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.

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

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.
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 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 $1.2\times$–$28.5\times$ 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-rutching 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 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 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 solving 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.
We use an application programming interface (API) similar to In

We also give a description of a higher-level API for expressing

The parent task then calculates

We also give a description of a higher-level API for expressing

The key challenge of implementing a work-stealing runtime on a

We also give a description of a higher-level API for expressing

3.1 Running Example

We use an application programming interface (API) similar to In
tell 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 meta-
data variable ready_count, also known as the reference counter.
This metadata tracks a task’s unfinished children. After a task fin-
ishes 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() fun-
tion to hold the logic of the concrete task. In this example (Fig-
ure 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 tem-
plated functions that support various parallel patterns. This in-
cludes parallel_invoke() for divide-and-conquer (Figure 3 (c)),
parallel_for() for parallel loops (Figure 3 (d)), and parallel_re-
duce() for parallel reduction (Figure 3 (e)).
we introduce three optimizations which enable work-stealing run-
workloads that do not have any reuse of input data). The naive
workload is able to beneﬁt from leveraging them (e.g., streaming
All cores access the same copy. Note that the atomics used for
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 coheren. 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 SCRAPTHPAD ENHANCED RUNTIME
Prior work has shown that leveraging the scratchpad memory is crit-
tical 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-
opimal 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 run-
times 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 program-
ners for user code. Reserving scratchpad space on Mosaic is real-
ized 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 scratch-
pad. 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 thread’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 scratch-
pad 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 deter-
mining an ideal switching depth which can vary from workload
to workload, while stack spilling cannot be realized without imple-
menting 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 ex-
tended 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 over-
flow threshold (i.e., lowest address of the scratch space in scratchpad),
and the other for storing the pointer to the DRAM overﬂow buffer.
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 points at which an overflow stack region will be swapped-in when replacements are performed when a frame is popped off the stack. To achieve this, 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 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 code at compilation-time to handle overflow of stack frames. 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 points at which an overflow stack region will be swapped-in when replacements are performed when a frame is popped off the stack.

### 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 queue, 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 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)).

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 points at which an overflow stack region will be swapped-in when replacements are performed when a frame is popped off the stack. 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.

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 queue, 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 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 4: Work-Stealing Runtime Implementations](image)

(a) Runtime Data in DRAM

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

(b) Runtime Data in Scratchpad

```c
void task::spawn( task* t ) {
    spm_lock.lock_aq();
    spm_tq.enq(t);
    spm_lock.lock_rl();
}
```
We also separate the spin lock protecting the task queue from the pointer of the lock as in the case where the lock is a member of the stack or the task queue. After implementing the two optimizations described above, processing OCN [26,44]. Our work-stealing runtime selects victims randomly, 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.

### 4.3 Read-Only Data Duplication

After implementing the two optimizations described above, profiling data collected from one of the apps (i.e., PageRank) shows an unexpected pattern. Figure 5 shows a heatmap 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.

### 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 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 backtracking 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.

<table>
<thead>
<tr>
<th>Cat</th>
<th>Name</th>
<th>PM</th>
<th>Input</th>
<th>DRAM Stack</th>
<th>SPM Stack</th>
<th>DRAM Stack</th>
<th>SPM Stack</th>
<th>DRAM Stack</th>
<th>SPM Stack</th>
<th>DRAM Stack</th>
<th>SPM Stack</th>
<th>DRAM Stack</th>
<th>SPM Stack</th>
<th>DRAM Stack</th>
<th>SPM Stack</th>
<th>DRAM Stack</th>
<th>SPM Stack</th>
<th>DRAM Stack</th>
<th>SPM Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>MatMul</td>
<td>pf</td>
<td>256</td>
<td>37 543</td>
<td>37 512</td>
<td>38 527</td>
<td>39 556</td>
<td>39 573</td>
<td>38 509</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>512</td>
<td>289 6914</td>
<td>289 6579</td>
<td>293 5049</td>
<td>295 5333</td>
<td>294 5321</td>
<td>297 5260</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SU</td>
<td>PageRank</td>
<td>npf</td>
<td>g14k16</td>
<td>11 1586</td>
<td>11 1685</td>
<td>23 1649</td>
<td>24 1451</td>
<td>23 1425</td>
<td>25 1343</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>email</td>
<td>11 5679</td>
<td>11 5384</td>
<td>27 1786</td>
<td>29 1638</td>
<td>24 1471</td>
<td>28 1358</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c-58</td>
<td>15 5136</td>
<td>15 5136</td>
<td>32 2257</td>
<td>40 2257</td>
<td>33 2044</td>
<td>38 1961</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SU</td>
<td>BFS</td>
<td>npf</td>
<td>bundle1</td>
<td>4 1483</td>
<td>4 1476</td>
<td>6 1005</td>
<td>7 995</td>
<td>6 1007</td>
<td>8 978</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>email</td>
<td>2 4144</td>
<td>2 4129</td>
<td>95 4046</td>
<td>132 3820</td>
<td>87 3657</td>
<td>132 3657</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c-58</td>
<td>3 3442</td>
<td>3 3444</td>
<td>10 1047</td>
<td>14 1012</td>
<td>11 1019</td>
<td>15 1009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SU</td>
<td>SpMV</td>
<td>pf</td>
<td>bundle1</td>
<td>42 50850</td>
<td>42 50718</td>
<td>183 12877</td>
<td>281 13409</td>
<td>189 12911</td>
<td>279 12992</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>email</td>
<td>22 47310</td>
<td>22 47343</td>
<td>1112 45864</td>
<td>1569 44351</td>
<td>1112 45456</td>
<td>1622 45391</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c-58</td>
<td>24 16570</td>
<td>24 16655</td>
<td>91 7568</td>
<td>123 7325</td>
<td>89 7222</td>
<td>129 7177</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB</td>
<td>Matrix</td>
<td>ss</td>
<td></td>
<td>512</td>
<td>– – – –</td>
<td>3 496</td>
<td>3 502</td>
<td>3 416</td>
<td>3 421</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transpose</td>
<td>ss</td>
<td></td>
<td>1024</td>
<td>– – – –</td>
<td>8 2238</td>
<td>9 2240</td>
<td>8 2031</td>
<td>8 1969</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DU</td>
<td>CilkSort</td>
<td>ss</td>
<td></td>
<td>16384</td>
<td>– – – –</td>
<td>7 304</td>
<td>9 279</td>
<td>6 264</td>
<td>8 253</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>131072</td>
<td>– – – –</td>
<td>30 1799</td>
<td>31 1658</td>
<td>5 1305</td>
<td>32 1264</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DU</td>
<td>NQueens</td>
<td>npf</td>
<td>8</td>
<td>4 1094</td>
<td>4 513</td>
<td>8 545</td>
<td>9 546</td>
<td>8 140</td>
<td>8 151</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>19 5371</td>
<td>19 2522</td>
<td>36 2478</td>
<td>37 2508</td>
<td>37 910</td>
<td>37 1026</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>100 24820</td>
<td>100 11691</td>
<td>177 11089</td>
<td>182 11381</td>
<td>181 6695</td>
<td>181 7367</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DU</td>
<td>UTS</td>
<td>npf</td>
<td>small-t1</td>
<td>11 90684</td>
<td>11 90228</td>
<td>53 5266</td>
<td>71 3236</td>
<td>55 3280</td>
<td>71 3156</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>small-t3</td>
<td>13 127199</td>
<td>13 126594</td>
<td>468 21028</td>
<td>665 21209</td>
<td>480 20878</td>
<td>680 20770</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
incurs significant overheads. MatMul 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 3x and 25x 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 3x 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 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.

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.

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.
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., 1800-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-cacheable 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-scarse 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.
Beyond Static Parallel Loops: Supporting Dynamic Task Parallelism on Manycore Architectures... ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada


[56] Pascal Vivet, Eric Guthmüller, Yvain Thonnart, Gael Pillonnet, Guillaume Moritz, Ivan Miro-Parades, Cesar Fuguet, Jean Durupt, Christian Bernard, Didier Varreau, Julian Pontes, Sebastien Therioux, David Coriat, Michel Harrand, Denis Dutuit, Didier Lattard, Lucile Arnaud, Jean Charbonnier, Perceval Coudrain, Arnaud Garnier, Frederic Berger, Alain Gueugnot, Alain Grenier, Quentin Meunier, Alexis Farcy, Alexandre Arriordaz, Serenaie Cheramy, and Fabien Clermidy. 2020. A 225GOPS 96-Core Processor with 6 Chiplets 3D-Stacked on an Active Interposer Offering 0.6ns/mm Latency, 3TB/s/mm2 Inter-Chiplet Interconnects and 150W/mm2@ 87%-Peak-Efficiency DC-DC Converters. Int’l Solid-State Circuits Conf. (ISSCC) (Feb 2020). https://doi.org/10.1109/ISSCC.2020.9062927


Received 2022-10-20; accepted 2023-03-19