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

Aras Atalar, Paul Renaud-Goud, Philippas Tsigas

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

Aras Atalar[1], Paul Renaud-Goud[2], and Philippas Tsigas[1]

[1]Chalmers University of Technology, aaras|philippas.tsigas@chalmers.se
[2]Informatics Research Institute of Toulouse, Paul.Renaud.Goud@irit.fr

Abstract

This paper considers the modelling 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 analysing lock-free 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 linked basic blocks, usually referred as nodes, that 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 structures (ii) *Read* events originated during traversals. This type of data structures are usually designed to accommodate a large number of data nodes, which makes the occurrence of an event on a given node rare at any given time. The throughput is defined by the number of events per operation in conjunction with the factors that impact the latencies of these events. We frame these impacting factors under capacity and coherence cache misses.

In this context, we model the events as Poisson processes that we can merge and split to estimate the latencies of the events based on the interleaving of events from different threads, and in turn estimate 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.

CONTENTS

I. 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 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 time line. 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 as a result of 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 will mention 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 the 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 rank different designs in various 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 of search data structures for a sequence of operations, generated by a memoryless and stationary access pattern. The threads execute the same piece of code on the same platform, throughput $\mathcal{T}$ can be estimated on the long-term as the expected latency of an operation (subjected to the distribution of the operations) divided by the number of threads P. 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 node 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 traversed 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 (*traversal*) 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 modelled as Poisson point 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 traversal latency associated with the triggered event. Once this information is grabbed, we roll back to the expectation of the traversal 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 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 II, then the problem is formulated in Section III. We present the framework in Section IV and the computation of throughput in Section V. In Section VI, we show how to initiate our model by considering the particularity of different search data structures. Finally, we describe the experimental results in Sections VII and VIII.

II. Related Work

The search path length of skiplists is analysed in [16], [21]. In [16], the search path length is split into vertical and horizontal components, where the horizontal cost is modelled with the number of right-to-left maximas (which corresponds to the traversed node) in a sequence of nodes with random heights. In [9], [22], [18], 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.

Previously mentioned 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 [24], data reuse distance patterns are modelled and then exploited to predict the cache miss ratio. In [11], 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 [10].

For the time complexity of lock-free search data structures, asymptotic amortized analyses [12], [5] are conducted since it is not possible to bound the execution time of a single operation, by definition. Apart from these theoretical studies, the performance of concurrent lock-free search data structures are studied and investigated through empirical studies in [14], [8]. In [7], it is shown experimentally that the conflicts between threads occur very rarely in the context of concurrent search data structures, which is confirmed by our analysis.

III. 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 asynchronous shared memory accesses. The threads are pinned to separate cores and call AbstractAlgorithm (see Figure 1) when they are spawned.

Procedure AbstractAlgorithm

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

Figure 1: Generic framework

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 the probability of selecting a key (resp. an operation type) is a constant.

A search data structure is modelled 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 (see Section VIII), 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.* the level of the cache where a node belongs to for a *Read* request. We summarize the parameters of our model as follows:

- *Algorithm parameters:* Expected latency of the application call t^{app}, expected computational cost to traverse a node t^{cmp}, 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; number of threads P.

IV. Framework

A. 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 traversed 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 (where a success means that a *Read* event on N_i occurs), where the probability of having a *Read* event during an interval depends on the associated operation (recall that the operation generating process 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 traversed nodes is then expected to be small in front of the set of all nodes. This implies that, given an operation, the probability that N_i belongs to the set of traversed 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 [3], [6], [1]. In particular, [4] has dealt with non-uniform Bernoulli processes. Their error bounds, which are proportional to the success probability, strengthen the use of Poisson processes in our context: the events on N_i are rare, thus the probabilities in Bernoulli processes are small and the approximation is well-conditioned.

Once the *Read* and *CAS* triggering events are modelled as Poisson processes for a single thread, the merge 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 modelled 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

$$q_k = \frac{\mathbb{P}\left[Op = op_k^{ins}\right]}{\mathbb{P}\left[Op = op_k^{ins}\right] + \mathbb{P}\left[Op = op_k^{del}\right]}.$$

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.

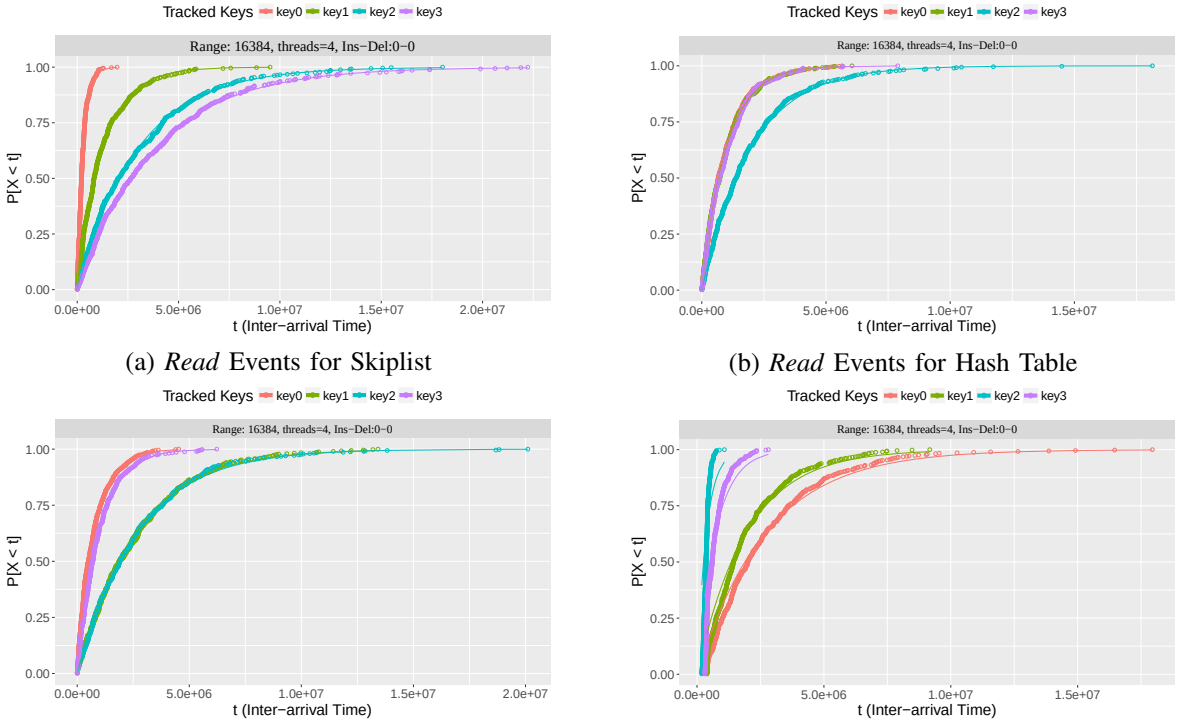
For a given node N_i, we denote by λ_i^{trav} (resp. λ_i^{read}, λ_i^{cas}) the rate of the events triggering a traversal (resp. *Read*, *CAS*) of N_i due to one thread, when $N_i \in D$. op_k^{del} (resp. op_k^{ins}, op_k^{src}) stands for a Delete (resp.

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 $\mathbb{P}[Op = op_k^o]$. $op_k^o \rightsquigarrow cas(N_i)$ (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:

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

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 traversal triggering events follows a Poisson process with rate $\lambda_i^{trav} = \lambda_i^{read} + \lambda_i^{cas}$, and that the read triggering events that originates from P' different threads and occurs at N_i follow a Poisson process with rate $P' \times \lambda_i^{read}$.

B. Validity of Poisson Process Hypothesis



(a) *Read* Events for Skiplist

(b) *Read* Events for Hash Table

(c) *Read* Events for Binary Tree

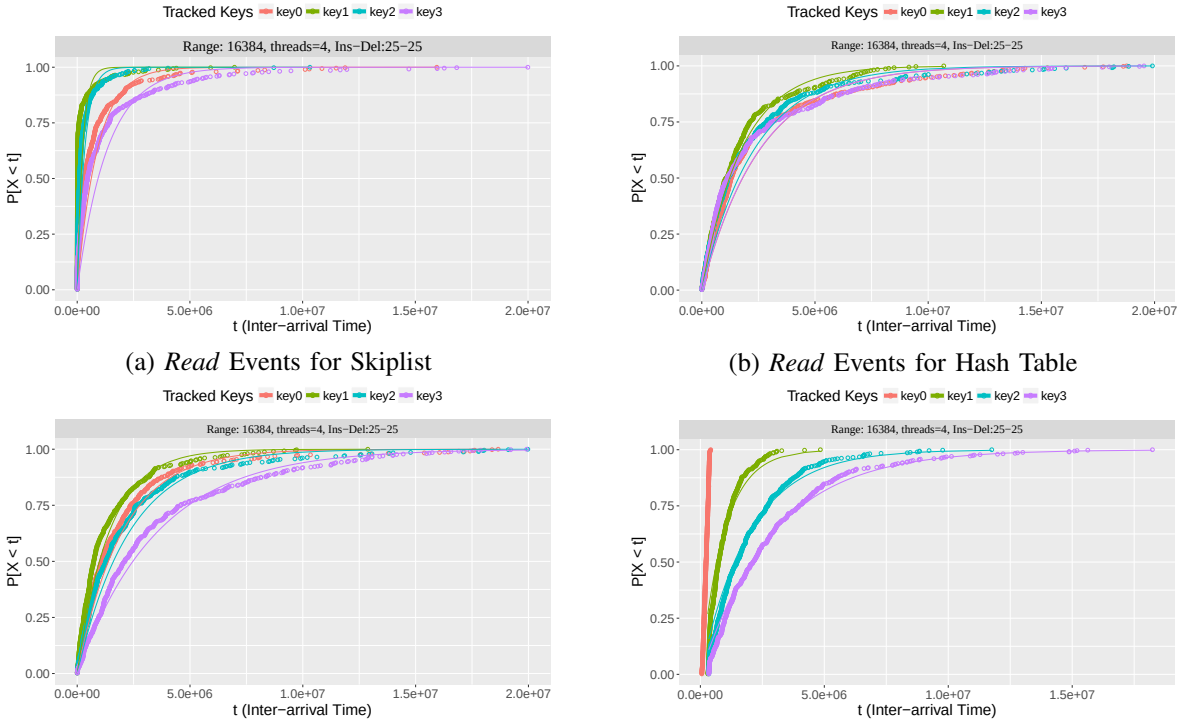
(d) *Read* Events for Linked List

Figure 2: Poisson Process Modeling - Search Only

To illustrate the validity of modeling the events as Poisson processes, we experimentally extract the cumulative distribution function of the inter-arrival latency of *Read* events that occur on a given node in a skip list and we compare it against the corresponding exponential distribution (recall that the time between events in a Poisson process is exponentially distributed).

We consider a search only scenario and $50/50$ search/update scenario. Each thread initially picks a random key and tracks the instants when a node associated with the chosen key is traversed during the execution. To facilitate the recording of the inter-arrival times, we disable the deletion of these particular keys (deletion is still enabled for any other key).

In Figure 2 and Figure 3, we illustrate the results, where the dots represent the experimental measurements and the lines are generated by exponential distributions. The mean of each distribution is instantiated as the mean of the experimental measurements. One can observe the grounds *a posteriori* of our Poisson process



(a) *Read* Events for Skiplist

(b) *Read* Events for Hash Table

(c) *Read* Events for Binary Tree

(d) *Read* Events for Linked List

Figure 3: Poisson Process Modeling - 50/50 Search/Update

modeling, and the variation of the event rates across keys, issuing from the differences between the node characteristics (key, height, location; see Section VI).

C. Impacting Factors

We have identified five factors that dominate the traversal latency of a node, distributed into two sets. On the one hand, the first set of factors only emerges in the parallel 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 traversals. Consider a cache of size C (fully associative), assume a node is traversed by a thread at time t and the next traversal (same thread and node) occurs at time t'. The thread would experience a capacity miss for the traversal at time t' if it has traversed 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 $Traverse_i$ the latency of traversing 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:

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

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.

D. 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 V-B), which links throughput with the expectation of the traversal latency of a node, computed from Sections V-A1 to V-A6. We show in these sections that they 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 VI-A (resp. VI-B, VI-C, VI-D) for linked lists (resp. hash tables, skip lists, binary trees).

V. Throughput Estimation

A. Traversal Latency

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

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

2) Stall Time: A thread experiences stall time while traversing 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 a traversal 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: $\mathbb{E}\left[CAS_i^{stall}\right] = \lambda_i^{cas}(P-1)t^{cas}(t^{cas}/2)$.

3) Invalidation Recovery: Given a thread, a coherence cache miss occurs if N_i is modified by any other thread in between two consecutive traversals 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 traversed, we look back at these events, and if among them, the last event was a *CAS* from another thread, a coherence miss occur: $\mathbb{P}[\text{Coherence Miss on } N_i] = \frac{\lambda_i^{cas}(P-1)}{\lambda_i^{cas}P + \lambda_i^{read}}$. We derive the expected latency of this factor during a traversal at N_k by multiplying this with the latency penalty of a coherence cache miss: $\mathbb{E}[CAS_i^{reco}] = \mathbb{P}[\text{coherence miss on } N_i] \times t^{rec}$.

4) Che's Approximation: Che's Approximation is a technique to estimate the hit ratio of a LRU cache, where the object (nodes for our case) accesses follow IRM (Independent Reference Model). Che's approximation is concerned with the capacity misses in a cache. We apply the approximation to the search data structures to estimate $\mathbb{E}\left[Hit_i^{cache_\ell}\right]$ and $\mathbb{E}\left[Hit_i^{tlb_\ell}\right]$. In this part, we give a brief discussion on Che's Approximation and in the following sections (see V-A5, V-A6), we have shown how we adapt this scheme for our purposes.

IRM is based on the assumption that the object references occur in an infinite sequence from a fixed catalog of $\mathcal{N}$ objects. The probability of referencing object i at any point in the sequence (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. Under LRU policy with cache of size C_ℓ^{dat} and subject to IRM demand of $\mathcal{N}$ objects, an object reference

would lead to a capacity miss if at least C_ℓ^{dat} unique object references take place after the previous reference to the same object. Let a reference to object i (O_i) occurs at time t_0, the characteristic time for the object i is defined by the random variable:

$$T_\ell^i = \inf\{t > 0 : X^i(t) = C_\ell^{dat}\}, \text{where,}$$

$$X^i(t) = \sum_{j=1, j\neq i}^{\mathcal{N}} \mathbf{1}_{t_0 < O_j \leq t}$$

Briefly, Che's approximation, first combines all T_ℓ^i, where $i \in [1..\mathcal{N}]$ in a single variable by assuming s_i is negligible compared to $\sum_{j=1}^{\mathcal{N}} s_j$ and then approximates T_ℓ^i with a constant T_ℓ^{dat} over objects. Consider a sequence of references that follows an IRM demand for $\mathcal{N}$ objects, with reference probability s_i, where $i \in [1..\mathcal{N}]$. The characteristic time T_ℓ^{dat} of a cache with size C_ℓ^{dat} is the unique solution of the following equation:

$$C_\ell^{dat} = \sum_{i=1}^{\mathcal{N}} (1 - e^{-s_i T_\ell^{dat}})$$

In [13], they analyse and illustrate the reason behind the accuracy of the approximations for a quite large spectrum of object reference distributions. Their argument relies on the random variable $X(t) = \sum_{j=1}^{\mathcal{N}} \mathbf{1}_{t_0 < O_j \leq t}$, that provides the number of unique object references that have occured in the interval $[0, t]$. As the crucial property, $X(t)$ is defined as the sum of independent random variables. Based on the central limit theorem, they show that a Gaussian approximation for this sum is quite reasonable, for all t.

Without loss of generality, let an object i is referenced consecutively at time 0 and t. We know that the second reference would be cache miss, in a cache of size C_ℓ^{dat}, if $X(t) > C_\ell^{dat}$, where by assumption $X(t)$ is a Gaussion random variable. The cache hit ratio of cacheline is given by:

$$hit_\ell^i = 1 - \int_0^{+\infty} \mathbb{P}\left[X(t) > C_\ell^{dat}\right] s_i e^{-s_i t} dt \tag{3}$$

Che's approximation, basically, approximates the cumulative distribution function of $X(t)$ with a step function that cuts this S-shaped cumulative distribution function at the $\mathbb{E}[X(t)] = \sum_{i=1}^{\mathcal{N}} (1 - e^{-s_i t})$, denoted by $m(t)$. Thus, it approximates hit_ℓ^i in Equation 3 with:

$$\begin{aligned} hit_\ell^i &\approx 1 - \int_0^{+\infty} \mathbf{1}_{m(t) > C_\ell^{dat}} s_i e^{-s_i t} dt \\ &= 1 - \int_0^{+\infty} \mathbf{1}_{t > T_\ell^{dat}} s_i e^{-s_i t} dt \end{aligned}$$

In this study, we have exploited Che's approximation to estimate the data and TLB cache hit ratios with a slight modification by keeping our arguments along the same lines with the ones presented above.

5) Cache Misses: We consider a data cache at level ℓ of size C_ℓ^{dat} and compute the hit latency due to *Read* events on this cache. We assume that N_i is either present in the search data structure or not, during the characteristic time of the cache. *Read* events at N_i are indeed much more frequent than the removal or insertion of N_i. This implies that if the characteristic time is long enough to accommodate the intervals where $N_i \in D$ and $N_i \notin D$, then the cache miss ratio of N_i should be quite low, which would be underestimated due to our assumption. We can employ the *Read* rates as popularities, *i.e.* $s_i = \lambda_i^{read}$, and modify Che's approximation to discriminate whether, at a random time, N_i is inside the data structure or not.

We integrate the masking variable P_i into Che's approximation. We have: $X^{cache}(t) = \sum_{i=1}^{\mathcal{N}} P_i \mathbf{1}_{0 < O_i \leq t}$, where O_i denotes the reference time of N_i. 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:

$$\mathbb{E}\left[X^{cache}(t)\right] = \sum_{i=1}^{\mathcal{N}} \mathbb{E}\left[P_i \mathbf{1}_{0<O_i\leq t}\right] = \sum_{i=1}^{\mathcal{N}} \mathbb{E}\left[P_i\right]\mathbb{E}\left[\mathbf{1}_{0<O_i\leq t}\right] = \sum_{i=1}^{\mathcal{N}} p_i(1-e^{-\lambda_i^{read}t}).$$

Lastly, we solve the equation for the characteristic time T_ℓ^{dat} of level ℓ cache: $\sum_{i=1}^{\mathcal{N}} p_i(1-e^{-\lambda_i^{read}T_\ell^{dat}}) = C_\ell^{dat}$ thanks to a fixed-point approach. After computing T_ℓ^{dat}, we estimate the cache hit ratio (on level ℓ) of N_i: $1-e^{-\lambda_i^{read}T_\ell^{dat}}$.

6) 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 managements 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 $s_i = \lambda_i^{read}$ for $i \in [1..\mathcal{N}]$ 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_i^j), leading to a success if N_i is mapped into page j, with probability $p_i/\mathcal{M}$ (hence Y_i^j does not depend on j). Under IRM, we can then express the page references as point processes with rate $r_j = \sum_{i=1}^{\mathcal{N}} Y_i^j s_i$, 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:

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

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 of 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

$$z_i = \lambda_i^{read} + \mathbb{E}\left[\sum_{j=1, j\neq i}^{\mathcal{N}} Y_j^k \lambda_j^{read}\right] = \lambda_i^{read} + \sum_{j=1, j\neq i}^{\mathcal{N}} p_j \lambda_j^{read}/\mathcal{M}.$$

7) Interactions: To be complete, we mention the interaction between impacting factors and the possibility of latency overlaps in the pipeline. Firstly, the traversal latency of different nodes can not be overlapped due to the semantic dependency for the linked nodes. For a single node traversal, the latency for cas execution and stall time can not be overlapped with any other factor. 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 (coherence) cost is induced (*i.e.* $\mathbb{E}\left[Hit_i^{cache_\ell}\right] = (1-\mathbb{P}\left[coherence\ miss\right])(\mathbb{P}\left[hit\ cache_l\right] - \mathbb{P}\left[hit\ cache_{l-1}\right])t_\ell^{dat}$).

B. Latency vs. Throughput

In the previous sections, we have shown how to compute the expected traversal latency for a given node. There remains to combine these traversal 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 $\lambda_i^{trav} = \lambda_i^{read} + \lambda_i^{cas}$. Thus the average arrival rate of threads to N_i is: $p_i\lambda_i^{trav}$. It can then be passed to Little's Law [17], which states that the expected number of threads (denoted by t_i) traversing N_i obeys to $t_i = p_i\lambda_i^{trav}\mathbb{E}\left[Traverse_i\right]$. 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 ($\mathbb{E}\left[Traverse_0\right] = t^{app}$) and its average arrival rate is equal to the throughput ($\lambda_0^{trav} = \mathcal{T}$). Then, we have: $\sum_{i=0}^{\mathcal{N}} t_i = \sum_{i=0}^{\mathcal{N}}(p_i\lambda_i^{trav}\mathbb{E}\left[Traverse_i\right])$, where λ_i^{trav} and $\mathbb{E}\left[Traverse_i\right]$ 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 $\lambda_i^{trav} = a_i\mathcal{T}$ and $\mathbb{E}\left[Traverse_i\right] = b_i\mathcal{T} + c_i$ 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 expected throughput, $\mathcal{T}$.

VI. Instantiating the Throughput Model

In this section, we show how to initialize 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 $\mathbb{P}\left[op_k^o \rightsquigarrow e(N_i) \,|N_i \in D\right]$, *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.

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.

A. Linked List

We start with the lock-free linked list implementation of Harris [15]. 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 $t \in \{\mathsf{Insert}, \mathsf{Delete}, \mathsf{Search}\}$ 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, $\mathbb{P}\left[op_{k'}^o \rightsquigarrow read(N_k)\right] = 1$ if $k \leq k'$ and $\mathbb{P}\left[op_{k'}^o \rightsquigarrow read(N_k)\right] = \prod_{i=k'}^{k-1}(1-p_i)$ 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:

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

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

B. Hash Table

We analyse here a chaining based hash table where elements are hashed to B buckets implemented with the lock-free linked list of Harris [15]. 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 traversal rate from node to node, but the node associated with a given key does not show significant variation in its traversal 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 traversal 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 traversed 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 traverse 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.

C. Skip List

There exist various lock-free skip list implementations and we study here the lock-free skip list [23]. 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.

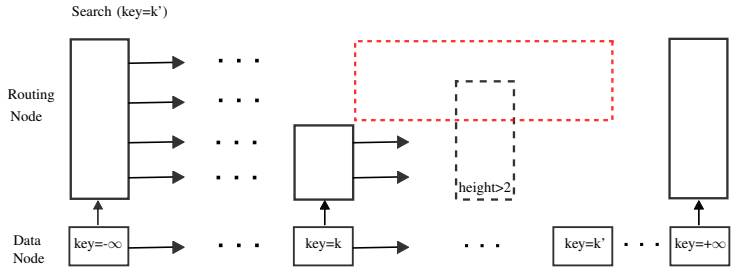


Figure 4: Skip List Events: Read Event Probability

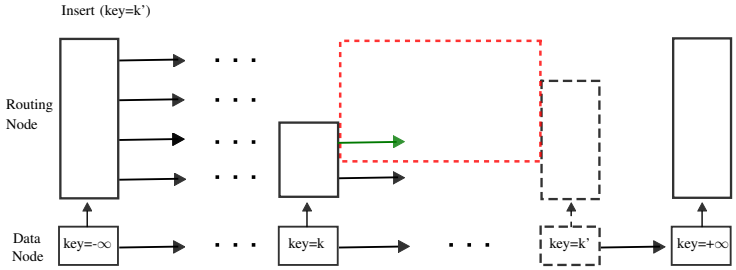


Figure 5: Skiplist Events: *CAS* Event Probability

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 4, 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:

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

Next, we apply a similar approach for *CAS* events. In Figure 5, we illustrate an example. A *CAS* event occurs at the green pointer, as a result of the removal (or insertion) of K_k if there is no node in the red frame. For all node and operation couples, $\mathbb{P}\left[op^o_{k'} \rightsquigarrow cas\left(N^z_{k,h}\right)\right]$ is simply obtained in those lines.

The insertion of an element with $K_{k'}$ introduces $N^z_{k',h}$ with probability $2^{-(h+1)}$ if $h \in [1..h_{max} - 1]$, and $1 - \sum_{i=0}^{h_{max}-1} 2^{-(h+1)}$ when the maximum height. The data node is linked to the list at the bottom layer with a *CAS* that is executed on the previous data node. If a routing node is introduced, it is linked to lists at h different layers, thus leads to h *CAS* instructions that are applied on the other nodes.

The deletion of an element is composed of two phases. The first phase is to mark the data node, $N^{dat}_{k',h}$ and the pointers in the routing node with height k', if it exists. If the height of the routing node is more than one, it is possible that multiple *CAS* intructions are executed on the same routing node. But, we only consider the

first one. The latency and also the effect of remaining ones would be negligible, as they are applied on the same cacheline one after each other. This repetitive behavior guarentees that the cacheline has already been exclusively owned before the next *CAS* instructions run. To recall, this is consistent with our assumption that an event can occur at most once per operation on a node. The second phase of deletion operation follows the same path with the insertion operation. Simply, a *CAS*, on the previous node, is executed for each layer that the data and routing nodes span.

We have denoted the success probability of an Insert operation with $q_{k'} = \frac{\mathbb{P}\left[op=op_{k'}^{\mathsf{Insert}}\right]}{\mathbb{P}\left[op=op_{k'}^{\mathsf{Insert}}\right]+\mathbb{P}\left[op=op_{k'}^{\mathsf{Delete}}\right]}$. Also, the factor $2^{-(h+1)}$ provides the probability of the insertion of a routing node with height h, coupled with its data node. Based on the non-existence of any node that overlaps with the area that is enclosed with the red frame in Figure 5, we obtain:

$$\mathbb{P}\left[op_{k'}^{ins} \rightsquigarrow cas\left(N_{k,h}^{z}\right)\right] = \begin{cases} (1-q_{k'})(\sum_{h=0}^{h_{max}} 2^{-(h+1)}(\prod_{x=k+1}^{k'-1}(1-(\sum_{y=h}^{h_{max}} p_{x,y}^{z})))), & \text{if } k < k' \\ 0, & \text{if } k \geq k' \end{cases}$$

$$\mathbb{P}\left[op_{k'}^{del} \rightsquigarrow cas\left(N_{k,h}^{z}\right)\right] = \begin{cases} 1, & \text{if } k = k' \\ q_{k'}(\sum_{h=0}^{h_{max}} 2^{-(h+1)}(\prod_{x=i+1}^{k'-1}(1-(\sum_{y=h}^{h_{max}} p_{x,y}^{z})))), & \text{if } k < k' \\ 0, & \text{if } k > k' \end{cases}$$

D. 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 as external nodes contain the key-value pair (valued node). We use the external binary tree of Natarajan [19] to initialize 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 traversed 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 [22]. In the same vein, we compute the traversal 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 traversing 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 traversed 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 traversed 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 traversed (if any): $N^{ext}_{k'}$, 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:

$$\mathbb{P}\left[op^o_{k'} \rightsquigarrow read\big(N^{int}_k\big)\right] = \begin{cases} 1/(1+\sum_{i=k'+1}^{k-1} p^{int}_i), \text{ if } k > k' \\ 1/(1+\sum_{i=k+1}^{k'} p^{int}_i), \text{ if } k \leq k' \end{cases},$$

$$\mathbb{P}\left[op^o_{k'} \rightsquigarrow read\big(N^{ext}_k\big)\right] = \begin{cases} 1, & \text{if } k = k' \\ \prod_{i=k+1}^{k'}(1-p^{ext}_i), & \text{if } k < k' \\ \prod_{i=1}^{k-1}(1-p^{ext}_i), & \text{if } k > k' \end{cases}$$

These probabilities finally lead to the computation of the *Read* (resp. *CAS*) rates $\lambda^{read}_{z,k}$ (resp. $\lambda^{cas}_{z,k}$) of N^z_k, 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^{int}_k depends on the size of the subtree whose root is N^{int}_k, which is expected to vary with the update operations on the tree. We dig more into this and reflect these variations by decomposing N^{int}_k into H_k virtual nodes, $N^{int}_{k,h}$, where $h \in [1..H_k]$. We define the *Read* rate $\lambda^{read}_{int,k,h}$ of these virtual nodes as a weighted sum of the initial node rate thanks the two equations $p^{int}_k = \sum_{h=1}^{H_k} p^{int}_{k,h}$ and $p^{int}_k \lambda^{read}_{int,k} = \sum_{h=1}^{H_k} p^{int}_{k,h} \lambda^{read}_{int,k,h}$.

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: $\lambda^{read}_{int,k,h} \propto h\lambda^{read}_{int,k}$. 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^{int}_k, we can evaluate the weight of the virtual nodes: $p^{int}_{k,h} = p^{int}_k \mathbb{P}\left[Sub_k = h\right]$.

We have computed $\lambda^{read}_{int,k}$. These values reflect the average behaviour along the whole execution. However, the average behavior is not enough to computethe traversal latency accurately for the internal nodes. In the execution, there are different time intervals where $\lambda^{read}_{int,k}$ show significant variation depending on the part of the tree that it is located. For instance, it is quite improbable to observe a cache miss at N^{int}_k when it is positioned at the root of the tree. One would observe a very high rate of traversals with low latency in this case, which decreases the expected traversal latency of N^{int}_k significantly. An accurate estimation for the cache misses requires the consideration of this particularity of the binary tree. To approximate the impact of this variation, we split N^{int}_k into a number (let H_k denotes this number for N^{int}_k) of independent virtual nodes (in the lines of independent reference model), each representing the behavior of N^{int}_k with a different *Read* rate. The virtual node, with *Read* rate $\lambda^{read}_{int,k,h}$, is denoted by $N^k_{h,int}$. We will obtain the *Read* rates $\lambda^{read}_{int,k,h}$ and presence probabilities $p^{int}_{k,h}$ for these virtual nodes by requiring that the average behaviors are still valid: $p^{int}_k = \sum_{h=1}^{H_k} p^{int}_{k,h}$ and $p^{int}_k \lambda^{read}_{int,k} = \sum_{h=1}^{H_k} p^{int}_{k,h} \lambda^{read}_{int,k,h}$.

Theorem 1. *For an external binary tree with N internal nodes, generated with the random permutation of insertions, the probability mass function of the size of the subtree (the random variable concerns only the number of the internal nodes and denoted by Sub_k) that is rooted at N^{int}_k is given by: $\mathbb{P}\left[Sub_k = N\right] = 1/N$ and $\mathbb{P}\left[Sub_k = s\right] = O(1/s^2)$.*

Proof. It is clear that $\mathbb{P}\left[Sub_k = N\right] = 1/N$ since it occurs iff N^{int}_k has the highest priority among all internal nodes. For the rest, we consider four different cases. Let σ_k denotes the index of N^{int}_k in the permutation of the sequence of N internal nodes that are arranged in the ascending order based on their keys.

(i) $\sigma_k + s \leq N$ and $\sigma_k - s \geq 1$: then there exist s distinct pairs of (N^{int}_j, N^{int}_i) such that $\sigma_i - \sigma_j = s+1$ and $\sigma_j < \sigma_k < \sigma_i$. Given a pair of such (N^{int}_j, N^{int}_i), $Sub_k = s$ if the priorities of N^{int}_j and N^{int}_i are higher than the priorities of all N^{int}_x, such that $\sigma_j < \sigma_x < \sigma_i$ and also N^{int}_k has a higher priority than all $N^{int}_{y \neq k}$ such that $\sigma_j < \sigma_y < \sigma_i$. This ($N^{int}_k$ is the root of subtree that includes all N^{int}_y, such that $\sigma_j < \sigma_y < \sigma_i$) can happen with probability, $\frac{2}{(s+2)(s+1)s}$. There exist s such non-overlapping cases. We have, $\mathbb{P}\left[Sub_k = s\right] = \frac{2}{(s+1)(s+2)}$.

(ii) $\sigma_k + s > N$ and $\sigma_k - s \geq 1$: then there exist a N^{int}_i such that $\sigma_i = N - s$. $Sub_k = s$ if N^{int}_i has higher priority than all N^{int}_x, such that $\sigma_i < \sigma_x \leq N$ and N^{int}_k has higher priority than all N^{int}_y, such that $\sigma_i < \sigma_{y \neq k} \leq N$. This can happen with probability, $\frac{1}{(s+1)s}$. In addition, there can be at least 0

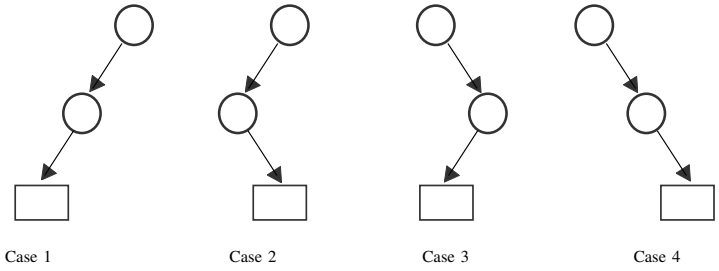


Figure 6: Binary Tree CAS Probability

and at most $s-1$ distinct pairs of (N_j^{int}, N_i^{int}) such that $\sigma_i - \sigma_j = s+1$ and $\sigma_j < \sigma_k < \sigma_i$. We have: $\frac{1}{(s+1)s} \leq \mathbb{P}[Sub_k = s] \leq \frac{1}{(s+1)s} + \frac{2(s-1)}{(s+1)(s+2)s}$.

(iii) $\sigma_k + s \leq N$ and $\sigma_k - s < 1$: The bound at (ii) applies to this case also.

(iv) $\sigma_k + s > N$ and $\sigma_k - s < 1$: then there exist a N_i^{int} such that $\sigma_i = N - s$ and a N_j^{int} such that $\sigma_j = s+1$. In addition, there can be at least 0 and at most $s-2$ distinct pairs of nodes (N_j^{int}, N_i^{int}) such that $\sigma_i - \sigma_j = s+1$ and $\sigma_j < \sigma_k < \sigma_i$. Similar to (i) and (ii), we obtain and sum the probabilities lead to $Sub_k = s$. We have: $\frac{2}{(s+1)s} \leq \mathbb{P}[Sub_k = s] \leq \frac{2}{(s+1)s} + \frac{2(s-2)}{(s+1)(s+2)s}$ □

We start with an observation. The *Read* rate of N_k^{int} is proportional to the size of the subtree that is rooted at N_k^{int}. Given a binary tree of N internal nodes, the size of the subtree can vary in the interval $[1, N]$, which means that we can have $H_k = N$ different *Read* rate levels ($\lambda_{int,k,h}^{read}$) associated with their presence probabilities $p_{k,h}^{int} = p_k^{int}\mathbb{P}[Sub_k = h]$. Relying on Theorem 1, one can observe that $\mathbb{P}[Sub_k = h]$ do not variate much from $c_1/(h+1)^2$ for the majority of different values of h and k. Therefore, we approximate $\mathbb{P}[Sub_k = h] \approx c_1/(h+1)^2$, with a single constant c_1 for all k and $h < H_k$. We know, $\sum_{h=1}^{H_k}\mathbb{P}[Sub_k = h] = 1$ and $\mathbb{P}[Sub_k = H_k] = 1/H_k$. So, we obtain $c_1 \approx 2$ by solving the equation $\int_{h=2}^{N}(c_1/h^2)dh = (N-1)/N$. We set $p_{k,h}^{int} = p_k^{int}(2/(h+1)^2)$ and $p_{k,H_k}^{int} = p_k^{int}/H_k$. Assuming $\lambda_{int,k,h}^{read} = c_2 h \lambda_{int,k}^{read}$ (*Read* rates are proportional to the subtree size), we require $p_k^{int}\lambda_{int,k}^{read} = \sum_{h=1}^{H_k} p_{k,h}^{int}\lambda_{int,k,h}^{read}$, which leads to $\lambda_{int,k}^{read} \approx c_2 + \int_{h=2}^{H_k}(2/h^2)c_2(h-1)\lambda_{int,k}^{read}dh$. We solve and obtain $c_2 \approx 1/(2\ln H_k)$. We set $\lambda_{int,k,h}^{read} = h\lambda_{int,k}^{read}/(2\ln H_k)$, for the virtual internal nodes.

Now, we consider the *CAS* events. Delete and Insert operation start with the search phase. Insert operation finalize with a *CAS* executed at the grandparent internal node of the inserted external key. Delete operation contains three *CAS*; (i) one at the grandparent internal node of the deleted external key; (ii) two that are executed consecutively at the parent node of the external key. Thus, we consider them as a single *CAS* instruction, since the second of the consecutive ones has a negligible cost because the cacheline has already been exclusively owned by the thread.

Similar to *Read* events, we first find the rate of *CAS* events for N_k^{int} and split these events to virtual nodes by requiring the average behavior is still valid: $p_k^{int}\lambda_{int,k}^{cas} = \sum_{h=1}^{H_k} p_{k,h}^{int}\lambda_{int,k,h}^{cas}$. To determine the target of *CAS* event, we need to determine the probability of an internal node N_k^{int} to be the grandparent or parent of the targetted $N_{k'}^{ext}$. We examine four different cases as illustrated in Figure 6. Given that we are in the first case, we look for the probability that $N_k^{int}, k' < k$, to possess the smallest or second smallest key, that is bigger than k', among the internal nodes that are present in the tree. Such internal nodes with the smallest key and the second smallest key corresponds to the parent and grandparent of $N_{k'}^{ext}$, respectively. For case 1, it is possible that the grandparent node is the node which has the xth, $x > 1$, smallest key that is bigger than i, that is present in the tree. But this probability decreases exponentially as x increases. That is why, we have attributed the *CAS* events that takes place at the granparent node to the node with second smallest key that is bigger than k'. For case 2, the parent corresponds to the smallest key that is bigger than k' and the grandparent corresponds to the biggest key that is smaller than k', that are present in the tree.

Formally, let $P_{k'}^B = \{i : i \geq k', N_i^{int} \in D\}$ and $P_{k'}^S = \{i : i < k', N_i^{int} \in D\}$. For the first case, we are interested in the probability that N_k^{int} is the grandparent or parent node of $N_{k'}^{ext}$. These are given by $\mathbb{P}\left[k = \sup\{P_{k'}^S - \sup\{P_{k'}^S\}\}\right]$ and $\mathbb{P}\left[k = \sup\{P_{k'}^S\}\right]$ respectively. For the second case, we are interested in

$\mathbb{P}\left[k = \sup\{P^S_{k'}\}\right]$ and $\mathbb{P}\left[k = \inf\{P^B_{k'}\}\right]$. The third and fourth cases follows the same lines as they are the flipped versions of the case one and two. For all non-sentinel nodes, we have $p^{int}_k = p$. First, we compute the following probabilities:

For $k \geq k'$ we have: (these probabilities are zero if $k < k'$)

$$\mathbb{P}\left[k = \sup\{P^S_{k'} - \sup\{P^S_{k'}\}\}\right] = p(k' - i)(1-p)^{(k'-k-1)}$$

$$\mathbb{P}\left[k = \sup\{P^S_{k'}\}\right] = (1-p)^{(k'-k)}$$

And for $k < k'$: (these probabilities are zero if $k \geq k'$)

$$\mathbb{P}\left[k = \inf\{P^B_{k'}\}\right] = (1-p)^{(k-k'-1)}$$

$$\mathbb{P}\left[k = \inf\{P^B_{k'} - \inf\{P^B_{k'}\}\}\right] = p(k - k' - 1)(1-p)^{(k-k'-2)}$$

Based on Lemma 1 (assuming a constant tree size), we obtain the expected number of internal nodes that route the search to its left child ($c_{k',l}$) and right child($c_{k',r}$) for an operation that is targetted to $key = k'$. On this route, we compute the probability of a random node to be the left (right) child of its parent, with $l_{k'} = c_{k',l}/(c_{k',l} + c_{k',r})$ (and similarly $r_{=}c_{k',r}/(c_{k',l} + c_{k',r})$). And, we estimate the probability of observing a case at a random time by using these values (*i.e.* $l^2_{k'}$ for Case 1, $l_{k'}r_{k'}$ for Case 2). And finally, we obtain:

$$\begin{aligned}\mathbb{P}\left[op^{del}_{k'} \rightsquigarrow cas\left(N^{int}_k\right)\right] =& p^{int}_{k'}(l^2_{k'}\mathbb{P}\left[k = \inf\{P^B_{k'} - \inf\{P^B_{k'}\}\}\right] \\ &+ l_{k'}(r_{k'}+1)\mathbb{P}\left[k = \inf\{P^B_{k'}\}\right] \\ &+ r_{k'}(l_{k'}+1)\mathbb{P}\left[k = \sup\{P^S_{k'}\}\right] \\ &+ r^2_{k'}\mathbb{P}\left[k = \sup\{P^S_{k'} - \sup\{P^S_{k'}\}\}\right])\end{aligned}$$

$$\begin{aligned}\mathbb{P}\left[op^{ins}_{k'} \rightsquigarrow cas\left(N^{int}_k\right)\right] =& (1 - p^{int}_{k'})(l^2_{k'}\mathbb{P}\left[k = \inf\{P^B_{k'} - \inf\{P^B_{k'}\}\}\right] \\ &+ l_{k'}r_{k'}\mathbb{P}\left[k = \inf\{P^B_{k'}\}\right] \\ &+ r_{k'}l_{k'}\mathbb{P}\left[k = \sup\{P^S_{k'}\}\right] \\ &+ r^2_{k'}\mathbb{P}\left[k = \sup\{P^S_{k'} - \sup\{P^S_{k'}\}\}\right])\end{aligned}$$

Lastly, we split the *CAS* events to the virtual nodes. *CAS* events can happen at the internal nodes only when they are in the last two levels of the tree (or similarly when the size of the subtree that is rooted at the concerned internal node is in the interval $[1, 3]$). We required the average behaviour to be valid and set $\lambda^{cas}_{int,k,x} = p^{int}_k \lambda^{cas}_{int,k}/(p^{int}_{k,1} + p^{int}_{k,2} + p^{int}_{k,3}), \forall x \in \{1, 2, 3\}$. For the cases where the operation key selection follows a zipf distribution, there exist a small region of the tree that the most operations concentrate. The update operations concentrate to that region so that the nodes are expected to change levels frequently. This means that the impact of invalidation recovery factor can be seen while the node is at an level. For this impacting factor, for zipf distribution, we split the events to virtual nodes evenly, $\forall h, \lambda^{cas}_{int,k,h} = \lambda^{cas}_{int,k}$.

VII. 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.

A. 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 [20], [2]. 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^{rec}_{low} (resp. t^{rec}_{high}), 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^{rec}_{low} and t^{rec}_{high}, 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 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 implementation, we have used ASCYLIB library [8] that is coupled with an epoch based memory management mechanism which introduces negligible latency.

B. Search Data Structures

1) Linked List: Figures 7, 8 and 9 illustrates the results for the lock-free linked list, for various scenarios that are described before (see VII). For the majority of the cases, our estimates look reasonable except the cases where the cache miss ratios are underestimated due to the limitations of the independent reference assumption. The assumption in the Independent Reference Model is that the event at the different nodes are independent Poisson Processes. A linked list operation reveals a high degree of spatial locality, implying that the Poisson Processes for the different nodes are indeed dependent. This inaccuracy illustrates indeed the importance of the accurate estimations for the event latencies that are needed to capture the practical performance.

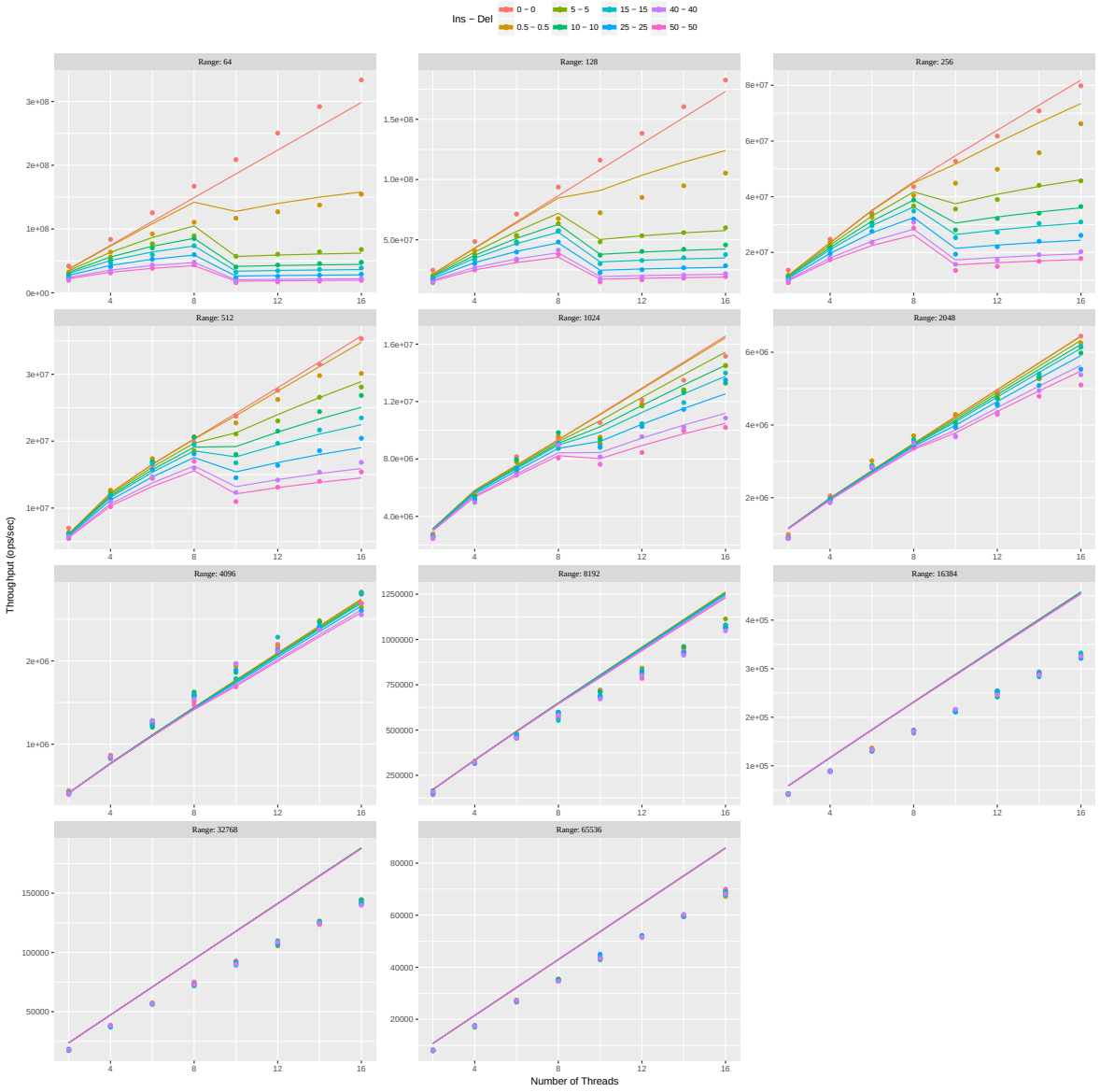


Figure 7: LL Uniform distribution for key selection

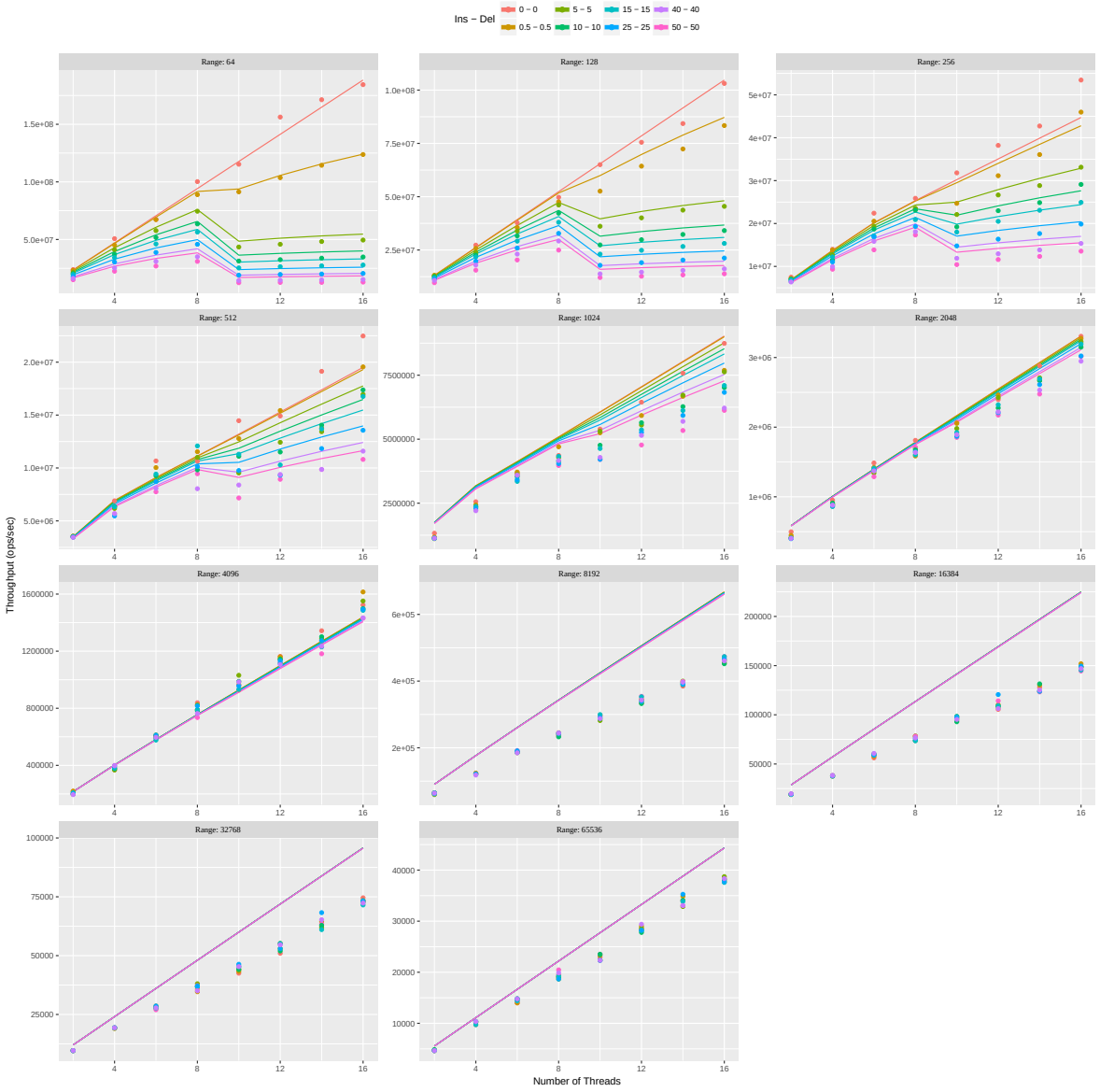


Figure 8: LL Zipf distribution for key selection

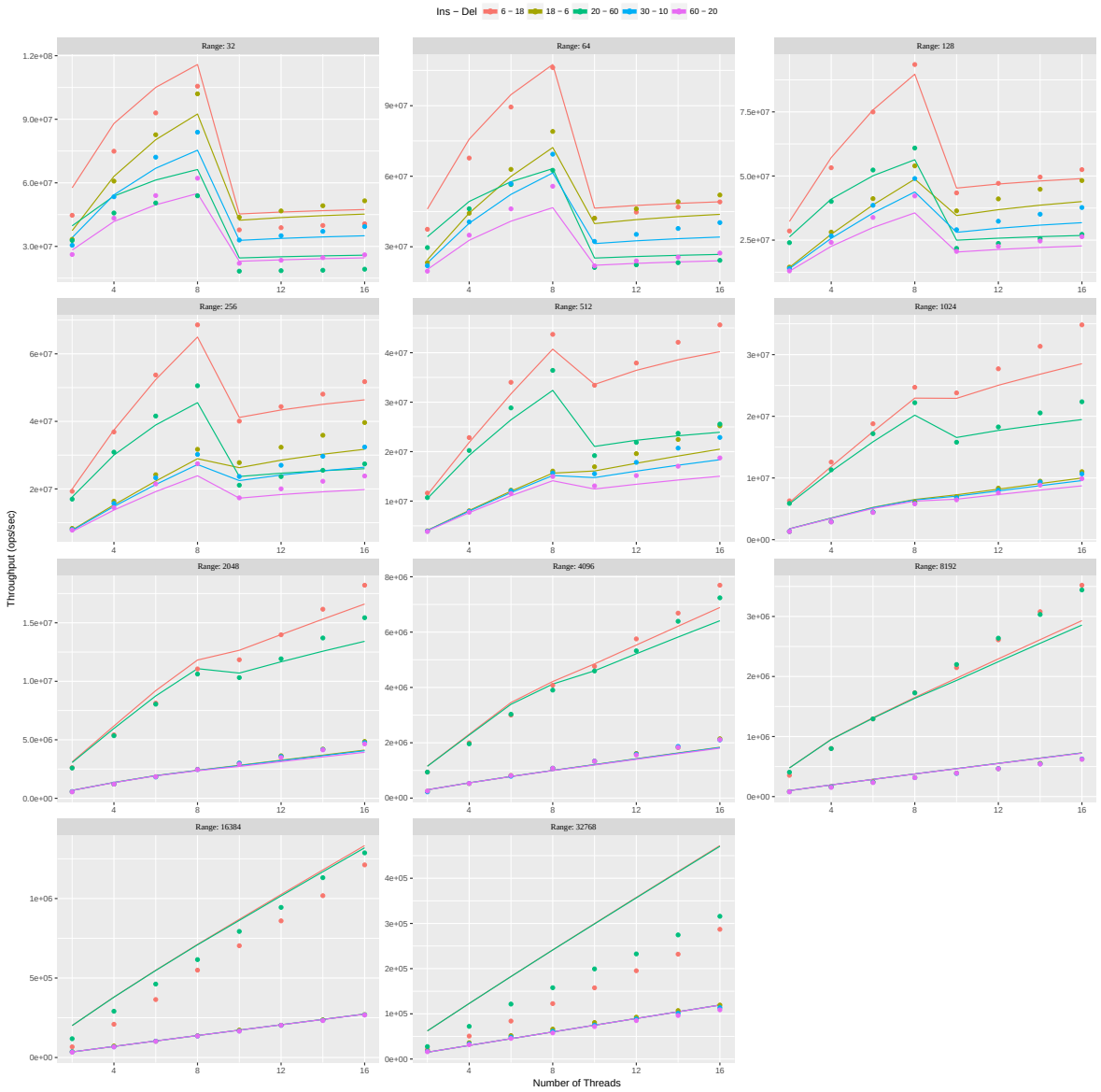


Figure 9: LL asymmetric update rates, uniform distribution for key selection

2) Hash Table: Figure 10, 11 and 12 illustrates the results for the lock-free hash table with different load factor values (number of slots per bucket) where the key selection process is initiated with uniform distribution. Figure 13 shows the results for a case where the selection process follows zipf distribution. Lastly, Figure 14 reveals the results for asymmetric delete and insert operation ratios where the key selection is done with uniform distribution. For the hash table, our estimates are able to capture the real behavior almost for all cases with satisfactory precision.

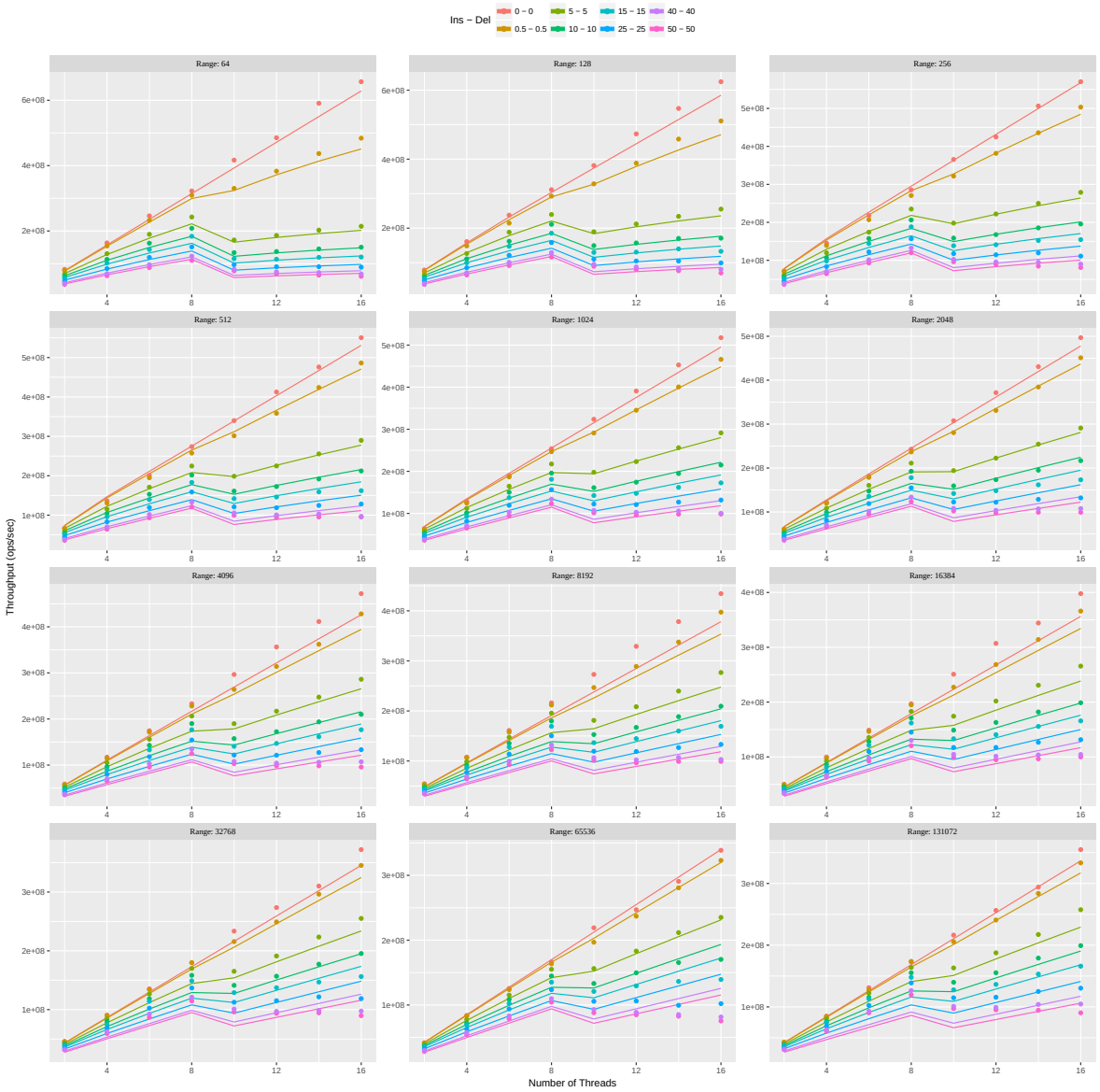


Figure 10: HT Uniform distribution for key selection, with load factor=2

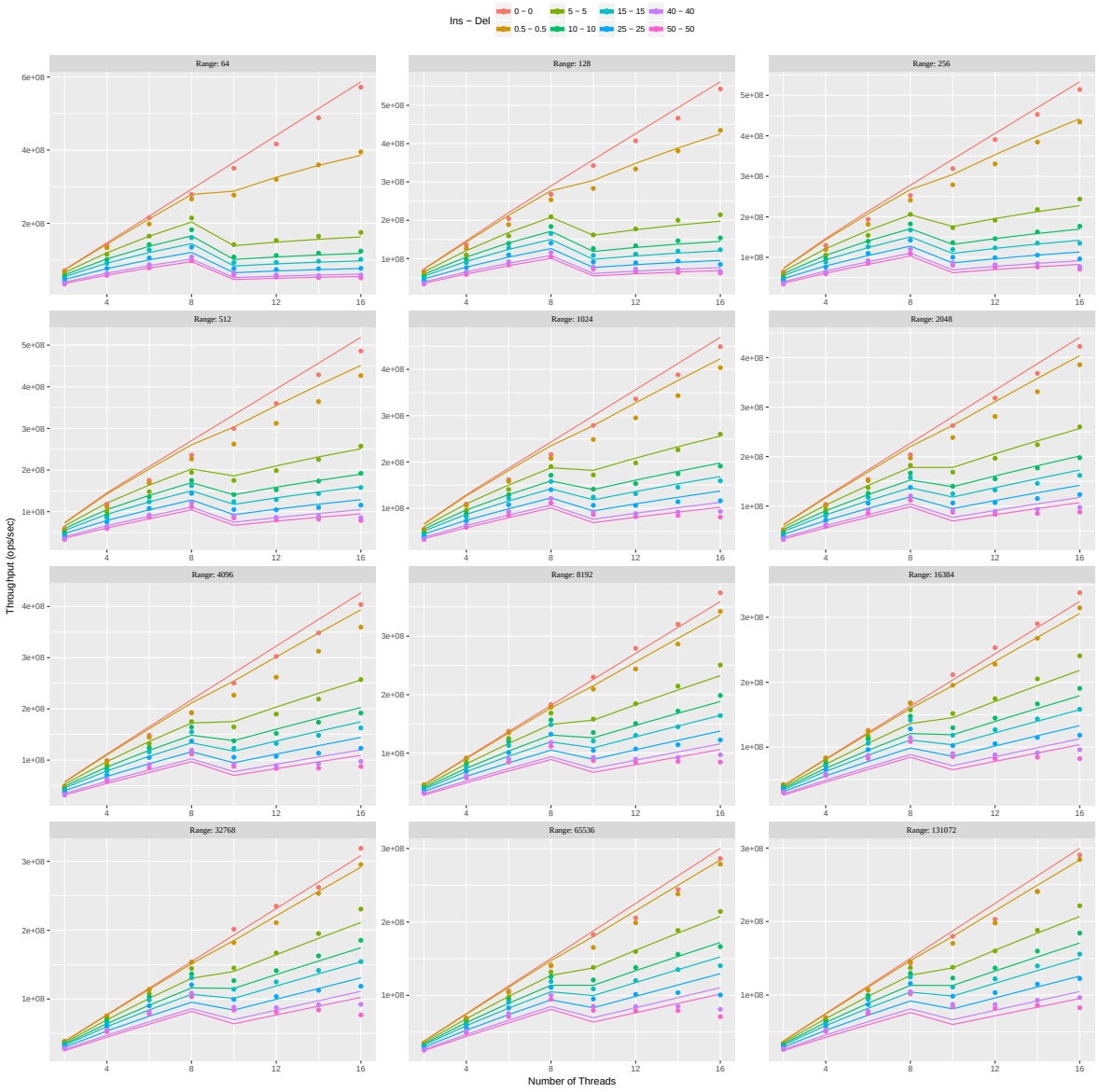


Figure 11: HT Uniform distribution for key selection, with load factor=4

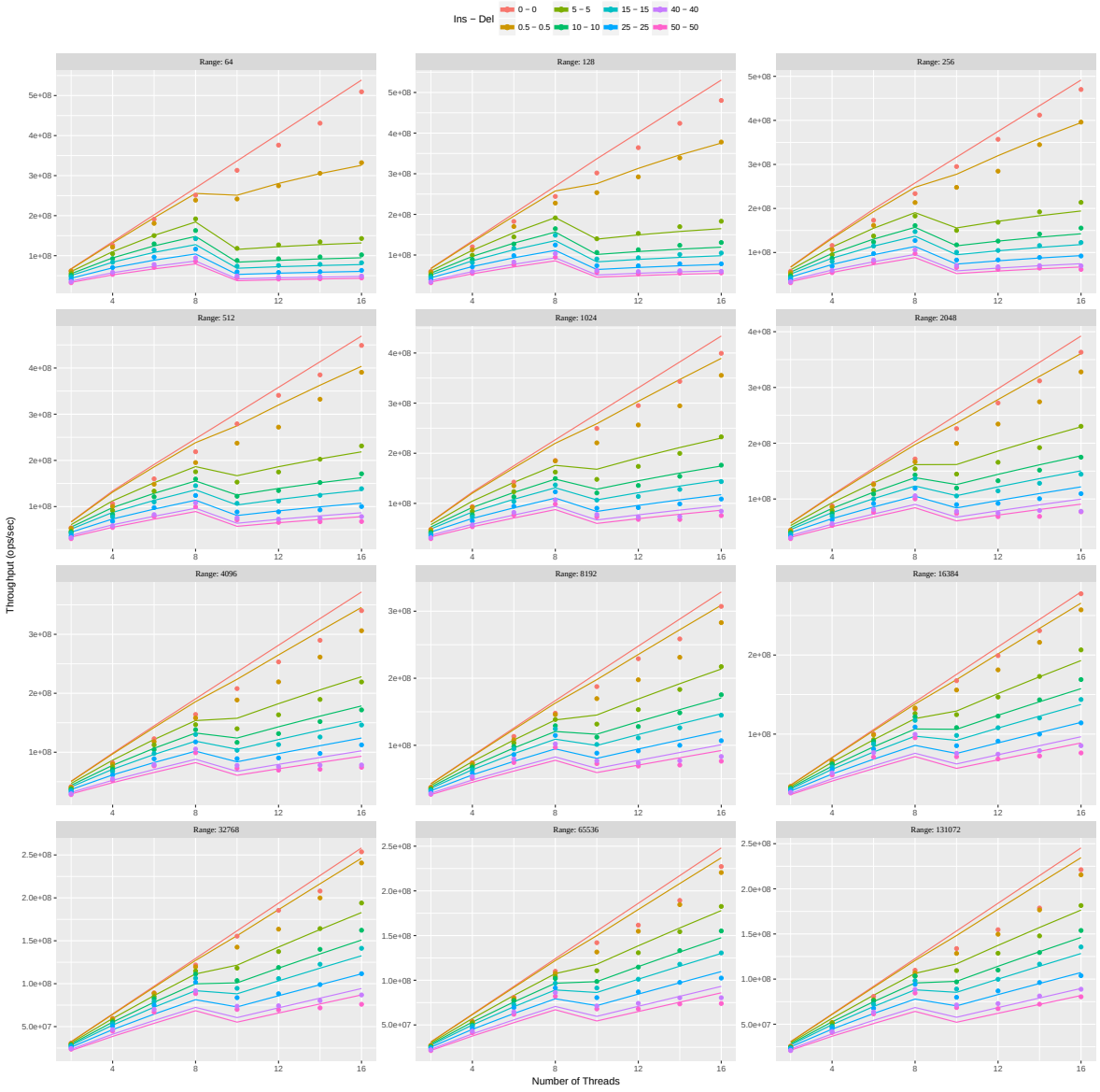


Figure 12: HT Uniform distribution for key selection, with load factor=8

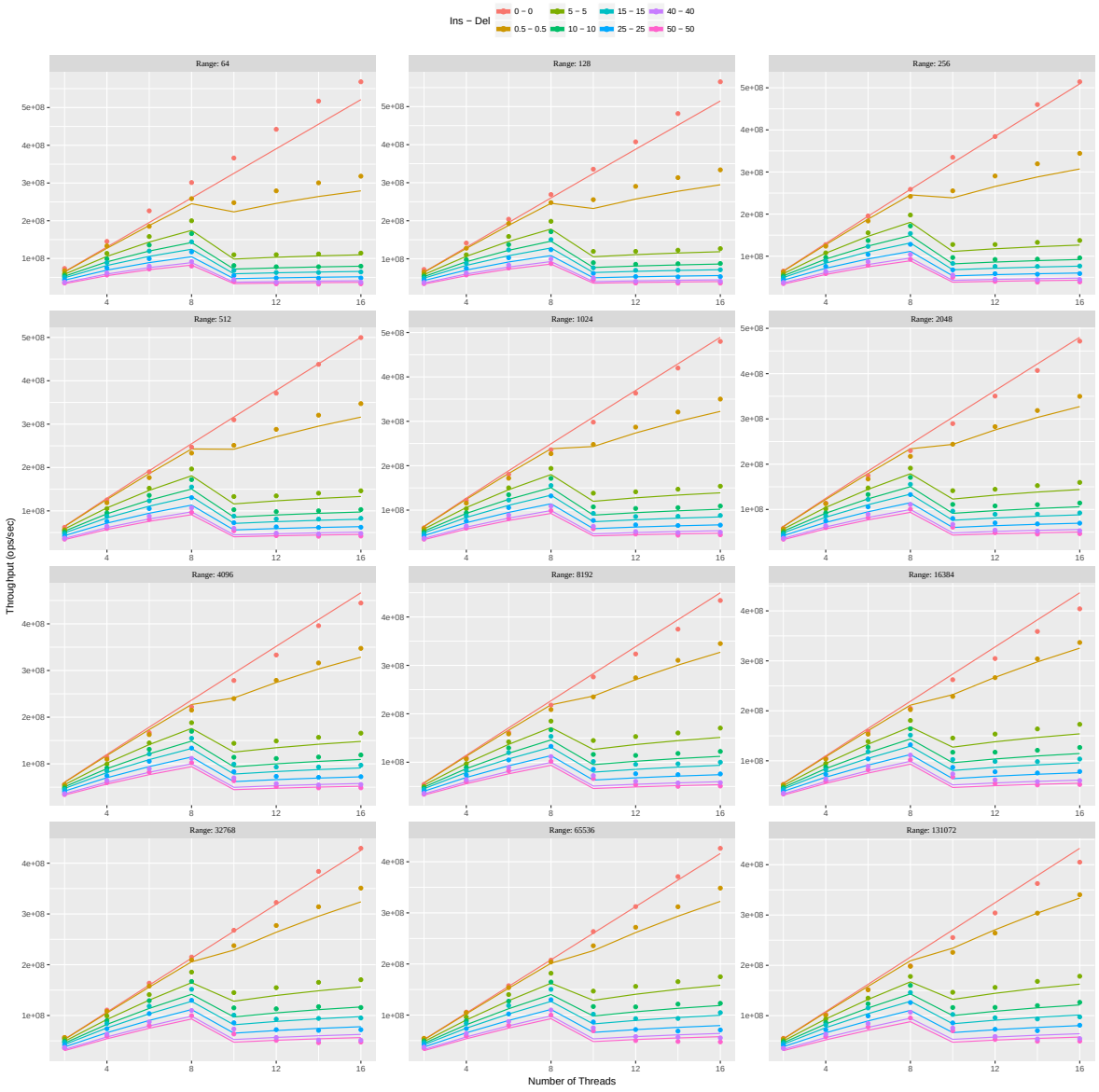


Figure 13: HT Zipf distribution for key selection, with load factor=2

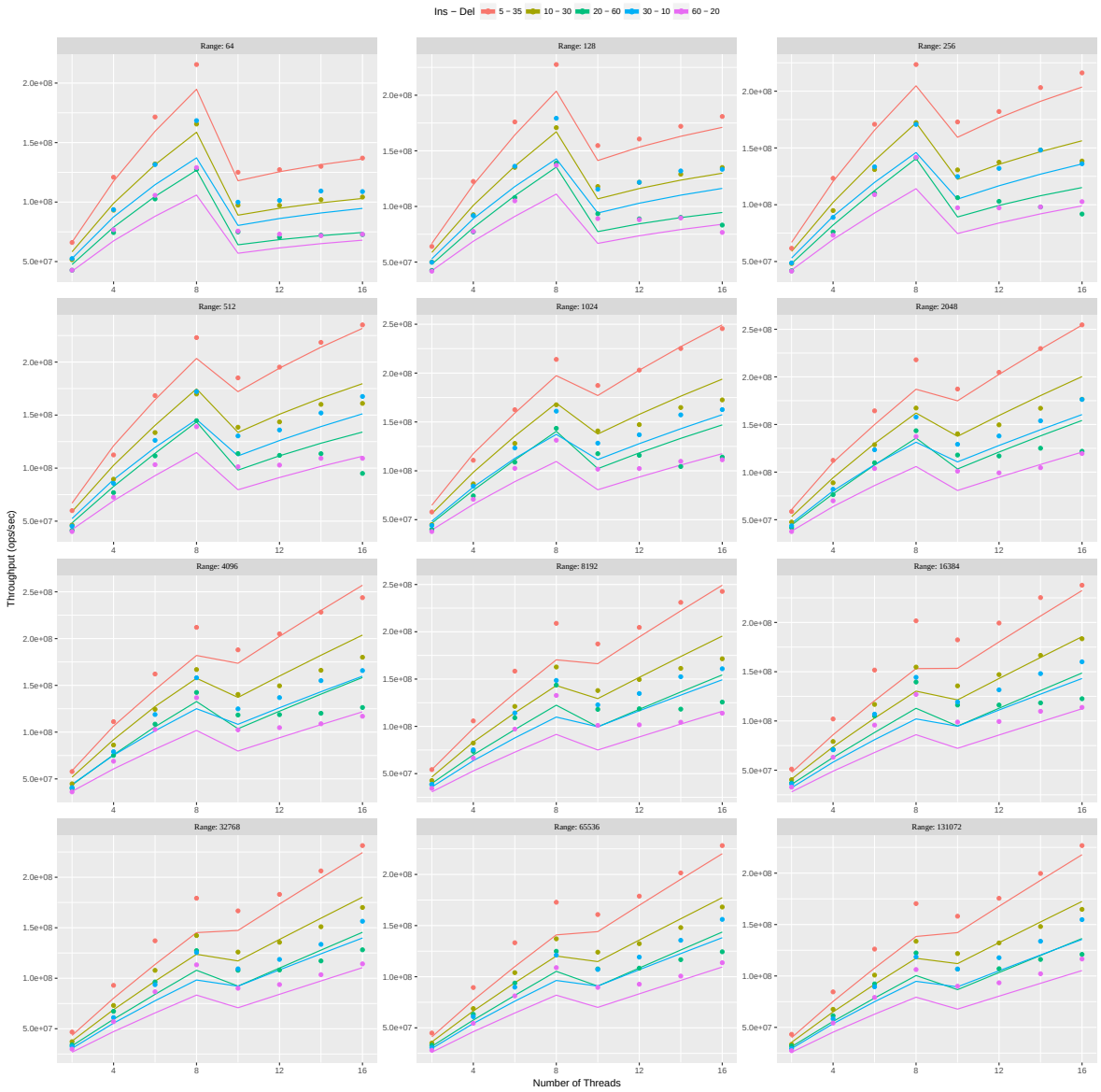


Figure 14: HT asymmetric update operations, Uniform distribution for key selection, with load factor=4

3) Skip List: Figure 15, 16 and 17 illustrates the results for the lock-free skip list, for various scenarios that are described before (see VII), where the estimations often closely follow the real behavior. In Figure 17, we observe that our estimation show some deviation from the real behavior, for the cases where key range is small and Delete ratio is higher than Insert. For such cases, the expected size of search data structure tends to be very small which might lead to inaccuracies.

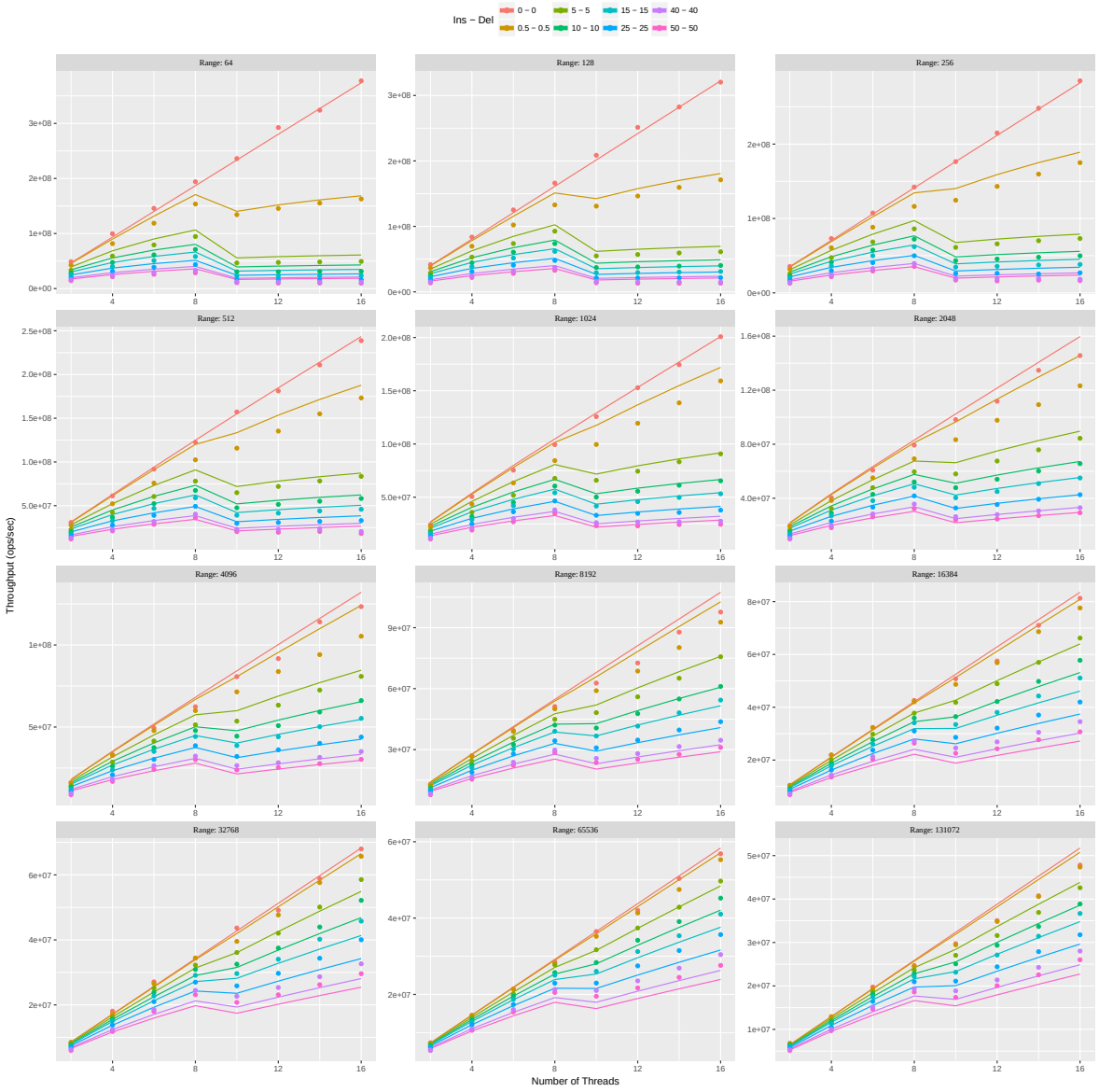


Figure 15: Skiplist Uniform distribution for key selection

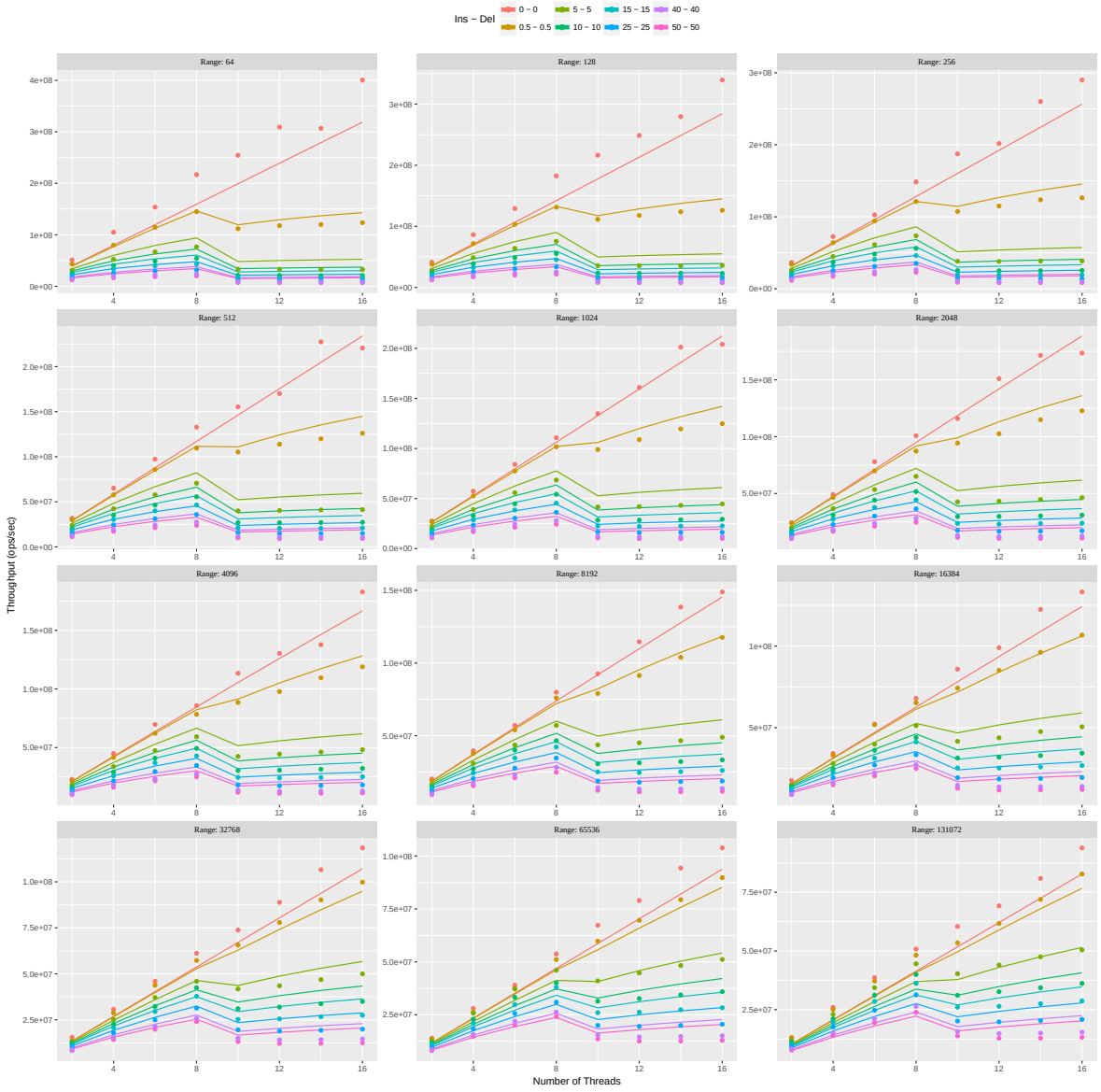


Figure 16: Skiplist Zipf distribution for key selection

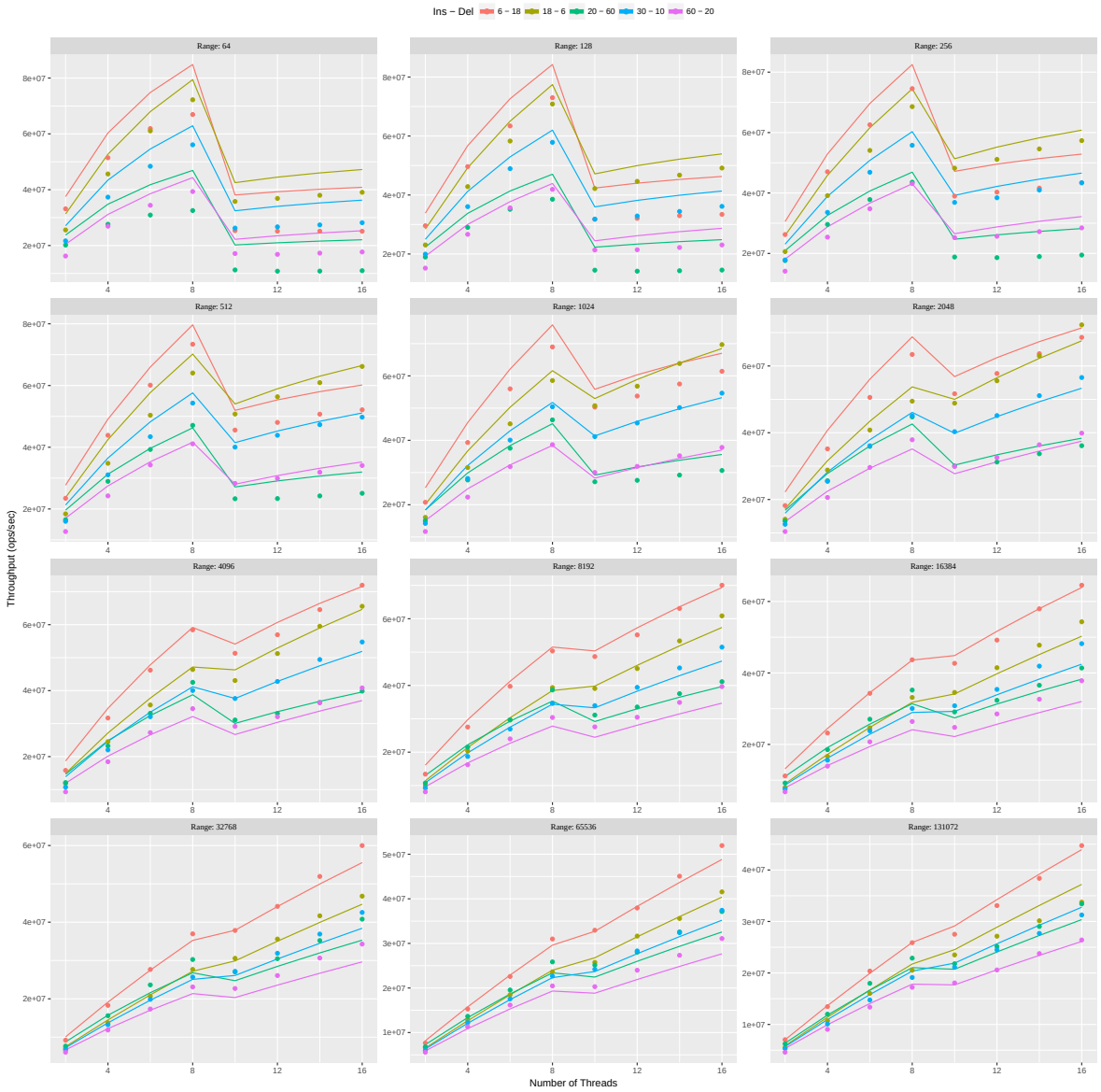


Figure 17: Skiplist asymmetric update rates, uniform distribution for key selection

4) Binary Tree: Figure 18, 19 and 20 illustrates the results for the binary tree, for various scenarios that are described before (see VII). Here, we observe that our estimations often closely follow the real behaviour.

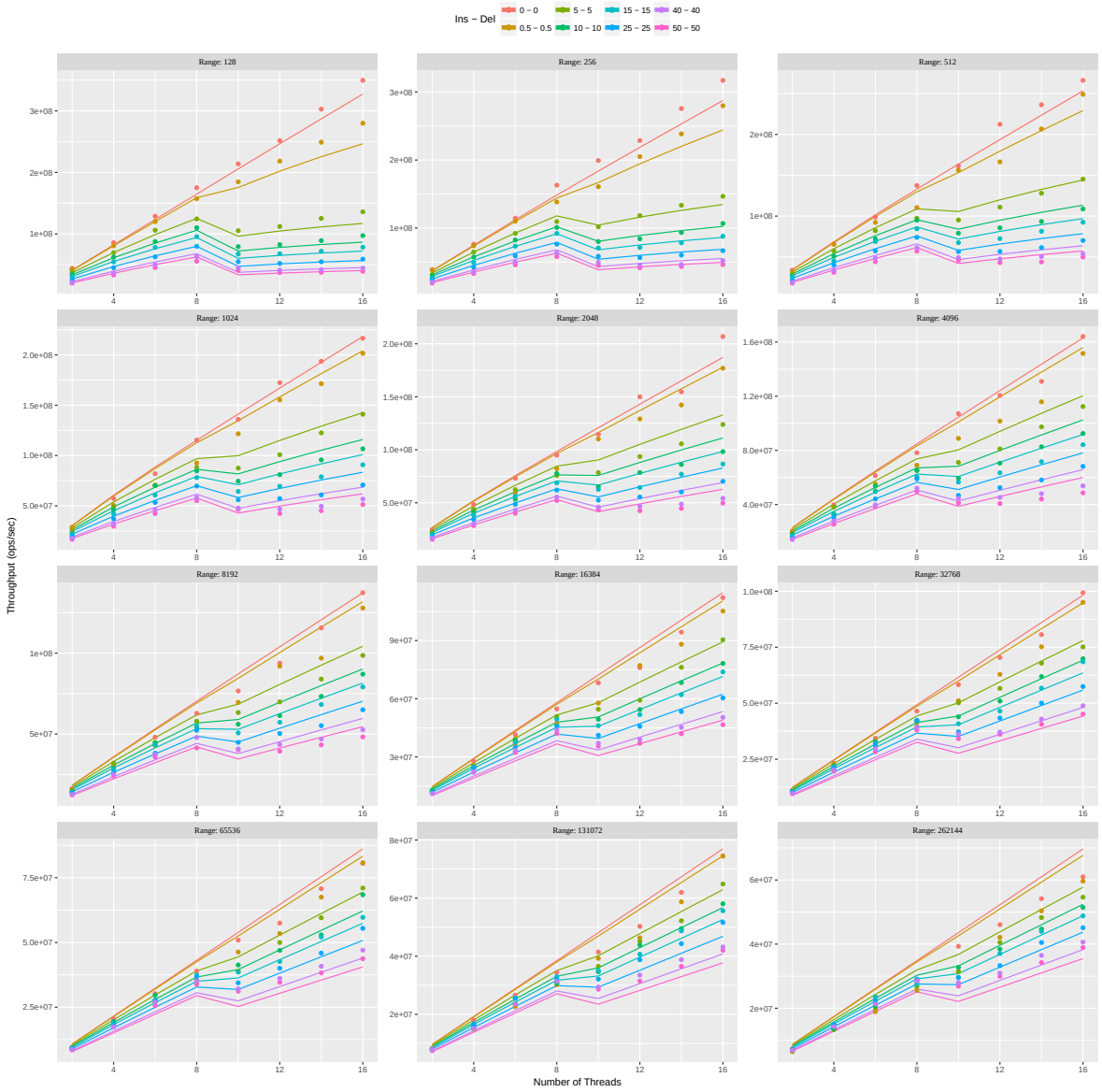


Figure 18: BST Uniform distribution for key selection

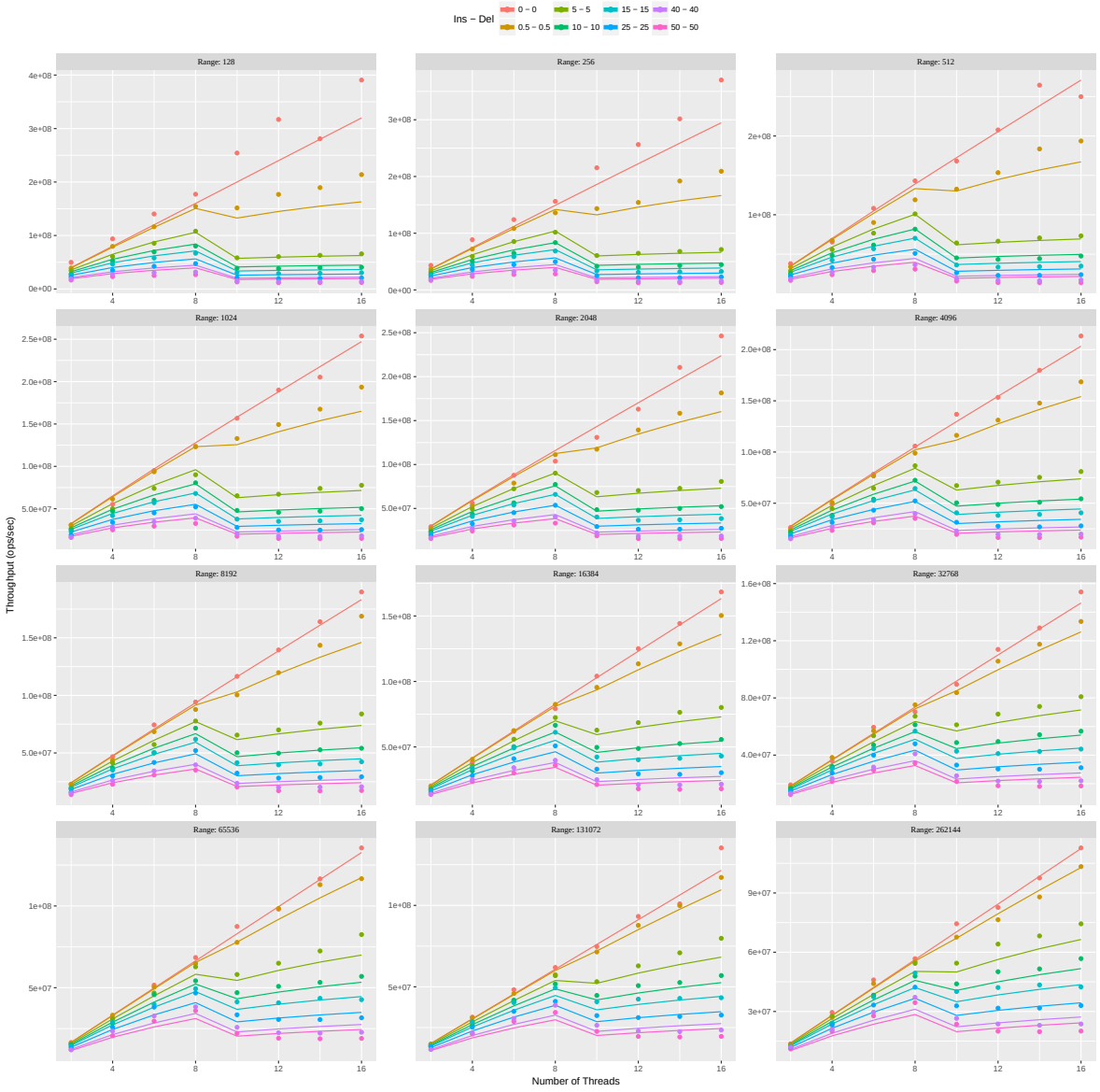


Figure 19: BST Zipf distribution for key selection

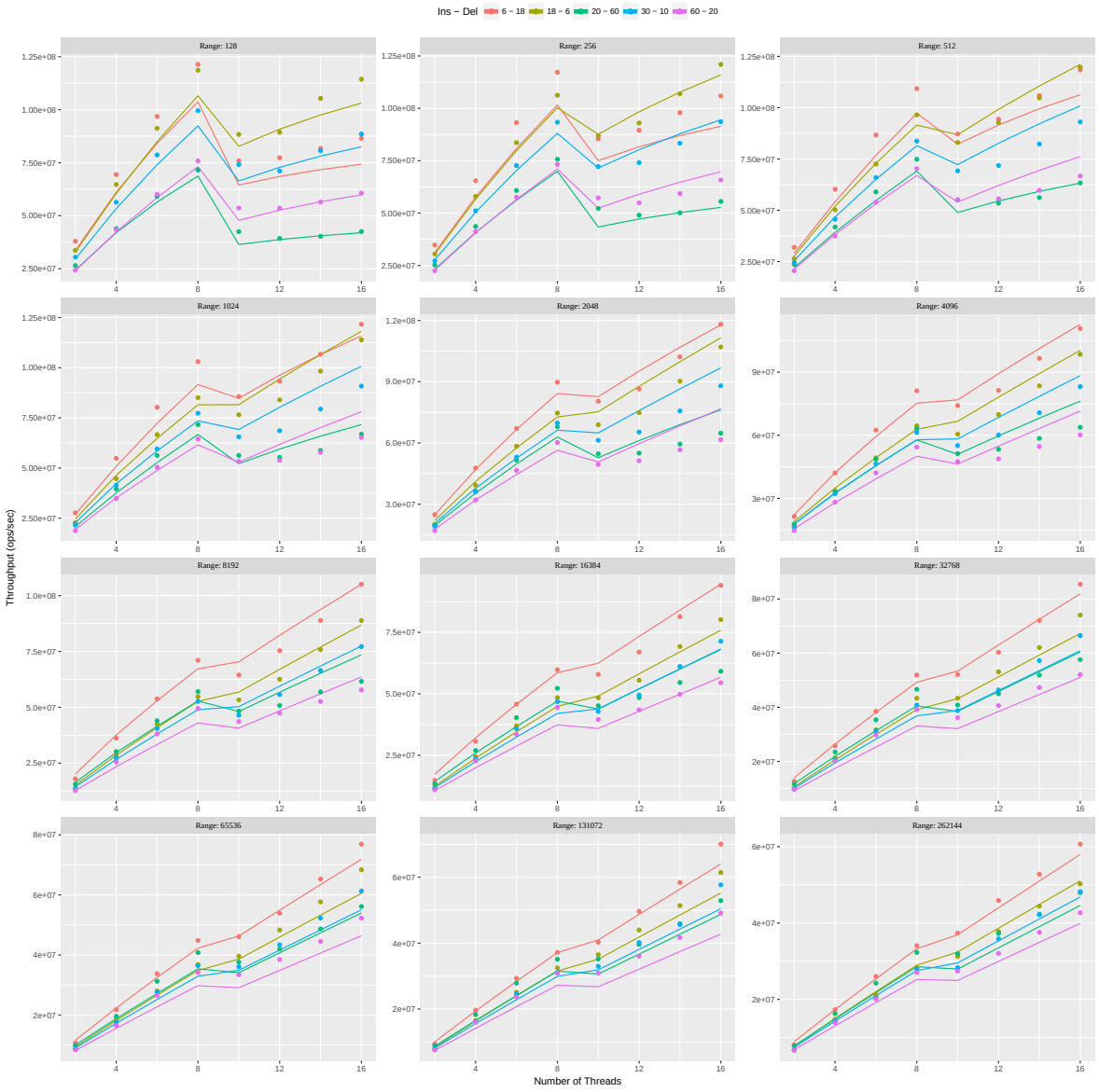


Figure 20: BST asymmetric update rates, uniform distribution for key selection

VIII. Applications: to Pad or not to Pad

In a non-padded (packed) configuration, multiple nodes are packed together into a single cacheline. This implies that a modification done at a node, could lead to a coherence cache miss in the traversal of the other nodes. It is often referred as *false sharing*. On the other hand, the packed configurations benefit from their compact representation by reducing the capacity misses.

Until now, we have assumed that the nodes are padded. Here, we extend the framework to estimate the performance of a packed configuration to facilitate the tuning process. In such a setting, where the nodes are inserted and deleted repeatedly, N_i can be alone in its cacheline with the old versions of a set of nodes that are not present any more in the data structure. Alternatively, it might be mapped to the same cacheline with some number of active nodes that are present in the search data structure and they all together contribute to the event rates that are originating from the same cacheline.

Firstly, we assume that at most two nodes can be packed to a cacheline (which is the case for the data structures that we consider) and we denote the total number of slots for the node allocations with $S = 2\mathcal{M}pageSize/cacheLineSize$ (recall that $\mathcal{M}$ is the number of pages that are used by the structure). We assume that the nodes are assigned uniformly to the slots; given that N_i and N_j are present in the structure, N_j is mapped to the same cacheline as N_i with probability: $1/(S-1)$. With the linearity of expectation, the expected additional event rate for the cacheline that N_i is mapped to can be given by the sum of event rates originating from different nodes. λ_i^{read} and λ_i^{cas} provides the event rates for N_i, and we introduce an additive factor to represent the average event rate contributions of other nodes to the cacheline of N_i: $\lambda_i^{read,addi}$ for *Read* events, and $\lambda_i^{cas,addi}$ for *CAS* events. N_j contribute to the *Read* event rates with λ_j^{read} if N_j and N_i are assigned to the same cacheline, which happens with probability $p_j/(S-1)$. Then, we have: $\lambda_i^{read,addi} = \sum_{j=1,j\neq i}^{\mathcal{N}} \lambda_j^{read}(\frac{p_j}{S-1})$ and $\lambda_i^{cas,addi} = \sum_{j=1,j\neq i}^{\mathcal{N}} \lambda_j^{cas}(\frac{p_j}{S-1})$.

With the node packing, we obtain additive components for *CAS* and *Read* events. Now, we show the integration of these additive components into the process.

1) Cache Misses: To begin with, packing would have a positive impact on the cache misses as it would increase the characteristic time (T) of the cache, that is the duration for C unique cacheline references. To recall, N_i could contribute to this C references only if $N_i \in D$ and we have embedded this effect into the process by introducing the random variable P_i (see V-A5). With the packing, this contribution becomes less probable, as the contribution would occur only if the reference to N_i occurs before the references to the other node that is mapped to the same cacheline with N_i. Otherwise, the reference to N_i would be ineffective for the characteristic time. To recall, the characteristic time is the solution of the following equation:

$$X^{cache}(t) = \sum_{j=1}^{N} P_i^{pack} \mathbf{1}_{0<O_j\leq t}$$

where P_i^{pack} is the variable that we modify in the process,

$$P_i^{pack} = \begin{cases} p_i(\lambda_i^{read}/(\lambda_i^{read} + \lambda_i^{read,addi})), & \text{if } P_i^{pack} = 1 \\ 1 - p_i(\lambda_i^{read}/(\lambda_i^{read} + \lambda_i^{read,addi})), & \text{if } P_i^{pack} = 0 \end{cases}$$

Having obtained the characteristic time, we involve the additive factor to estimate the cache miss rate of N_i. This is because a reference leads to a cache miss (in a cache of size C) only if the previous C cacheline references do not include the cacheline that N_i is mapped to.

$$Hit_i^{cache} = 1 - e^{(-(\lambda_i^{read}+\lambda_i^{read,addi}))/P)T}$$

2) Page Misses: Secondly, packing can improve the TLB cache hit ratios. This simply happens because it reduces the total number of pages that the search data structure spans. To recall, the total number of pages is a parameter of the process that computes the expected latency for the impacting factor (Hit_i^{tlb}). Packing do not influence the process, so we just need to update the value of the parameter.

3) CAS Execution: On the downside, packing is expected to reduce the performance through the *CAS* related impacting factors. To recall, CAS_i^{reco} represents the expected latency per traversal at N_i for executing *CAS* instructions targeted to N_i. This factor is proportional to the throughput, and packing do not change the probability of executing a *CAS* at N_i while traversing it. So, packing does not have a direct impact on this component.

4) Invalidation Recovery: The most important performance impacting *CAS* related factor is the invalidation recovery. For each traversal of N_i, there exist a possibility to pay for a coherence cache miss due to the previous *CAS* executions at the cacheline, that N_i is mapped to. To compute the probability of a coherence miss, one needs to consider the previous events on the cacheline. The traversal (by a thread at N_i) would not experience the coherence miss if the previous traversal (on the cacheline that N_i is mapped to) of the same thread is not followed by *CAS* event of another thread. Thus, we consider the additive factor for both type of events and modify the process as follows:

$$\mathbb{P}\left[\text{Coherence Miss on traversal of } N_i\right] = \frac{(\lambda_i^{cas} + \lambda_i^{cas,addi})(P-1)}{(\lambda_i^{cas} + \lambda_i^{cas,addi})P + (\lambda_i^{read} + \lambda_i^{read,addi})}$$

5) Stall Time: Finally, packing has a potential to increase the ratio of time that the cacheline (that N_i is mapped to) is blocked due to *CAS* executions. We simply update the process by involving the additive factor:

$$\mathbb{E}\left[CAS_i^{stall}\right] = (\lambda_i^{cas} + \lambda_i^{cas,addi})(P-1)t^{cas}\frac{t^{cas}}{2}$$

6) Experiments: In Figures 22 and 21, the results are depicted for configurations with padding (dashed lines), packing(dots) and our packing based estimations(lines), for the linked list and hash table (nodes for tree and skiplist is too large to be packed in a single cacheline or already packed). The key selection is done with the uniform distribution. For almost every case, we observe that the packing increases the performance and the performance do not degrade due to the false sharing, even when the update rate is high. The stall time ($\mathbb{E}\left[CAS_i^{stall}\right]$) often is not significant and the invalidation recovery ($\mathbb{E}\left[CAS_i^{reco}\right]$) dominates the performance when there are update operations. As an observation, the latency induced by this factor do not increase with packing, presumably because:

$$\frac{(\lambda_i^{cas} + \lambda_i^{cas,addi})(P-1)}{(\lambda_i^{cas} + \lambda_i^{cas,addi})P + (\lambda_i^{read} + \lambda_i^{read,addi})} \approx \frac{\lambda_i^{cas}(P-1)}{\lambda_i^{cas}P + \lambda_i^{read}}$$

This might explain us the reason why the false sharing do not degrade the performance, as opposed to one might expect. However, the cache and page misses influence the performance positively, as expected.

Our estimations show that these effects are captured by our framework. We observe a slight increase in almost all the curves that is coupled with a slight increase in our estimations, due to the reduced capacity cache misses.

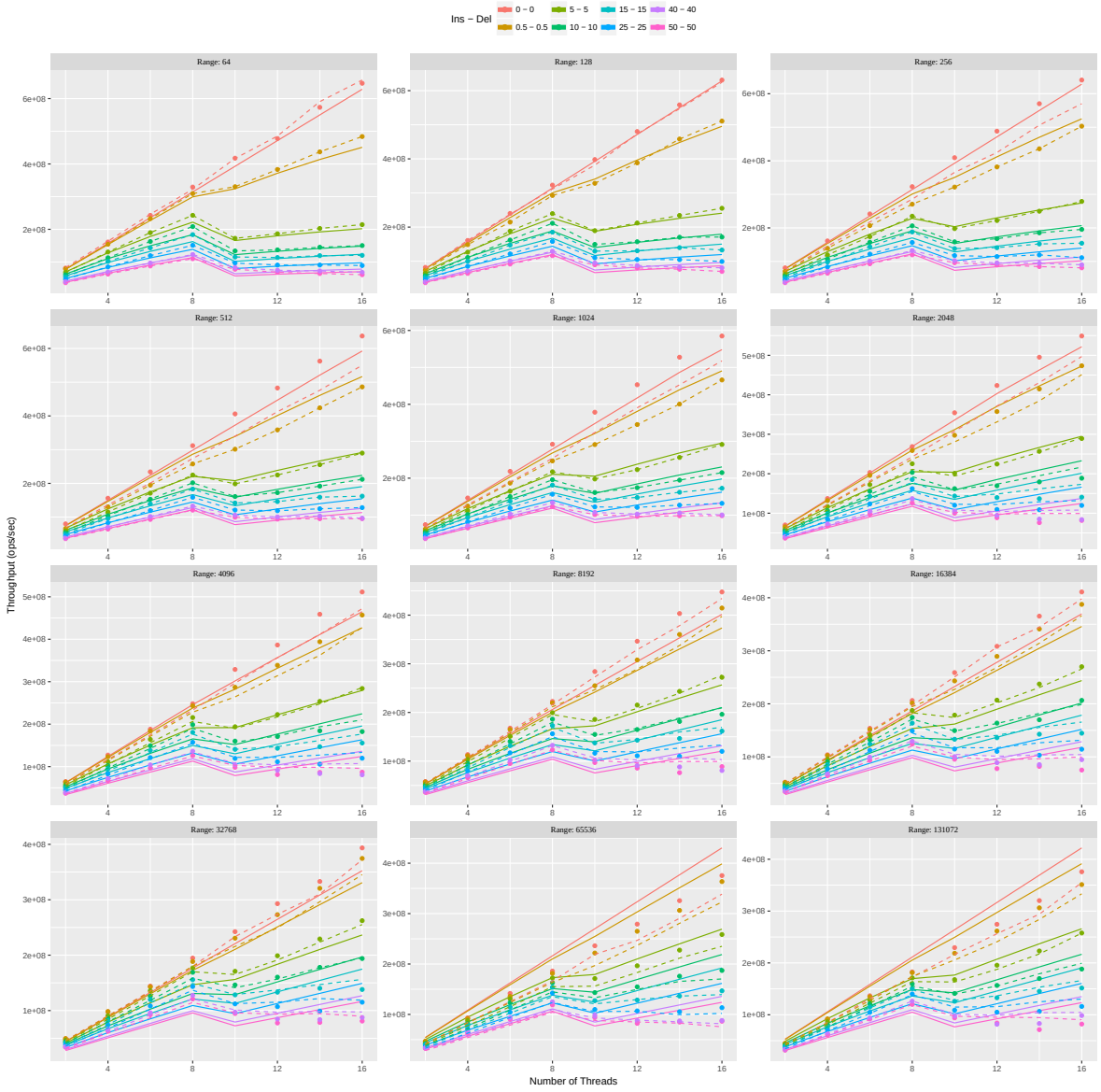


Figure 21: Packed nodes for Hash Table, with load factor=2

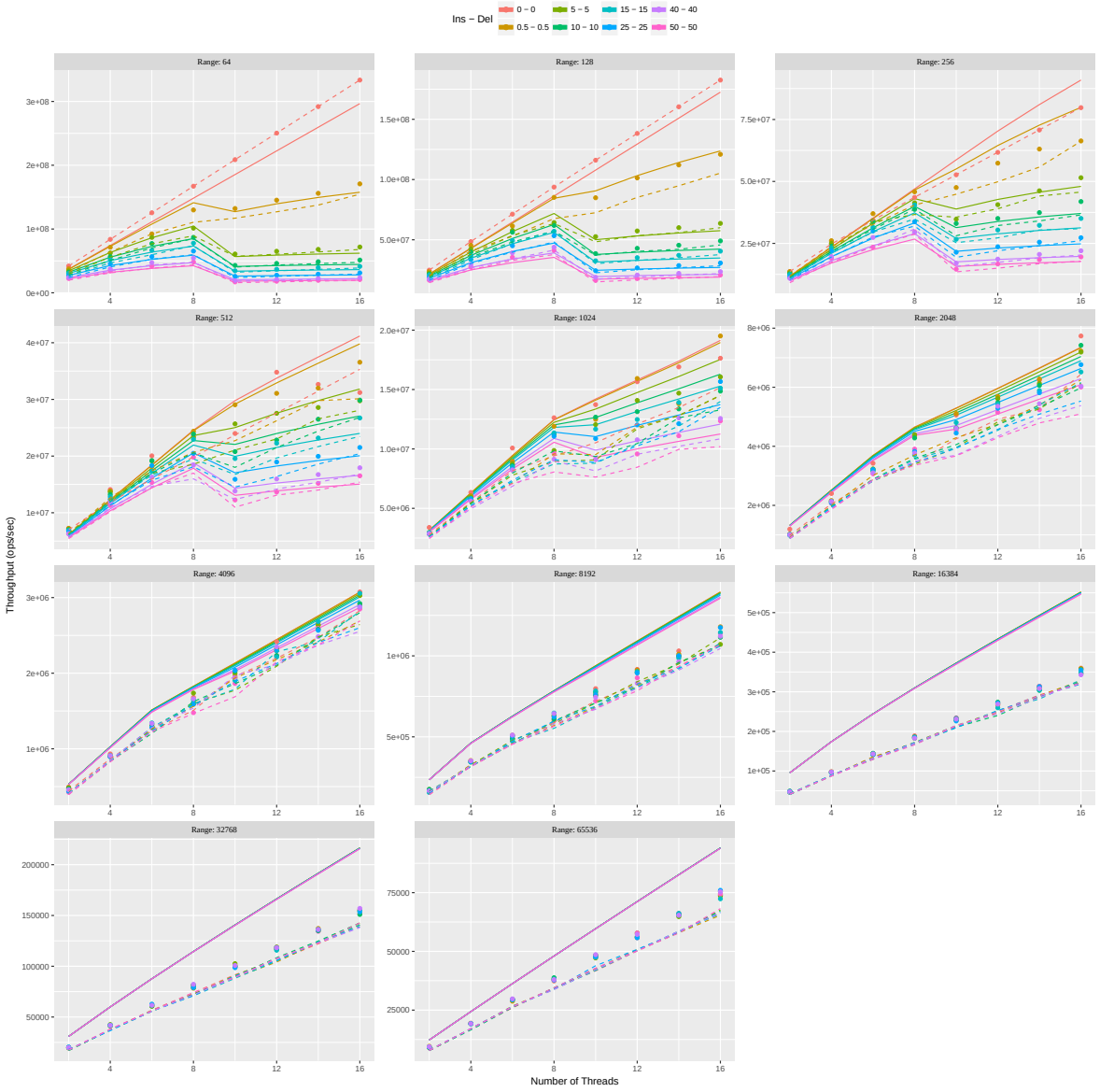


Figure 22: Packed nodes for Linked List

IX. Conclusion

In this paper, we have modelled and analysed 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 modelled 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] Richard Arratia, Larry Goldstein, and Louis Gordon. Poisson approximation and the chen-stein method. *Statistical Science*, 5(4):403–424, 1990.

[2] Vlastimil Babka and Petr Tuma. Investigating cache parameters of x86 family processors. In *SPEC Benchmark Workshop*, volume 5419 of *Lecture Notes in Computer Science*, pages 77–96. Springer, 2009.

[3] A.D. Barbour and T.C. Brown. Stein's method and point process approximation. *Stochastic Processes and their Applications*, 43(1):9 – 31, 1992.

[4] Timothy C. Brown, Graham V. Weinberg, and Aihua Xia. Removing logarithms from poisson process error bounds. *Stochastic Processes and their Applications*, 87(1):149 – 165, 2000.

[5] Bapi Chatterjee, Nhan Nguyen Dang, and Philippas Tsigas. Efficient lock-free binary search trees. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*, pages 322–331. ACM, 2014.

[6] Louis H. Y. Chen and Adrian RÃűllin. Approximating dependent rare events. *Bernoulli*, 19(4):1243–1267, 09 2013. URL: https://doi.org/10.3150/12-BEJSP18, doi:10.3150/12-BEJSP18.

[7] Tudor David and Rachid Guerraoui. Concurrent search data structures can be blocking and practically wait-free. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 337–348. ACM, 2016.

[8] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 631–644. ACM, 2015.

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

[10] James D. Fix. The set-associative cache performance of search trees. In *Proceedings of the ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 565–572, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644203.

[11] 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. URL: https://doi.org/10.1016/0166-218X(92)90177-C, doi:10.1016/0166-218X(92)90177-C.

[12] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*, pages 50–59. ACM, 2004.

[13] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. *CoRR*, abs/1202.3974, 2012. URL: http://arxiv.org/abs/1202.3974, arXiv:1202.3974.

[14] Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 1–10. ACM, 2015.

[15] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing (DISC)*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2001.

[16] Peter Kirschenhofer and Helmut Prodinger. The path length of random skip lists. *Acta Informatica*, 31(8):775–792, 1994. URL: https://doi.org/10.1007/BF01178735, doi:10.1007/BF01178735.

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

[18] Hosam M. Mahmoud and Ralph Neininger. Distribution of distances in random binary search trees. *The Annals of Applied Probability*, 13(1):253–276, 01 2003. URL: https://doi.org/10.1214/aoap/1042765668, doi:10.1214/aoap/1042765668.

[19] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328. ACM, 2014.

[20] Gabriele Paoloni. How to benchmark code execution times on Intel® ia-32 and ia-64 instruction set architectures. Technical Report 324264-001, Intel, September 2010.

[21] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. URL: http://doi.acm.org/10.1145/78973.78977, doi:10.1145/78973.78977.

[22] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[23] 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.

[24] Yutao Zhong, Steven G. Dropsho, Xipeng Shen, Ahren Studer, and Chen Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Transaction on Computers*, 56(3):328–343, 2007.