CHALMERS
UNIVERSITY OF TECHNOLOGY

Lock-Free Search Data Structures: Throughput Modeling with Poisson Processes

Lock-Free Search Data Structures: Throughput Modeling with Poisson Processes

Aras Atalar
Chalmers University of Technology, S-41296 Göteborg, Sweden
aaras@chalmers.se

Paul Renaud-Goud
Informatics Research Institute of Toulouse, F-31062 Toulouse, France
Paul.Renaud.Goud@irit.fr

Philippas Tsigas
Chalmers University of Technology, S-41296 Göteborg, Sweden
philippas.tsigas@chalmers.se

Abstract

This paper considers the modeling and the analysis of the performance of lock-free concurrent search data structures. Our analysis considers such lock-free data structures that are utilized through a sequence of operations which are generated with a memoryless and stationary access pattern. Our main contribution is a new way of analyzing lock-free concurrent search data structures: our execution model matches with the behavior that we observe in practice and achieves good throughput predictions.

Search data structures are formed of basic blocks, usually referred to as nodes, which can be accessed by two kinds of events, characterized by their latencies; (i) CAS events originated as a result of modifications of the search data structure (ii) Read events that occur during traversals. An operation triggers a set of events, and the running time of an operation is computed as the sum of the latencies of these events. We identify the factors that impact the latency of such events on a multi-core shared memory system. The main challenge (though not the only one) is that the latency of each event mainly depends on the state of the caches at the time when it is triggered, and the state of caches is changing due to events that are triggered by the operations of any thread in the system. Accordingly, the latency of an event is determined by the ordering of the events on the timeline.

Search data structures are usually designed to accommodate a large number of nodes, which makes the occurrence of an event on a given node rare at any given time. In this context, we model the events on each node as Poisson processes from which we can extract the frequency and probabilistic ordering of events that are used to estimate the expected latency of an operation, and in turn the throughput. We have validated our analysis on several fundamental lock-free search data structures such as linked lists, hash tables, skip lists and binary trees.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Lock-free, Search Data Structures, Performance, Modeling, Analysis

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.9

1 Introduction

A search data structure is a collection of $\langle key, value \rangle$ pairs which are stored in an organized way to allow efficient search, delete and insert operations. Linked lists, hash tables, binary trees are some widely known examples. Lock-free implementations of such concurrent data

22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 9; pp. 9:1–9:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

structures are known to be strongly competitive at tackling scalability by allowing processors to operate asynchronously on the data structure.

Performance (here throughput, *i.e.* number of operations per unit of time) is ruled by the number of events in a search data structure operation (*e.g.* $O(\log \mathcal{N})$ for the expected number of steps in a skip list or a binary tree). The practical performance estimation requires an additional layer as the cost (latency) of these events need to be mapped onto the hardware platform; typical values of latency varies from 4 cycles for an access to the first level of cache, to 350 cycles for the last level of remote cache. To estimate the latency of events, one needs to consider the misses, which are sensitive to the interleaving of these events on the timeline. On the one hand, a capacity miss in data or TLB (Translation Lookaside Buffer) caches with LRU (Least Recently Used) policy arise when the interleaving of memory accesses evicted a cacheline. On the other hand, the coherence cache misses arise due to the modifications, that are often realized with *Compare-and-Swap* (*CAS*) instructions, in the lock-free search data structure. The interleaving of events that originate from different threads, determine the frequency and severity of these misses, hence the latencies of the events.

In the literature, there exist many asymptotic analyses on the time complexity of sequential search data structures and amortized analyses for the concurrent lock-free variants that involve the interaction between multiple threads. But they only consider the number of events, ignoring the latency. On the other side, there are performance analyses that aim to estimate the coherence and capacity misses for the programs on a given platform, with no view on data structures. We go through them in the related work. However, there is a lack of results that merge these approaches in the context of lock-free data structures to analytically predict their practical performance.

An analytical performance prediction framework could be useful in many ways: (i) to facilitate design decisions by providing an extensive understanding; (ii) to compare different designs in various execution contexts; (iii) to help the tuning process. On this last point, lock-free data structures come with specific parameters, *e.g.* padding, back-off and memory management related parameters, and become competitive only after picking their hopefully optimal values.

In this paper, we aim to compute the average throughput ($\mathcal{T}$) of search data structures for a sequence of operations, generated by a memoryless and stationary access pattern. Throughput is directly linked to the latency of operations. As the traversal of a search data structure is light in computation, the latency of an operation is dominated by the memory access costs to the nodes that belong to the path from the entry of the data structure to the targeted node.

Therefore, part of this paper is dedicated to the discovery of the route(s) followed by a thread on its way to reach any node in the data structure. In other words, what is the sequence of nodes that are accessed when a given key is targeted by an operation.

As the latency of an operation is the sum of the latency of each memory access to the nodes that are on the path, we obviously need to estimate the individual latency of each accessed node. Even if, in the end, we are interested in the average throughput, this part of the analysis cannot be satisfied with a high-level approach, where we would ignore which thread accesses which node across time. For instance, the cache, whose misses are expected to greatly impact throughput, should be taken carefully into account. This can only be done in a framework from which the interleaving of memory accesses among threads can be extracted. That is why we model the distribution of the memory accesses for every thread.

More precisely, a memory access can be either the read or the modification of a node, and two point distributions per node represent the triggering instant of either a *Read* or a *CAS*.

These point distributions are modeled as Poisson processes, since they can be approximated by Bernoulli processes, in the context of rare events. Knowing the probabilistic ordering of these events gives a decisive information that is used in the estimate of the access latency associated with the triggered event. Once this information is grabbed, we roll back to the expectation of the access of a node, then to the expectation of the latency of an operation.

We validate our approach through a large set of experiments on several lock-free search data structures that are based on various algorithmic designs, namely linked lists, hash tables, skip lists and binary trees. We feed our experiments with different key distributions, and show that our framework is able to predict and explain the observed phenomena.

The rest of the paper is organized as follows. We discuss related work in Section 2, then the problem is formulated in Section 3. We present our framework in Section 4 and analysis of throughput in Section 5. In Section 6, we show how to initiate our model by considering the particularity of different search data structures. Finally, we describe the experimental results in Sections 7.

2 Related Work

When it comes to estimating the performance of concurrent lock-free data structures, the focus of the previous work has been on studying the contention overhead that occurs due to the existence of concurrent operations which overlap in time and access the same shared memory locations. In such cases, the contention manifests in the form of stall times [13] due to hardware conflicts and extended operation execution times due to logical conflicts in the algorithmic level (*e.g.* re-execution of the retry loop iteration).

For the time complexity of lock-free search data structure operations, previous work considered asymptotic amortized analysis [16] since it is not possible to bound the execution time of a single operation, by definition. The analysis is parameterized with a measure of contention (*e.g.* point contention [5]) that bounds the extra cost of failed attempts that are billed to a successful operation. Also, it is common to model a concurrent execution as an adversary that schedules the steps of concurrent processes in order to analyze the impact of contention [8, 13]. These studies target theoretical worst-case execution times. Closer to the practical domain, the *expected* system and individual operation latencies are analyzed for a general class of lock-free algorithms under a uniform stochastic scheduler in [1].

In our previous work [2, 3], we aim at estimating the average throughput performance of some lock-free data structures, that is observed in practice. We consider a universal construction [20] of lock-free data structures in its practical use. Data structures that have inherent sequential bottlenecks (*e.g.* stacks, counters and queues) are targeted; the universal construction can not be exploited in practice for the *efficient* designs of search data structures since it inhibits the potential disjoint access parallelism.

Conversely, in this work, we study the performance of the efficient designs of lock-free search data structures. These designs employ fine-grained synchronization, in which the modifications are spread to many different shared memory locations. As a result of this characteristic, hardware (stall time) and logical conflicts between threads, that were playing a central role in our previous work, occurs very rarely for search data structures and the performance is driven by different impacting factors. The performance of concurrent lock-free search data structures is studied and investigated through empirical studies in [18, 11]. To the best of our knowledge, we attempt for the first time to model and analyze the performance of lock-free search data structures and obtain estimates that are close to what is observed in practice on top of actual hardware platforms.

Procedure AbstractAlgorithm

```
1 while ! done do
2 |   key ← SelectKey(keyPMF);
3 |   operation ← SelectOperation(operationPMF);
4 |   result ← SearchDataStructure(key, operation);
```

Figure 1 Generic framework.

On the other hand, various performance metrics for search data structures have been studied for the sequential setting. The search path length of skip lists is analyzed in [21]. In [12, 24], various performance shapers for the randomized trees are studied, such as the time complexity of operations, the expectation, and distribution of the depth of the nodes based on their keys. However, these studies are not concerned with the interaction between the algorithms and the hardware. The following approaches rely on the independent reference model (IRM) for memory references and derive theoretical results or performance analysis. In [15], the exact cache miss ratio is derived analytically (computationally expensive) for LRU caches under IRM. As an outcome of this approach, the cache miss ratio of a static binary tree is estimated by assigning independent reference probabilities to the nodes in [14].

3 Problem Statement

We describe in this section the structure of the algorithm and the system that is covered by our model. We target a multicore platform where the communication between threads takes place through shared memory accesses. The threads are pinned to separate cores and call AbstractAlgorithm (see Figure 1) when they are spawned.

A concurrent search data structure is a shared collection of data elements, each associated with a key, that support three basic operations holding a key as a parameter. `Search` (resp. `Insert`, `Delete`) operation returns (resp. inserts, deletes) the element if the associated key is present (resp. absent, present) in the search data structure, otherwise returns *null*.

The applications that use a search data structure can be seen as a sequence of operations on the structure, interleaved by application-specific code containing at least the key and operation selection, as reflected in AbstractAlgorithm.

The access pattern (*i.e.* the output of the key and operation selections) should be considered with care since it plays a decisive role in the throughput value. An application that always looks for the first element of a linked list will obviously lead to very high throughput rates. In this study, we consider a memoryless and stationary key and operation selection process *i.e.* such that for any operation in the sequence the probability of selecting a key (resp. an operation type) is a constant.

A search data structure is modeled as a set of basic blocks called nodes, which either contain a value (*valued nodes*) or routes towards nodes (*router nodes*). W.l.o.g. the key set can be reduced to $[1..\mathcal{R}]$, where $\mathcal{R}$ is the number of possible keys. We denote by $(N_i)_{i\in[1..\mathcal{N}]}$ the set of $\mathcal{N}$ potential nodes, and by K_i the key associated with N_i. Until further notice , we assume that we have exactly one node per cacheline.

An operation can trigger two types of events in a node. We distinguish these events as *Read* and *CAS* events. The latency of an event is based on the state of the hardware platform at the time that the event occurs, *e.g.* for a *Read* request, the level of the cache that a node belongs to. We summarize the parameters of our model as follows:

- *Algorithm parameters:* Expected latency of the application specific-code (interleaves data structure operations) t^{app}, local computational cost while accessing a node t^{cmp} (a constant cost of a few cycles for the key comparisons, local updates for pointer chasing), probability mass functions for the key and operation selection.
- *Platform parameters:* Cache hit latencies (resp. capacity) from level ℓ: t_ℓ^{dat} (resp. C_ℓ^{dat}) for the data caches and t_ℓ^{tlb} (resp. C_ℓ^{tlb}) for TLB caches; other memory instruction latencies (that depends on P): t^{cas} for a *CAS* execution and t^{rec} to recover from an invalid state (*Read* at an invalid cacheline, that is in Modified state in another thread's local cache); number of threads P.

4 Framework

4.1 Event Distributions

We consider first a single thread running AbstractAlgorithm on a data structure where only search operations happen, and we observe the distribution of the *Read* triggering events on a given node N_i. The execution is composed of a sequence of search operations, where each operation is associated with a set of accessed nodes, which potentially includes N_i. If we slice the time into consecutive intervals, where an interval begins with a call to an operation, we can model the *Read* events as a Bernoulli process (a successful Bernoulli trial implies that a *Read* event on N_i occurs in the respective operation), where the probability of having a *Read* event during an interval depends on the parameters of the associated operation (recall that the process that generates the operation parameters is stationary and memoryless).

Search data structures have been designed as a way to store large data sets while still being able to reach any node within a short time: the set of accessed nodes is then expected to be small compared to the total number of nodes. This implies that, given an operation, the probability that N_i belongs to the set of accessed nodes is small. Therefore we can map the Bernoulli process on the timeline with constant-sized interval of length $\mathcal{T}^{-1}$ instead of mapping it with the actual operation intervals: as the probability of having a *Read* event within an operation is small, the duration between two events is big, and this duration is close to the number of initial intervals within this duration, multiplied by $\mathcal{T}^{-1}$ (with high probability, because of the Central Limit Theorem).

When we increase the scope of the operations to insertion and deletion, the structure is no longer static and the probability for a node to appear in an interval is no longer uniform, since it can move inside the data structure. There exists a long line of research in approximating Bernoulli processes by Poisson point processes [7], in which not only the number events but also their respective locations on the timeline are approximated. In particular, [9] has dealt with non-uniform Bernoulli processes, in which the success probabilities of trials are not necessarily same. Their error bounds, which are proportional to the success probabilities, strengthen the use of Poisson processes in our context: the events on N_i are rare, thus the probabilities in Bernoulli trials are small and the approximation is well-conditioned.

Once *Read* and *CAS* triggering events are modeled as Poisson processes for a single thread, the merge (superposition) of several Poisson processes models the multi-thread execution.

Lastly, we specify a point on the dynamicity: since we have insertions and deletions, nodes can enter and leave the data structure. This is modeled by the masking random variable P_i which expresses the presence of N_i in the structure. At a random time, we denote by D the set of nodes that are inside the data structure, and P_i is set to 1 iff $N_i \in D$. We denote by p_i its probability of success ($p_i = \mathbb{P}[P_i = 1]$). Its evaluation will often rely on the probability that the last update operation on key k was an `Insert`; we denote it by q_k, and

$q_k = \mathbb{P}\left[Op = op_k^{ins}\right] / (\mathbb{P}\left[Op = op_k^{ins}\right] + \mathbb{P}\left[Op = op_k^{del}\right])$. Note that the search data structures contain generally several *sentinel nodes* which define the boundaries of the structure and are never removed from the structure: their presence probability is 1.

For a given node N_i, we denote by λ_i^{acc} (resp. λ_i^{read}, λ_i^{cas}) the rate of the events triggering an access (resp. *Read*, *CAS*) of N_i due to one thread, when $N_i \in D$. op_k^{del} (resp. op_k^{ins}, op_k^{src}) stands for a `Delete` (resp. `Insert`, `Search`) on node key k. The probability for the application to select op_k^o, where $o \in \{ins, del, src\}$ is denoted by $\mathbb{P}\left[Op = op_k^o\right]$. $op_k^o \rightsquigarrow cas(N_i)$ (resp. $read\ (N_i)$) means that during the execution of op_k^o, a *CAS* (resp. a *Read*) occurs on N_i. Putting all together, we derive the rate of the triggering events:

$$\forall e \in \{cas, read\}\ :\ \lambda_i^e = \frac{\mathcal{T}}{P} \times \sum_{o \in \{ins,del,src\}} \sum_{k=1}^{\mathcal{R}} \mathbb{P}\left[Op = op_k^o\right] \times \mathbb{P}\left[op_k^o \rightsquigarrow e(N_i) \,|\, N_i \in D\right] \quad (1)$$

Recall for later that Poisson processes have useful properties, *e.g.* merging two Poisson processes produces another Poisson process whose rate is the sum of the two initial rates. This implies especially that the access triggering events follows a Poisson process with rate $\lambda_i^{acc} = \lambda_i^{read} + \lambda_i^{cas}$, and that the read triggering events that originates from P' different threads and occurs at N_i follow a Poisson process with rate $P' \times \lambda_i^{read}$.

To quantify the error in Poisson process approximations, we experimentally extract the cumulative distribution function of the inter-arrival latency of events that occur on a given node. Then, we apply the Kolmogorov-Smirnov test to compare it against exponential distributions (recall that the time between events in a Poisson process is exponentially distributed). Please see [4] for this comprehensive set of comparisons.

4.2 Impacting Factors

We have identified five factors that dominate the access latency of a node, distributed into two sets. On the one hand, the first set of factors only emerges in the concurrent executions as a result of the coherence issues on the search data structures. Atomic primitives, such as a *CAS*, are used to modify the shared search data structures asynchronously. To execute a *CAS* in multi-core architectures, the cache coherency protocol enforces exclusive ownership of the target cacheline by a thread (pinned to a core) through the invalidation of all the other copies of the cacheline in the system, if needed. One can guess the performance implications of this process that triggers back and forth communication among the cores. As the first factor, *CAS* instruction has a significant latency. The thread that executes the *CAS* pays this latency cost. Secondly, any other thread has to stall until the end of the *CAS* execution if it attempts to access (read or modify) the node while the *CAS* is getting executed. Last and most importantly, any thread pays a cost to bring a cacheline to a valid state if it attempts to access a node that resides in this cacheline and that has been modified by another thread after its previous access to this node.

On the other hand, the capacity misses in the data and TLB caches are other performance impacting factors for the node accesses. Consider a cache of size C, assume a node is accessed by a thread at time t and the next access (same thread and node) occurs at time t'. The thread would experience a capacity miss for the access at time t' if it has accessed at least C distinct nodes in the interval (t, t'). The same applies for TLB caches where the references to the distinct pages are counted instead of the nodes.

At a given instant, we denote by $Access_i$ the latency of accessing node N_i, either due to a *Read* event or a *CAS* event, for a given thread. This latency is the sum of random variables that correspond to the previous respective five impacting factors and the constant

local computation cost (≈ 4 cycles):

$$Access_i = t^{cmp} + CAS_i^{exe} + CAS_i^{stall} + CAS_i^{reco} + \sum_\ell Hit_i^{cache_\ell} + \sum_\ell Hit_i^{tlb_\ell}, \quad (2)$$

where, at a random time, CAS_i^{exe} is the latency of a CAS, CAS_i^{stall} the stall time implied by other threads executing a CAS on N_i, CAS_i^{reco} the time needed to fetch the data from another modifying thread, $Hit_i^{cache_\ell}$ the latency resulting from a hit on the data cache in level ℓ, and $Hit_i^{tlb_\ell}$ the latency coming from a hit on the TLB cache in level ℓ.

4.3 Solving Process

The solving decomposes into three main steps. Firstly, we can notice that Equation 1 exposes $2\mathcal{R}+1$ unknowns (the $2\mathcal{R}$ access rates and throughput) against $2\mathcal{R}$ equations. To end up with a unique solution, a last equation is necessary. The first two steps provide a last sufficient equation thanks to Little's law (see Section 5.2), which links throughput with the expectation of the access latency of a node, computed in Sections 5.1. We show in this section that the each component of the access latency can be expressed according to the access rates λ_i^{read} and λ_i^{cas}. The last step focuses on the values of the probabilities in Equation 1, which are strongly related with the particular data structure under consideration; they are instantiated in Section 6.1 (resp. 6.2, 6.3, 6.4) for linked lists (resp. hash tables, skip lists, binary trees).

5 Throughput Estimation

5.1 Access Latency

Applying expectation to Equation 2 leads to $\mathbb{E}[Access_i] = t^{cmp} + \mathbb{E}[CAS_i^{exe}] + \mathbb{E}\left[CAS_i^{stall}\right] + \mathbb{E}[CAS_i^{reco}] + \mathbb{E}\left[\sum_\ell Hit_i^{cache_\ell}\right] + \mathbb{E}\left[\sum_\ell Hit_i^{tlb_\ell}\right]$. We express here each term according to the rates at every node $\lambda_\star^{cas}$ and $\lambda_\star^{read}$.

CAS Execution

Naturally, among all access events, only the events originating from a CAS event contribute, with the latency t^{cas} of a CAS: $\mathbb{E}[CAS_i^{exe}] = t^{cas} \cdot \lambda_i^{cas}/(\lambda_i^{read} + \lambda_i^{cas})$.

Stall Time

A thread experiences stall time while accessing N_i when a thread, among the $(P-1)$ remaining threads, is currently executing a CAS on the same node. As a first approximation, supported by the rareness of the events, we assume that at most one thread will wait for the access to the node.

Firstly, we obtain the rate of CAS events generated by $(P-1)$ threads through the merge of their Poisson processes. Consider an access of N_i at a random time; (i) the probability of being stalled is the ratio of time when N_i is occupied by a CAS of $(P-1)$ threads, given by: $\lambda_i^{cas}(P-1)t^{cas}$; (ii) the stall time that the thread would experience is distributed uniformly in the interval $[0, t^{cas}]$. Then, we obtain: $\mathbb{E}\left[CAS_i^{stall}\right] = \lambda_i^{cas}(P-1)t^{cas}(t^{cas}/2)$.

Invalidation Recovery

Given a thread, a coherence cache miss occurs if N_i is modified by any other thread in between two consecutive accesses of N_i. The events that are concerned are: (i) the CAS events from any thread; (ii) the *Read* events from the given thread. When N_i is accessed, we look back at these events, and if among them, the last event was a CAS from another thread, a coherence miss occur: $\mathbb{P}\left[\text{Coherence Miss on } N_i\right] = \frac{\lambda_i^{cas}(P-1)}{\lambda_i^{cas}P+\lambda_i^{read}}$. We derive the expected latency of this factor during an access at N_i by multiplying this with the latency penalty of a coherence cache miss: $\mathbb{E}\left[CAS_i^{reco}\right] = \mathbb{P}\left[\text{Coherence Miss on } N_i\right] \times t^{rec}$.

Che's Approximation

Che's Approximation [10] is a technique to estimate the hit ratio of an LRU cache of size C, where the object (here, node) accesses follow IRM (Independent Reference Model). IRM is based on the assumption that the object references occur in an infinite sequence from a fixed catalog of $\mathcal{N}$ objects. The popularity of object i (denoted by s_i, where $i \in [1..\mathcal{N}]$) is a constant that does not depend on the reference history and does not vary over time.

Starting from $t = 0$, let the time of reference to object i be denoted by O_i, then the time for C unique references is given by: $t_c = \inf\{t > 0 : X(t) = C\}$, where $X(t) = \sum_{j=1}^{\mathcal{N}} \mathbf{1}_{0<O_j\leq t}$. Che's approximation estimates the hit ratio of object i with $Hit_i \approx 1 - e^{-s_i T}$, where the so-called characteristic time (denoted by T that approximates t_c) is the unique solution of the following equation: $C = \sum_{j=1}^{\mathcal{N}}(1 - e^{-s_j T}) = \mathbb{E}\left[X(T)\right]$. The accuracy of the approximation is rooted at the random variable $X(t)$ that is approximately Gaussian since it is defined as the sum of many independent random variables (Central Limit Theorem). We provide a more detailed discussion on this approximation in [4] based on the analysis in [17].

Cache Misses

We consider a data cache at level ℓ of size C_ℓ^{dat} and compute the hit ratio of N_i on this cache. On a given data structure, we have compared two related scenarios: a search only scenario, leading to a given expected size of the data structure, against a scenario with updates, that leads to the same expected size. We have observed that the capacity cache miss ratio were similar in both cases. Therefore, we consider that with or without updates, $N_\star$ is either present in the search data structure or not, *during the characteristic time of the cache.*

We can employ the access rates as popularities, *i.e.* $s_x = \lambda_x^{acc}$ for $x \in [1..\mathcal{N}]$, and modify Che's approximation to distinguish whether, at a random time, N_x is inside the data structure or not.

We integrate the masking variable P_x into Che's approximation. We have: $X^{cache}(t) = \sum_{x=1}^{\mathcal{N}} P_x \mathbf{1}_{0<O_x\leq t}$, where O_x denotes the reference time of N_x. We can still assume $X^{cache}(t)$ is Gaussian, as a sum of many independent random variables. We estimate the characteristic time as follows with the linearity of expectation and the independence of the random variables: $\mathbb{E}\left[X^{cache}(t)\right] = \sum_{x=1}^{\mathcal{N}} \mathbb{E}\left[P_x \mathbf{1}_{0<O_x\leq t}\right] = \sum_{x=1}^{\mathcal{N}} \mathbb{E}\left[P_x\right]\mathbb{E}\left[\mathbf{1}_{0<O_x\leq t}\right] = \sum_{x=1}^{\mathcal{N}} p_x(1 - e^{-\lambda_x^{acc} t})$. Lastly, we solve the equation for the characteristic time T_ℓ^{dat} of level ℓ cache: $\sum_{x=1}^{\mathcal{N}} p_x(1 - e^{-\lambda_x^{acc} T_\ell^{dat}}) = C_\ell^{dat}$ thanks to a fixed-point approach. After computing T_ℓ^{dat}, we estimate the cache hit ratio (on level ℓ) of N_i: $1 - e^{-\lambda_i^{acc} T_\ell^{dat}}$.

Page Misses

In this paragraph, we aim at computing the page hit ratio of N_i for the TLB cache at level ℓ of size C_ℓ^{tlb}. The total number $\mathcal{M}$ of pages that are used by the search data structure can be regulated by a parameter of the memory management scheme (frequency of recycling attempts

for the deleted nodes), as the total number of nodes is a function of $\mathcal{R}$. Different from the cachelines (corresponding to the nodes), we can safely assume that a page accommodates at least a single node that is present in the structure at any time.

We cannot apply straightforwardly Che's approximation since the page reference probabilities are unknown. However, we are given the cacheline reference probabilities $s_x = \lambda_x^{acc}$ for $x \in [1..\mathcal{N}]$ and we assume that $\mathcal{N}$ cachelines are mapped uniformly to $\mathcal{M}$ pages, $[1..\mathcal{N}] \to [1..\mathcal{M}]$, $\mathcal{N} > \mathcal{M}$. Under these assumptions, we know that the resulting page references would follow IRM because aggregated Poisson processes form again a Poisson process.

We follow the same line of reasoning as in the cache miss estimation. First, we consider a set of Bernoulli random variables (Y_x^j), leading to a success if N_x is mapped into page j, with probability $p_x/\mathcal{M}$ (hence Y_x^j does not depend on j). Under IRM, we can then express the page references as point processes with rate $r_j = \sum_{x=1}^{\mathcal{N}} Y_x^j s_x$, for all $j \in [1..\mathcal{M}]$.

Similar to the previous section, we denote the time of a reference to page j with O_j and we define the random variable $X^{page}(t) = \sum_{j=1}^{\mathcal{M}} \mathbf{1}_{0<O_j\leq t}$ and compute its expectation:

$$\begin{aligned}
\mathbb{E}\left[X^{page}(t)\right] &= \sum_{j=1}^{\mathcal{M}} \mathbb{E}\left[\mathbf{1}_{0<O_j\leq t}\right] = \sum_{j=1}^{\mathcal{M}} \mathbb{E}\left[1 - e^{-r_j t}\right] = \sum_{j=1}^{\mathcal{M}} \mathbb{E}\left[1 - e^{-\sum_{x=1}^{\mathcal{N}} Y_x^j \lambda_x^{acc} t}\right] \\
&= \sum_{j=1}^{\mathcal{M}} \left(1 - \prod_{x=1}^{\mathcal{N}} \mathbb{E}\left[e^{-Y_x^j \lambda_x^{acc} t}\right]\right) = \sum_{j=1}^{\mathcal{M}} \left(1 - \prod_{x=1}^{\mathcal{N}} \left(\frac{\mathcal{M} - p_x}{\mathcal{M}} + \frac{p_x e^{-\lambda_x^{acc} t}}{\mathcal{M}}\right)\right) \\
\mathbb{E}\left[X^{page}(t)\right] &= \mathcal{M}\left(1 - \prod_{x=1}^{\mathcal{N}} \left(\frac{\mathcal{M} - p_x}{\mathcal{M}} + \frac{p_x e^{-\lambda_x^{acc} t}}{\mathcal{M}}\right)\right),
\end{aligned}$$

Assuming $X^{page}(t)$ is Gaussian as it is sum of many independent random variables, we solve the following equation for the constant T_ℓ^{tlb} (characteristic time of a TLB cache of size C): $\mathbb{E}\left[X^{page}(T_\ell^{tlb})\right] = C_\ell^{tlb}$.

Lastly, we obtain the TLB hit rate for N_i by relying on the average *Read* rate of the page that N_i belongs to; we should add to the contributions of N_i, the references to the nodes that belong to the same page as N_i. Then follows the TLB hit ratio: $1 - e^{-z_i T_\ell^{tlb}}$, where $z_i = \lambda_i^{acc} + \mathbb{E}\left[\sum_{x=1, x\neq i}^{\mathcal{N}} Y_x^j \lambda_x^{acc}\right] = \lambda_i^{acc} + \sum_{x=1, x\neq i}^{\mathcal{N}} p_x \lambda_x^{acc}/\mathcal{M}$.

Interactions

To be complete, we mention the interaction between impacting factors and the possibility of latency overlaps in the pipeline. Firstly, the access latency of different nodes can not be overlapped due to the semantic dependency for the linked nodes. For a single node access, the latency for *CAS* execution and stall time can not be overlapped with any other factor. Based on the cache coherency protocol behavior, we do not charge invalidation recovery cost for *CAS* events. We consider inclusive data and TLB caches. It is not possible to have a cache hit on level l, if the cache on level $l-1$ is hit, and we do not consider any cost for the data cache hit if invalidation recovery or *CAS* execution (coherence) cost is induced (*i.e.* $\mathbb{E}\left[Hit_i^{cache_\ell}\right] = (1 - \mathbb{P}\left[coherence\ cost\right])(\mathbb{P}\left[hit\ cache_l\right] - \mathbb{P}\left[hit\ cache_{l-1}\right])t_\ell^{dat}$).

5.2 Latency vs. Throughput

In the previous sections, we have shown how to compute the expected access latency for a given node. There remains to combine these access latencies in order to obtain the throughput of the search data structure. Given $N_i \in D$, the average arrival rate of threads to N_i is

$\lambda_i^{acc} = \lambda_i^{read} + \lambda_i^{cas}$. Thus the average arrival rate of threads to N_i is: $p_i\lambda_i^{acc}$. It can then be passed to Little's Law [22], which states that the expected number of threads (denoted by t_i) accessing N_i obeys to $t_i = p_i\lambda_i^{acc}\mathbb{E}[Access_i]$. The equation holds for any node in the search data structure, and for the application call occurring in between search data structure operations. Its expected latency is a parameter ($\mathbb{E}[Access_0] = t^{app}$) and its average arrival rate is equal to the throughput ($\lambda_0^{acc} = \mathcal{T}$). Then, we have: $\sum_{i=0}^{\mathcal{N}} t_i = \sum_{i=0}^{\mathcal{N}}(p_i\lambda_i^{acc}\mathbb{E}[Access_i])$, where λ_i^{acc} and $\mathbb{E}[Access_i]$ are linear functions of $\mathcal{T}$. We also know $\sum_{i=0}^{\mathcal{N}} t_i = P$ as the threads should be executing some component of the program. We define constants with a_i, b_i, c_i for $i \in [0..\mathcal{N}]$. And, we represent $\lambda_i^{acc} = a_i\mathcal{T}$ and $\mathbb{E}[Access_i] = b_i\mathcal{T} + c_i$ and we obtain the following second order equation: $\sum_{i=0}^{\mathcal{N}}(p_i a_i b_i)\mathcal{T}^2 + \sum_{i=0}^{\mathcal{N}}(p_i a_i c_i)\mathcal{T} - P = 0$. This second order equation has a unique positive solution that provides the throughput, $\mathcal{T}$.

6 Instantiating the Throughput Model

In this section, we show how to instantiate our model with widely known lock-free search data structures, that have different operation time complexities. In order to obtain a throughput estimate for a structure, we need to compute the rates $\lambda_\star^{read}$ and $\lambda_\star^{cas}$, and $\mathbb{P}[op_k^o \rightsquigarrow e(N_i) \,|N_i \in D]$, *i.e.* the probability that, at a random time, an operation of type o on key k leads to a memory instruction of type e on node N_i, knowing that N_i is in the data structure. For the ease of notation, nodes will sometimes be doubly or triply indexed, and when the context is clear, we will omit $|N_i \in D$ in the probabilities. As a remark, the properties of the access pattern (memoryless and stationary) are critical here because they allow us to extract the probability of the state that the data structure could be in (based on the presence of nodes) at a random time, which is then used to find $\mathbb{P}[op_k^o \rightsquigarrow e(N_i) \,|N_i \in D]$.

We first estimate the throughput of linked lists and hash tables, on which we can directly apply our method, then we move on more involved search data structure, namely skip lists and binary trees, that need a particular attention.

6.1 Linked List

We start with the lock-free linked list implementation of Harris [19]. All operations in the linked list start with the search phase in which the linked list is traversed until a key. At this point all operations terminate except the successful update operations that proceed by modifying a subset of nodes in the structure with *CAS* instructions. The structure contains only valued node and two sentinel nodes N_0 and $N_{\mathcal{R}+1}$, so that $\mathcal{N} = \mathcal{R} + 2$ and for all $i \in [1..\mathcal{R}]$, N_i holds key i, *i.e.* $K_i = i$.

First, we need to compute the probabilities of triggering a *Read* event and *CAS* event on a node, given that the node is in the search data structure, for all operations of type $t \in \{\texttt{Insert}, \texttt{Delete}, \texttt{Search}\}$ targeted to key k.

At a random time, N_k, for $k \in [1..\mathcal{R}]$, is in the linked list iff the last update operation on key k is an insert: $p_k = q_k$, by definition of q_k. Moreover, when N_k is in the structure (condition that we omit in the notation), $op_{k'}^t$ reads N_k, either if N_k is before $N_{k'}$, or if it is just after $N_{k'}$. Formally, $\mathbb{P}[op_{k'}^o \rightsquigarrow read(N_k)] = 1$ if $k \leq k'$ and $\mathbb{P}[op_{k'}^o \rightsquigarrow read(N_k)] = \prod_{i=k'}^{k-1}(1-p_i)$ if $k > k'$.

CAS events can only be triggered by successful `Insert` and `Delete` operations. A successful `Insert` operation, targeted to $N_{k'}$, is realized with a *CAS* that is executed on N_k, where $k = \sup\{\ell < k' : N_\ell \in D\}$. The probability of success, which conditions the *CAS*'s, follows from the presence probabilities:

$$\mathbb{P}\left[op^{ins}_{k'} \rightsquigarrow cas(N_k)\right] = \begin{cases} 0, & \text{if } k \geq k' \\ \prod_{i=k+1}^{k'} (1-p_i), & \text{if } k < k' \end{cases} ; \quad \mathbb{P}\left[op^{del}_{k'} \rightsquigarrow cas(N_k)\right] = \begin{cases} 1, & \text{if } k = k' \\ 0, & \text{if } k > k' \\ p_{k'} \prod_{i=k+1}^{k'-1} (1-p_i), & \text{if } k < k' \end{cases}$$

6.2 Hash Table

We analyze here a chaining based hash table where elements are hashed to B buckets implemented with the lock-free linked list of Harris [19]. The structure is parametrized with a load factor lf which determines B through $B = \mathcal{R}/lf$. The hash function $h : k \mapsto \lceil k/lf \rceil$ maps the keys sequentially to the buckets, so that, after including the sentinel nodes (2 per bucket), we can doubly index the nodes: $N_{b,k}$ is the node in bucket b with key k, where $b \in [1..B]$ and $k \in [1..lf]$ (the last bucket may contain less elements).

$$\mathbb{P}\left[op^{o}_{b',k'} \rightsquigarrow read\big(N_{b,k}\big)\right] = \begin{cases} 0, & \text{if } b' \neq b \\ 1, & \text{if } b' = b \text{ and } k' \geq k \\ \prod_{j=k'}^{k-1} (1-p_{b,j}), & \text{if } b' = b \text{ and } k' < k \end{cases}$$

$$\mathbb{P}\left[op^{ins}_{b',k'} \rightsquigarrow cas\big(N_{b,k}\big)\right] = \begin{cases} 0, & \text{if } b' \neq b \text{ or } k' \leq k \\ \prod_{j=k+1}^{k'} (1-p_{b,j}), & \text{if } b' = b \text{ and } k' > k \end{cases}$$

$$\mathbb{P}\left[op^{del}_{b',k'} \rightsquigarrow cas\big(N_{b,k}\big)\right] = \begin{cases} 0, & \text{if } b' \neq b \text{ or } k' < k \\ 1, & \text{if } b' = b \text{ and } k' = k \\ p_{b,k'} \prod_{j=k+1}^{k'-1} (1-p_{b,j}), & \text{if } b' = b \text{ and } k' > k \end{cases}$$

In the previous two data structures, we do observe differences in the access rate from node to node, but the node associated with a given key does not show significant variation in its access rate during the course of the execution: inside the structure, the number of nodes preceding (and following) this node is indeed rather stable. In the next two data structures, node access rates can change dramatically according to node characteristics, that may include its position in the structure. In a skip list, a node N_i containing key K_i with maximum height will be accessed by any operation targeting a node with a higher key. However, N_i can later be deleted and inserted back with the minimum height; the operations that access it will then be extremely rare. The same reasoning holds when comparing an internal node with key K_i of a binary tree located at the root or close to the leaves.

As explained before, an accurate cache miss analysis cannot be satisfied with average access rates. Therefore, the information on the possible significant variations of rates should not be diluted into a single access rate of the node. To avoid that, we pass the information through virtual nodes: a node of the structure is divided into a set of virtual nodes, each of them holding a different flavor of the initial node (height of the node in the skip list or subtree size in the binary tree). The virtual nodes go through the whole analysis instead of the initial nodes, before we extract the average behavior of the system hence throughput.

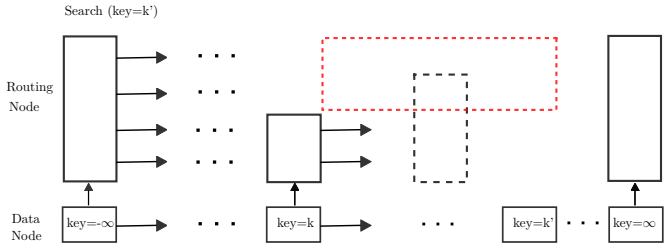

Figure 2 Skip List Events: Read Event Probability.

6.3 Skip List

There exist various lock-free skip list implementations and we study here the lock-free skip list [25].

Skip lists offer layers of linked lists. Each layer is a sparser version of the layer below where the bottom layer is a linked list that includes all the elements that are present in the search data structure. An element that is present in the layer at height h appears in layer at height $h+1$ with a fixed appearance probability (1/2 for our case) up to some maximum layer h_{max} that is a parameter of the skip list.

Skip list implementations are often realized by distinguishing two type of nodes: (i) valued nodes reside at the bottom layer and they hold the key-value pair in addition to the two pointers, one to the next node at the bottom layer and one to the corresponding routing node (could be *null*); (ii) routing nodes are used to route the threads towards the search key. Being coupled with a valued node, a routing node does not replicate the key-value pair. Instead, only a set of pointers, corresponding to the valued node containing the next key in different layers, are packed together in a single routing node (that fits in a cacheline with high probability). Every *Read* event in a routing node is preceded by a *Read* in the corresponding valued node.

We denote by $N^{rou}_{k,h}$ the routing node containing key k, whose set of pointers is of height h, where $h \in [1..h_{max}]$. A valued node containing the key k is denoted by $N^{dat}_{k,h}$ when connected to $N^{rou}_{k,h}$ ($h = 0$ if there is no routing node). Furthermore, there are four sentinel nodes $N^{dat}_{0,h_{max}}$, $N^{rou}_{0,h_{max}}$, $N^{dat}_{\mathcal{R}+1,h_{max}}$, $N^{rou}_{\mathcal{R}+1,h_{max}}$. The presence probabilities result from the coin flips (bounded by h_{max}): for $z \in \{dat, rou\}$, $p^z_{k,h} = 2^{-(h+1)}q_k$ if $h < h_{max}$, $p^z_{k,h} = q_k - \sum_{\ell=0}^{h_{max}-1} p^z_{k,\ell}$ otherwise.

By decomposing into three cases, we compute the probability that an operation $op^o_{k'}$ of type $o \in \{ins, del, src\}$, targeted to k', causes a *Read* triggering event at $N^z_{k,h}$ when $N^z_{k,h} \in D$. Let assume first that $k' > k$. The operation triggers a *Read* event at node $N^z_{k,h}$ if for all (x, y) such that $y > h$ and $k < x \leq k'$, $N^z_{x,y}$ is not present in the skip list (*i.e.* in Figure 2, no node in the skip list overlaps with the red frame). Let assume now $k' < k$. The occurrence of a *Read* event requires that: for all (x, y) such that $y \geq h$ and $k' \leq x < k$, $N^z_{x,y}$, is not present in the structure. Lastly, a *Read* event is certainly triggered if $k' = k$. The final formula is given by:

$$\mathbb{P}\left[op^o_{k'} \rightsquigarrow read\left(N^z_{k,h}\right)\right] = \begin{cases} \prod_{x=k+1}^{k'} \left(1 - \left(\sum_{y=h+1}^{h_{max}} p^z_{x,y}\right)\right), & \text{if } k \leq k' \\ \prod_{x=k'}^{k-1} \left(1 - \left(\sum_{y=h}^{h_{max}} p^z_{x,y}\right)\right), & \text{if } k > k' \end{cases}$$

To be complete, we describe in [4] how to compute the probability for *CAS* events, following a similar approach.

6.4 Binary Tree

We show here how to estimate the throughput of external binary trees. They are composed of two types of nodes: internal nodes route the search towards the leaves (routing nodes) and store just a key, while leaves, referred to as external nodes contain the key-value pair (valued node). We use the external binary tree of Natarajan [23] to instantiate our model. The search traversal starts and continues with a set of internal nodes and ends with an external node. We denote by N_k^{int} (resp. N_k^{ext}) the internal (resp. external) node containing key k, where $k \in [1..\mathcal{R}]$. The tree contains two sentinel internal nodes that reside at the top of the tree (hence are accessed by all operation): N_{-1}^{int} and N_0^{int}.

Our first aim is to find the paths followed by any operation through the binary tree, in order to obtain the access triggering rates, thanks to Equation 1. Binary trees are more complex than the previous structures since the order of the operations impact the positioning of the nodes. The random permutation model proposes a framework for randomized constructions in which we can develop our model. Each key is associated with a priority, which determines its insertion order: the key with the highest priority is inserted first. The performance characteristics of the randomized binary trees are studied in [24]. In the same vein, we compute the access probability of the internal node with key k in an operation that targets key k'.

▶ **Lemma 1.** *Given an external binary tree, the probability of accessing N_k^{int} in an operation that targets key $K_{k'}$ is given by: (i) $1/f(k,k')$ if $k' \geq k$; (ii) $1/(f(k',k)-1)$ if $k' < k$, where $f(x,y)$ provides the number internal nodes whose keys are in the interval $[x,y]$.*

Proof. N_k^{int} would be accessed if it is on the search path to the external node with key k'. Given $k' \geq k$, this happens iff N_k^{int} has the highest priority among the internal nodes in the interval $[k,k']$. This interval contains $f(k,k')$ internal nodes, thus, the probability of N_k^{int} to possess the highest priority is $1/f(k,k')$. Similarly, if $k' < k$, then N_k^{int} is accessed iff it has the highest priority in the interval $(k',k]$. Hence, the lemma. ◀

Even if in the binary tree, nodes are inserted and deleted an infinite number of times, Lemma 1 can still be of use. The number of internal nodes in the interval $[k,k']$ (or $(k',k]$ if $k' < k$) is indeed a random variable which is the sum of independent Bernoulli random variables that models the presence of the nodes. As a sum of many independent Bernoulli variables, the outcome is expected to have low variations because of its asymptotic normality. Therefore, we replace this random variable with its expected value and stick to this approximation in the rest of this section. The number of internal nodes in any interval come out from the presence probabilities: $p_k^z = q_k$, where $z \in \{int, ext\}$.

In an operation is targeted to key k', a single external node is accessed (if any): $N_{k'}^{ext}$, if present, else the external node with the biggest key smaller than k', if it exists, else the external node with the smallest key. Then, we have:

$$\mathbb{P}\left[op_{k'}^o \rightsquigarrow read\left(N_k^{int}\right)\right] = \begin{cases} 1/(1+\sum_{i=k'+1}^{k-1} p_i^{int}), \text{ if } k > k' \\ 1/(1+\sum_{i=k+1}^{k'} p_i^{int}), \text{ if } k \leq k' \end{cases},$$

$$\mathbb{P}\left[op_{k'}^o \rightsquigarrow read\left(N_k^{ext}\right)\right] = \begin{cases} 1, & \text{if } k = k' \\ \prod_{i=k+1}^{k'}(1-p_i^{ext}), & \text{if } k < k' \\ \prod_{i=1}^{k-1}(1-p_i^{ext}), & \text{if } k > k' \end{cases}$$

These probabilities finally lead to the computation of the *Read* (resp. *CAS*) rates $\lambda_{z,k}^{read}$ (resp. $\lambda_{z,k}^{cas}$) of N_k^z, where $z \in \{int, ext\}$, that will be used in the last following step.

We focus now on the *Read* rate of the internal nodes. We have found the average behavior of each node in the previous step; however, the node can follow different behaviors during the execution since the *Read* rate of N_k^{int} depends on the size of the subtree whose root is N_k^{int}, which is expected to vary with the update operations on the tree. We dig more into this and reflect these variations by decomposing N_k^{int} into H_k virtual nodes, $N_{k,h}^{int}$, where $h \in [1..H_k]$. We define the *Read* rate $\lambda_{int,k,h}^{read}$ of these virtual nodes as a weighted sum of the initial node rate thanks the two equations $p_k^{int} = \sum_{h=1}^{H_k} p_{k,h}^{int}$ and $p_k^{int}\lambda_{int,k}^{read} = \sum_{h=1}^{H_k} p_{k,h}^{int}\lambda_{int,k,h}^{read}$.

We connect the virtual nodes to the initial nodes in two ways. On the one hand, one can remark that the *Read* rate is proportional to the subtree size: $\lambda_{int,k,h}^{read} \propto h\lambda_{int,k}^{read}$. On the other hand, based on the probability mass function of the random variable Sub_k representing the size of the subtree rooted at N_k^{int}, we can evaluate the weight of the virtual nodes: $p_{k,h}^{int} = p_k^{int}\mathbb{P}[Sub_k = h]$.

More details are to be found in [4], on how to obtain the mass function of the random variable Sub_k to compute *Read* rates for the virtual nodes and how to deal with the *CAS* events.

7 Experimental Evaluation

We validate our model through a set of well-known lock-free search data structure designs, mentioned in the previous section. We stress the model with various access patterns and number of threads to cover a considerable amount of scenarios where the data structures could be exploited. For the key selection process, we vary the key ranges and the distribution: from uniform (*i.e.* the probability of targeting any key is constant for each operation) to zipf (with $\alpha = 1.1$ and the probability to target a key decreases with the value of the key). Regarding the operation types, we start with various balanced update ratios, *i.e.* such that the ratio of `Insert` (among all operations) equals the ratio of `Delete`. Then, we also consider asymmetric cases where the ratio of `Insert` and `Delete` operations are not equal, which changes the expected size of the structure.

7.1 Setting

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets, each containing eight physical cores. The system is equipped with Intel Xeon E5-2687W v2 CPUs. Threads are pinned to separate cores. One can observe the performance change when number of threads exceeds 8, which activates the second socket.

In all the figures, y-axis provides the throughput, while the number of threads is represented on x-axis. The dots provide the results of the experiments and the lines provide the estimates of our framework. The key range of the data structure is given at the top of the figures and the percentage of update operations are color coded.

We instantiate all the algorithm and architecture related latencies, following the methodologies described in [6] In line with these studies, we observed that the latencies of t^{cas} and t^{rec} are based on thread placement. We distinguish two different costs for t^{cas} according to the number of active sockets. Similarly, given a thread accessing to a node N_i, the recovery latency is low (resp. high), denoted by t_{low}^{rec} (resp. t_{high}^{rec}), if the modification has been performed by a thread that is pinned to the same (resp. another) socket. Before the execution, we measure both t_{low}^{rec} and t_{high}^{rec}, and instantiate t^{rec} with the average recovery latency, computed in the following way for a two-socket chip. For $s \in \{1, 2\}$, we denote by P_s the number of threads that are pinned to socket numbered s. By taking into account

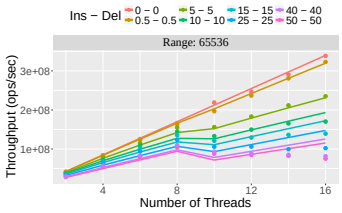

Figure 3 Hash Table with load factor 2.

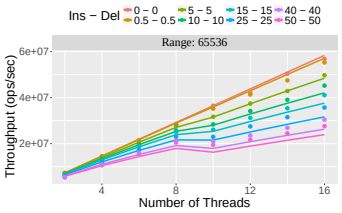

Figure 4 Skip List.

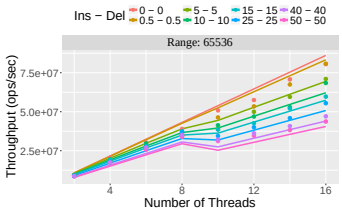

Figure 5 Binary Tree.

all combinations, we have $t^{rec} = (P_1(P_1 t^{rec}_{low} + P_2 t^{rec}_{high}) + P_2(P_2 t^{rec}_{low} + P_1 t^{rec}_{high}))/P^2$. Since $P = P_1 + P_2$, we obtain $t^{rec} = t^{rec}_{low} + 2(P_1/P)(1 - P_1/P)(t^{rec}_{high} - t^{rec}_{low})$.

For the data structure implementations, we have used ASCYLIB library [11] that is coupled with an epoch based memory management mechanism which has negligible latency.

7.2 Search Data Structures

In Figure 3, 4 and 5, we provide the results for the hash table, skip list and binary tree where the key selection is done with the uniform distribution. One can see that the performance drops as the update rate increases, due to the impact of *CAS* related factors. This impact is magnified with the activation of the second socket (more than 8 threads) since the event becomes more costly. When there is no update operation, the performance scales linearly with the number of threads. Since the threads access disjoint parts of structure, we observe a similar behaviour for the cases with updates. See [4] for a more comprehensive set of experiments and applications.

8 Conclusion

In this paper, we have modeled and analyzed the performance of search data structures under a stationary and memoryless access pattern. We have distinguished two types of events that occur in the search data structure nodes and have modeled the arrival of events with Poisson processes. The properties of the Poisson process allowed us to consider the thread-wise and system-wise interleaving of events which are crucial for the estimation of the throughput. For the validation, we have used several fundemental lock-free search data structures.

As a future work, it would be of interest to study to which extent the application workload can be distorted while giving satisfactory results. Putting aside the non-memoryless access patterns, the non-stationary workloads such as bursty access patterns, could be covered by splitting the time interval into alternating phases and assuming a stationary behaviour for each phase. Furthermore, we foresee that the framework can capture the performance of lock-based search data structures and also can be exploited to predict the energy efficiency of the concurrent search data structures.

References

1 Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? In David B. Shmoys, editor, *STOC*, pages 714–723. ACM, June 2014.

2 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model. In *DISC*, pages 341–355. Springer, 2015.

3 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses. In *OPODIS*, pages 23:1–23:17, 2016.

4 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. Lock-Free Search Data Structures: Throughput Modelling with Poisson Processes. *CoRR*, abs/1805.04794, 2018. URL: `http://arxiv.org/abs/1805.04794`.

5 Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *JACM*, 50(4):444–468, 2003.

6 Vlastimil Babka and Petr Tuma. Investigating Cache Parameters of x86 Family Processors. In *SPEC Benchmark Workshop*, pages 77–96. Springer, 2009.

7 A.D. Barbour and T.C. Brown. Stein's method and point process approximation. *Stochastic Process. Appl.*, 43(1):9–31, 1992.

8 Naama Ben-David and Guy E. Blelloch. Analyzing Contention and Backoff in Asynchronous Shared Memory. In *PoDC*, pages 53–62, 2017.

9 Timothy C. Brown, Graham V. Weinberg, and Aihua Xia. Removing logarithms from Poisson process error bounds. *Stochastic Process. Appl.*, 87(1):149–165, 2000.

10 Hao Che, Ye Tung, and Zhijun Wang. Hierarchical Web caching systems: modeling, design and experimental results. *IEEE Communications Society Press*, 20(7):1305–1314, 2002.

11 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS*, pages 631–644. ACM, 2015.

12 Luc Devroye. A note on the height of binary search trees. *JACM*, 33(3):489–498, 1986.

13 Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *JACM*, 44(6):779–805, 1997.

14 James D. Fix. The set-associative cache performance of search trees. In *SODA*, pages 565–572, 2003.

15 Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier. Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-Organizing Search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.

16 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PoDC*, pages 50–59. ACM, 2004.

17 Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. *CoRR*, abs/1202.3974, 2012.

18 Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, pages 1–10. ACM, 2015.

19 Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2001.

20 Maurice Herlihy. A Methodology for Implementing Highly Concurrent Objects. *TOPLAS*, 15(5):745–770, 1993.

21 Peter Kirschenhofer and Helmut Prodinger. The Path Length of Random Skip Lists. *Acta Informatica*, 31(8):775–792, 1994.

22 John D. C. Little. A proof for the queuing formula: L= λ W. *Operations research*, 9(3):383–387, 1961.

23 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328. ACM, 2014.

24 Raimund Seidel and Cecilia R. Aragon. Randomized Search Trees. *Algorithmica*, 16(4/5):464–497, 1996.

25 Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.