CircusTent: A Benchmark Suite for Atomic Memory Operations

Brody Williams
brody.williams@ttu.edu
Texas Tech University
Lubbock, Texas

John D. Leidel
jleidel@tactcomplabs.com
Tactical Computing Laboratories
Muenster, Texas

Xi Wang
xi.wang@ttu.edu
Texas Tech University
Lubbock, Texas

David Donofrio
ddonofrio@tactcomplabs.com
Tactical Computing Laboratories
San Francisco, California

Yong Chen
yong.chen@ttu.edu
Texas Tech University
Lubbock, Texas

ABSTRACT

A paradigm shift is currently taking place in the field of computer architecture. Consistent silicon-level processor improvements, relied upon in the past to drive the augmentation of system scalability, have stalled. As such, it is widely believed that future systems, wherein the design of hardware and software are more closely coupled, will need to leverage an increased degree of heterogeneity in order to realize further improvements.

Parallel processing and corresponding programming models, already ubiquitous to high performance computing, will play a crucial role in these systems. Consequently, it is critically important to understand the interaction between these components. However, the behavior of atomic operations and associated synchronization primitives, which already represent a bottleneck in current systems, is difficult to quantify.

Therefore, in this work, we introduce CircusTent, an open source, modular, and natively extensible benchmark suite for shared and distributed memory systems that is designed to measure the performance of a target architecture's memory subsystem with respect to atomic operations. Herein, we first detail the design of CircusTent, which includes eight different kernels designed to replicate common atomic memory access patterns using two atomic primitives. We then demonstrate the capabilities of CircusTent through an evaluation of fourteen different platforms using our OpenMP benchmark implementation. In short, we believe CircusTent will prove to be an invaluable tool for the design and prototyping of emerging architectures.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Computing methodologies → Parallel computing methodologies**; • **Hardware → Memory and dense storage**; • **General and reference → Performance**.

MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA

ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422789

KEYWORDS

Benchmark, Atomic Memory Operations, OpenMP, MPI, OpenSHMEM

ACM Reference Format:
Brody Williams, John D. Leidel, Xi Wang, David Donofrio, and Yong Chen. 2020. CircusTent: A Benchmark Suite for Atomic Memory Operations. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3422575.3422789

1 INTRODUCTION

The impending demise of both Moore's Law and Dennard Scaling has produced a renaissance in the field of computer architecture. Unable to continue leveraging silicon-level processor improvements to further enhance performance and scalability, system architects have been forced to explore other options. In this new era of heterogeneous architectures and hardware/software codesign, parallel processing and associated programming paradigms have become of paramount importance. Alongside an increased prominence in traditional high performance computing (HPC) environments in academia, government laboratories, and industry, parallel processing has also become imperative for consumer-level devices. As such, a better understanding of the behavior of parallel applications, and their interaction with the architectures they run on, is highly desirable in order to ensure optimal performance.

One well understood characteristic of parallel applications is that they frequently suffer from scalability issues as the number of cooperating processing elements (PEs) grows. Often, this performance degradation can be directly attributed to overheads associated with the synchronization of active PEs. Consequently, well written applications utilize synchronization primitives, such as barriers and mutual exclusion locks, as infrequently as possible. Regrettably, the nature of parallel applications typically precludes the removal of all such synchronization methods as they are necessary to prevent race conditions associated with access to shared data structures and ensure program correctness. Given the impact synchronization plays on the scalability and performance of parallel applications, understanding and optimization of these routines is key.

In many cases, the synchronization primitives described above are constructed using atomic memory operations. Abstractly, an atomic memory operation can be defined as an operation that is uninterruptible. Although an atomic operation may be composed

of several smaller "sub-operations" that would, under other circumstances, necessitate the execution of multiple instructions to complete, herein these sub-operations are treated as a single, cohesive unit. In this work, we focus on a particularly prominent class of atomic memory operations known as read-modify-write (RMW) atomics. Most modern architectures, including x86, RISC-V, and those produced by ARM, include ISA-level RMW atomic instructions and provide microarchitectural support for realization of their execution. As their designation would suggest, RMW atomics load a single value of a given data type into a specified register, modify said value, and finally write it back to memory in one unified step. In this manner, RMW atomic operations can be utilized to safely set variables underlying more complex synchronization primitives, such as barriers and locks, in parallel environments.

Further, RMW atomics also constitute a particularly efficient and fine-grained synchronization primitive in and of themselves. In many cases, application developers can replace mutex lock encapsulated critical code segments with "lock-free" atomic based synchronization. Doing so allows parallel execution to continue to the greatest degree possible and often significantly improves application performance. Many graph-based computational kernels, wherein only access to shared vertex and/or edge data need be coordinated, employ such an atomic-based synchronization scheme. In our previous work, we examined the GAP Benchmark Suite [4], which is designed to replicate the memory access patterns of graph workloads in a shared memory environment, in order to quantify the proportion of atomic operations. Therein, we found that, across all six included kernels, an average of 17.46% of the total instructions were RMW atomic operation instructions [25]. In this and similar scenarios, the frequency of atomic memory operations results in contention that, while less significant than lock-based synchronization, has measurable effects on application performance.

A variety of different shared and distributed memory parallel programming paradigms exist today in order to provide efficient scaling both within a single node, and across distinct nodes, respectively. Unsurprisingly, variants of high-level synchronization constructs and atomic memory operations are critical for physically shared memory paradigms as well as distributed memory paradigms that utilize a shared memory abstraction. For models designed specifically for physically shared memory systems, such as OpenMP or Pthreads, high-level synchronization primitives and API level atomic operations can be realized through simple wrappers around the appropriate ISA-level atomic instructions. However, for distributed shared memory parallel programming models, the implementation of "remote atomics" and associated synchronization primitives is more complex. In these models, access to a node's local shared memory must be coordinated not only between local PEs, but also among those located across a network fabric. Therefore, these remote atomic operations are typically built upon some combination of RDMA verbs, local barrier synchronizations, and ISA-level atomic instructions [7][15]. Utilizing a node's network interface card to perform the local atomic operations can further optimize system performance in these models [25]. The prevalence of distributed memory environments in high performance computing, in conjunction with inherent limits to multicore scaling [9], illustrate the need to understand the behavior of these remote atomic operations when designing future systems.

In this work, we propose CircusTent, an open source, modular, and natively extensible benchmark suite for atomic operations. Designed to replicate common atomic memory access patterns in both shared and distributed memory environments, CircusTent provides system architects insight into the performance and scalability of a target architecture's memory hierarchy. As such, we believe CircusTent will serve as an invaluable tool for the design and prototyping of future systems [1].

The remainder of this work is organized as follows. Section 2 discusses relevant previous works pertaining to synchronization, atomic memory operations, and memory subsystem performance. Section 3 details the design of the CircusTent benchmark suite and its constituent kernels. Section 4 presents an evaluation of the CircusTent benchmark suite conducted using a variety of different architectures and the OpenMP programming model. Section 5 reports our final analysis and conclusions. Finally, Section 6 provides a brief overview of planned future work.

2 PREVIOUS WORK

2.1 Atomic Operations & Synchronization

Several previous studies have examined the behavior and performance characteristics of atomic operations and synchronization primitives. Villa et al. explored the efficiency and scalabilty of barrier based synchronization on manycore architectures utilizing intraprocessor interconnects modeled after the design of network-on-chip paradigms [24]. In this study, the authors evaluated four different barrier algorithms, implemented in both hardware and software, using a cycle-accurate simulator. Trials conducted using up to 128 cores, arranged in five different topologies, revealed that, at least for similar configurations, hardware based barrier implementations exhibit better performance for intraprocessor synchronization.

In a similar study, David et al. conducted an extensive investigation of synchronization that spanned multiple hardware and software methodologies [8]. Based on the results of their evaluation, the authors of this work made several important observations. First, they note that, regardless of the implementation, synchronization across sockets is far more expensive than intrasocket synchronization. In fact, even in the absence of contention, the authors found that the latency of operations performed on cache lines across sockets increased 2-7.5x as compared to their intrasocket analogs. Second, they observe that the organization and behavior of the of the last-level cache (LLC) plays an important role in synchronization scalability within a socket. Finally, they perceive that, as the number of threads contending for access to shared data grows, message-passing mechanisms often outperform locking schemes. Overall, for physically shared memory systems such as those utilized in this work, the authors conclude that the scalability of synchronization is directly correlated to the system's architecture.

In [21], Schweizer et al. develop a methodology for analyzing the latency and bandwidth of atomic operations. In particular, they study the effects of different cache coherency states and complex memory hierarchies on these operations. Using their proposed methodology, they then evaluate three different RMW atomic operations on a number of x86 platforms. As part of this evaluation, they show that, contrary to popular belief, all of the tested atomic

operations exhibit comparable performance in terms of latency and bandwidth. Further, they find that, even in the absence of dependencies between instructions, the design of atomic operations on the tested platforms inherently prevents instruction-level parallelism.

Hoseini et al. also study the properties of atomic operations in physically shared memory systems [14]. For their investigation, they monitor accesses to shared cache lines in conditions that simulate both high and low levels of contention. Notably, for the high contention environment, wherein requests to a single cache line become serialized, they examine the scheduling of thread accesses. For one platform utilizing a Xeon-E5 processor, they were unable to discern any deterministic scheduling pattern. However, for the other platform featuring a Knights Landing Xeon Phi processor, they observe that threads pinned to cores coresident on the same tile were always scheduled sequentially. This is logical as this processor employs L2 caches that are shared between cores on the same tile, but does not include an LLC shared between all cores.

Summary: Although each of the studies detailed above make important contributions and advance our understanding of atomic operations and synchronization primitives, they do so only in a limited number of carefully controlled scenarios. As such, the behaviors and performance characteristics they observe may not be fully generalizable to varied architectures and applications. Furthermore, the specialized approaches they employ are applicable only for physically shared memory systems and not easily replicated. In contrast, CircusTent provides an infrastructure for benchmarking the performance of atomic operations that is built directly on conventional shared and distributed shared memory parallel programming models. In this manner, CircusTent offers an easily accessible, uniform means of measuring memory subsystem performance in common application environments across diverse platforms.

2.2 Transactional Memory

Transactional memory (TM) represents an orthogonal approach to synchronization that has become increasingly prominent in recent years. Similar to the use of atomic operations, this paradigm seeks to avoid the performance overheads associated with lock-based synchronization. However, it targets concurrent accesses to shared data at a more coarse-grained level. Rather than operating on a single value, as with an atomic operation, this approach encapsulates a series of instructions into a "transaction". In many ways, these transactions correspond to critical code segments traditionally protected using lock-based synchronization. In contrast, however, simultaneous accesses to the data underlying a transaction by distinct processing elements are not inherently prevented. Instead, operations on this data are speculatively performed and tracked across processing elements. In the absence of a conflict, changes to the data are committed. If a conflict is detected, these changes are discarded instead. In this manner, transactional memory represents an optimistic approach to synchronized data access in a parallel environment.

Previous studies have implemented transactional memory in both hardware and software. Hardware Transactional Memory (HTM) was originally proposed by Herlihy and Moss [13]. In this first work on transactional memory, the authors exploited access right policies within existing cache coherency protocols to realize their implementation. This implementation added new ISA-level instructions, as well as a distinct *transactional cache*, to enable hardware-level support for the paradigm. Locations accessed within the scope of a transaction by these new transactional load and store variants were traced using an associated *read set* and *write set*, respectively. Ananian et al. described and developed an extension to HTM that allows transactions to scale to near virtual memory levels through the use of a memory-resident logging structure [2].

Software Transactional Memory (STM) was first demonstrated by Shavit and Touitou [22]. The methodology detailed in this work provided the capabilities of non-blocking transactional memory to existing systems at the software level through the utilization of only standard load-linked and store-conditional ISA instructions. Moreover, it also introduced the concept of helper policies for transactions between processing elements. This initial implementation was, however, only applicable to static transactions that accessed a predetermined sequence of memory locations. This limitation was largely overcome in a subsequent work by Herlihy et al. [12]. Bocchino et al. further expanded the notion of STM and developed the first implementation targeted towards Partitioned Global Address Space models in distributed memory environments [6]. Herein, the authors found that, when utilized in conjunction with the GASNet library, their implementation was able to efficiently scale to up to 512 processors. In addition to standalone HTM and STM methodologies, some studies have also sought to couple the performance of HTM together with the extensibility of STM in a hybrid approach [3].

Summary: An in-depth comparison of transactional memory and atomic-based synchronization is beyond the scope of this work. However, it is worth noting that utilization of HTM is limited to systems with underlying microarchitecural support. Further, lock-based synchronization cannot always be directly translated to analogous transactional memory routines [5]. As such, the performance of atomic operations and associated synchronization primitives will be of continued importance to future systems.

2.3 Memory Benchmarks

The Spatter Benchmark, developed by Lavin et al., is also directly relevant to this work [17]. Although Spatter does not analyze the performance of synchronization primitives or atomic operations, similar to CircusTent, it is designed to benchmark the memory hierarchies of target architectures. More specifically, Spatter measures an architecture's memory performance with respect to an emerging class of indexed memory access patterns known as scatter and gather operations. Highly tunable, Spatter provides an efficient means of measuring these irregular, non-uniform memory access patterns increasingly common to HPC applications in a manner previous existing benchmarks could not. Currently, Spatter supports both OpenMP and CUDA backend implementations for both CPU and GPU-based platforms. As detailed in Section 3.3, CircusTent also integrates kernels that replicate these memory access patterns, but does so using atomic operations in lieu of traditional loads and stores.

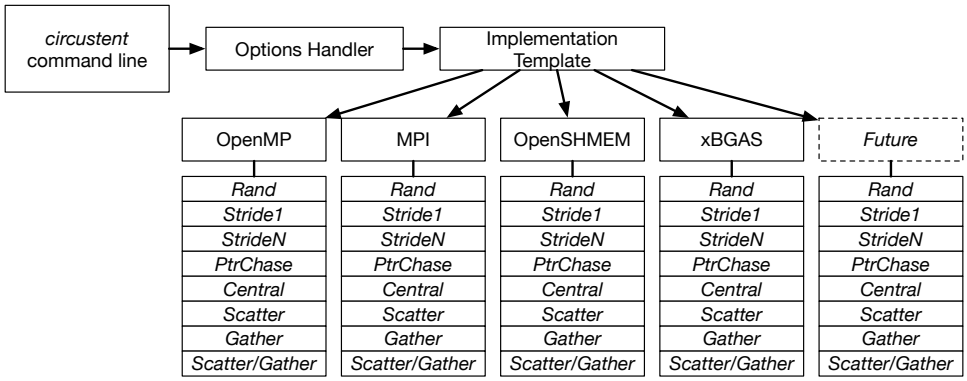

Figure 1: CircusTent Architecture

3 CIRCUSTENT

3.1 Benchmark Overview

The CircusTent benchmark infrastructure is designed to provide users and system architects the ability to derive normalized, quantitative performance data for atomic memory operations on parallel systems [1]. For this, we define the following driving requirements.

Derivation of Normalized Results. One of the primary difficulties in designing future scalable systems is the ability to derive and utilize quantifiable data to assist in the process of evaluating architectural trade offs. While application benchmark data is of paramount importance, it is often difficult to synthesize performance characteristics across applications or application kernels. As a result, the CircusTent benchmark derives a normalized, quantifiable performance value regardless of the size and implementation of the target system.

Programming Model Support. One of the unique characteristics of the CircusTent infrastructure is its ability to support a multitude of programming models for shared and distributed memory utilizing heterogeneous system architectures. Unlike previous memory benchmarks that were implemented using a single programming model, CircusTent seeks to exploit the intersection of the system architecture and programming model. Disparate programming models present atomic memory operations using a variety of methods that include compiler intrinsics, programming model directives, and explicit function calls. Therefore, CircusTent permits architects to utilize the benchmark with the application programming model most relevant to their constituent users.

System-Specific Optimizations. In addition to supporting a myriad of programming model backends, the CircusTent infrastructure also has the ability to support system-specific optimizations. As each programming model implementation is supported via a base implementation template, users have the ability to prescriptively utilize system-specific optimizations such as inline assembly language or compiler intrinsics to induce system-specific behavior.

Pathological Kernels. Finally, in order to mimic a wide array of application memory access patterns, CircusTent is designed to support eight pathological kernels that replicate common memory access patterns of interest. The patterns include unit stride, non unit stride, irregular, or entirely random accesses to memory locations using a variety of loop constructs. Further, each pathological kernel can be implemented using various different styles of atomic memory operations. We currently implement CircusTent backends using atomic *Add* and *Compare-and-Swap* (CAS).

As we see in Figure 1, the CircusTent infrastructure is constructed using a C++ inheritance model whereby operations such as command line parsing and benchmark option handling are common across all supported programming models. The implementation template is a C++ base class that is utilized to encapsulate each individual programming model implementation. Operations such as memory allocation, deallocation, and programming model initialization are provided by base-level member routines that are overridden by each respective programming model. In this manner, each programming model implementation has the ability to utilize it's respective initialization and allocation routines.

In addition to the aforementioned initialization and allocation routine functions, each programming is also required to implement individual versions of each benchmark algorithm (Section 3.3). For each algorithm defined, the implementation can be optimized using the respective programming model constructs in order to induce optimal performance.

3.2 Programming Models

The current set of programming models supported by CircusTent includes OpenMP, MPI, OpenSHMEM, and xBGAS [18]. As mentioned above, each respective implementation contains a unique set of initialization, memory allocation, and execution functions. The initialization functions perform any necessary device discovery, library initialization, and rudimentary setup as required for each programming model. Similarly, the memory allocation routines are utilized by each programming model to allocate and initialize two data structures. The `VAL` data structure is a linear array that may span multiple nodes and contains target data. The `IDX` data structure is also a linear array that may span multiples nodes and contains indices within the scope of the `VAL` data structure. Currently, all the backend implementations utilize unsigned 64-bit integers (`uint64_t`) for each data and index value. Notably, the initialization and allocation time is not included in the algorithmic runtime reported by the top-level driver functions. Finally, each

of the respective models contains an optimized implementation of each kernel that utilizes both target atomic memory operation types, atomic *Add* and *Compare-and-Swap*.

3.2.1 OpenMP. The first programming model utilized to implement a CircusTent backend is OpenMP. The OpenMP backend is rather simple and does not include a large number of CPU or memory-specific optimizations. As OpenMP is utilized in only physically shared memory environments, this backend employs standard `malloc` routines for the `VAL` and `IDX` memory management. Further, as we sought to make the OpenMP backend as portable as possible, it does not utilize any explicit NUMA optimizations. For each of the target kernels, we utilize the OpenMP `for` loop construct to parallelize the algorithms across multiple PEs. The default scheduling paradigm, as defined by the OpenMP runtime, is also utilized. For each atomic operation, we exploit the GNU atomic builtins using the `__ATOMIC_RELAXED` argument such that architectures with weak memory ordering may have multiple requests in flight.

3.2.2 MPI. The second programming model utilized to implement a CircusTent backend is MPI. Specifically, we utilize the MPI-3 specification [10] [11] to implement this distributed memory backend. The MPI backend is constructed using MPI-3 *window* constructs for dynamic, one-sided RMA operations. Each of the `VAL` and `IDX` constructs are locally allocated and mapped into a dynamic window. Within the kernel implementations of each algorithm, we utilize one sided operations, `MPI_Get` and `MPI_Put`, to fetch normal index values. Similarly, the `MPI_Compare_and_swap` and `MPI_Fetch_and_op` MPI-3 atomic operation function calls are utilized for realization of the actual atomics. Depending upon the MPI library utilized at compile time, each of the aforementioned MPI atomic operations may be implemented utilizing combinations of OS system calls, microarchitectural atomic operations, and remote direct memory access (RDMA) operations on the interconnect.

3.2.3 OpenSHMEM. The next programming model utilized to implement a CircusTent backend is OpenSHMEM [23] [20]. OpenSHMEM consists of a runtime infrastructure and set of common function interfaces that provide distributed memory systems a shared memory view via a symmetric heap managed by the runtime. Similar to other parallel programming models such as MPI, OpenSHMEM provides native interfaces for atomics, one-sided communication, and memory allocation routines. For CircusTent, we utilize the standard set of OpenSHMEM memory management routines (`shmem_malloc`) in order to allocate the `VAL` and `IDX` constructs in the symmetric heap. Similar to the MPI-3 implementation, we utilize native OpenSHMEM one-sided operations, `shmem_long_get` and `shmem_long_put`, to fetch normal index values. We utilize the native OpenSHMEM atomic function interfaces, `shmem_long_fadd` and `shmem_long_cswap`, for atomic *Add* and *Compare-and-Swap*, respectively.

3.2.4 xBGAS. The final programming model currently utilized to implement a CircusTent backend is the xBGAS runtime infrastructure [18] [16]. xBGAS is a project to construct a microarchitectural extension to the RISC-V instruction set in order to provide instruction-level shared memory access across distributed memory systems. The xBGAS runtime infrastructure mimics the form and function of the OpenSHMEM infrastructure. It provides rudimentary memory management routines for hardware-accelerated symmetric heap allocation (`xbrtime_malloc`) as well as function wrappers for xBGAS-accelerated global shared memory access instructions. The xBGAS backend directly employs atomic variants of these functions, `xbrtime_ long_atomic_add` and `xbrtime_long_atomic_compare_swap`, to realize its constituent atomic operations.

3.3 Algorithms

The CircusTent infrastructure contains eight individual benchmark kernels. Each kernel is described in terms of a generic atomic memory operation, or *AMO*. However, each kernel may be implemented using any platform-supported atomic operations. In the case of this study, we utilize atomic *Add* and atomic *Compare-and-Swap* operations to implement each kernel, respectively.

3.3.1 Random Access. The first kernel is a basic random access kernel (Algorithm 1). This kernel allocates two array structures. The `VAL` array contains a series of values. The `IDX` array contains a series of valid indices within the scope of the `VAL` array. Prior to the execution of the kernel, these indices are randomly selected and written to the `IDX` array using a linear congruential randomizer. For each iteration of the loop, a single `VAL` array entry is updated using an atomic operation. In this manner, the random access kernel contains one memory load (`IDX[i]`) and one atomic operation for each iteration of the loop. The goal of this kernel is to observe the performance of atomic operations when the platform has a limited ability to cache data for subsequent iterations.

Algorithm 1: Random Access Kernel

```
for i  0 to iters by 1 do
  | AMO(VAL[IDX[i]])
end
```

3.3.2 Stride-1. The second kernel encapsulates a simple, stride-1 kernel (Algorithm 2). The kernel allocates a single array (`VAL`) that contains a series of values. For each iteration of the loop, the kernel updates a single value in the array in linear fashion using a single atomic operation. In this manner, a platform may utilize data prefetching and/or caching in order to optimize the access to data members in this kernel in an optimal manner similar in form to dense vectors or matrices.

Algorithm 2: Stride-1 Kernel

```
for i  0 to iters by 1 do
  | AMO(VAL[i])
end
```

3.3.3 Stride-N. The third kernel is similar in form to the second kernel. In this kernel, we utilize the same `VAL` array structure as mentioned above, but we permit the user to define the unit stride by which we access the array (Algorithm 3). For example, if the user seeks to determine what the raw memory bandwidth is of parallel

atomics by forcing every access to induce a cache line miss, the stride-n kernel can accomplish this. Further, for each parallel PE participating in the kernel execution, the starting index is at least *iters* distance from the previous PE.

Algorithm 3: Stride-N Kernel

```
for i 0 to iters by stride do
  | AMO(VAL[i])
end
```

3.3.4 Pointer Chase. The fourth kernel included in the CircusTent suite is a pointer chasing kernel (Algorithm 4). Similar to the random access kernel, this kernel makes use of an `IDX` array. In this case, however, each preassigned random index within the array corresponds to another element of `IDX`. Each PE begins the kernel loop by performing an atomic operation to an array location, `start`, determined using the PE's rank identifier. For each subsequent iteration, the executed atomic operation is directly dependent on the index determined in the previous repetition. This kernel therefore replicates the irregular memory access patterns common to many applications that utilize linked data structures such as graphs.

Algorithm 4: Pointer Chase Kernel

```
for i 0 to iters by 1 do
  | start = AMO(IDX[start])
end
```

3.3.5 Central. The fifth kernel is unique among those included in the CircusTent suite. Rather than emulate a typical memory access pattern, the Central kernel is designed to measure performance in a worst case scenario (Algorithm 5). Within each iteration of this kernel, every active PE performs an atomic operation to the same shared memory location, given by `VAL[0]`. As a result, these accesses become serialized and performance quickly plateaus as the level of contention rises. Depending on the underlying architecture, this behavior can also severely tax the cache hierarchy and associated interconnects. For distributed shared memory systems, it also stresses the network interconnect. Given the above, this kernel can be used to estimate minimum performance for applications that feature frequent memory hot spots.

Algorithm 5: Central Kernel

```
for i 0 to iters by 1 do
  | AMO(VAL[0])
end
```

3.3.6 Sca er. The sixth kernel replicates the scatter memory access pattern found in many modern HPC applications (Algorithm 6). This pattern is characterized by the combination of sequential loads together with randomly indexed stores. As such, this kernel, wherein the `VAL` and `IDX` arrays are constructed as in the random access kernel, performs multiple atomic operations during each of its iterations. In the first step of a given iteration, the target atomic operation is used to obtain a random destination index, given by `dest`, from the `IDX` array. Next, a value to be stored, `val`, is similarly obtained. Finally, an atomic operation is executed on the memory location denoted by `VAL[dest]` using argument `val`.

Algorithm 6: Scatter Kernel

```
for i 0 to iters by 1 do
  | dest = AMO(IDX[i+1])
  | val = AMO(VAL[i])
  | AMO(VAL[dest], val) // VAL[dest] = val
end
```

3.3.7 Gather. The Gather kernel (Algorithm 7) can be considered the inverse of the Scatter kernel detailed above. Whereas the latter utilizes sequential loads in conjunction with random stores, the former combines loads from randomly indexed locations with sequential stores. Here, an atomic operation first procures a random index, `src`, from the `IDX` array. A store value, `val`, is then set from the memory location corresponding to `src` using a subsequent atomic operation. In the final step, `val` is used as an argument to the atomic operation that writes to the `VAL` array in a sequential manner across iterations.

Algorithm 7: Gather Kernel

```
for i 0 to iters by 1 do
  | src = AMO(IDX[i+1])
  | val = AMO(VAL[src])
  | AMO(VAL[i], val) // VAL[i] = val
end
```

3.3.8 Sca er/Gather. The final kernel included in the CircusTent suite is also the most complex, utilizing a total of four atomic operations (Algorithm 8). Aptly named, the Scatter/Gather kernel combines the random access components of Algorithms 6 & 7. Within each loop iteration of this kernel, the first and second atomic operations set random indices `src` and `dest`, corresponding to source and destination memory locations, respectively, using the `IDX` array. The third atomic operation then sets `val` from the location given in `src`. The final atomic operation is executed on the location denoted by `dest` using `val` as an argument. The memory access patterns exhibited by kernels 6, 7, and 8 imitate those commonly found in applications that perform computation using sparse matrices.

Algorithm 8: Scatter/Gather Kernel

```
for i 0 to iters by 1 do
  | src = AMO(IDX[i])
  | dest = AMO(IDX[i+1])
  | val = AMO(VAL[src])
  | AMO(VAL[dest], val) // VAL[dest] = val
end
```

Benchmark	AMOs Per Iteration
Rand	1
Stride-1	1
Stride-N	1
Pointer Chase	1
Central	1
Scatter	3
Gather	3
Scatter/Gather	4

Table 1: Atomic Operation Distribution

We summarize the number of atomic operations required to perform each kernel in Table 1. However, this may vary depending upon how each platform implements a respective atomic operation. The CircusTent implementation infrastructure supports the ability to override these defaults for each platform/programming model.

3.4 Normalizing the Results

Given the native extensibility of CircusTent to support a multitude of programming models and platforms, we seek to develop a normalized metric such that we can compare results across platforms and differing degrees of execution parallelism. For this purpose, we introduce the *GAMs* metric.

The *GAMs*, or *billions of atomic operations per second*, metric encapsulates the number of parallel execution elements (*PEs*), the atomic operation algorithmic complexity, and the wall clock execution time into a single metric. As we see in Equation 1, the metric is a ratio of the total number of atomic operations executed for all parallel execution elements (in billions) across all iterations and the wall clock execution time. The atomic operation algorithmic complexity is the total number of atomic operations required for a single *PE* to execute a single iteration of the target CircusTent kernel. This is equivalent to the *AMOs Per Iteration* column in Table 1.

$$GAMs = \frac{^{1}PEs \quad Iters \quad AMOs\_Per\_Iter^{0} \bullet 1e^{9}}{time} \tag{1}$$

4 BENCHMARK EVALUATION

In order to verify and demonstrate the capabilities of the CircusTent benchmark suite, we conducted an evaluation of a diverse set of platforms with respect to atomic memory operations. In the interest of space, and in order to provide a comprehensive introduction to CircusTent and its constituent kernels in Section 3, we evaluated only our OpenMP backend in this work. The evaluation of other backend implementations is left for a future work. We first introduce our test platforms and briefly describe pertinent details of their architecture in Section 4.1. We next detail our evaluation methodology in Section 4.2. Results for each of the CircusTent benchmark kernels are presented in Section 4.3. Finally, we analyze and discuss observed patterns and performance characteristics in Section 4.4.

4.1 Platforms

We evaluate CircusTent across a total of fourteen distinct platforms that encompass device classes ranging from embedded systems to those used in petascale-class supercomputers. These systems feature single and dual socket configurations utilizing processors from Intel, AMD, and ARM with varied instruction set architectures, clock frequencies, and core counts. Similarly, the total memory capacity differs across platforms. Table 2 provides an overview of our test platform specifications wherein each system is denoted by its processor model. The size of the `VAL` array used during our evaluation, as well as the compiler used to build CircusTent, is also shown for each platform.

As the organization of each system's cache hierarchy is an important factor to its benchmark performance, some further detail in this regard is warranted. Many of the processors featured in our test platforms adhere to a somewhat standard cache hierarchy design. Each of these processors feature a three-level hierarchy wherein the L1 and L2 caches are private to each core and the L3 caches are shared between cores in a given socket. Each L1 cache is subdivided into L1i and L1d caches for instructions and data, respectively. For the sake of brevity, we detail only systems whose cache organization deviates from this norm, and how they do so, below.

Cortex-A53 - The Arm Cortex-A53 is hosted on a Raspberry Pi Model 3B+. This model features four cores, each with a private 32KiB L1 cache and a 512KiB shared L2 cache. The Cortex-A53 does not have any cache levels beyond L2.

Cortex-A72 - Similar to the A53, the A72 has two cache levels. The L1 cache is split between a 32KiB instruction cache and a 48KiB data cache. The 1MiB L2 cache is shared amongst all the cores with no further caching layers.

Ryzen V1605B - The AMD Ryzen V1605B embedded x86_64 socket is hosted via an Udoo Bolt platform. The processor features an L1 cache split between a 256KiB instruction cache and a 128KiB data cache. The L1 instruction cache is 4-way set associative and the data cache is 8-way set associative. The L2 cache is 2MiB using an 8-way set associative configuration. The L3 cache is also 8-way set associative with 4MiB of capacity.

Core i7-4980HQ - In addition to a conventional 6 MiB L3 cache shared between cores, this processor also features a 128 MiB eDRAM L4, or "Crystal Well", cache that is shared between the CPU cores and the integrated GPU.

Xeon Phi 7250 - The 68 cores present in this processor are arranged in 34 tiles of 2 cores each. Herein, cores coresident within a tile share a 1MiB L2 cache. Although it does not incorporate a true shared last level cache, this processor features 16 GiB of MCDRAM that can be used as addressable memory, a shared cache across all cores, or in a hybrid configuration. For our evaluation, the MCDRAM is utilized in the cache configuration.

System	Clock Frequency	Cores / Socket	Total Sockets	LLC Size / Socket	Total Memory	Array Size	Operating System	Compiler
Cortex-A53	1.40Ghz	4	1	512KiB	512MiB	256MiB	Ubuntu 18.04 4.15.0	GCC 7.4.0
Cortex-A72	1.50Ghz	4	1	1MiB	4GiB	256MiB	Debian 10.1 4.19.75	GCC 8.3.0
Ryzen V1605B	1.58Ghz	4	1	4MiB	32GiB	15GiB	Ubuntu 19.04 5.2.10	GCC 8.3.0
Opteron 4130	2.60Ghz	4	2	6MiB	64GiB	15GiB	Centos7 3.10.0	GCC 8.3.1
Core i5-3210M	2.50Ghz	2	1	3MiB	4GiB	256MiB	macOS 10.13.6	clang 9.1.0
Core i7-3930K	3.20Ghz	6	1	12MiB	64GiB	15GiB	Linux Mint 18.3 4.15.0	GCC 5.4.0
Core i7-4980HQ	2.80Ghz	4	1	6MiB L3 + 128MiB L4	16GiB	15GiB	macOS 10.15.3	GCC 9.2.0
Xeon Phi 7250	1.40Ghz	68	1	16GiB MCDRAM	96GiB	15GiB	SLES 4.12.14	GCC 8.3.0
Xeon E5620	2.40Ghz	4	2	12MiB	48GiB	15GiB	Ubuntu 16.04 4.4.0	GCC 5.4.0
Xeon X5650	2.67Ghz	6	2	12MiB	64GiB	15GiB	Ubuntu 18.04 4.15.0	GCC 7.5.0
Xeon E5-2620 v3	2.40Ghz	6	1	15MiB	64GiB	15GiB	Ubuntu 16.04 4.4.0	GCC 5.4.0
Xeon E5-2670 v2	2.50Ghz	10	2	25MiB	64GiB	15GiB	Centos7 3.10.0	GCC 7.3.0
Xeon E5-2695 v4	2.10Ghz	18	2	45MiB	192GiB	15GiB	Centos7 3.10.0	GCC 7.3.0
Xeon E5-2698 v3	2.30Ghz	16	2	40MiB	128GiB	15GiB	SLES 4.12.14	GCC 8.3.0

Table 2: Benchmark System Configurations

4.2 Methodology

Consistent with previous studies [24][8][21][14], we perform our initial evaluation of the CircusTent suite in the context of a physically shared memory environment. In order to do so, we execute each of the CircusTent benchmark kernels, detailed in Section 3.3, on the test platforms introduced above using our OpenMP backend. For each kernel, we conduct trials using implementations based on both the atomic *Add* and *Compare-and-Swap* primitives. A uniform stride size of $N = 9$ is utilized for each trial of the StrideN kernel.

We collect performance results for each platform. In order to eliminate any performance volatility associated with simultaneous multithreading, we vary the thread count during our trials from a single thread, up to one thread per physical core, for each platform. Similarly, 64-bit operands are used throughout the evaluation to prevent any inconsistencies. Moreover, to better simulate real-world behavior, we allow the operating system and programming model to perform the mapping of threads to processor cores.

Where feasible, a uniform size of approximately 15 GiB is utilized for the `VAL` array on each platform. For the Cortex A-53, Cortex A-72, and Core i5-3210M systems, wherein physical memory limitations make this configuration impractical, a 256 MiB `VAL` array is utilized instead. In order to generate sufficient runtime such that observable patterns of behavior emerge, twenty million iterations of each kernel loop are run during every benchmark trial. We utilize our normalized *GAMs* metric, as introduced in Section 3.4, throughout our evaluation to measure and compare the performance of our diverse set of test platforms.

4.3 Results

4.3.1 Random Access. We first examine our benchmark results from the Random Access kernel as shown in Figure 2. Overall, our test platforms manifest lower *GAMs* performance for this kernel than in the majority of subsequent benchmarks. This behavior can be directly attributed to the irregular, unpredictable memory access patterns inherent in this kernel. In particular, the Core i7-4980HQ, Opteron 4130, and Cortex-A72 systems showcase the poorest performance across all our test platforms when utilizing four or fewer threads. For the same number of threads, the Xeon E5-2695 v4, Core i7-3930k, and Xeon-2698 v3 systems record the highest GAMs. Interestingly, with the exception of the Xeon E5-2695 v4, the Core i5-3210M system outperforms every other platform when executing with one or two threads. It is also notable that *Compare-and-Swap* implementations of this kernel outperform atomic *Add* implementations for the same platform. A noticeable drop in performance,

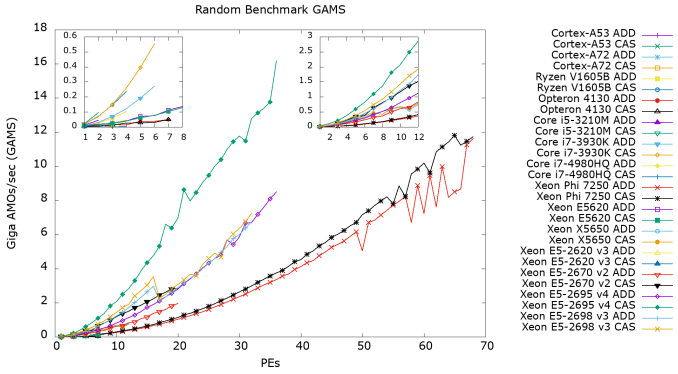

Figure 2: Random Benchmark GAMS

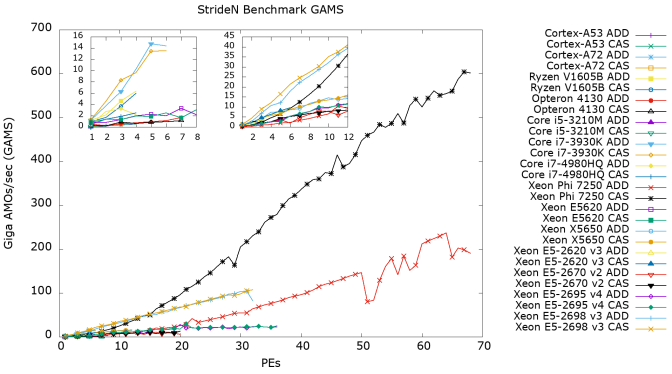

Figure 4: Stride-N Benchmark GAMS

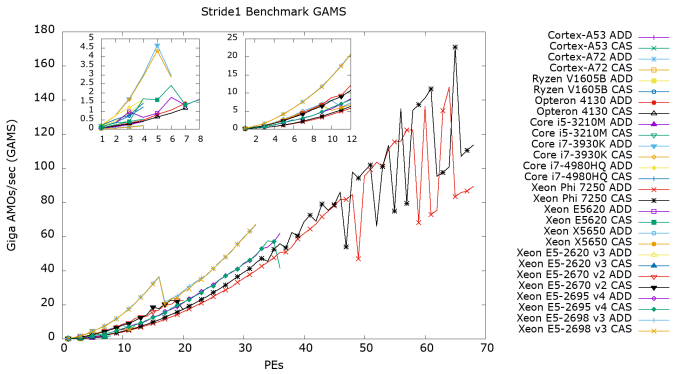

Figure 3: Stride-1 Benchmark GAMS

corresponding to thread placement across sockets, is also distinguishable for the Xeon E5-2698 v3 platform in this benchmark.

4.3.2 Stride-1. Our results for the Stride-1 kernel benchmark, shown in Figure 3, clearly contrast with those of the previous kernel. Herein, we see a marked improvement in the *GAMs* performance of the majority of our test platforms. This divergence is particularly prominent for some of our more powerful platforms such as the Xeon E5 class systems and the Core i7-3930k. However, the Core i7-4980HQ platform, which performed poorly in the Random Access benchmark, also makes unmistakable improvements. Orthogonally, the Core i5-3210M system performs much more poorly in comparison. Similar to the previous benchmark, a noticeable dip in performance occurs at 17 threads for the Xeon X5-2698 v3 system. Unlike the Random Access benchmark, there is little discernible performance variation between CAS and Add kernel implementations within a platform. For this benchmark, some unusual behavior is also demonstrated by the Xeon Phi 7250 platform, wherein the performance becomes erratic around 46 threads. We detail our conclusions in regard to this system in Section 4.4.2.

4.3.3 Stride-N. The results of our Stride-N kernel benchmark, illustrated in Figure 4, align most closely with those demonstrated by the Stride-1 benchmark. Here, the *GAMs* performance exhibited by each platform is much higher than it was for the Random Access kernel. Indeed, the *GAMs* recorded for this kernel often exceed those shown for the Stride-1 kernel. We attribute this behavior to the fact that each thread operates most often on distinct memory segments within this kernel. This improves each thread's private cache utilization while minimizing cache line invalidations. It also explains the absence of the cross-socket performance degradation demonstrated by the Xeon E5-2698 v3 system in other kernel benchmarks. The platforms that performed well during the Stride-1 benchmark, such as the Core i7-3930k and Xeon E5-2698 v3 systems, also exhibit impressive performance for this kernel. Our Ryzen V1605B platform also performs well in comparison. Likewise, those that performed poorly in the Stride-1 benchmark, including the Opteron 4130 and Cortex systems, often replicate that behavior here. In fact, the Core i5-3210M system manifests the poorest performance in this benchmark across all tested platforms. Also similar to the Stride-1 benchmark, with the exception of the Xeon Phi 7250 system, very little performance variation is measured between the CAS and Add kernel implementations. Orthogonal to the Stride-1 benchmark, there appears to be a hard upper bound in regard to the scalability of each platform for this kernel. Given the behavior of this kernel and our demonstrated results, we ascribe this bound to be a function of each platform's cache size. The presence of the on-package 16GiB MCDRAM, configured as a last-level cache, may also help explain why this generalization does not apply to the CAS kernel implementation on the Xeon Phi 7250 platform.

4.3.4 Pointer Chase. We next investigate the results of our Pointer Chase kernel benchmark. As shown in Figure 5, the *GAMs* performance for this kernel across platforms is lower than each of the previously detailed benchmarks. Similar to the Random Access kernel, this kernel exhibits irregular, unpredictable memory access patterns. However, in contrast to the Random Access kernel, the Pointer Chase kernel performs atomic operations to memory locations within the `IDX` array. Since the size of this array is based on a static calculation, and is typically much smaller than the `VAL` array, this behavior results in an increased number of cache line invalidations and associated cache misses. Consequently, the ability of the cache to improve system performance is severely diminished. As a result, the benefits to performance offered by large caches are effectively nullified in this benchmark and, in most cases, our more powerful systems perform little better than their counterparts for

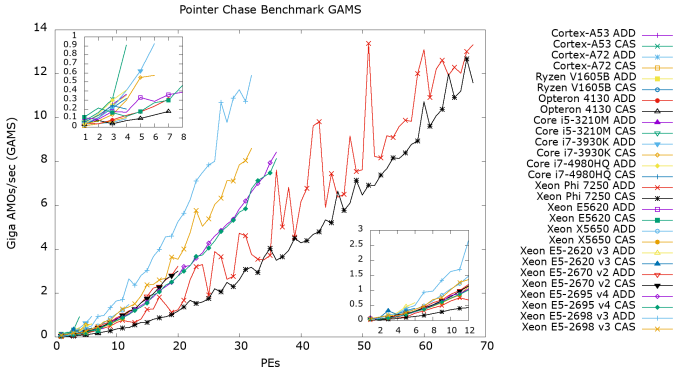

Figure 5: Pointer Chase Benchmark GAMS

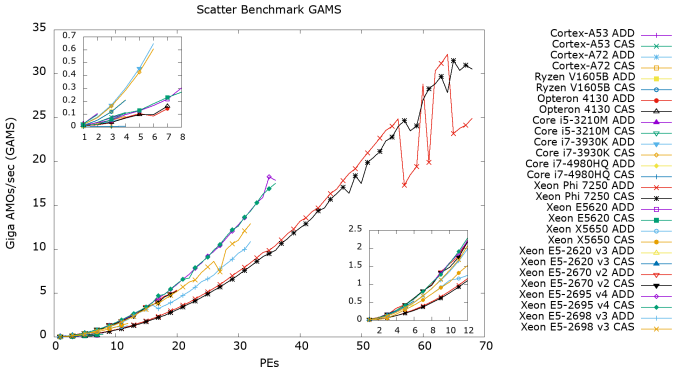

Figure 7: Scatter Benchmark GAMS

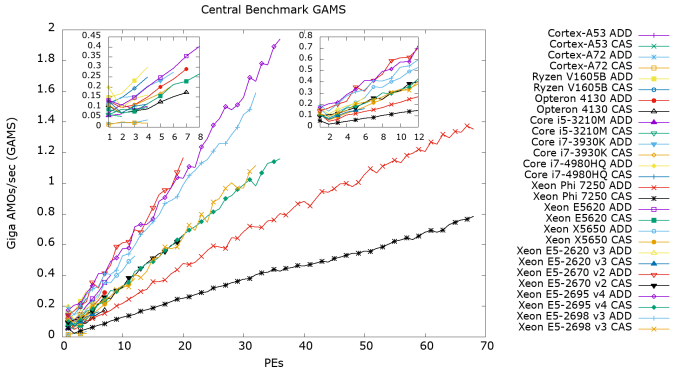

Figure 6: Central Benchmark GAMS

the same number of threads. Notably, the Xeon Phi 7250 platform almost uniformly records the lowest *GAMs* performance regardless of thread count. Further, its erratic behavior as the number of threads increases, as noted for other kernels, is particularly prominent in this benchmark and begins at a much lower number of threads. The results from this benchmark are also the first to showcase a distinct performance benefit for atomic Add based kernel implementations as compared to CAS implementations.

4.3.5 Central. Although the Random Access and Pointer Chase kernels perform more poorly than the Stride-1 and Stride-N kernels, the *GAMs* measured in our Central kernel benchmark, as shown in Figure 6, are by far the lowest of any in the CircusTent suite. As each thread performs an atomic operation to a common memory location during each iteration of the kernel loop, the measured *GAMs* performance for this kernel is directly dependent on the characteristics of a target architecture's memory subsystem. Since the number of requests any memory hierarchy can service within a given time frame is limited, the *GAMs* performance of this kernel for a given platform will eventually plateau as the number of PEs increases. Although several of our systems, such as the Xeon E5 class systems, may not have reached their zenith, our results demonstrate that many of our other platforms quickly reach such a plateau. Indeed, none of our test platforms ever exceed 2 *GAMs* in this benchmark. As contention for memory access represents a limiting factor in this benchmark, other components that contribute to a system's performance are less visible in these results. This explains the absence of any cross-socket performance decline for the Xeon E5-2698 v3, or any other, platform. In addition to our Xeon E5 platforms, which continue to perform well across disparate kernels, our Ryzen V1605B system also registers exceptional performance in this benchmark. In fact, the Ryzen system has the highest *GAMs* performance of any platform for 4 or fewer threads. In contrast, our Cortex-A72, Cortex-A53, Xeon Phi 7250, and Core i5-3210M systems again represent the lowest performers for this benchmark. However, the Opteron 4130 platform performs slightly better than in previous trials. Similar to the Pointer Chase kernel, the Central kernel seems to elicit higher performance from atomic Add implementations.

4.3.6 Sca er. The results for our Scatter kernel benchmark are shown in Figure 7. The most immediately observable characteristic of this graph is the steadily increasing performance of each platform as the number of threads increases. The only major exception to this behavior is manifested by the Xeon Phi 7250 system beyond approximately 56 threads. Another observation can be made in regard to the relative performance of our test platforms in this benchmark. In general, each system exhibits improved performance as compared to the Random Access and Pointer Chase benchmarks, but poorer performance compared to the Stride-1 and Stride-N benchmarks. Given what we know about the memory access patterns of these previous kernels, these results imply that the indexed access patterns present in the Scatter kernel are able to leverage the cache hierarchy to improve performance to some degree. The higher performance exhibited by our Xeon and Core i7-3930k systems lend some credence to this conclusion. However, the fact that our Core i5-3210M platform records the highest *GAMs* across all tested platforms suggests other factors may also play a role. Moreover, the poor performance of the Core i7-4980HQ system, which features an L4 cache in addition to a conventional L3, further increases this likelihood. Another notable characteristic of this kernel is that our Xeon e5-2698 v3 platform again experiences a drop in performance associated with cross-socket thread placement. However, this behavior only occurs for the atomic Add kernel implementation. Whereas other kernels demonstrated often demonstrated improved performance for a particular atomic primitive, the Scatter

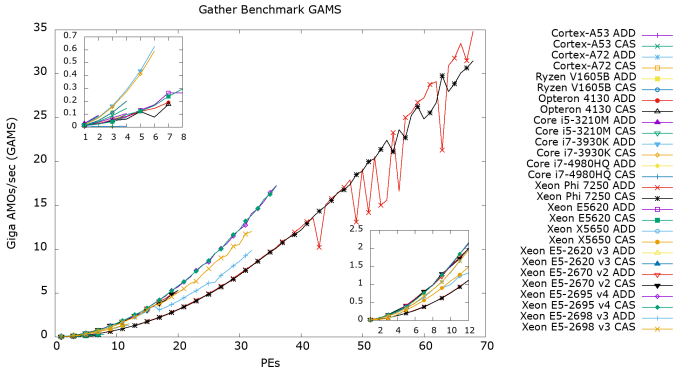

Figure 8: Gather Benchmark GAMS

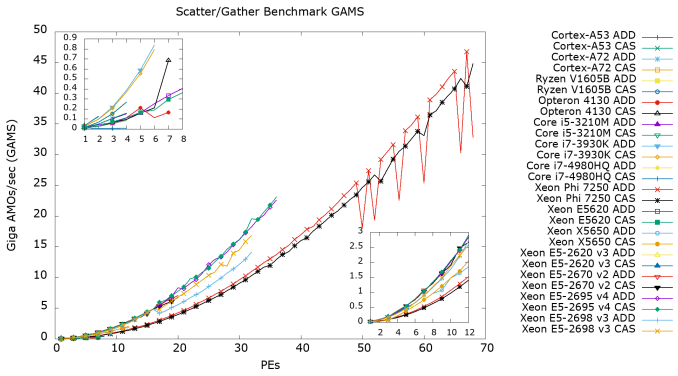

Figure 9: Scatter/Gather Benchmark GAMS

kernel seems to perform equally well for either CAS or Add kernel implementations.

4.3.7 Gather. Given the relationship between the Scatter and Gather kernels, it is unsurprising that our Gather benchmark results share many similarities with those detailed in Section 4.3.6. In fact, as shown in Figure 8, the *GAMs* performance across platforms for this kernel closely emulate those exhibited by the Scatter kernel. Both benchmarks demonstrate the same predictable improvement to the measured *GAMs* performance as the number of threads increases. Further, the *GAMs* performance of given a platform in the Gather benchmark is approximately equivalent to its performance in the Scatter benchmark. For our Xeon E5-2698 v3 platform, the now familiar cross-socket performance decline is also replicated only for the atomic Add implementation of the Gather kernel. With the exception of this system, the performance of the Gather kernel is likewise comparable for atomic Add and CAS implementations across systems and thread counts. Perhaps the most readily visible difference between the two sets of results is the fact that the erratic performance characteristics associated with our Xeon Phi 7250 system begin at a lower thread count in the Gather benchmark.

4.3.8 Sca er/Gather. The results of our final benchmark, performed using the Scatter/Gather kernel, are shown in Figure 9. As might be expected, the results for this kernel are highly congruous with those demonstrated by the individual Scatter and Gather benchmarks. The same cross-socket behavior exhibited by our Xeon E5-2698 v3 platform in these benchmarks is repeated for the Scatter/Gather kernel. Similarly, the Scatter/Gather kernel does not display an affinity for either atomic Add or CAS kernel implementations. Notably, however, the *GAMs* performance across platforms for this benchmark is slightly improved as compared to the Scatter and Gather benchmarks. This behavior is particularly pronounced at higher thread counts and for our more powerful platforms. As such, we conclude that the combination of the Scatter and Gather memory access patterns within a single kernel leads to an improved cache hit rate as the level of concurrency rises. Our Core i5-3210M system again manifests surprisingly impressive performance in this benchmark. However, the otherwise dominant performance of our Xeon and Core i7-3930k systems further support our conjecture. As with the Scatter and Gather benchmarks, our Core i7-4980HQ system presents exceptionally poor performance for this kernel. The Cortex-A72, Opteron 4130, and Xeon E5620 systems round out the lowest performing platforms for this benchmark.

4.4 Analysis

4.4.1 Expected Correlations. In many ways, the results gathered during our CircusTent evaluation conform to expectations. The variance of *GAMs* performance across different kernels, regardless of platform, is one such example. Here, the measured performance is highest in kernels that utilize uniform memory access patterns, such as the Stride-N and Stride-1 kernels. These uniform access patterns leverage the principle of spatial locality to service a higher percentage of memory requests from the cache. As such, they improve performance for atomic operations in much the same manner that they do so for traditional loads/stores. In contrast, kernels that utilize unpredictable memory access patterns are often forced satisfy requests from main memory. Accordingly, the recorded *GAMs* performance of the Random Access and Pointer Chase kernels is much lower. The performance of the Scatter, Gather, and Scatter/Gather CircusTent kernels, which utilize semi-random, indexed access patterns, falls between these two classifications. Predictably, the Central kernel, wherein concurrent atomic operations to a single memory location are repeatedly performed, exhibits the lowest performance and quickly reaches an upper bound as the degree of contention rises.

A similar generalization can be made in regard to the correlation between cache size and platform performance. Notably, the systems that sustain the highest *GAMs* performance across the distinct CircusTent kernels, such as the Xeon E5-2670 v2, Xeon E5-2695 v4, and Xeon E5-2698 v3, correspond to the those that feature the largest last-level caches. In contrast, the systems that often exhibit the poorest performance, including the Cortex-A53, Cortex-A72, and Opteron 4130, typically feature much smaller caches. This disparity directly illustrates the benefits a large cache size offers for atomic operation performance. These observations, consistent with conventional architectural wisdom, increase our confidence in the correctness and viability of the CircusTent benchmark suite.

4.4.2 New Insights. Our CircusTent evaluation also provides some new insights into the performance of atomic operations that are not readily apparent. The unexpectedly higher performance of our

CAS kernel implementations over their atomic Add analogs in some scenarios is one such example. Conventional wisdom dictates that the data overheads associated with the CAS instruction render it less efficient than other atomic instructions such as Add. However, our test platforms uniformly measured higher *GAMs* performance for the CAS implementation of the CircusTent Random Access kernel. We attribute this behavior to several contributing factors. First, our OpenMP Random Access kernel implemented using the CAS atomic primitive is written such that the value of *VAL[IDX[i]]* is compared against itself to determine the proper store value. In the event the values are equivalent, which is always the case for this implementation, the CAS instruction again stores this same value to the specified location. Herein, it is likely that the microarchitecture recognizes that the value to be written and the value already stored are, in fact, the same and optimizes performance by skipping the store component of the CAS operation. In this case, the time taken to execute each CAS instruction is reduced in a manner that the Add instructions are not. Further, writing the same value into each memory location during successive iterations of the kernel loop prevents cache-line invalidations. In scenarios wherein cache blocks are present in multiple private caches, this directly improves overall performance by increasing the cache-hit rate at higher levels of the cache hierarchy. Taken together, these behaviors explain the apparent performance discrepancy between the CAS and ADD implementations. Moreover, these observations suggest that future compilers may be able to further optimize performance by replacing atomic Add instructions with CAS analogs when values are expected to be modified infrequently.

We also observe that the performance characteristics of two of our test platforms deviate from our expectations. Since our Core i7-4980HQ system features a 128 MiB shared L4 cache in addition to its L3, we anticipated that this platform would be one of the higher performers across kernels. For four of our benchmarks, however, this system generated only mediocre *GAMs* performance that seems more inline with its 6MiB L3 cache. Further, in our Random Access, Scatter, Gather, and Scatter/Gather benchmarks, it was by far the poorest performing platform. Given the random memory access patterns inherent in these kernels, it seems likely that many cache lines would be evicted from the L3 cache into the L4 during benchmark execution. Although we assume the eDRAM-based L4 cache has a higher latency than its SRAM counterparts, it would still be expected to service requests faster than main memory. As such, we are currently exploring the microarchitecture of this system's memory hierarchy to better understand these results.

The performance of our Xeon Phi 7250 platform was also somewhat surprising. Although the cores present on this processor do not share a true last-level cache, we anticipated the 16GB of MCDRAM operating in the cache configuration to appreciably improve the performance of this platform. However, when compared to trials performed on other systems with equivalent thread counts, the Xeon Phi 7250 system performed poorly across the disparate set of CircusTent kernels. Moreover, its performance in many benchmarks was abnormally erratic. Notably, this behavior became increasingly aggravated as the thread count rose and was particularly prominent in our Stride-1 and Pointer Chase benchmark results. We believe this system's erratic behavior and sub-par performance can be attributed to several factors associated with its complex memory subsystem. First, the absence of a true-last level cache on this platform directly entails some degree of performance degradation. As the L1 and L2 caches on this system cannot cache all the necessary information, requests must often be serviced from the MCDRAM. Although one might expect this MCDRAM to improve performance in comparison to main memory, the opposite effect occurs. Here, the increased latency of the MCDRAM combined with the irregular memory access patterns exhibited by the CircusTent kernels diminish the platform's performance [19]. As such, any requests that would be serviced by the L3 cache, or even main memory, on other systems often incur a significant penalty for the Xeon Phi 7250 platform. Second, and perhaps even more importantly, the 34 L2 caches resident on each tile in this processor are kept fully coherent. As any invalidation for shared cache lines must be propagated across this complex system, these operations are exceedingly costly. These expensive operations, combined with the fact that any core possessing an invalidated cache line must then service a subsequent request from the MCDRAM, make shared cache lines across cores not resident within the same tile an expensive proposition for this system. As execution with an increasing thread count, as well as the Stride-1 and Pointer Chase kernels themselves, represent scenarios wherein cache line sharing is progressively likely, we feel this reinforces our conclusion regarding the performance of this platform. Further, this behavior can also explain the variation between the performance of the CAS and Add Stride-N kernel implementations. As our Stride-N CAS implementation operates in the same manner as the Random Access implementation previously detailed, cache line invalidations are similarly not generated in this benchmark. Thus, the absence of these invalidations allows the CAS implementation to far surpass that of the atomic Add implementation on our Xeon Phi 7250 platform.

Beyond the considerable impact cache size and organization has on atomic operation performance, the operating system and compiler may play a bigger role than originally anticipated. For example, we observe that our Ryzen V1605B system performs significantly better than expected in several of our kernel benchmarks despite its modest cache size. As we have not yet been able to isolate any microarchitectural factors that would explain this anomaly, it may be a result of the newer operating system and compiler used on this platform. While our Core i5-3210M also exceeds expectations in several benchmarks, we suspect this performance may be a function of the smaller `VAL` array utilized on this system. As our Xeon E5620 system performed significantly more poorly than similar systems with identical cache organizations, the operating system and compiler may also be at fault here. Finally, we note that while our Xeon E5-2698 v3 system demonstrated cross-socket performance degradations in several of our benchmarks, none of our other dual-socket platforms manifested the same behavior. Despite our efforts to avoid simultaneous multithreading, it is possible the operating systems of these platforms attempted to optimize performance by scheduling threads to the same physical core rather than cross socket boundaries.

5 CONCLUSIONS

The performance of atomic operations plays a critical role in the scalability of existing systems. As the degree of heterogeneity and

complexity of memory hierarchies in future systems increases, this trend can only be expected to continue, if not intensify. However, to the best of our knowledge, a comprehensive methodology for measuring the performance of memory subsystems with respect to atomic operations has yet to emerge.

In this work, we introduced the open source CircusTent benchmark suite to fill this void. Orthogonal to previous works, CircusTent measures the performance of disparate architectures in a generalized manner using common parallel programming paradigms. Herein, we showcased the modular design of CircusTent that enables native extensibility for future refinements and additional programming models. We explored our current backend implementations, designed for both physically shared and distributed shared memory systems, built upon the OpenMP, MPI, and OpenSHMEM programming models as well the xBGAS microarchitecture extension. We also detailed the eight kernels, designed to replicate memory access patterns common in high-performance computing applications, that constitute the CircusTent suite.

Finally, utilizing our OpenMP backend, we performed an extensive evaluation of CircusTent across fourteen diverse test platforms. Through the use of our normalized *GAMs* metric, we were able to directly measure and compare the performance of these disparate systems. In line with previous work and conventional wisdom, our results showed that both the memory access patterns of a given application and the cache organization of a target system significantly affect its performance with respect to atomic operations. However, our evaluation also revealed some new insights and diverged from our expectations with regard to several of our test platforms.

We feel our evaluation clearly demonstrates the capabilities and value of the CircusTent benchmark suite. As such, we believe that CircusTent will prove to be a useful tool that aids in the benchmarking of existing systems as well as the design and prototyping of future systems.

6 FUTURE WORK

Although the current CircusTent infrastructure supports a number of modern programming models, we would like to further expand this set of models to better support heterogeneous platforms that include diverse components such as GPUs and FPGAs. This will require adding support for the OpenMP *target* construct and/or inclusion of additional backends such as OpenACC or CUDA-based implementations. Alternatively, a more holistic approach could be applied through integration of a modern heterogeneous compilation and runtime infrastructure such as SYCL [26].

In addition to the aforementioned heterogeneous system infrastructure, we also seek to expand our benchmark evaluation to include distributed memory platforms and programming models. Given the current support for MPI and OpenSHMEM, it would be advantageous to execute CircusTent on a variety of interconnects (Infiniband, Cray Aries, Ethernet) at scale in order to derive the performance parameters of atomic memory operations for large-scale system deployments.

Finally, as parallel programming models continue to evolve and adapt to new system architectures, we will continually update the current CircusTent backend implementations to exploit the latest in programming model optimizations. Further, we fully expect to continue developing new programming model backends for CircusTent in order to evaluate additional models for future scalable systems.

ACKNOWLEDGMENTS

Research reported in this publication was supported by the U.S. Department of Defense under Contract FA8075 14 D 0002. The authors would also like to thank Los Alamos National Laboratory for use of the Trinitite system. This work is authorized for release under LA-UR-20-26175.

REFERENCES

[1] 2019. CircusTent Benchmark Suite Repository. https://github.com/tactcomplabs/circustent.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. 2006. Unbounded Transactional Memory. *IEEE Micro* 26, 1 (Jan. 2006), 59–69.

[3] Lee Baugh, Naveen Neelakantam, and Craig Zilles. 2008. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *2008 International Symposium on Computer Architecture*. IEEE, 115–126.

[4] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[5] Colin Blundell, E Christopher Lewis, and Milo MK Martin. 2006. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters* 5, 2 (2006), 17–17.

[6] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. 2008. Software Transactional Memory for Large Scale Clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. Association for Computing Machinery, New York, NY, USA, 247–258. https://doi.org/10.1145/1345206.1345242

[7] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. 2017. Fast In-Memory Transaction Processing Using RDMA and HTM. *ACM Trans. Comput. Syst.* 35, 1, Article Article 3 (July 2017), 37 pages. https://doi.org/10.1145/3092701

[8] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 33–48. https://doi.org/10.1145/2517349.2522714

[9] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 365–376. https://doi.org/10.1145/2000064.2000108

[10] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[11] William Gropp. 2012. MPI 3 and Beyond: Why MPI Is Successful and What Challenges It Faces. In *Recent Advances in the Message Passing Interface*, Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–9.

[12] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. 2003. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (PODC '03)*. Association for Computing Machinery, New York, NY, USA, 92–101. https://doi.org/10.1145/872035.872048

[13] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

[14] Fazeleh Hoseini, Aras Atalar, and Philippas Tsigas. 2019. Modeling the Performance of Atomic Primitives on Modern Architectures. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article Article 28, 11 pages. https://doi.org/10.1145/3337821.3337901

[15] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, USA, 437–450.

[16] Tactical Computing Labs. [n.d.]. RISC-V Extended Addressing Architecture Extension Specification Codenamed: xBGAS. https://github.com/tactcomplabs/xbgas-archspec

[17] Patrick Lavin, Jeffrey Young, Jason Riedy, Richard Vuduc, Aaron Vose, and Dan Ernst. 2018. Spatter: A Tool for Evaluating Gather / Scatter Performance. arXiv:cs.PF/1811.03743

[18] John D. Leidel, Xi Wang, Frank Conlon, Yong Chen, David Donofrio, Farzad Fatollahi-Fard, and Kurt Keville. 2018. XBGAS: Toward a RISC-V ISA Extension for Global, Scalable Shared Memory. In *Proceedings of the Workshop on Memory Centric High Performance Computing (MCHPC'18)*. Association for Computing Machinery, New York, NY, USA, 22–26. https://doi.org/10.1145/3286475.3286478

[19] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis. 2017. Exploring the Performance Benefit of Hybrid Memory System on HPC Environments. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 683–692.

[20] Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. 2011. *OpenSHMEM - Toward a Unified RMA Model*. Springer US, Boston, MA, 1379–1391. https://doi.org/10.1007/978-0-387-09766-4_490

[21] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation PACT)*. IEEE, 445–456.

[22] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. Association for Computing Machinery, New York, NY, USA, 204–213. https://doi.org/10.1145/224964.224987

[23] Open Source Software Solutions. [n.d.]. OpenSHMEM 1.4 Specification. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf

[24] Oreste Villa, Gianluca Palermo, and Cristina Silvano. 2008. Efficiency and Scalability of Barrier Synchronization on NoC Based Many-Core Architectures. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '08)*. Association for Computing Machinery, New York, NY, USA, 81–90. https://doi.org/10.1145/1450095.1450110

[25] Xi Wang, Brody Williams, John D. Leidel, Alan Ehret, Michel Kinsy, and Yong Chen. 2020. Remote Atomic Extension (RAE) for Scalable High Performance Computing. In *Proceedings of the 57th Annual Design Automation Conference 2020 (DAC '20)*.

[26] Michael Wong and Ruyman Reyes. 2018. What's New in SYCL 1.2.1 and How to Explore the Features. In *Proceedings of the International Workshop on OpenCL (IWOCL '18)*. Association for Computing Machinery, New York, NY, USA, Article 11, 1 pages. https://doi.org/10.1145/3204919.3204930