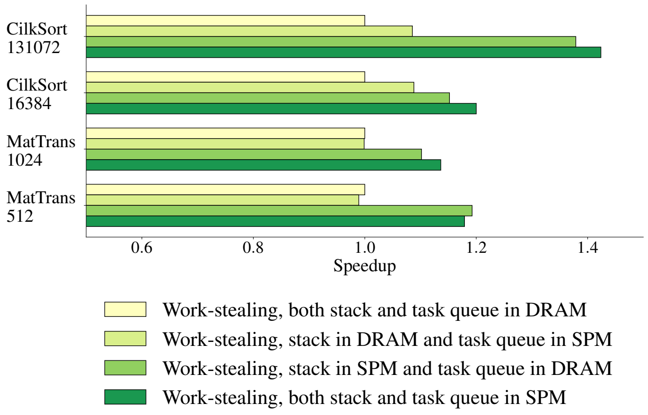

Figure 10: Performance of CilkSort and MatrixTranspose – normalized to having both stack and task queue in SPM; MatTrans = MatrixTranspose. Note that the X-axis starts at 0.5.

incur significant overheads. *NQueens* has heavy reads and writes to the stack as it frequently copies stack allocated arrays. Allocating the stack in DRAM leads to severe performance degradation.

Comparing the static scheduler that places stack in SPM to our baseline work-stealing runtime that has both the stack and the task queue in DRAM, we can observe that we either only incur minimal overheads over a traditional static runtime (e.g., in the cases of *MatMul-256* and *NQueens-8*) or achieve non-trivial performance improvement (e.g., *PR-email* and *UTS-t1* are able to achieve 3× and 25× better performance, respectively). This demonstrates the benefit of running irregular workloads with a work-stealing runtime on manycores. As expected, *PageRank*, *SpMV*, and *SpMatrixTranspose* show input dependent behavior and achieve different speedups on different inputs (e.g., *PageRank* shows only moderate speedup on the synthetic graph *g14k16*, but achieves 3× speedup on real-world graph *email*). *MatMul* with 512 × 512 input matrices shows an unexpected 25% performance improvement over the static baseline. This is because while there is no inherent load imbalance in our tiled implementation, cores experience non-uniform memory latency due to their locations in the 2-D mesh OCN. Dynamic load-balancing helps mitigate this difference by scheduling more compute to cores with lower memory latency.

Different workloads show varied benefit from our optimization techniques that leverage the SPM space not claimed by the programmer. *PageRank* is able to benefit from both optimizations and achieves best performance when both the stack and the task queue are in SPM. *BFS* can only outperform the static baseline with optimizations enabled, and SPM-allocated stack has a higher impact on *BFS* than SPM-allocated task queue. *NQueens* utilizes the stack heavily and achieves the best performance when the SPM is reserved solely for the stack. In this configuration, fewer stack frames are overflowed to DRAM. We also observe that as the input size increases from 8 to 10, more moderate speedup is achieved by our work-stealing runtime compared to the static baseline. This is because larger inputs incur deeper stacks and thus more stack frame overflows to DRAM, *NQueens* becomes more DRAM bandwidth bound. *MatrixTranspose* and *CilkSort* are also able to benefit from

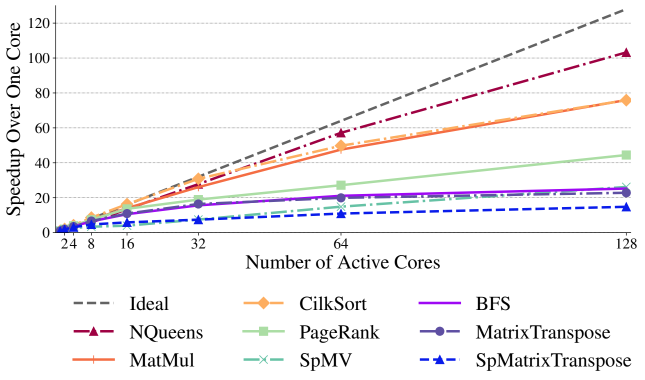

Figure 11: Workload Scaling – inputs: MatMul = 256; PageRank = g14k16; MatrixTranspose = 512; NQueens = 8; BFS = g18k8; CilkSort = 131072; SpMV = u16k32; SpMatrixTranspose = c-58. Data collected on work-stealing runtime with both task and task queue in SPM.

having the stack in SPM (see Figure 10). *SpMV*, *SpMatrixTranspose*, and *UTS* do not have either frequent stack or frequent task queue operations. Moreover, both *SpMV* and *SpMatrixTranspose* are already DRAM bandwidth bounded. Extra traffic to DRAM incurred by allocating both stack and task queue in DRAM has only insignificant impact. As a result, our optimizations do not yield better performance on these three workloads.

Across all workloads, we observe an increase in the number of dynamic instructions on work-stealing runtimes vs. on static runtimes (see Table 1). This is expected as it is well-known that work-stealing runtimes add overheads from various sources (e.g., task creation and scheduling), especially when working with very fine-grained tasks. We also observe an increase in the number of dynamic instructions when the SPM-allocated task queue optimization is enabled. This is because with reduced task queue access latency, cores can perform stealing attempts faster and fail more when there is no task to steal. These instructions are executed by idle cores that cannot find ready tasks and they are not part of the critical path.

We also conduct a scalability study with all workloads except *UTS*. We did not include *UTS* due to its extensively long simulation time. Results are shown in Figure 11. *NQueens* scales the best since, with more cores, more stack allocated data can be kept in SPM. *CilkSort*, as the name suggests, is an algorithm well suited to a dynamic task parallel runtime and is also well balanced, minimizing the overhead from stealing. *MatMul* is another balanced workload that scales well; it has high arithmetic intensity and loads from DRAM infrequently. *MatrixTranspose* is memory intensive and its scalability is limited by memory bandwidth. *BFS*, *PageRank*, *SpMV*, and *SpMatrixTranspose* are similarly bounded by memory, and in addition they can suffer from severe imbalance. While our runtime is a major boon to these workloads (static scheduling fairs much worse), task stealing becomes more frequent on unbalanced inputs as the core count increases.

To summarize, the proposed work-stealing runtime: (1) either improves performance of static-balanced workloads by migrating tasks away from cores that have long memory latency or induces only minimal overheads; (2) improves performance of irregular workloads which show input dependent behavior when there is input induced load imbalance; (3) efficiently supports dynamic-balanced and dynamic-unbalanced workloads to achieve high performance, and (4) provides high scalability. Our proposed optimization techniques which automatically leverage SPM are able to improve performance of applications that have frequent stack and/or frequent task queue operations (i.e., *NQueens*, *MatrixTranspose*, *PageRank*, and *BFS*) and incur only minimal overheads on workloads that cannot benefit from them.

7 RELATED WORK

Early manycore research prototypes integrated 16–110 cores on a single die [22, 23, 35, 39, 54–56]. The industry has adopted the manycore approach as well and products available typically include 64–256 cores [6, 21, 27, 28, 32, 46, 59, 60]. Recent research prototypes have scaled core counts by an order-of-magnitude to over a thousand cores (e.g., 1000-core KiloCore [11], 1024-core Epiphany-V [41], and 4096-core Manticore [62])

A number of prior works explored work-stealing runtimes on manycore architectures that provide software-centric cache coherence. Long et al. [36] implemented a Cilk-like runtime on a 64-core manycore architecture with a shared L2 cache and non-coherent private L1 caches. They attacked the shared data coherence issue by leveraging a bloom filter based hardware mechanism, Coherence Vector, to identify memory locations that should not be cached in non-coherent private L1 caches. The proposed runtime stores all runtime-related shared data (e.g., task queues) into the Coherence Vector. For user data with parent-child dependency, they exploit the DAG-consistency [7] and insert L1 invalidate and write-back instructions in the runtime. Similarly, Wang et al. [58] worked on a similar system (i.e., big.TINY) and also proposed inserting L1 cache invalidation and write-back instructions at proper locations in their Cilk-like runtime. Unlike Long et al. who identified runtime shared data as non-cachable locations, Wang et al. proposed to leverage the same self-invalidation and self-flush mechanism for keeping runtime shared data coherent. For example, after locking a task queue, a core performs a L1 cache invalidation to avoid reading stale data when accessing the task queue. To mitigate the frequent L1 cache invalidation and write-back induced by task queue operations, Wang et al. proposed a hardware-based mechanism, direct task stealing, which makes task queue a private data structure. Stealing is made possible by having the thief send a user-level interrupt to the victim. The victim then pops a task from its task queue on behalf of the thief. Tagliavini et al. [53] implemented an OpenMP runtime on a manycore architecture that has non-coherent private L1 caches. Similar to both works mentioned above, the private L1 caches need to be self-invalidated and self-flushed at proper time to maintain coherence. Unlike the two Cilk-like runtimes that have per thread task queues, their proposal leverages a centralized task queue. All three works studied manycore architectures with software-centric cache coherence, while our work targets architectures that have only software-managed scratchpads. Orr et al. [43] implemented a Cilk-like work-stealing runtime on GPGPUs with software-centric caches.

Although not a manycore, the Cray T3D/E architectures [4, 16] bear similarities to HammerBlade. Both are global shared memory architectures capable of parallel work-sharing programming models. A notable difference is that the Cray machines' notion of local memories pertains to abundant-but-slow DRAM, as opposed to HammerBlade's local memories being fast-but-scarce SRAM. Nonetheless, we believe that techniques from this work could be applied to these Cray machines.

Zakkak et al. [61] proposed an implementation of the Java virtual machine on a SPM manycore and adopted work-dealing instead of work-stealing. Our work, to the best of our knowledge, describes the first implementation of a Cilk-like work-stealing runtime for manycore architectures with only software-managed SPM. Alvarez et al. [3] described a task-based parallel runtime which can transparently use the SPM for holding input and output data in a hybrid memory hierarchy. Prior work also studied work-stealing runtimes on PGAS or distributed memory clusters, including [19, 45, 50]. Li et al. [34] studied efficient implementations of conditional division on manycore architectures. Their work focused on improving the work scheduling efficiency on top of an existing work-stealing runtime and is orthogonal to ours. Chen et al. [14] and Margerm et al. [37] explored generating task parallel accelerators with coherent caches. Our work can be applied to support accelerators with SPMs.

8 CONCLUSION

We demonstrate that, in contrast to conventional wisdom, a work-stealing runtime is viable and beneficial on manycore architectures with only software-managed scratchpad memories. This work provides programmers a familiar programming model for efficient software development on manycore architectures like HammerBlade, and achieves significant performance improvements over traditional programming models such as statically scheduled parallel loops (i.e., up to 3.94× speedup for workloads that can be statically scheduled and up to 28.5× speedup for workloads with dynamic parallelism). This work is a small yet important step towards solving the manycore architecture programmability challenge. While we evaluated our work-stealing runtime on HammerBlade, our techniques are applicable to other PGAS manycore architectures that have software-managed scratchpads memories.

ACKNOWLEDGMENTS

The HammerBlade architecture development was funded exclusively by DARPA SDH Award #FA8650-18-2-7863. The dynamic task-task parallel programming framework and work-stealing runtime was supported in part by NSF PPoSS Award #2118709 and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, as well as equipment donations from Intel. The authors acknowledge and thank Moyang Wang and Tuan Ta for useful discussions on implementing state-of-the-art work-stealing runtimes. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and DARPA or the U.S. Government.

REFERENCES

[1] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Ian Galton, Rajesh K. Gupta, Patrick P. Mercier, Mani Srivastava, Michael B. Taylor, and Zhiru Zhang. 2017. Celerity: An Open-Source RISC-V Tiered Accelerator Fabric. *Symp. on High Performance Chips (Hot Chips)* (Aug 2017).

[2] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Anuj Rao, Austin Rovinski, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Rajesh K. Gupta, Michael B. Taylor, and Zhiru Zhang. 2017. Experiences Using the RISC-V Ecosystem to Design an Accelerator-Centric SoC in TSMC 16nm. *Workshop on Computer Architecture Research with RISC-V (CARRV)* (Oct 2017).

[3] Lluc Alvarez, Miquel Moretó, Marc Casas, Emilio Castillo, Xavier Martorell, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. 2015. Runtime-Guided Management of Scratchpad Memories in Multicore Architectures. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)* (Oct 2015). https://doi.org/10.1109/PACT.2015.26

[4] E. Anderson, J. Brooks, C. Grassl, and S. Scott. 1997. Performance of the CRAY T3E Multiprocessor. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)* (Nov 1997), 39–39. https://doi.org/10.1145/509593.509632

[5] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *IEEE Trans. on Parallel and Distributed Systems (TPDS)* 20, 3 (Mar 2009), 404–418. https://doi.org/10.1109/TPDS.2008.105

[6] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, Dave Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. 2008. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. *Int'l Solid-State Circuits Conf. (ISSCC)* (Feb 2008). https://doi.org/10.1109/ISSCC.2008.4523070

[7] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. 1996. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. *Symp. on Parallel Algorithms and Architectures (SPAA)* (Jun 1996). https://doi.org/10.1145/237502.237574

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. *Symp. on Principles and Practice of Parallel Programming (PPoPP)* (Jul 1995). https://doi.org/10.1145/209937.209958

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (Aug 1996), 55–69.

[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sep 1999), 720–748. https://doi.org/10.1145/324133.324234

[11] Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M. Baas. 2017. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits (JSSC)* 52, 4 (Apr 2017), 891–902. https://doi.org/10.1109/JSSC.2016.2638459

[12] Ajay Brahmakshatriya, Emily Furst, Victor Ying, Claire Hsu, Changwan Hong, Max Ruttenberg, Yunming Zhang, Dai Cheol Jung, Dustin Richmond, Michael Taylor, Julian Shun, Mark Oskin, Daniel Sanchez, and Saman Amarasinghe. 2021. Taming the Zoo: The Unified GraphIt Compiler Framework for Novel Architectures. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2021). https://doi.org/10.1109/ISCA52012.2021.00041

[13] P. Charles, C. Grothoff, V. Sarkar, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)* (Oct 2005). https://doi.org/10.1145/1103845.1094852

[14] Tao Chen, Shreesha Srinath, Christopher Batten, and Edward Suh. 2018. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. *Int'l Symp. on Microarchitecture (MICRO)* (Oct 2018). https://doi.org/10.1109/MICRO.2018.00014

[15] Lin Cheng, Peitian Pan, Zhongyuan Zhao, Krithik Ranjan, Jack Weber, Bandhav Veluri, Seyed Borna Ehsani, Max Ruttenberg, Dai Cheol Jung, Preslav Ivanov, Dustin Richmond, Michael B. Taylor, Zhiru Zhang, and Christopher Batten. 2022. A Tensor Processing Framework for CPU-Manycore Heterogeneous Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 6 (2022), 1620–1635. https://doi.org/10.1109/TCAD.2021.3103825

[16] Cray Research, Inc 1993. *CRAY T3D System Architecture Overview.* Cray Research, Inc. http://www.bitsavers.org/pdf/cray/HR-04033_CRAY_T3D_System_Architecture_Overview_Sep93.pdf

[17] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. 2012. CPU DB: Recording Microprocessor History. *ACM Queue* (Apr 2012), 10–27.

[18] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawaj, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald G. Dreslinski, Christopher Batten, and Michael B. Taylor. 2018. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 38, 2 (Mar/Apr 2018), 30–41. https://doi.org/10.1109/MM.2018.022071133

[19] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable Work Stealing. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)* (Nov 2009). https://doi.org/10.1145/1654059.1654113

[20] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Jun 1998). https://doi.org/10.1145/277652.277725

[21] Tom R. Halfhill. 2020. ThunderX3's Cloudburst of Threads: Marvell Previews 96-core 384-thread Arm Server Processor. *Microprocessor Report, The Linley Group* (Apr 2020).

[22] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. 2007. A 5-GHz Mesh Interconnect for a Teraflops Processor. *IEEE Micro* 27, 5 (Sep/Oct 2007), 51–61. https://doi.org/10.1109/MM.2007.4378783

[23] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. 2010. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. *Int'l Solid-State Circuits Conf. (ISSCC)* (Feb 2010). https://doi.org/10.1109/ISSCC.2010.5434077

[24] Intel Corporation 2012. *Intel Cilk Plus Language Extension Specification.* Intel Corporation. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1665.htm

[25] Intel Corporation 2019. *Intel Threading Building Blocks.* Intel Corporation. https://software.intel.com/en-us/intel-tbb

[26] Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. 2020. Ruche Networks: Wire-Maximal, No-Fuss NoCs : Special Session Paper. *Int'l Symp. on Networks-on-Chip (NOCS)* (Apr 2020). https://doi.org/10.1109/NOCS50636.2020.9241586

[27] Kalray 2022 (accessed Aug 2022). Kalray MPPA Products. Online Webpage. https://www.kalrayinc.com/products/mppa-technology/.

[28] David Kanter. 2015. Knights Landing Reshapes HPC.

[29] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. 2009. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2009). https://doi.org/10.1145/1555754.1555774

[30] Khronos Working Group 2011. *OpenCL Specification, v1.2.* Khronos Working Group. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[31] Charles E. Leiserson. 2009. The Cilk++ Concurrency Platform. *Design Automation Conf. (DAC)* (Jul 2009). https://doi.org/10.1145/1629911.1630048

[32] L. Li, J. Fang, H. Fu, J. Jiang, W. Zhao, C. He, X. You, and G. Yang. 2018. swCaffe: A Parallel Framework for Accelerating Deep Learning Applications on Sunway TaihuLight. *Int'l Conf. on Cluster Computing* (Sep 2018). https://doi.org/10.48550/arXiv.1903.06934

[33] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *Computer Architecture Letters (CAL)* (Jul 2020). https://doi.org/10.1109/LCA.2020.2973991

[34] Zheng Li, Jose Duato, Olivier Certner, and Olivier Temam. 2010. Scalable Hardware Support for Conditional Parallelization. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)* (Sep 2010).

[35] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Ilia Lebedev, and Srinivas Devadas. 2013. *Hardware-Level Thread Migration in a 110-Core Shared-Memory Multiprocessor.* Technical Report 512. MIT CSAIL CSG.

[36] Guo-Ping Long, Jun-Chao Zhang, and Dong-Rui Fan. 2008. Architectural Support and Evaluation of Cilk Language on Many-Core Architectures. *Chinese Journal of Computers* 31, 11 (2008), 1975–1985. https://doi.org/10.3724/SP.J.1016.2008.01975

[37] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. TAPAS: Generating Parallel Accelerators from Parallel Programs. *Int'l Symp. on Microarchitecture (MICRO)* (Oct 2018). https://doi.org/10.1109/MICRO.2018.00028

[38] Michael McCool, Arch D. Robinson, and James Reinders. 2012. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann.

[39] Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Jonathan Balkind, Alexey Lavrov, Mohammad Shahrad, Samuel Payne, and David Wentzlaff. 2017. Piton: A Manycore Processor for Multitenant Clouds. *IEEE Micro* 37, 2 (Mar/Apr 2017), 70–80. https://doi.org/10.1109/MM.2017.36

[40] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2006. UTS: An Unbalanced Tree Search Benchmark. *Int'l*

Workshop on Lanaguages and Compilers for Parallel Computing (LCPC) (Nov 2006). https://doi.org/10.1007/978-3-540-72521-3_18

[41] Andreas Olofsson. 2016. Epiphany-V: A 1024-processor 64-bit RISC System-On-Chip. *Computing Research Repository (CoRR)* arXiv:abs/1610.01832 (Aug 2016). https://doi.org/10.48550/arXiv.1610.01832

[42] OpenMP Architecture Review Board 2013. *OpenMP Application Program Interface, Version 4.0.* OpenMP Architecture Review Board. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[43] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. 2014. Fine-Grain Task Aggregation and Coordination on GPUs. *Int'l Symp. on Computer Architecture (ISCA)* (Jul 2014). https://doi.org/10.1109/ISCA.2014.6853209

[44] Yanghui Ou, Shady Agwa, and Christopher Batten. 2020. Implementing Low-Diameter On-Chip Networks for Manycore Processors Using a Tiled Physical Design Methodology. *Int'l Symp. on Networks-on-Chip (NOCS)* (Sep 2020). https://doi.org/10.1109/NOCS50636.2020.9241710

[45] Guilherme P. Pezzi, Marcia C. Cera, Elton Mathias, Nicolas Maillard, and Philippe O. A. Navaux. 2007. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. *Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)* (Oct 2007). https://doi.org/10.1109/SBAC-PAD.2007.36

[46] Carl Ramey. 2011. TILE-Gx100 ManyCore Processor: Acceleration Interfaces and Architecture. *Symp. on High Performance Chips (Hot Chips)* (Aug 2011). https://doi.org/10.1109/HOTCHIPS.2011.7477491

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

[48] Austin Rovinski, Chun Zhao, Khalid Al-Hawaj, Paul Gao, Shaolin Xie, Christopher Torng, Scott Davidson, Aporva Amarnath, Luis Vega, Bandhav Veluri, Anuj Rao, Tutu Ajayi, Julian Puscar, Steve Dai, Ritchie Zhao, Dustin Richmond, Zhiru Zhang, Ian Galton, Christopher Batten, Michael B. Taylor, and Ron G. Dreslinski. 2019. A 1.4 GHz 695 Giga RISC-V Inst/s 496-core Manycore Processor with Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS. *Symp. on VLSI Technology and Circuits (VLSI)* (Jun 2019). https://doi.org/10.23919/VLSIC.2019.8778031

[49] Austin Rovinski, Chun Zhao, Khalid Al-Hawaj, Paul Gao, Shaolin Xie, Christopher Torng, Scott Davidson, Aporva Amarnath, Luis Vega, Bandhav Veluri, Anuj Rao, Tutu Ajayi, Julian Puscar, Steve Dai, Ritchie Zhao, Dustin Richmond, Zhiru Zhang, Ian Galton, Christopher Batten, Michael B. Taylor, and Ron G. Dreslinski. 2019. Evaluating Celerity: A 16nm 695 Giga-RISC-V Instructions/s Manycore Processor with Synthesizable PLL. *IEEE Solid-State Circuits Letters (SSCL)* 2, 12 (Dec 2019), 289–292. https://doi.org/10.1109/LSSC.2019.2953847

[50] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. 2011. Lifeline-Based Global Load Balancing. *SIGPLAN Not.* (feb 2011), 201–212. https://doi.org/10.1145/2038037.1941582

[51] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Interemdiate Representation. *Symp. on Principles and Practice of Parallel Programming (PPoPP)* (Feb 2017). https://doi.org/10.1145/3155284.3018758

[52] Julian Shun and Guy Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Symp. on Principles and Practice of Parallel Programming (PPoPP)* (Feb 2013). https://doi.org/10.1145/2517327.2442530

[53] Giuseppe Tagliavini, Daniele Cesarini, and Andrea Marongiu. 2018. Unleashing Fine-Grained Parallelism on Embedded Many-Core Accelerators with Lightweight OpenMP Tasking. *IEEE Transactions on Parallel and Distributed Systems* 29, 9 (2018), 2150–2163. https://doi.org/10.1109/TPDS.2018.2814602

[54] Guangming Tan, Dongrui Fan, Junchao Zhang, Andrew Russo, and Guang R. Gao. 2008. Experience on Optimizing Irregular Computation for Memory Hierarchy in Manycore Architecture. *Symp. on Principles and Practice of Parallel Programming (PPoPP)* (Feb 2008). https://doi.org/10.1145/1345206.1345255

[55] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. 2003. A 16-Issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network. *Int'l Solid-State Circuits Conf. (ISSCC)* (Feb 2003). https://doi.org/10.1109/ISSCC.2003.1234253

[56] Pascal Vivet, Eric Guthmuller, Yvain Thonnart, Gael Pillonnet, Guillaume Moritz, Ivan Miro-Panadès, Cesar Fuguet, Jean Durupt, Christian Bernard, Didier Varreau, Julian Pontes, Sebastien Thuries, David Coriat, Michel Harrand, Denis Dutoit, Didier Lattard, Lucile Arnaud, Jean Charbonnier, Perceval Coudrain, Arnaud Garnier, Frederic Berger, Alain Gueugnot, Alain Greiner, Quentin Meunier, Alexis Farcy, Alexandre Arriordaz, Severine Cheramy, and Fabien Clermidy. 2020. A 220GOPS 96-Core Processor with 6 Chiplets 3D-Stacked on an Active Interposer Offering 0.6ns/mm Latency, 3Tb/s/mm2 Inter-Chiplet Interconnects and 156mW/mm2@ 82%-Peak-Efficiency DC-DC Converters. *Int'l Solid-State Circuits Conf. (ISSCC)* (Feb 2020). https://doi.org/10.1109/ISSCC19947.2020.9062927

[57] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: Scalable Threads for Internet Services. *Symp. on Operating Systems Principles (SOSP)* (Oct 2003), 268–281. https://doi.org/10.1145/945445.945471

[58] Moyang Wang, Tuan Ta, Lin Cheng, and Christopher Batten. 2020. Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2020). https://doi.org/10.1109/ISCA45697.2020.00025

[59] David Wentzlaff, Patrick Griffin, Henry Hoffman, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. 2007. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro* 27 (Sep/Oct 2007), 15–31. Issue 5. https://doi.org/10.1109/MM.2007.4378780

[60] Bob Wheeler. 2020. Ampere Maxes Out at 128 Cores. *Microprocessor Report, The Linley Group* (Jul 2020).

[61] Foivos S. Zakkak and Polyvios Pratikakis. 2016. Building a Java™ Virtual Machine for Non-Cache-Coherent Many-Core Architectures. *Int'l Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)* (Aug 2016). https://doi.org/10.1145/2990509.2990510

[62] Florian Zaruba, Fabian Schuiki, and Luca Benini. 2021. Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing. *IEEE Micro* (Mar/Apr 2021). https://doi.org/10.48550/arXiv.2008.06502

Received 2022-10-20; accepted 2023-01-19