Performance Analysis and Modelling of Concurrent Multi-access Data Structures

Performance Analysis and Modelling of Concurrent Multi-access Data Structures

Adones Rukundo
Chalmers University of Technology
Gothenburg, Sweden
adones@chalmers.se

Aras Atalar
Chalmers University of Technology
Gothenburg, Sweden
aaras@chalmers.se

Philippas Tsigas
Chalmers University of Technology
Gothenburg, Sweden
tsigas@chalmers.se

ABSTRACT

The major impediment to scaling concurrent data structures is memory contention when accessing shared data structure *access-points*, leading to thread serialisation, hindering parallelism. Aiming to address this challenge, significant amount of work in the literature has proposed multi-access techniques that improve concurrent data structure parallelism. However, there is little work on analysing and modelling the execution behaviour of concurrent multi-access data structures especially in a shared memory setting.

In this paper, we analyse and model the general execution behaviour of concurrent multi-access data structures in the shared memory setting. We study and analyse the behaviour of the two popular random access patterns: shared (Remote) and exclusive (Local) access, and the behaviour of the two most commonly used atomic primitives for designing lock-free data structures: Compare and Swap, and, Fetch and Add. We model the concurrent multi-accesses by splitting the thread execution procedure into five logical sessions: i) side-work, ii) *access-point* search iii) *access-point* acquisition, iv) *access-point* data acquisition and v) *access-point* data operation. We model the acquisition of an *access-point*, as a system of closed queuing networks with parallel servers, and data acquisition in terms of where the data is located within the memory system.

We evaluate our model on a set of concurrent data structure designs including a counter, a stack and a FIFO queue. The evaluation is carried out on two state of the art multi-core processors: Intel Xeon Phi CPU 7290 with 72 physical cores and Intel Xeon E5-2695 with 14 physical cores. Our model is able to predict the throughput performance of the given concurrent data structures with 80% to 100% accuracy on both architectures.

CCS CONCEPTS

• **Theory of computation → Concurrency**; **Parallel computing models**; • **Computing methodologies → Concurrent algorithms**; • **Information systems → Data structures**.

KEYWORDS

concurrency, data structures, locality, multi-access, semantic relaxation, performance modelling, parallel programming, lock-free, parallelism, cache, multi-core, queuing theorem.

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9146-7/22/07.
https://doi.org/10.1145/3490148.3538578

ACM Reference Format:
Adones Rukundo, Aras Atalar, and Philippas Tsigas. 2022. Performance Analysis and Modelling of Concurrent Multi-access Data Structures. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3490148.3538578

1 INTRODUCTION

Concurrent data structures are designed to provide highly concurrent accesses to shared data. Although concurrent data structures can benefit from parallelism, scaling concurrent data structures to parallel hardware, such as multi-core processors, is one of the most important challenges in computing today. A major impediment to scaling concurrent data structures is process (*thread*) synchronisation at given shared memory locations (*access-points*). Synchronisation is generally achieved by guaranteeing some notion of atomicity, in that, a given sequence of instructions executed by a single thread appears to take effect instantaneous to the external observer. For a given data structure, there can be one or more *access-points*, from where threads have to compute, consistently, the current state of the given data structure. Synchronisation is vital to achieving data structure correctness with respect to given sequential semantics, and therefore cannot be eliminated [8].

However, there is a trade-off between synchronization and scalability, that manifests as contention between concurrent threads at a given shared *access-point* [10, 12, 14–17, 22]. Contention builds up quickly and limits scalability as the number of concurrent threads trying to synchronise access at a given shared *access-point* increases. As an example, a stack has one *access-point* (head) where concurrent threads have to contend while trying to add or remove an item to/from the stack. This is an example of an inherently sequential data structure that requires special techniques to achieve better parallelism.

Reducing contention arising from limited data structure *access-points*, and consequently improving scalability, is and has been a major challenge for concurrent data structure researchers. Several design techniques trying to address this challenge have been proposed and applied in the literature, including, elimination [1, 23, 36], combining [22, 37], dynamic elimination-combining [9] and semantic relaxation [2–4, 6, 18, 19, 24, 31, 32, 35, 38, 39]. Some of these techniques have gone further to localise thread accesses to given *access-points* (locality) [18, 32, 39]. Locality is achieved by giving a thread exclusive access to a given *access-point* for a given number of operations.

The general idea behind most of the proposed techniques is to increase the number of *access-points* from which concurrent threads can access the data structure and complete their operations in parallel [27]. Increasing the number of *access-points* of a data

structure has the potential to improve parallelism, and thus harness the high throughput performance capabilities of the highly parallel multi-core processors. However, for some data structures, the only way to do so efficiently is by relaxing their semantics [24, 31, 32]. Although relaxing semantics has been studied and shown to significantly improve throughput performance, it has also been shown, that semantic relaxation, through increasing the number of *access-points*, is inversely proportional to the data structure *accuracy* (degree of relaxation) [5, 31, 32]. Furthermore, increasing the number of data structure *access-points* has a memory consumption trade-off. Memory consumption increases with the increase in the number of *access-points* which in turn increases the cost of data structure access [20].

Scaling throughput performance is as important as data structure accuracy and memory efficiency. Therefore, understanding the balance between the trade-offs mentioned above is key to the designing of scalable concurrent data structures. The goal is to understand the sweat-spot up to where we do not counteract the performance benefits from contention reduction through increasing the number of *access-points* [32]. This implies that as the number of *access-points* increases beyond a certain point, the trade-offs can out weigh the performance gain. Although this is true, there is little work in the literature that models and analyses concurrent multiple *access-points* execution behaviour in terms of throughput performance, especially in a multi-core execution environment. Modelling and analysing the practical throughput performance of concurrent data structures with multiple *access-points*, is an essential missing resource in the literature. Such a resource can go a long way to help concurrent data structure designers and researchers choose a good configuration for their applications. To try and address this gap, specific system models have to be developed integrating both access patterns and hardware configurations.

In this paper, we analyse and model the general execution behaviour of concurrent *access-points* (multi-accesses), in a multi-core and many-core shared memory execution environment. We use a three step methodology. First we design a simple concurrent multi-access execution model macro benchmark that we use to study the two basic random access patterns: remote and localised accesses, and two atomic primitives: Compare and Swap (*CAS*[1]) and Fetch and Add (*FAA*[2]). We study *CAS* and *FAA* as they are the most common atomic primitives when designing lock-free data structures. The aim of this work is to provide a generic model that can be augmented later on with the specifics of particular data structure designs. For this reason, we design a generic model that excludes data structure specific operations, at the beginning, and only focuses on the concurrent *access-points* relevant operations. As we describe later on, this model can be extended to incorporate the specifics of the data structure and hardware on hand.

Next we model the concurrent multi-accesses by splitting the thread execution procedure into five logical sessions: i) side-work, ii) *access-point* search, iii) access-point acquisition, iv) access-point data acquisition and v) access-point data update. The first and fifth sessions are independent of the access patterns, and therefore we focus on the second, third and fourth sessions for our analysis. We model the acquisition of an *access-point*, as a system of closed queuing networks with parallel servers, where each server corresponds to an *access-point*. Then we model the data acquisition (*memory latency*) in terms of where the data is located within the memory system (*cache hit/miss*). We use the memory and data operation latency to estimate the service time distribution of the queue network. Combining the operational costs of the five sessions, we are able to come up with a predictive model that can predict the throughput performance of concurrent multi-access data structures in a multi-core shared memory system. We see our model as a stepping stone towards modelling and analysing complex concurrent shared memory data structures that integrate disjoint multi-accesses as part of their design.

We first validate our model by predicting the throughput performance of our macro benchmark. Then we evaluate the accuracy of our model on a set of common, practical multi-access data structures, including a counter, a stack and a FIFO queue. Both the validation and evaluation experiments are run on a state of the art multi-core and a state of the art many-core system: Intel Xeon E5-2695 with 14 physical cores and Intel Xeon Phi CPU 7290 with 72 physical cores respectively. Our model is able to predict the throughput performance of all the experimented benchmarks with 80% to 100% accuracy, for different execution configurations and algorithmic designs as presented in Section 5.

The rest of the paper is organised as follows. We discuss the literature in Section 2, present our macro benchmark framework in Section 3 followed by the throughput analysis in Section 4. We validate our model in Section 5 and conclude in Section 6.

2 RELATED WORK

As mention in Section 1, there is a trade-off between synchronization and scalability in the form of contention. Towards understanding the impact of contention to concurrent data structure performance in shared memory, different studies have been presented in the literature [10, 15–17, 22]. Results in the literature show that scalability of synchronisation is a property of hardware [14]. However, most contention based performance analysis studies abstract away numerous hardware features, including the memory latency, the cost of synchronising the threads [34] and the fact that memory is partitioned into modules that service requests sequentially. In our study, we consider such hardware features to estimate the service time at each *access-point* and estimate how long a thread would wait in the queue.

Although getting a picture of how synchronisation and their underlying hardware atomic primitives behave in every single context is difficult; studies have been conducted to try and quantify the cost of synchronisation and how it affects scalability [12, 14]. Across the different layers experimented in the studies, results show that limited *access-points* leads to a higher cost of synchronisation which inhibits scalability. It was also observed that the underlying hardware atomic primitives have a high impact on the synchronisation cost. As an example, CAS based synchronisation was more costly than FAA. Insights from the studies indicate that concurrent data structures have to be designed to overcome the hardware synchronisation cost limitations. However, a general mechanism on

[1] *CAS* atomically compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value

[2] *FAA* atomically increments the contents of a memory location by a specified value

how to overcome such factors to improve concurrent data structure performance is limited. This motivates the need to analyse and model concurrent multi-access data structures, and quantify the performance benefits of integrating multi-accesses in designing concurrent data structure.

The instruction level latency of hardware atomic primitives is predominately memory latency. This implies that when evaluated on the same hardware, atomic primitives such as CAS and FAA can have identical instruction level latency as shown in [34]. The latency evaluation presented in [34] shades light on cache-to-cache transfer latency for the given hardware. However, when applied to concurrent data structures, the overall atomic primitive latency can be far way from the observed cache-to-cache transfer latency. Whereas in [34] CAS and FAA are observed to have similar latency, our results discussed later in Section 5 show that CAS has a higher latency than FAA. The difference can partly be attributed to the variability in the number of operands involved and the notion of *wasted* work introduced by the CAS semantics (retry loop). It is also observed in [34], that due to the limitation of atomic's instruction level parallelism, bandwidth reduces with the increase of number of concurrent threads (executing an atomic operation) contending on the same memory location. This observation clarifies the scalability limits, that we discussed in the previous section, that arise from *access-point* contention. Although the evaluation gives an insight into the cost of atomics, the results are restricted to the instruction level cost, independent of other factors such as contention and operand variability.

A set of sophisticated benchmarks for cache-to-cache transfer latency and bandwidth measurements to arbitrary locations in the memory subsystem have been proposed in the literature [20]. The benchmarks consider the coherency state of cache-lines to analyse the cache coherency protocols and their performance impact. Results from different multi-core processors reveal that the location and coherence state of data within the memory hierarchy have a tremendous affect on the performance of an application accessing the given data [21, 29, 30, 33].

Concurrent data structure throughput performance has been analysed in terms of hardware and logical conflicts [7]. In the analysis, hardware conflicts are viewed as concurrent calls to a hardware atomic primitive, whereas logical conflicts are viewed as concurrent operations on the shared data structure. The analysis relies on the estimation of two impacting factors that lower the throughput: the serialisation of threads executing the atomic primitives within a retry loop, and the wasted thread retries that fail on the atomic primitive. Furthermore, an atomic primitives performance model is presented in [25], covering performance metrics including energy consumption, fairness among threads and throughput. The model is built on a single cache-line bouncing process as a way of modelling concurrent access to a shared memory resource. However, the two studies are limited to a single shared access-point and specific to a single access pattern. This limits the memory latency to a single cache-line residing in the level one cache of the tested hardware. Implying that the movement of data within the memory hierarchy is not fully captured.

Performance studies have been carried out in the literature targeting different computing system's components and configurations such as hardware primitives, memory hierarchies and cache coherence protocols. Some of the studies have been used to analyse and model performance of specific algorithms and data structures designs. Although intricate performance analysis and models have been presented in the literature, work incorporating concurrent multi-accesses to shared memory in the context of concurrent data structures is still lacking. There is need to analyse and model the practical throughput performance of the ever growing number of multi-access concurrent data structure design techniques.

3 EXECUTION MODEL

In this section, we present our micro benchmarks that we use to model the execution behaviour of concurrent *access-points*, locally and remotely accessed. We base the benchmarks on a simple lock-free concurrent access to shared memory (implementing a simple counter), considering the two most commonly used atomic primitives for designing lock-free data structures: *FAA* and *CAS*. Using either *FAA* or *CAS*, each thread updates a memory location corresponding to a given *access-point* as shown in Algorithms 1 and 2. We benchmark local *access-point* (*locality*) execution behaviour by allowing a thread to exclusively access a specific *access-point*, on which it can perform its update locally without contending or sharing data with any other thread in the system. To benchmark shared *access-points* (*remote*) execution behaviour, we let a thread select an *access-point* uniformly at random, from an array of shared *access-points* on which it can perform its update.

Algorithm 1: Fetch and Add (*FAA*)

```
1.1  Allocate sharedCounters[K];
1.2  Allocate localCounters[N];
1.3  Function ThreadProcedure(P):
1.4      while ElapsedTime < duration do
1.5          operation ← randomOp(opRatio);
1.6          SIDEWORK(randomCycles(maxCycles));
1.7          if operation = local then
1.8              FAA(localCounters[P],1);
1.9          else
1.10             index ← randomIndex(K);
1.11             FAA(sharedCounters[index],1);
1.12         success ++;
1.13     return success;
```

Threads call the ThreadProcedure function and execute their operations in a loop for a given *duration* (Line 1.4, 2.4). A thread has to uniformly at random decide (Line 1.5, 2.5), whether, to operate on a local *access-point* (Line 1.8, 2.10) or, a shared *access-point* (Line 1.11, 2.16) that is also selected uniformly at random (Line 1.10, 2.13). *FAA* unlike *CAS* always succeeds on acquiring access to an *access-point*. Typically, a thread that synchronises using *CAS*, accesses an *access-point* twice to complete its operation. The first access is to read the state of the *access-point* (Line 2.14) and prepare the new value to be used for updating accordingly (Line 2.15). The second access is to try and perform the required *access-point* operation by executing the *CAS* instruction (Line 2.16). *CAS* can only succeed if

Algorithm 2: Compare and Swap (*CAS*)

```
2.1  Allocate sharedCounters[K];
2.2  Allocate localCounters[N];
2.3  Function ThreadProcedure(P):
2.4      while ElapsedTime < duration do
2.5          operation ← randomOp(opRatio);
2.6          SIDEWORK(randomCycles(maxCycles));
2.7          if operation = local then
2.8              curr ← localCounters[P];
2.9              new ← curr + 1;
2.10             CAS(localCounters[P],curr,new);
2.11         else
2.12             while true do
2.13                 index ← randomIndex(K);
2.14                 curr ← sharedCounter[index];
2.15                 new ← curr + 1;
2.16                 if CAS(sharedCounter[index],curr,new) then
2.17                     break;
2.18         success ++;
2.19     return success;
```

the state of the *access-point* has not changed from the first access state (Line 2.14).

We split the thread procedure into five logical sessions: i) side-work, ii) *access-point* search iii) *access-point* acquisition, iv) *access-point* data acquisition and v) *access-point* data operation. A description of these five logical sessions is presented in Figure 1. We note that the order of the sessions is a typical one. The order of some sessions might be permuted, for example a thread might perform side-work after performing an *access-point* search, and in other cases sessions might overlap.

3.1 Side-work

We define side-work, as the work done by the thread independent of the number of *access-points* available. In other words, when performing side-work the thread does not access the data structure. However, the amount of side-work can depend on the data structure design. The data structure might require a thread to do some independent work in preparation to access the data structure. For example, a thread might run as part of the side-work, a hash function before accessing a hash table or allocate memory for a new data structure item before it is added to the data structure. Also, the data structure can be part of an application that requires threads to perform data structure independent tasks between accesses to the data structure. Side-work can further be expanded to include thread delays caused by external interference such as thread preemption. To cater for side-work in our execution model, we include a variable number of random pauses cycles between operations (Line 1.6, 2.6). We use the randomly selected pause cycles to mimic the variability in thread delays due to the side-work and the randomness of thread *access-point* acquisition, covering a wide range of different application scenarios.

3.2 Access-point Search

A thread has to search for an *access-point* from the given shared *access-points* through which it can access the data structure. The search session is depicted in Figure 1 as access-point *search*, starting from the time t_1 when the thread starts the search process, to the time t_2 when the thread selects an *access-point* and issues an access request for the given *access-point*. In our micro benchmark, we use a simple search process that is comprised of a single call to a random function (Line 1.10, 2.13). However, some data structures designs such as 2D-relaxed [32] and random choice of two [5, 31] have more complex search processes that involve reading multiple data points within memory before selecting an *access-point*. As an extension of the *access-point* search session, data structure specific search models can be added to our model to better suit the given data structure designs with more complex search patterns.

3.3 Access-point Acquisition

During this session, the thread tries to gain access to the data structure through a given *access-point*. The duration of this session is depicted in Figure 1 as *Waiting time*, starting from the time t_2 when the thread requests for access at a given *access-point*, to the time t_3 when the thread acquires the access through the given *access-point*. When accessing a concurrent data structure, threads have to synchronise their accesses to maintain the data structure correctness with respect to the respective sequential semantics. Thread synchronisation is achieved through guaranteeing some notion of atomicity, where, thread operation at a given *access-point* appears to occur in a single instant between acquiring access and completing the data operation. Figure 1 describes the sessions that require atomicity as *Atomic execution*, starting from acquiring access at the time t_2 to when the data operation is completed at time t_4.

We model the *access-point* acquisition session as a system of closed queuing networks with parallel servers, where each server corresponds to an *access-point*. Due to atomicity, an *access-point* can only service one thread at a time. This therefore implies, that concurrent threads trying to access the same *access-point* have to wait for each other to be serviced. In the literature this is commonly referred to as *contention*, and significant work towards contention reduction is available in the literature. As discussed in Section 1, increasing the number of *access-points* is one of the most popular design technique proposed in the literature to alleviate *contention*. To study this behaviour, we vary the number of shared *access-points* using the parameter $\mathcal{K}$ (Lines 1.1, 2.1). For a given number of threads, as the number of *access-points* increases, contention reduces by a factor, discussed and analysed in Section 4. Reduction in contention leads to an increase in throughput performance. However, the performance increase is subject to the data location within the memory hierarchy as we show in Section 5.

3.4 Access-point Data Acquisition

A thread has to acquire data that corresponds to the selected *access-point* on which to perform its data operation. This session is depicted in Figure 1 as *Fetching data*, starting from the time t_3, when the thread acquires exclusive access to a given *access-point*, to the time t_4, when the thread acquires the given *access-point* data. The thread data fetching period relies on how much time it takes for

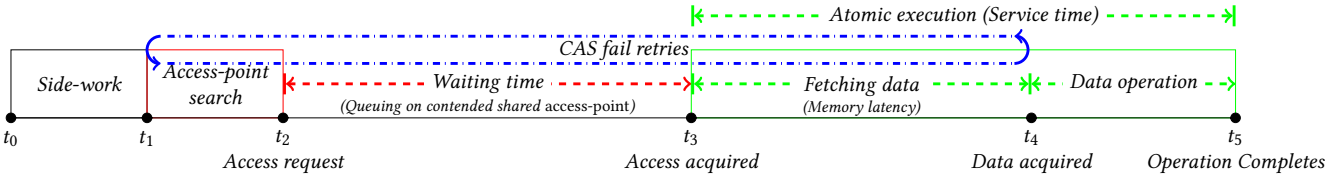


Figure 1: Thread execution procedure breakout. The *Atomic execution* section represents the instructions carried out by the thread while holding the *access-point*, i.e. other words, while executing the atomic instruction *CAS*/*FAA*. When a *CAS* fails, it exists the *Atomic execution* by releasing the *access-point*.

the hardware to deliver the data to the thread (*memory-latency*). Memory-latency varies depending on the hardware design and memory configuration (*memory hierarchies*).

We model the data acquisition session in terms of where the data is located within the memory system. Multi-core systems have complex memory hierarchies, including several levels of private and shared cache memories (L1, L2, often L3, and rarely even L4) [13, 21, 29]. Cache levels have different capacities and latencies, with L1 having the smallest capacity and lowest latency. The higher the cache level the higher the capacity and latency. The smallest transferable unit of data within the cache hierarchy is a *cache-line*. Several copies of the same cache-line can exist in different caches of different cores at the same time, especially in shared data access executions. Cache coherent protocols such as MESI[3], are used to maintain the multiple cache-line copies coherent across the different cores. For example, an update on one of the cache-line copies, invalidates all the other copies available in other caches. Also cache level latencies differ depending on the coherence state of the data being accessed [20].

We consider two types of data acquisition; local (Line 1.8, 2.10) and remote (Line 1.11, 2.16). Local data acquisition is when a valid copy of the required *access-point* data is fetched from the cache of the local core (a core on which the requesting thread is running). While remote data acquisition, is when a valid copy of the required *access-point* data has to be fetched from the cache of a remote core. Typically, data requests are made to the local cache, starting from the lowest cache level (L1) up until the last level cache. Only if, the requested data is not found within the local cache levels, then a remote data request is issued to other remote caches. Data can be found (*cache-hit*) or not found (*cache-miss*) at a given cache level. Cache-misses can be caused by two factors; the requested data might have been evicted from cache due to cache limited capacity (*capacity-miss*), or, the requested data might have been invalidated by a remote update on the data (*coherence-miss*) [21, 29, 30, 33].

The rate of cache-hits versus cache-misses contributes in determining the overall performance of concurrent multi-access data structures in shared memory. To benchmark cache hits/misses, we vary the number of local accesses and remote accesses together with varying the number of shared *access-points*. Each thread decides uniformly at random whether to access a local *access-point* or a shared *access-point* (Line 1.5, 2.5). This way, we can vary the number of coherent-misses and study their execution behaviour.

[3]MESI cache coherent protocol, maintains cache-lines in one of the the four states: Modified, Exclusive, Shared and Invalid.

Furthermore, to benchmark capacity-misses, we vary the number of *access-points* ($\mathcal{K}$) to fill up the capacities of different cache levels. Filling up caches induces data evictions, leading to capacity-misses when required data is found to have been evicted from the given cache level.

3.5 Access-point Data Operation

This is the last session of the thread execution procedure, described as *Data operation* in Figure 1. At this point, the required data has been delivered to the thread to carry out the necessary data operations. During this session, a thread performs the required data operations specific to the given *access-point*. Core clock frequency is the only performance determining factor for this session. The higher the frequency the shorter the time the data operation will take to complete.

4 THROUGHPUT ANALYSIS

In this section, we analyse the throughput of our execution model Algorithms 1 and 2. We start by modelling the overall throughput as a system of closed network of queues where each *access-point* corresponds to system servers. Then we model the memory system and estimate the memory latency for accessing an *access-point*. We then use the memory latency and thread data operation execution time to derive the service time distribution of the *access-points* (*Atomic execution* depicted in Figure 1). Finally, we incorporate side-work, *access-point* search, local accesses and *CAS* failures into the general model to estimate the overall throughput of each algorithm.

4.1 Closed Network of Queues

We analyse a closed network of queues, as a system of $\mathcal{K}$ number of *access-points* (servers) and $\mathcal{N}$ number of threads. Our approach is based on Mean Value Analysis. We estimate the expected delay that a thread is subject to, starting from its arrival instant to the server, until it completes its operation. This expected delay is then used to compute the throughput of the queuing network.

We decompose the delay and define it as the sum of its components. Then, we obtain the expectation of the total delay, relying on linearity of expectation, by estimating and summing the expectation of the components.

First, we estimate the expectation of the queuing delay at the arrival instant, which can be further split into the residual service time of the thread in service (denoted by $\mathcal{R}_t$) and the sum of service times for threads that are waiting in the queue (denoted by Q_t). In addition, we consider the service delay for the thread to get its

own service (denoted by $\mathcal{S}_t$). As a remark, we have removed the subscripts for the expected delay of these components because they are equal for all *access-points* due to the symmetry among *access-points*. Then, the system throughput (denoted by $\mathcal{T}$) and *access-point* i throughput (denoted by $\mathcal{T}_i$) can be computed as follows:

$$\mathcal{T} = \frac{\mathcal{N}}{\mathbb{E}(\mathcal{R}_t) + \mathbb{E}(\mathcal{Q}_t) + \mathbb{E}(\mathcal{S}_t)} \tag{1}$$

$$\mathcal{T}_i = \frac{\mathcal{T}}{\mathcal{K}} \tag{2}$$

To estimate $\mathbb{E}(\mathcal{R}_t)$ and $\mathbb{E}(\mathcal{Q}_t)$, we assume that an arriving thread observes the system in equilibrium with itself removed (basically we assume that the arrival theorem [11] still holds for general service time distribution).

With this assumption, the utilisation of queue i,

$$u_i(\mathcal{N}-1) = \mathbb{E}(\mathcal{S}_t)\frac{\mathcal{T}\frac{\mathcal{N}-1}{\mathcal{N}}}{\mathcal{K}}$$

provides the probability of encountering a thread in service at the arrival instant. Let the expected number of threads waiting for *access-point* i at a random time be $\mathcal{Q}_i(\mathcal{N})$, and expected number of threads waiting for *access-point* i at the instant of an arrival be $\mathcal{A}_i(\mathcal{N})$. Based on the linearity of expectation, the expected number of threads waiting in the queue i is then:

$$\mathcal{A}_i(\mathcal{N}) - u_i(\mathcal{N}-1), \text{where } \mathcal{A}_i(\mathcal{N}) = \mathcal{Q}_i(\mathcal{N}-1).$$

We compute:

$$\mathbb{E}(\mathcal{Q}_t) = (\mathcal{Q}_i(\mathcal{N}-1) - u_i(\mathcal{N}-1))\mathbb{E}(\mathcal{S}_t) \tag{3}$$

Theorem 1. *Given that a thread arrives while a job (thread) is under service, the residual time for the job is given by* $\mathbb{E}(\mathcal{S}_t)\frac{1+c_S}{2}$, *where* c_S *is the coefficient of variation for service time distribution.*

Proof. Let $f_{\mathcal{S}_t}(x)$ denote the probability density function for the random variable $\mathcal{S}_t$ (service time). Given that a thread arrives at a random time during a service period of length X, the expected residual time is given by:

$$\mathbb{E}(\mathcal{R}_t|\text{arrival during a job of length X}) = \frac{X}{2}$$

We can remove the conditioning on $\mathbb{E}(\mathcal{R}_t)$ by first considering the fraction of time that a job of length X is under service, which can be calculated as $\frac{Xf_{\mathcal{S}_t}(X)}{\mathbb{E}(\mathcal{S}_t)}$ and then integrating over all possible lengths of the service time as follows:

$$\begin{aligned}\mathbb{E}(\mathcal{R}_t) &= \int_0^\infty \frac{x}{2}\frac{xf_{\mathcal{S}_t}(x)}{\mathbb{E}(\mathcal{S}_t)}dx \\ &= \frac{1}{2\mathbb{E}(\mathcal{S}_t)}\int_0^\infty x^2 f_{\mathcal{S}_t}(x)dx = \frac{\mathbb{E}(\mathcal{S}_t^2)}{2\mathbb{E}(\mathcal{S}_t)} \\ &= \frac{Var(\mathcal{S}_t) + \mathbb{E}(\mathcal{S}_t)^2}{2\mathbb{E}(\mathcal{S}_t)} = \mathbb{E}(\mathcal{S}_t)\frac{1+c_S^2}{2}\end{aligned}$$

□

Given that there is a thread in service at the arrival instant, and that arriving thread arrives at a random time during the service period, the residual time is then given as follows, relying on Theorem 1:

$$\mathbb{E}(\mathcal{R}_t) = u_i(\mathcal{N}-1)\mathbb{E}(\mathcal{S}_t)\frac{1+c_S^2}{2} \tag{4}$$

Finally, we plug Equations 3 and 4 into Equation 1 to obtain the system throughput, and then the throughput of queue i using Equation 2. We have represented the utilisation of a queue as a function of system throughput ($u_i(\mathcal{N}-1) = \mathbb{E}(\mathcal{S}_t)\frac{\mathcal{T}\frac{\mathcal{N}-1}{\mathcal{N}}}{\mathcal{K}}$) which leads to a quadratic equation for the system throughput as follows:

$$\begin{aligned}\mathcal{T}_i &= \frac{\mathcal{N}}{\mathcal{K}}\frac{1}{\mathbb{E}(\mathcal{R}_t) + \mathbb{E}(\mathcal{Q}_t) + \mathbb{E}(\mathcal{S}_t)} \\ &= \frac{\mathcal{N}}{\mathcal{K}}\frac{1}{u_i(\mathcal{N}-1)\mathbb{E}(\mathcal{S}_t)\frac{1+c_S^2}{2} + (\mathcal{Q}_i(\mathcal{N}-1) - u_i(\mathcal{N}-1))\mathbb{E}(\mathcal{S}_t) + \mathbb{E}(\mathcal{S}_t)} \\ &= \frac{\mathcal{N}}{\mathcal{K}}\frac{1}{(\mathbb{E}(\mathcal{S}_t)\frac{\mathcal{T}\frac{\mathcal{N}-1}{\mathcal{N}}}{\mathcal{K}})\mathbb{E}(\mathcal{S}_t)\frac{1+c_S^2}{2} + (\frac{\mathcal{N}-1}{\mathcal{K}} - (\mathbb{E}(\mathcal{S}_t)\frac{\mathcal{T}\frac{\mathcal{N}-1}{\mathcal{N}}}{\mathcal{K}}))\mathbb{E}(\mathcal{S}_t) + \mathbb{E}(\mathcal{S}_t)}\end{aligned}$$

$$\begin{aligned}\mathcal{T} &= \frac{\mathcal{N}}{(\mathbb{E}(\mathcal{S}_t)\frac{\mathcal{T}\frac{\mathcal{N}-1}{\mathcal{N}}}{\mathcal{K}})\mathbb{E}(\mathcal{S}_t)\frac{1+c_S^2}{2} + (\frac{\mathcal{N}-1}{\mathcal{K}} - (\mathbb{E}(\mathcal{S}_t)\frac{\mathcal{T}\frac{\mathcal{N}-1}{\mathcal{N}}}{\mathcal{K}}))\mathbb{E}(\mathcal{S}_t) + \mathbb{E}(\mathcal{S}_t)} \\ \Rightarrow \mathcal{T}^2\frac{\mathcal{N}-1}{\mathcal{N}\mathcal{K}}\mathbb{E}(\mathcal{S}_t)^2\frac{c_S^2-1}{2} &+ \mathcal{T}\mathbb{E}(\mathcal{S}_t)(\frac{\mathcal{N}-1}{\mathcal{K}}+1) - \mathcal{N} = 0\end{aligned}$$

When $c_S^2 = 1$, the quadratic term vanishes and the throughput is given by:

$$\mathcal{T} = \frac{\mathcal{N}}{\mathbb{E}(\mathcal{S}_t)(\frac{\mathcal{N}-1}{\mathcal{K}}+1)} \tag{5}$$

For $c_S^2 \neq 1$, we rely on Theorem 2.

Theorem 2. *The quadratic equation has at most one solution that obeys* $0 \leq u_i(\mathcal{N}) \leq 1$.

Proof. First, we show that, if $c_S^2 \neq 1$, the quadratic equation ($a\mathcal{N}^2 + b\mathcal{N} + c = 0$) has two distinct solutions since $b^2 - 4ac > 0$, where $a = \frac{\mathcal{N}-1}{\mathcal{N}\mathcal{K}}\mathbb{E}(\mathcal{S}_t)^2\frac{c_S^2-1}{2}$, $b = \mathbb{E}(\mathcal{S}_t)(\frac{\mathcal{N}-1}{\mathcal{K}}+1)$ and $c = -\mathcal{N}$.

We have $\mathcal{N} \geq 1$, then we obtain:

$$\begin{aligned}b^2 - 4ac &= \mathbb{E}(\mathcal{S}_t)^2(\frac{\mathcal{N}-1}{\mathcal{K}}+1)^2 + 4\frac{\mathcal{N}-1}{\mathcal{N}\mathcal{K}}\mathbb{E}(\mathcal{S}_t)^2\frac{c_S^2-1}{2}\mathcal{N} \\ &= \mathbb{E}(\mathcal{S}_t)^2(\frac{\mathcal{N}-1}{\mathcal{K}})^2 + 2\frac{\mathcal{N}-1}{\mathcal{K}}+1) \\ &\quad + 2\frac{\mathcal{N}-1}{\mathcal{K}}\mathbb{E}(\mathcal{S}_t)^2c_S^2 - 2\frac{\mathcal{N}-1}{\mathcal{K}}\mathbb{E}(\mathcal{S}_t)^2 \\ &= \mathbb{E}(\mathcal{S}_t)^2(\frac{\mathcal{N}-1}{\mathcal{K}})^2 + 1 + 2\frac{\mathcal{N}-1}{\mathcal{K}}c_S^2 > 0\end{aligned}$$

Now, we require that the utilization of queue i satisfies the condition $0 \leq u_i(\mathcal{N}) \leq 1$, where $u_i(\mathcal{N}) = \mathcal{T}_i\mathbb{E}(\mathcal{S}_t) = \frac{\mathcal{T}\mathbb{E}(\mathcal{S}_t)}{\mathcal{K}}$. We consider two cases. For $c_S^2 > 1$, the quadratic equation has a positive ($S^{(0)}$) and a negative ($S^{(1)}$) solution. We reject the negative solution. For $c_S^2 < 1$, the equation has two positive solutions, denoted by $S^{(0)} = \frac{-b-\sqrt{b^2-4ac}}{2a}$ and $S^{(1)} = \frac{-b+\sqrt{b^2-4ac}}{2a}$. We reject the solution

$S^{(1)}$ because setting $\mathcal{T} = S^{(1)}$ leads to $u_i(\mathcal{N}) > 1$:

$$u_i(\mathcal{N}) = \mathcal{T}\frac{\mathbb{E}(\mathcal{S}_t)}{\mathcal{K}} = S^{(1)}\frac{\mathbb{E}(\mathcal{S}_t)}{\mathcal{K}}$$
$$= \frac{-b+\sqrt{b^2-4ac}}{2a}\frac{\mathbb{E}(\mathcal{S}_t)}{\mathcal{K}}$$
$$= \frac{-(\mathbb{E}(\mathcal{S}_t)(\frac{\mathcal{N}-1}{\mathcal{K}}+1)) + \sqrt{\mathbb{E}(\mathcal{S}_t)^2(\frac{\mathcal{N}-1}{\mathcal{K}})^2+1+2\frac{\mathcal{N}-1}{\mathcal{K}}c_S^2}}{2(\frac{\mathcal{N}-1}{\mathcal{N}\mathcal{K}}\mathbb{E}(\mathcal{S}_t)^2\frac{c_S^2-1}{2})}\frac{\mathbb{E}(\mathcal{S}_t)}{\mathcal{K}}$$
$$= \frac{(\frac{\mathcal{N}-1}{\mathcal{K}}+1)+\sqrt{(\frac{\mathcal{N}-1}{\mathcal{K}})^2+1+2\frac{\mathcal{N}-1}{\mathcal{K}}c_S^2}}{\frac{\mathcal{N}-1}{\mathcal{N}}(1-c_S^2)}$$
$$> \frac{1}{\frac{\mathcal{N}-1}{\mathcal{N}}(1-c_S^2)} > 1$$

For both cases, we have at most one solution ($S^{(0)}$) that might obey the utilization condition. □

Based on Theorem 2, we set $\mathcal{T} = S^{(0)}$, if it obeys the utilization condition.

4.2 Memory Latency

In this section, we model the memory latency for accessing an *access-point*. As discussed in Section 3.4, latency mainly depends on the location of the *access-point* data within the memory hierarchy and the location of the thread that requests for the data. A multi-core memory system is typically composed of several cache levels, some private to the core and others shared. The cache at the first level (L1), provides the lowest latency but smallest in size. As one goes to the higher levels of the hierarchy, the latency grows together with the size of the cache. Here, we consider each thread to be pinned to one core (one-to-one mapping).

When a thread (let say P_x) accesses an *access-point*, the *access-point* data (let say D_d) is delivered to its L1 cache. Data delivery can lead to two events. Firstly, if the capacity of L1 is full, some data (let say D_e) will be evicted to create space for the delivered D_d. Data eviction can lead to a cache *capacity-miss*, if P_x subsequently requests for D_e. Secondly, if D_d was fetched from a private cache of thread (let say P_y) and updated by P_x, the copy of D_d in the P_y cache will be invalidated. As a result, P_y will experience a cache *coherence-miss* if it subsequently accesses an *access-point* associated with D_d. P_y will have to fetch D_d from P_x cache or shared memory.

To model cache access patterns and subsequently estimate memory latency, we split the memory system into cache sub-sets, which we refer to as access domains. Domains are specific to each thread, for example, if $P_x \in (domain_i)$, then $domain_i$ can be a local domain for P_x but will be remote for threads $\notin (domain_i)$. A thread incurs a specific memory latency when accessing data that was last updated by a thread in a given domain, this can be local or remote due to a coherence-miss. Within each domain, data can be fetched from a different cache level considering the possibility of capacity-misses. We denote the cache level j in domain i as C_{ij}, and denote the memory latency of fetching data in a given cache coherence state s from C_{ij} as $\mathcal{L}_{ij}^s$. Typically $s \in (Modified, Exclusive, Shared, Invalidated)$ for multi-core/many-core systems.

Recall that threads access *access-points* uniformly at random, which gives a uniform random distribution of *access-points*' data across caches. Given a system with $\mathcal{K}$ number of shared *access-points*, $\mathcal{J}$ number of cache levels and $\mathcal{D}$ number of domains. We calculate the probability of fetching a given *access-point* data from C_{ij} as follows, where $\mathcal{K}_{ij}$ is the number of *access-points*' data in C_{ij}:

$$\mathbb{P}(C_{ij}) = \frac{\mathcal{K}_{ij}}{\mathcal{K}}$$

We then compute the expected overall memory latency incurred by a thread accessing given *access-points* as follows:

$$\mathbb{E}(\mathcal{M}_{latency}) = \sum_{\substack{0<i\leq\mathcal{D}\\0<j\leq\mathcal{J}}} \frac{\mathcal{K}_{ij}}{\mathcal{K}} \times \mathcal{L}_{ij}^s$$

Due to the symmetry of core/thread memory resources, the memory latency is similar for all threads.

4.3 Throughput Estimate

In this section, we estimate the throughput following our micro benchmark execution models, Algorithms 1 and 2. Let the time it takes to complete a data operation be $\mathcal{OP}_t$, the expected memory latency be $\mathbb{E}\left(\mathcal{M}_{latency}^l\right)$ and $\mathbb{E}\left(\mathcal{M}_{latency}^s\right)$ for fetching an exclusive local *access-point* and shared *access-point* data respectively. Also, let the probability of a thread accessing its exclusive local *access-point* be $\mathcal{H}$. We estimate the local service time as:

$$\mathbb{E}\left(\mathcal{S}_t^l\right) = \mathcal{H} \times \left(\mathbb{E}\left(\mathcal{M}_{latency}^l\right) + \mathcal{OP}_t\right)$$

A processor will only access a shared *access-point* when it is not accessing its exclusive local *access-point* (with probability: 1-$\mathcal{H}$). Therefore, the expected service time distribution for shared *access-points* is given by:

$$\mathbb{E}\left(\mathcal{S}_t^s\right) = (1-\mathcal{H}) \times \left(\mathbb{E}\left(\mathcal{M}_{latency}^s\right) + \mathcal{OP}_t\right)$$

Threads accessing their exclusive local *access-points*, reduce the number of threads contending (queuing) for the shared *access-points*, which in return reduces the expected queue size distribution. From Equation 5, the general queue size distribution is given by $\frac{\mathcal{N}-1}{\mathcal{K}}$. To incorporate local accesses, we compute the expected queue size distribution as:

$$\mathbb{E}(Q_s) = (1-\mathcal{H}) \times \frac{\mathcal{N}-1}{\mathcal{K}}$$

Let the side-work execution time be $\mathcal{W}_t$ and *access-point* search execution time be $\mathcal{A}_t$. In addition to $\mathcal{S}_t^l$ and $\mathcal{S}_t^s$, $\mathcal{W}_t$ and $\mathcal{A}_t$ contribute to the overall thread throughput. Replacing $\frac{\mathcal{N}-1}{\mathcal{K}}$ with $\mathbb{E}(Q_s)$ in Equation 5, we estimate the throughput of our execution model Algorithms 1 and 2 in Section 4.3.1 and 4.3.2 respectively bellow, where $\mathcal{N}$ is the number of threads.

*4.3.1 Execution Model Algorithm 1 (*FAA*).* *FAA* memory updates always succeed on first attempt, in other words, a thread executing a *FAA* instruction will always complete its data operation on acquiring access to a given *access-point*. This implies that all queuing threads will complete their operations (will be serviced) before rejoining the queue.

$$\mathcal{T}^{faa} = \frac{\mathcal{N}}{\left(\mathbb{E}\left(\mathcal{S}_t^s\right)\left(\mathbb{E}\left(Q_s\right)+1\right)\right)+\mathbb{E}\left(\mathcal{S}_t^l\right)+\mathcal{A}_t+\mathcal{W}_t}$$

*4.3.2 Execution Model Algorithm 2 (*CAS*).* Unlike *FAA* where a thread always succeeds on acquiring access to a given *access-point*. A thread executing a *CAS* instruction on an *access-point* can fail to perform its data operation. Typically, a thread that fails retries by searching for another *access-point* in a retry loop as shown in Algorithm 2 (Line 2.12). A retrying thread has to exit the *Atomic execution* by releasing the *access-point* and retry on a randomly selected *access-point* as illustrated in Figure 1.

Although *CAS* has fail retries, there will always be one thread that succeeds in each retry loop. With this guarantee, the queue size will reduce by one until all the threads initially in the queue have succeeded. Using Arithmetic progression, we calculate the expected *CAS* fail queue distribution as follows:

$$\mathbb{E}\left(Q_s^{cas}\right) = \left(\frac{1+\mathbb{E}\left(Q_s\right)}{2}\right) \times \mathbb{E}\left(Q_s\right)$$

A thread will only fail on a CAS try when accessing a shared *access-point* (1-$\mathcal{H}$). Also, for every *CAS* try, a thread has to fetch data for the given *access-point* from within the memory hierarchy. Therefore, the expected time distribution wasted on *CAS* fails is given by:

$$\mathbb{E}\left(\mathcal{F}_t\right) = (1-\mathcal{H}) \times \mathbb{E}\left(Q_s^{cas}\right) \times \mathbb{E}\left(\mathcal{M}_{latency}^s\right)$$

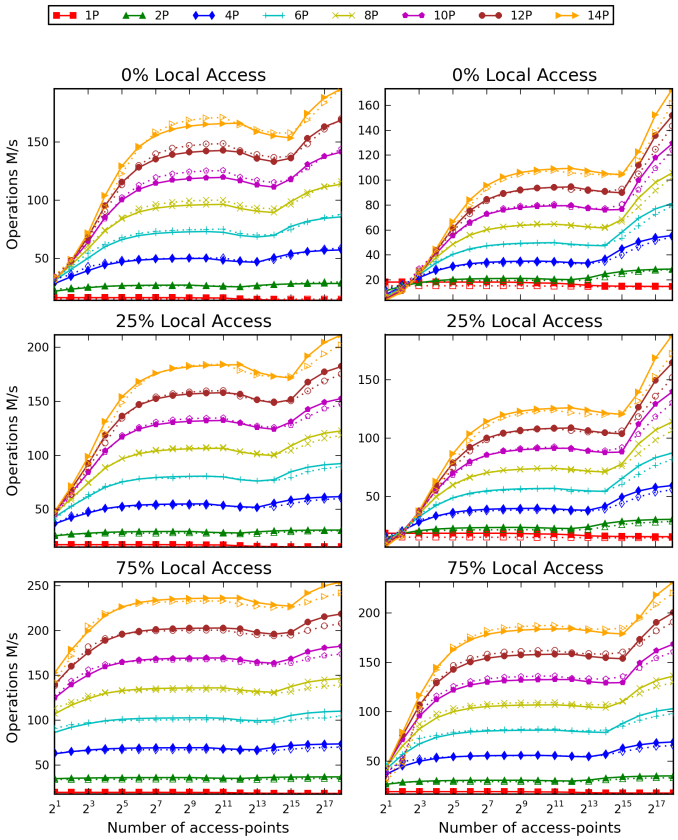
This implies that, a thread has to wait for a duration equivalent to $\mathbb{E}\left(\mathcal{F}_t\right)$, plus, a succeeding thread per retry, before it succeeds. Incorporating the CAS retries, we then estimate the throughput of Algorithms 2 as follows:

$$\mathcal{T}^{cas} = \frac{\mathcal{N}}{\left(\mathbb{E}\left(\mathcal{S}_t^s\right)\left(\mathbb{E}\left(Q_s\right)+1\right)\right)+\mathbb{E}\left(\mathcal{F}_t\right)+\mathbb{E}\left(\mathcal{S}_t^l\right)+\mathcal{A}_t+\mathcal{W}_t}$$

5 EVALUATION

In this section, we discuss how we obtain the system and algorithmic arguments that we use to validate and evaluate the accuracy of our model. Firstly we validate our model and present several insights into the performance of different access patterns and the accuracy of our predictions. Consequently, we evaluate the accuracy of our model on three random choice, multi-access, semantically relaxed data structures, including a counter, a stack and a FIFO queue [5, 31, 32]. Our discussion also shows how our model can be used in other applications apart from the data structures that we used.

The validation and evaluation experiments are conducted on two machines with different hardware configurations:



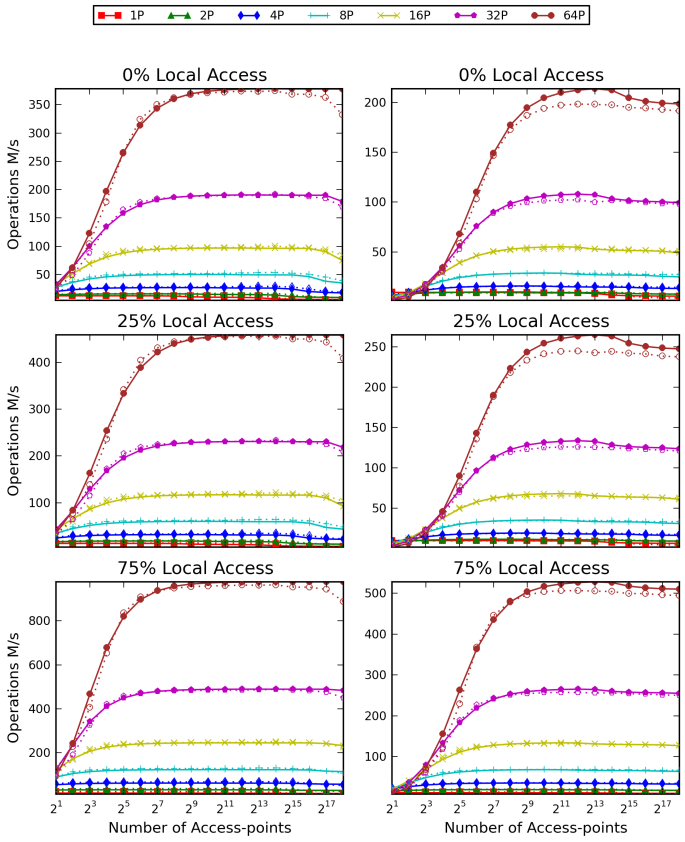
(a) *FAA* execution model **(b) *CAS* execution model**

Figure 2: General model validation, *Xeon* CPU. Solid lines present the model throughput prediction, dashed lines present the actual measured throughput, P stands for thread(s).

(1) Intel Xeon CPU E5-2695 v4 @ 2.3 GHz (*Xeon*). Has 14 physical cores with three cache levels; 32KB L1 and 256KB L2 private to each core and 35MB L3 shared among the 14 cores.
(2) Intel Xeon Phi CPU 7290 v2 @ 1.5 GHz (*XeonPhi*). Has 36 tiles with two cache levels; 32KB L1 and 1MB L2 private to each tile. Each tile has two physical cores sharing the tile L2 cache making a total 72 physical cores.

Our benchmarks were compiled using GCC 8.2.0 at O3 optimisation level and threads are pinned one per physical core/tile. However for 64 threads on *XeonPhi*, two threads share a tile with each thread pinned on an individual tile core. Throughput is measured as an average of the number of successful access operations performed per second out of five runs per benchmark.

Figures 2, 4, 3 and 5 present the measured and predicted throughput for the respective machines. The x-axis provides the number of *access-points* while the y-axis provides the throughput in form of million operations per second (M/s). The solid lines present the model throughput prediction, whereas the dashed lines present



(a) *FAA* execution model **(b) *CAS* execution model**

Figure 3: General model validation validation, *XeonPhi* CPU. Solid lines present the model prediction, whereas the dashed lines present the actual measured throughput. P stands for thread(s).

the actual measured throughput. Plot colours and markers represent different number of concurrent threads P as indicated in the respective legends.

5.1 System Parameters

Both *Xeon* and *XeonPhi* use write-back inclusive cache and a MESI (Modified, Exclusive, Shared, Invalid) cache coherence protocol [26]. Processor frequencies are fixed to 2.3 GHZ for *Xeon* and 1.5 GHZ for *XeonPhi* for all the experiments. The memory system is divided into three access domains; local, local-shared and remote. For a given thread, a local domain contains its core private cache levels and the rest of the caches are remote if not shared with the given thread. The local-shared domain contains cache levels shared between cores, in the case of *Xeon*, L3 belongs to the shared domain common to all the available 14 cores, whereas for *XeonPhi*, L2 belongs to a limited local-shared domain for two cores within the same tile. This means that for *XeonPhi*, L2 of a given tile is remote for threads not belonging to the given tile.

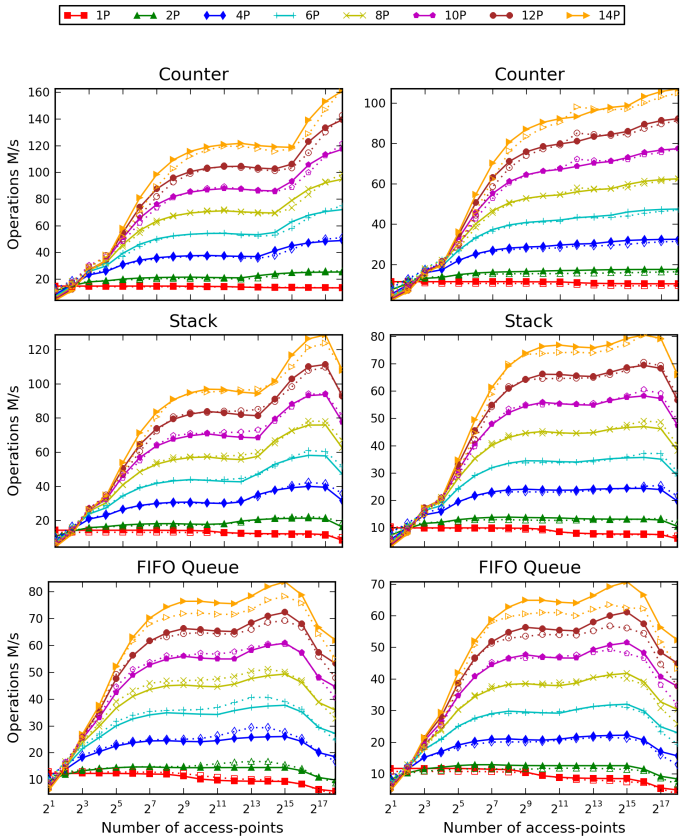
Each *access-point* is aligned to a full cache-line of 64Bytes to avoid false sharing and simplify data location and coherence state estimates. We estimate the data transfer latency between domains ($\mathcal{L}^s_{ij}$) using BenchIT [28], a commonly used tool to measure cache-to-cache transfer latency [20, 21, 29, 30, 33]. Data is transferred in form of cache-lines within the memory system, and cache-lines can be in either of the four MESI coherence states for both *Xeon* and *XeonPhi*. In our case, for localised accesses, *access-point* cache-lines will mostly be in a modified state, therefore we use the measured data transfer latency for a cache-line in modified state. Whereas for remote accesses, *access-point* cache-lines will transition between the four different MESI coherence states, making it hard to predict the cache-line coherence state at any given point in time. We therefore measure the cache transfer latency for the four different MESI coherence states, and calculate the average as the overall remote access cache-line transfer latency.

We estimate the execution duration of each atomic operation, by running the algorithms with a single thread executing the atomic operation locally. Subtracting the measured L1 cache latency, we obtain the operation time ($\mathcal{OP}_t$). Recall from Figure 1, $\mathcal{OP}_t$ is part of the atomic execution.

In systems where there might be external interfere such as data transfer or thread interference (for example thread preemption), the model provides parameters such as memory latency or side-work that can be used to add the estimated interference respectively. We also note that the cache levels are in most cases shared with other processes running on the computing system. It is almost impossible to measure the available cache space dedicated to the algorithm execution. However, by monitoring the system load, we are able to estimate the average cache space being utilised by the given algorithm. With this information, we can then calculate the maximum number of *access-points* that can reside within a given cache level, and then be able to estimate the number of *access-points* in a given cache level for a given access domain at any point in time ($\mathcal{K}_{ij}$).

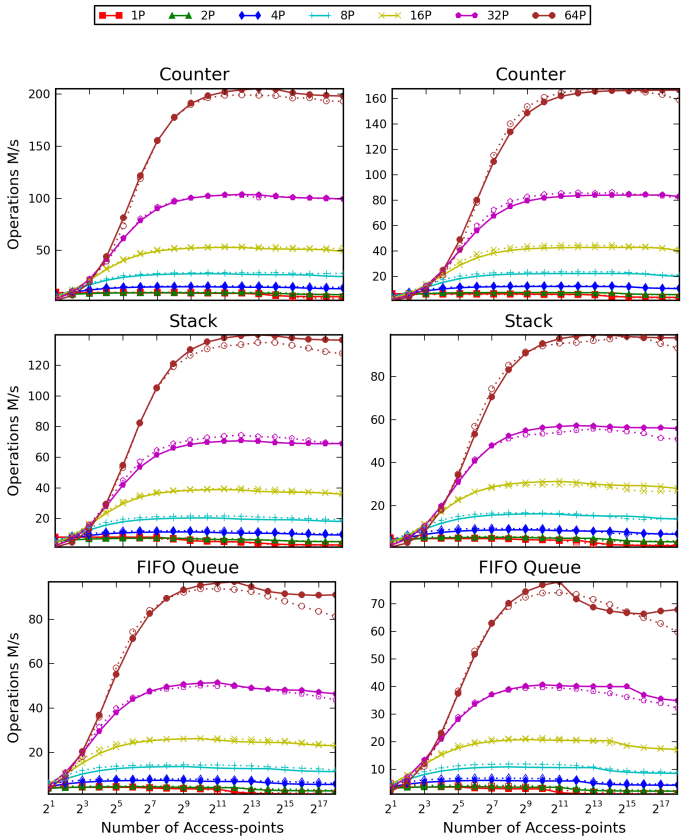
5.2 Algorithmic Parameters

Data structures (FIFO queue and the Stack) are initiated with 2^{18} items for each experiment, meaning that part of the cache capacity is consumed by the item's data. A thread randomly decides whether to add or remove an item from the data structure with a 50% chance for either operation. This implies that approximately each data structure maintains the same initial number of items (2^{18}). The algorithms are each composed of multiple instantiations (sub-structures) of a given data structure, with each instantiation being assigned an independent *access-point*. A thread has to select an *access-point* either using the random choice of one or random choice of two access pattern, an execution step we measure as the search cost. Under random choice of one, a thread selects one *access-point* uniformly at random and proceeds to try and acquire the given *access-point*. Whereas under random choice of two, a thread selects two *access-points* uniformly at random, then selects the most semantically correct *access-point* from the two given *access-points* depending on the data structure semantics. Using a stack as an example, a thread performing a pop operation will select from among the two *access-points*, an *access-point* with the most recently pushed



(a) Random choice of one (b) Random choice of two

Figure 4: Data structure model validation, *Xeon* CPU. Solid lines present the model prediction, whereas the dashed lines present the actual measured throughput. P stands for thread(s).



(a) Random choice of one (b) Random choice of two

Figure 5: Data structures model validation, *XeonPhi* CPU. Solid lines present the model prediction, whereas the dashed lines present the actual measured throughput. P stands for thread(s).

(added) item. Apart from the counter, a thread has to also fetch item data when accessing a given *access-point*, a cost we add to the memory latency parameter.

We assume a uniform distribution of data among threads since *access-points* are accessed uniformly at random. This also means that *access-points* are distributed uniformly within the different core cache levels. Therefore we can estimate the location of both the *access-point* ($\mathcal{K}_{ij}$) and the item data within the memory hierarchy since a fixed number of items are maintained for each data structure. However, this can be different in real life where the access of the data structure is not uniformly distributed at random. In this case, as an extension, a specific location model estimating the location of *access-points* ($\mathcal{K}_{ij}$) can be added to our model for specific data structure designs. To estimate the size of side-work ($\mathcal{W}_t$), we run the algorithms without executing the *access-point* requests.

5.3 Results

Generally, we observe that our model is able to predict throughput on both hardware platforms for all the benchmarks with accuracy between 80% to 100%, for all experiments. Figures 2 and 3 present results for our *FAA* and *CAS* execution models for the purposes of validating our model in terms of local vs remote accesses. In our execution models, captured by Algorithm 1 and 2, we vary the local operations to give an analytical insight into the execution behaviour of techniques such as, work stealing [39], local linearisation [18] and 2D relaxation [32] that allow threads to alternate between operating locally and accessing global variables. To validate our local operations analysis, we vary the probability of a thread accessing its local *access-point* (from 0% to 75% Local Access). As the percentage of local accesses increases, we observe an increase in throughput. Our model is able to predict this increase due to the incorporation of local access probabilities, which is part of the service time that was split into local ($\mathcal{S}_t^l$) and shared ($\mathcal{S}_t^s$). In our analysis we predict that when accesses to the local *access-point* increase, contention on the shared *access-points* reduce, subsequently reducing the queue time ($\mathbb{E}(Q_s)$). Also when operating locally, the processor is fetching *access-point* data from the local/local-shared domain, incurring less memory latency ($\mathbb{E}\left(\mathcal{M}_{latency}^l\right)$) than when fetching data from the

remote domain ($\mathbb{E}\left(\mathcal{M}^{s}_{latency}\right)$). Locality is especially helpful in the contention prone executions, where the number of *access-points* is less than the number of threads as observed in Figures 2 and 3. Having more threads than *access-points* ($\mathcal{N} < \mathcal{K}$) means that threads will more often try to access the same *access-point* at the same time leading to contention.

Another observable difference is the lower throughput performance of the *CAS* execution model (Figure 2b and 3b) as compared to the *FAA* one (Figure 2a and 3a). This is because of the *CAS* retry loop that expands the thread waiting queue time. Our model is able to predict this difference using the estimated number of *CAS* fails (Q^{cas}_{s}) that we incorporate in the general model. Another factor that plays role in this performance gap, is the double access request incurred by the *CAS* algorithm. In our model, this effect is integrated in the *CAS* fail memory latency parameter since a thread that succeeds does not pay an extra cost for the read. This observation shows that instruction level latency [34] in itself is not enough, when comparing the different atomic primitives for the purpose of predicting concurrent data structure performance.

We vary the number of threads to evaluate the model prediction when multiple concurrent threads are accessing data from different access domains. $1P$, which stands for one thread, shows results for a single thread. For a single thread execution, we do not expect coherence-misses but expect capacity-misses as the number of *access-points* increases. From the results, we observe a reduction in throughput as the number of *access-points* increases, and our model predicts accurately the behaviour observed. This behaviour is attributed to capacity-misses that force the thread to fetch data from slower cache levels as the number of *access-points* increase beyond what the capacity of the smaller faster cache levels can hold. This behaviour is more apparent in data intensive data structures such as the FIFO queue that has double the *access-points* (head and tail) as compared to the counter and the stack (Figure 4 and 5).

Now consider the situation when multiple threads are active concurrently. As the number of *access-points* increases, there is an increase in throughput until a saturation point, where the queuing time ($\mathbb{E}\left(Q_s\right)$) is negligible. Based on queue time analysis, our model is able to predict the increase in throughput and the saturation points accurately. Higher numbers of *access-points* lead to data evictions from the small private cache levels to the larger shared L3 for the case of *Xeon* CPU. Although L3 is the slowest cache level when accessed locally, it is faster to fetch data from L3 than from a remote L1 or L2 as observed in Figure 2 and 4. Our model is able to predict this behaviour because our memory analysis incorporates data evictions by considering cache level capacities and domains. Predicting the saturation point is especially important for semantically relaxed data structures. Beyond the saturation point, data structure quality keeps dropping without enough throughput gain to justify the trade-off [31, 32].

Comparing the different data structures, we observe that the cache capacity is utilised differently. This is due to the difference in data intensity. The counter has to fetch a single cache-line to complete a counter operation, whereas a stack has to fetch two cache-lines, one for the top of the stack *access-point* and another for the stack item. The FIFO queue is the most data intensive of the three data structures, fetching three cache-lines for the *deque* operation, that is, the head of the queue *access-point*, tail of the queue *access-point* and the queue item cache-lines. This explains the difference in cache capacity saturation points. For example we observe that the FIFO queue saturates the last level cache (L3 for *Xeon* at around 2^{16} *access-points* and L2 for *XeonPhi* at around 2^{11} *access-points*) earlier than both the stack and the counter, dropping its throughput as it accesses off chip memory for both *Xeon* and *XeonPhi* for high number of *access-points* ($\mathcal{K}_{i3} > L3$, $\mathcal{K}_{i2} > L2$ respectively).

Also our model correctly predicts the performance difference between the two access patterns with choice of one performing better than choice of two. This is due to the higher search cost ($\mathcal{A}_t$) for the choice of two access pattern, where a thread has to fetch two random *access-points* from which to select an *access-point*.

6 CONCLUSION

In this paper, we have analysed and modelled the throughput performance of concurrent multi-accesses of lock-free data structures in multi-core/many-core shared memory systems. Multi-access techniques have been proposed and introduced in the design of concurrent data structures in order to improve their throughput performance and scalability and as expected they play a significant role in their overall performance. We considered multi-access techniques that typically use two types of memory access patterns, locally and remotely. We considered two classes of atomic operations: Repeat until Condition (Compare and Swap) and Atomically Modify (Fetch and Add), that are the typical atomic primitives used in the design of lock-free data structures. We have modelled the acquisition of a data structure *access-point* in memory, as a system of queuing networks with parallel servers, where each server corresponds to an *access-point*. We also model memory latency in terms of cache location and data coherence status. For the validation and evaluation, we have predicted the throughput performance of a multi-access micro benchmark and a set of multi-access semantically relaxed concurrent data structures (counter, stack and FIFO queue) using two contemporary hardware platforms. The results show that our model follows closely the actual execution behaviour without significant deviations independently of the number of *access-points* or concurrent threads used.

ACKNOWLEDGMENTS

This work has been supported by SIDA/Bright Project (317) under the Makerere-Sweden bilateral research programme 2015-2022, Mbarara University of Science and Technology, Swedish Research Council (Vetenskapsrådet) Under Contract No.: 2021-05443 and The Swedish Foundation for International Cooperation in Research and Higher Education (STINT), grant no. SG2021-8934.

REFERENCES

[1] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. 2010. Scalable Producer-Consumer Pools Based on Elimination-Diffraction Trees. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II* (Ischia, Italy) *(Euro-Par'10)*. Springer-Verlag, Berlin, Heidelberg, 151–162.

[2] Yehuda Afek, Guy Korland, and Eitan Yanovsky. 2010. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In *Proceedings of the 14th International Conference on Principles of Distributed Systems* (Tozeur, Tunisia) *(OPODIS'10)*. Springer-Verlag, Berlin, Heidelberg, 395–410.

[3] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Zheng Li, and Giorgi Nadiradze. 2018. Distributionally Linearizable Data Structures. In *Proceedings of the*

30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018, Christian Scheideler and Jeremy T. Fineman (Eds.). ACM, New York, NY, USA, 133–142. https://doi.org/10.1145/3210377.3210411

[4] Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. 2018. Relaxed Schedulers Can Efficiently Parallelize Iterative Algorithms. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, Calvin Newport and Idit Keidar (Eds.). ACM, New York, NY, USA, 377–386. https://dl.acm.org/citation.cfm?id=3212756

[5] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. 2017. The Power of Choice in Priority Scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. ACM, New York, NY, USA, 283–292. https://doi.org/10.1145/3087801.3087810

[6] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A scalable relaxed priority queue. *ACM SIGPLAN Notices* 50, 8 (2015), 11–20.

[7] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. 2015. Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9363)*, Yoram Moses (Ed.). Springer, 341–355. https://doi.org/10.1007/978-3-662-48653-5_23

[8] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. 2011. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. *SIGPLAN Not.* 46, 1 (Jan. 2011), 487–498. https://doi.org/10.1145/1925844.1926442

[9] Gal Bar-Nissan, Danny Hendler, and Adi Suissa. 2011. A Dynamic Elimination-Combining Stack Algorithm. In *Principles of Distributed Systems*, Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 544–561.

[10] Naama Ben-David and Guy E. Blelloch. 2017. Analyzing Contention and Backoff in Asynchronous Shared Memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/3087801.3087828

[11] Richard J. Boucherie and Nico M. van Dijk. 1997. On the arrivai theorem for product form queueing networks with blocking. *Performance Evaluation* 29, 3 (1997), 155–176. https://doi.org/10.1016/S0166-5316(96)00045-4

[12] Daniel Cederman, Bapi Chatterjee, Nhan Nguyen Dang, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2013. A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*. IEEE Computer Society, 1309–1320. https://doi.org/10.1109/IPDPS.2013.91

[13] Men-Chow Chiang and Gurindar S. Sohi. 1992. Evaluating Design Choices for Shared Bus Multiprocessors in a Throughput-Oriented Environment. *IEEE Trans. Comput.* 41, 3 (March 1992), 297–317. https://doi.org/10.1109/12.127442

[14] 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* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 33–48. https://doi.org/10.1145/2517349.2522714

[15] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. 1997. Contention in Shared Memory Algorithms. *J. ACM* 44, 6 (nov 1997), 779–805. https://doi.org/10.1145/268999.269000

[16] Faith Ellen, Danny Hendler, and Nir Shavit. 2012. On the Inherent Sequentiality of Concurrent Objects. *SIAM J. Comput.* 41, 3 (2012), 519–536. https://doi.org/10.1137/08072646X arXiv:https://doi.org/10.1137/08072646X

[17] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. 1998. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. *SIAM J. Comput.* 28, 2 (1998), 733–769. https://doi.org/10.1137/S009753979427491 arXiv:https://doi.org/10.1137/S009753979427491

[18] Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. 2016. Local Linearizability for Concurrent Container-Type Data Structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada (LIPIcs, Vol. 59)*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.6

[19] Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. 2013. Distributed Queues in Shared Memory: Multicore Performance and Scalability Through Quantitative Relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '13)*. ACM, New York, NY, USA, Article 17, 9 pages. https://doi.org/10.1145/2482767.2482789

[20] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. 2009. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. ACM, New York, NY, USA, 413–422. https://doi.org/10.1145/1669112.1669165

[21] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. 2009. Comparing Cache Architectures and Coherency Protocols on X86-64 Multicore SMP Systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. ACM, New York, NY, USA, 413–422. https://doi.org/10.1145/1669112.1669165

[22] Danny Hendler and Shay Kutten. 2006. Constructing Shared Objects That Are Both Robust and High-Throughput. In *Proceedings of the 20th International Conference on Distributed Computing* (Stockholm, Sweden) *(DISC'06)*. Springer-Verlag, Berlin, Heidelberg, 428–442. https://doi.org/10.1007/11864219_30

[23] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2010. A scalable lock-free stack algorithm. *J. Parallel and Distrib. Comput.* 70, 1 (2010), 1–12.

[24] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative Relaxation of Concurrent Data Structures. *SIGPLAN Not.* 48, 1 (Jan. 2013), 317–328. https://doi.org/10.1145/2480359.2429109

[25] Fazeleh Sadat 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, Kyoto, Japan, August 05-08, 2019*. ACM, 28:1–28:11. https://doi.org/10.1145/3337821.3337901

[26] Intel Corporation 2014. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

[27] Amos Israeli and Lihu Rappoport. 1994. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, California, USA) *(PODC '94)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/197917.198079

[28] G. Juckeland, S. Börner, M. Kluge, S. Kölling, W.E. Nagel, S. Pflüger, H. Röding, S. Seidl, T. William, and R. Wloch. 2004. BenchIT - Performance measurement and comparison for scientific applications. In *Parallel Computing*, G.R. Joubert, W.E. Nagel, F.J. Peters, and W.V. Walter (Eds.). Advances in Parallel Computing, Vol. 13. North-Holland, 501–508. https://doi.org/10.1016/S0927-5452(04)80064-9

[29] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E. Nagel. 2015. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*. IEEE Computer Society, 739–748. https://doi.org/10.1109/ICPP.2015.83

[30] Sabela Ramos and Torsten Hoefler. 2017. Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 297–306. https://doi.org/10.1109/IPDPS.2017.30

[31] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. ACM, New York, NY, USA, 80–82. https://doi.org/10.1145/2755573.2755616

[32] Adones Rukundo, Aras Atalar, and Philippas Tsigas. 2019. Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary (LIPIcs, Vol. 146)*, Jukka Suomela (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 31:1–31:15. https://doi.org/10.4230/LIPIcs.DISC.2019.31

[33] Andreas Sandberg, David Black-Schaffer, and Erik Hagersten. 2012. Efficient Techniques for Predicting Cache Sharing and Throughput. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. ACM, New York, NY, USA, 305–314. https://doi.org/10.1145/2370816.2370861

[34] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the Cost of Atomic Operations on Modern Architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 445–456. https://doi.org/10.1109/PACT.2015.24

[35] Nir Shavit. 2011. Data Structures in the Multicore Age. *Commun. ACM* 54, 3 (March 2011), 76–84. https://doi.org/10.1145/1897852.1897873

[36] Nir Shavit and Dan Touitou. 1995. Elimination Trees and the Construction of Pools and Stacks: Preliminary Version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures* (Santa Barbara, California, USA) *(SPAA '95)*. ACM, New York, NY, USA, 54–63. https://doi.org/10.1145/215399.215419

[37] Nir Shavit and Asaph Zemach. 2000. Combining funnels: a dynamic approach to software combining. *J. Parallel and Distrib. Comput.* 60, 11 (2000), 1355–1387.

[38] Edward Talmage and Jennifer L. Welch. 2017. Relaxed Data Types as Consistency Conditions. In *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10616)*, Paul G. Spirakis and Philippas Tsigas (Eds.). Springer, 142–156. https://doi.org/10.1007/978-3-319-69084-1_10

[39] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. 2015. The lock-free k-LSM relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, Albert Cohen and David Grove (Eds.). ACM, 277–278. https://doi.org/10.1145/2688500.2688547