Understanding and Improving Bloom Filter Configuration for Lazy Address-Set Disambiguation

by

Mark C. Jeffrey

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Understanding and Improving Bloom Filter Configuration for Lazy Address-Set Disambiguation

Mark C. Jeffrey

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2011

Many parallelization systems detect memory access conflicts across concurrent threads by *disambiguating* address-sets using bit-vector-based Bloom filters, which are efficient, but can report false conflicts that do not exist. Systems with lazy conflict detection often use Bloom filters unconventionally by testing sets for null-intersection via Bloom filter intersection, contrasting with the conventional approach of issuing membership queries into the Bloom filter. In this dissertation we develop much-needed theory for probability of false conflicts in Bloom filter null-intersection tests, notably demonstrating that Bloom filter intersection requires substantially larger bit-vectors to provide equivalent statistical behavior to querying. Furthermore, we recognize that our theoretical implications counter practical intuition, and thus use RingSTM to evaluate theory in practice by implementing and comparing the Bloom filter configurations. We find that despite its overheads, the queue-of-queries approach reduces execution time and is thus the most compelling alternative to Bloom filter intersection for lazy address-set disambiguation.

Acknowledgements

A great deal of thanks goes to my supervisor, Professor Greg Steffan, for accepting and encouraging my blend of interests in theory and systems implementation. Through our frequent discussions, he convinced me of the merit of this study to the parallel programming community, and helped me to improve the communication of my research ideas. Professor Bruce Francis was of great help through his tutorial, *Elements of Mathematical Style*, and his general feedback on our analytical work. Thanks go to Hratch Mangassarian for our discussion on asymptotic notation, and Professor James Tuck for initial discussion on the need for better theoretical understanding of Bloom filters in address-set disambiguation. I acknowledge the NSERC Alexander Graham Bell Canada Graduate Scholarship (CGS-M) for its financial support.

Thank you to my fellow graduate students for much advice and support (proof-reading, slide-reading, and code-optimizing), and for being a great fan club at concerts. Thanks to the students of CARG for (unintentionally) leading me to related work, building the case for a study of null-intersection tests. Many thanks to Skule Music for the perfect break from research. Finally, thank you to friends and family for their understanding and support when I would disappear for weeks at a time, and greeting me cheerfully at the finish line.

Contents

List of Acronyms

BoB Batch-of-Bloom-filters

FSO False Set-Overlap

HTM Hardware Transactional Memory

QoQ Queue-of-Queries

SIMD Single Instruction Multiple Data

SSE Streaming SIMD Extensions

STM Software Transactional Memory

TLS Thread-Level Speculation

TM Transactional Memory

List of Symbols and Abbreviations

$[N]$ Set of integers $\{1, 2, \ldots, N\}$

$\text{FP}_{\in}$ False positive predicate for a Bloom filter query

$\text{FSO}_{\cap}$ Predicate for false set-overlap by Bloom filter intersection

$\text{FSO}_{\in}$ Predicate for false set-overlap by a queue-of-queries into a Bloom filter

$\mathcal{H}_{\frac{m}{k}}$ Set of all hash function tuples mapping U to the $\frac{m}{k}$-bit subfields of a partitioned Bloom filter

h_{pre} Prefiltering hash function for a batch-of-Bloom-filters

$\mathcal{H}_m$ Set of all hash function tuples mapping U to the full m-bit range of an unpartitioned Bloom filter

b Number of Bloom filters internal to a batch-of-Bloom-filters

$BF(S)$ Set of asserted bits in a Bloom filter that represents set S

k Number of hash functions that index a Bloom filter

m Length of the Bloom filter bit vector (in bits)

U Universe (or address space for TM)

List of Tables

List of Figures

Chapter 1

Introduction

Chip-multiprocessors demonstrate the scalability required to exploit Moore's Law transistor counts, but require parallel programs to achieve their full performance potential. Unfortunately, creating bug-free high-performance parallel programs is extremely difficult, with synchronizing concurrent accesses to shared memory as the most arduous task. A number of tools and runtime systems have been developed to help programmers detect, tolerate, or avoid memory access conflicts and bugs—for example, Transactional Memory (TM) [22] allows the programmer to simply label regions of code as transactions that are then executed atomically and in isolation. The foundation of such *parallelization* systems and tools is a mechanism for detecting conflicts, i.e., unsafe interleavings of memory accesses across execution *epochs*, including transactions, critical sections, race-free episodes, or "chunks" of sequentially-consistent instructions [12].

For any conflict detection strategy, memory reads and writes are accumulated into per-thread read- and write-sets, respectively, over the course of an epoch. Conflicts generally manifest when a memory address appears in the write-set of one thread, and the read- or write-set of another concurrent thread, and are detected according to one of two schedules: *eagerly*, at the time of memory access, or *lazily* at the end of an epoch

or some validation step. In the case of TM, to repair a conflict, one of the transactions must be aborted and its global effects rolled back.

Bloom Filters Given a need for fast runtime address-set comparisons, the bit-vector-based Bloom filter [7] has emerged as the address-set representation of choice for many systems in hardware TM (HTM) [13,30,55,63,67], software TM (STM) [17,35,47,48,58], hybrid TM [11,38,65], Thread-Level Speculation (TLS) [13,20], transaction contention management [5,6,57], loop-parallelizers [4], and even concurrency debugging tools [3,23, 24,28,32,33,42,44,45,49,69]. A Bloom filter is conventionally used to provide a constant-time set membership query via hash-indexing a bit-vector, at the cost of a potential *false positive* where an address is incorrectly identified as a member of a read/write-set to which it does not belong. False positives can lead to unnecessary transactional aborts, false bug reports, and other mispredictions [5], and the probability of a false positive increases as bit-vector length decreases—hence Bloom filter size and access-timing constraints must be carefully balanced. Despite this trade-off, the Bloom filter does not threaten program correctness, and has made numerous appearances in both hardware systems for its unbounded set representation, and in software systems for its fast set operations.

1.1 Breaking Convention: Bloom filters for Lazy Conflict Detection

The main concern for designers of parallelization systems is to configure the Bloom filters appropriately to achieve an acceptable false conflict rate. An accurate model for probability of false conflicts would thus be extremely helpful for system designers, enabling a fast theoretical design space exploration, rather than time-consuming empirical explorations. Bloom filters are apparently well-suited to eager conflict detection, by

offering fast set membership queries with a tunable accuracy that can be guided by the well-understood false positive probability distribution [7–9, 16]. Configuration in such scenarios includes choice of bit-vector length and number of hash functions. In this dissertation, we bring attention to the unconventional use of Bloom filters in lazy parallelization systems, where instead of supporting periodic membership queries, Bloom filters are often used unconventionally to compare entire address sets for any access overlap, via bit-vector *intersection*. With delayed conflict detection, lazy systems typically compare sets of two or more addresses, affording new creativity in Bloom filter use, but this alternate method of intersection has not been formally studied or modeled, further expanding the Bloom filter configuration space.

These Bloom-filter-based *null-intersection tests* can be implemented in three ways: (i) queue-of-queries, (ii) unpartitioned intersection, and (iii) partitioned intersection. The first approach uses Bloom filters most intuitively, by issuing a *queue-of-queries* (QoQ)—i.e., by maintaining one complete set of accessed addresses, and issuing a query of each address into the Bloom filter representation of the other set (e.g. SigTM [38]). The second and third approaches avoid this linear time complexity by *intersecting* two Bloom filters using bitwise `AND`, and analyzing the remaining bits to determine whether the resulting intersection is empty [1, 4, 5, 12, 13, 17, 33, 35, 42, 45, 47, 48, 50, 57, 58, 69]. For the second approach, an *unpartitioned* Bloom filter, the hash functions map to the entire bit-vector range, and following intersection, any bits set to one imply that the intersection cannot be empty. In contrast, for the third approach, *partitioned* Bloom filters designate disjoint bit-fields of the bit-vectors, and these partitions are pairwise intersected—a result of all-zero for any partition indicates a null intersection.[1] When two input address sets are in fact disjoint, each null-intersection test might return a *false set-overlap* (FSO), the lazy analogue to a false positive, which falsely indicates that the two sets shared some common

[1]Partitioned and unpartitioned bit-vectors are asymptotically equivalent [9,60] for membership query false positives, but the distinction is important for implementation [53].

elements. Although the three null-intersection tests are in wide use, to the best of our knowledge, their respective probabilities of false set-overlap have neither been studied analytically nor conclusively compared in prior work.

1.2 Theory and Practice of Bloom filter Null-Intersection Tests

In this dissertation we provide system designers with a new analytical model for the probability of false set-overlap for Bloom filter null-intersection tests. We conclusively show which bit-vector configuration admits fewer false conflicts, and prove that to achieve equivalent probability of false set-overlap, intersection-based usage requires Bloom filters that are at least a factor of the square root of set cardinality larger than query-based usage. Our models suggest that a change from unpartitioned Bloom filters to 2- or 4-way partitioning of the bit-vector will yield considerable reduction of false conflicts in a number of existing parallelization systems [47, 58], and even further reductions can be observed when using the QoQ approach.

Competing with the reduced false conflicts, our theoretical recommendations unfortunately contradict practical intuition by introducing complexities: additional hash functions add overhead to Bloom filter insertions in software, and dynamically maintaining a set in hardware for QoQ adds considerable complexity relative to the fixed-sized Bloom filter bit-vector. To demonstrate the utility of this work, we thus experimentally validate our theoretical recommendations on commodity hardware using the STAMP benchmarks [37] and RingSTM [58]—a write-buffered software TM that detects conflicts with single-partitioned Bloom filter intersection. We focus on program execution time using efficient implementations, to demonstrate whether reductions in false abort rates outweigh the additional overheads of more refined Bloom filter null-intersection tests. In particular, we implement and compare the following configurations.

Unpartitioned Intersection This is the baseline, single-partitioned Bloom filter implementation provided in RingSTM.

Partitioned Intersection We extend RingSTM to support two- and four-way partitioned Bloom filters. While our models suggests that properly-configured partitioned intersection reduces false conflicts, it does require the computation of additional hash functions and hence increases per-access overheads.

Queue-of-Queries (QoQ) We implement QoQ by exploiting the existing associative write-buffers in RingSTM, avoiding the overheads of maintaining additional queue structures. The challenge with this approach is that the evaluation of many queries at validation-time could be prohibitively expensive for applications with large write-sets.

Batch-of-Bloom-Filters (BoB) We introduce a new null-intersection test called intersecting a *Batch-of-Bloom-filters* (BoB), that is intuitively a hybrid between the partitioned and QoQ approaches, designed to capture the benefits of each—i.e., to reduce false conflicts while continuing to exploit the speed of bitwise `AND` operations. The basic idea of BoB is, rather than a queue-of-individual-queries, to instead maintain a queue-of-Bloom-filters that we call a *batch*. We use a "prefilter" hash function to map each address to a certain filter in the batch, this way distributing addresses across the batch such that each filter in the batch represents a disjoint subset of addresses.

SIMD Optimization Recognizing that the bit-vector operations offer much parallelism, we evaluate the impact of exploiting Intel SSE2 and SSE4.1 vector instructions.

1.3 Research Goals

The focus of this dissertation is to develop a better understanding and improved implementation of Bloom filter configurations when used in null-set-intersection testing, considering both theory (for its ease of approximately exploring the design space) and

practice (to compel the community to heed our controversial recommendations). To this end, we have the following goals:

1. to provide a theory for Bloom-filter-based null-intersection testing, including the derivations of probability of false set-overlap between two disjoint (address) sets for each of the three existing Bloom filter configurations, and an analytical comparison of these distributions;

2. to pursue a new Bloom filter configuration for performing null-intersection tests that offers a compromise between the accuracy of querying, and the implementation-simplicity of bitwise intersection;

3. to implement and evaluate the performance of the four null-intersection tests in a real STM using queue-of-queries (QoQ), and partitioned, unpartitioned, and BoB intersection.

1.4 Organization

The rest of the dissertation is organized as follows. Chapter 2 provides a background on the Bloom filter data structure, including fundamental operations and theory, related work, and more details on its application in parallelization tools. Chapter 3 introduces and develops a formal theory for Bloom filter use in null-intersection testing, including comparisons among the three approaches. Chapter 4 describes our proposed batch-of-Bloom-filters intersection, demonstrating its statistical compromise between querying and bitwise intersection. Chapter 5 shifts the focus to practical considerations, describing the efficient software implementation of null-intersection tests. Chapter 6 presents a performance evaluation of the four null-intersection tests. In Chapter 7, we summarize this work, including contributions, conclusions, and future directions.

Chapter 2

Background on Bloom Filters

Since the introduction of Bloom filter *signatures* [13] to hardware TM, a flurry of research followed, studying both their use in new parallelization tools, and improvement upon initial Bloom filter implementations. In this chapter we introduce the fundamentals of Bloom filter operations and theory, survey the Bloom filter literature related to our work, and briefly review the applications in parallelization tools.

2.1 Fundamentals

This section gives a brief background on the relevant aspects of Bloom filters [7]. For preliminary notation, let $[N]$ denote the set $\{1, \ldots, N\}$. A Bloom filter is a data structure that compactly represents a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements from some universe U. The filter is a bit-vector of m bits indexed by a hash function tuple of k (ideally) mutually-independent hash functions $h(x) = (h_1(x), \ldots, h_k(x))$, supporting operations such as element insertion, membership queries, set union, and set intersection. We refer to the collection as the hash function tuple, and otherwise refer to individual hash functions. Two configurations of bit indexing are widely used, called unpartitioned and partitioned. The k-tuple hash function of an unpartitioned Bloom filter maps to the entire m-bit range

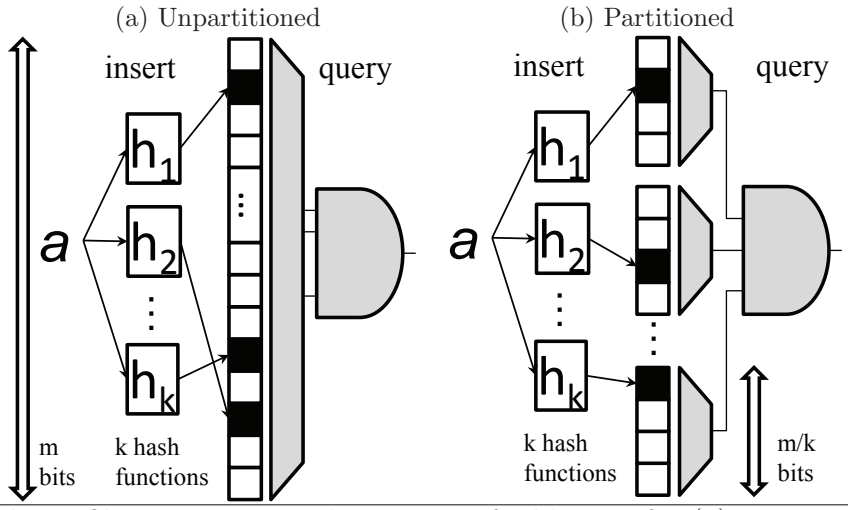


Figure 2.1: Bloom filter insertion and querying of address a for (a) an unpartitioned and (b) partitioned Bloom filter. In both cases, the filter has length m bits, and a tuple of k truly random hash functions. Addresses are inserted by asserting the bits indexed by the k hash values (dark boxes). A query accepts an address as a member of the set iff all k indexed bits are set to 1. Inspiration for figures is from [51]

of the bit-vector,[1] $h : U \to [m]^k$. In contrast, a partitioned Bloom filter distinguishes k disjoint subfields of the filter, with each hash function of the tuple mapping to an integer $\frac{m}{k}$-bit partitioned range, $h : U \to \left[\frac{m}{k}\right]^k$ [41]. In the context of address disambiguation, S is a set of memory addresses from a v-bit address space (or universe): $S \subset U = \{0, 1, \ldots, 2^v - 1\}$.

Figure 2.1 illustrates the individual element operations on a Bloom filter. To initialize an empty set, all bits of the vector are set to 0. Each element $x \in S$ is subsequently inserted in the filter by asserting the k bits indexed by each hash of x; the $h_i(x)$'th bit is set to 1 for $1 \leq i \leq k$. We denote the Bloom filter representation of S as $BF(S)$, which corresponds to the set of asserted bits after inserting all $x \in S$ [18, 51]. With a fully-constructed Bloom filter for S, an address $y \in U$ can be quickly tested for membership in S. The membership query accepts $y \in S$ if all of the $h_i(y)$'th bits of the bit-vector are 1, and otherwise indicates that $y \notin S$.

[1] The hash function output is ideally uniform.

For the remainder of this paper, we assume that the bit-vector length, m, and number of hash functions, k, are constant. Let $\mathcal{H}_m = \{h|h : U \to [m]^k\}$ be the set of all hash function tuples mapping from the address space to the full m-bit range of an unpartitioned Bloom filter. Similarly, let $\mathcal{H}_{\frac{m}{k}} = \{h|h : U \to [\frac{m}{k}]^k\}$ represent the set of all hash function tuples from the address space to the $\frac{m}{k}$-bit subfields of a partitioned Bloom filter.

2.1.1 False Positives

By encoding elements of a large universe into a compact bit-vector, there is a small probability that an element y (that is not in S) has collisions on each of its k hashes with some elements in the set S. Both hash *aliasing* and filter *density* (fraction of asserted bits) can lead to membership queries being falsely accepted, and so the Bloom filter actually represents a superset of the original address set: $S \subseteq BF(S)$. In essence, the membership query "is y in S?" is not answered by no or yes, but rather no or *maybe*. The following definition formalizes this notion.

Definition 1. *Let $S = \{x_1, \ldots, x_n\} \subset U$ be represented by an m-bit Bloom filter, $BF(S)$, using the k-tuple hash function $h \in (\mathcal{H}_m \cup \mathcal{H}_{\frac{m}{k}})$. When testing some element $y \notin S$ for membership in S, we define the* false positive *predicate $\mathrm{FP}_{\in}(S, y, h)$ to be true when the query accepts y as a member of $BF(S)$—i.e., when $y \notin S$, but $\forall i \in [k]$, $h_i(y) = h_i(x_j)$ for some $x_j \in S$.*

This definition describes the false positive event for both unpartitioned and partitioned Bloom filters, as $h \in (\mathcal{H}_m \cup \mathcal{H}_{\frac{m}{k}})$; however, throughout the paper we will specify the particular Bloom filter indexing via conditioning on h from one of the two hash function sets. The probability of false positives for a Bloom filter is well understood and estimated in a straightforward fashion [7, 9, 51]—the reader is directed to prior work for a formal proof. Assuming the partitioned Bloom filter indexing scheme, the distribution is as follows:

Lemma 1. [7, 9, 41, 60] *Let $h \in \mathcal{H}_{\frac{m}{k}}$ be a truly random k-tuple hash function. For any fixed set $S \subset U$ and element $y \notin S$, the probability that y is accepted in a membership query of partitioned $BF(S)$ is*

$$\Pr\left[\mathrm{FP}_{\in}(S, y, h) \mid h \in \mathcal{H}_{\frac{m}{k}}\right] = \left(1 - \left(1 - \frac{k}{m}\right)^{|S|}\right)^{k}. \tag{2.1}$$

For unpartitioned Bloom filters, $\Pr[\mathrm{FP}_{\in}(S, y, h) \mid h \in \mathcal{H}_m]$ is less than the result above, since a partitioned filter typically has more asserted bits. Notably, the two distributions asymptotically approach $\left(1 - e^{-\frac{k|S|}{m}}\right)^k$ [9],[2] and the latter approximation is minimized when $k = \frac{m}{|S|} \ln 2$ [9, 60].

2.1.2 Approximate Set Intersection

Beyond individual element operations, Bloom filters can be used to perform set union and intersection. In this work we focus on set intersection and its application in testing two sets for an empty (null) intersection. Let $S_1, S_2 \subset U$ be two sets that are represented by Bloom filters, $BF(S_1)$ and $BF(S_2)$, that use the same m and hash functions. The filter $BF(S_1 \cap S_2)$ is computed by hash-encoding elements of the actual intersection of these sets. The Bloom filter representations of S_1 and S_2 are insufficient to accurately compute $BF(S_1 \cap S_2)$, but their *Bloom filter intersection*, $BF(S_1) \cap BF(S_2)$, is quickly computed by the bitwise `AND` of their bit-vectors. Note that the set of bits asserted in $BF(S_1 \cap S_2)$ are certainly set to one in the Bloom filter intersection, but it is possible that additional bits could unnecessarily persist to be one following the `AND` operation. Bloom filter intersection thus provides an approximation to set intersection that maintains the original querying property of never returning false negatives [9, 18, 46].

Guo *et al.* [18] quantify the uncertainty in approximating set intersection with Bloom filter intersection. Assuming the unpartitioned Bloom filter configuration, the theorem

[2] Since $(1 - k/m)^n \approx e^{-\frac{kn}{m}}$, provided $m > nk$ [9].

by Guo is stated as a lemma toward our own contributions; readers are directed to the original work for a proof.

Lemma 2. [18] *Assuming the same m and random hash function $h \in \mathcal{H}_m$ are used in the Bloom filters of S_1, S_2, and $S_1 \cap S_2$, then $BF(S_1 \cap S_2) = BF(S_1) \cap BF(S_2)$ with probability*

$$(1 - 1/m)^{k^2 \times |S_1 - S_1 \cap S_2| \times |S_2 - S_1 \cap S_2|}.$$

Apparently, the asserted Bloom filter bits of a set intersection are not necessarily equivalent to the bits asserted by Bloom filter intersection of the sets; they are equivalent with non-negligible probability.

2.1.3 Accuracy of the False Positive Probability

Recent work [8,16] indicates that the "classic" analysis of the Bloom filter that proves the above Lemmas 1 and 2 is optimistic. The result attributed to Bloom (and republished in decades of subsequent work) is in fact a strict lower bound to the correct false positive probability. The new insight by Bose *et al.* [8] and Christensen *et al.* [16] has only focused on unpartitioned Bloom filters; applying their methods to the partitioned configuration, and subsequently repairing Lemma 2 is left as future work, beyond the scope of this dissertation. Regardless, the approximation provided by these lemmas is sufficient for this work, as Christensen *et al.* [16] demonstrated that the relative error diminishes with the larger m (≥ 1024 bits) typically used in parallelization systems [12, 13, 17, 23, 32, 38, 45, 51, 53, 58, 67, 68].

2.2 Related Work

Given a foundation in Bloom filter operations, in this section we discuss related work in Bloom filter design, application, and optimization. Specifically, we list prior Bloom

filter optimizations for parallelization systems, the Bloom filter descendants that inspired our proposed Batch-of-Bloom-filters, and other existing applications of Bloom filter intersection, namely in databases.

Bloom filters have been studied and highly optimized for tracking addresses in parallelization tools and TM systems, but no work has yet focused on enhancing their operation for null-intersection tests. LogTM-SE [67] has been augmented to demonstrate that: (i) partitioned Bloom filters are hardware-amenable for queries and insertions, with minimal impact to false abort rates [53]; (ii) simple bit-selection induces more false aborts than H_3 [10, 53] or PBX hashing [68]; (iii) the spatial locality of an address stream can be exploited to avoid false aborts [51], or to retain those false positives that stall or abort a transaction that would have later shown a true abort [15]; and (iv) that read- and write-sets should share the same Bloom filter as a general solution to the asymmetry in read- and write-set cardinalities [52]. Application-specific address hashing [30] has also been proposed.

All of these proposals optimize the false abort rate of an eager system that employs Bloom filter membership queries, and all except the multiset signature [52] optimize the Bloom filter hash functions. In contrast, we optimize the false abort rate of a lazy system by considering Bloom filter configuration for null-intersection tests (querying vs. intersection and partitioned vs. batch). Many of the previous proposals are orthogonal to null-intersection tests, and would be excellent additions to our own experiments in hardware. Our case study is in software TM, thus we are limited in our selection of hash functions; we seek to demonstrate that these configuration decisions are robust in lieu of non-ideal (but fast) hash functions.

Many high-level changes have been proposed for new Bloom filter designs, including the use of a collection of filters, and using one hash function as a prefilter to sub-Bloom filters or other hash functions [60]. Our BoB proposal is the first to combine these two ideas for null-intersection tests. Muzahid *et al.* [44] represent the most closely

related work, with a brief mention to a similar design in their atomicity violation tracker. They maintain four "physical" filters for each "logical" Bloom filter, and each address is hash encoded into one of the four filters depending on its value. Their implementation was used solely for querying, not for intersection. Furthermore, they do not explain the implemented mapping of addresses to physical filters, nor do they empirically demonstrate the improvement over regular querying.

Outside of parallelization tools, Tarkoma *et al.* [60] survey collection-of-Bloom-filter and multiple-choice prefilter hash function designs. The collection-based designs include Hierarchical Bloom Filters [54], Filter Banks [14, 31], Dynamic Bloom Filters [18], Split Bloom Filters [66], and Scalable Bloom Filters [2]. Papapetrou *et al.* proposed Dynamic Block-Partitioned Bloom filters [46] that use a dynamic matrix of Bloom filters to increase or decrease the total space used, to match dynamic cardinality requirements. Hao *et al.* [21] introduce partitioned hashing, which selects a group of hash functions that will minimize the number of bits set in the filter based on the prefilter hash of the input element. The design reduces the false positive probability in applications where the set of keys changes slowly relative to incoming membership queries.

Prior to parallelization systems, the bitwise intersection of Bloom filters was applied in the database community to accelerate relational join operations: approximating set intersection, and subsequently performing membership queries to the remaining bits [36]. Estimation of join cardinality has also benefited from this fast intersection [9, 43, 46]. However, unlike address-set disambiguation, database applications generally do not strive for the intersection result to be an empty set. Our work builds on these studies of Bloom filter intersection, with a focus on address-set disambiguation, and hence targeting intersections that return empty sets in the ideal case.

2.3 Parallelization Tools and Runtime Systems

The over-arching challenge for parallel programming stems from detecting and managing data access conflicts between parallel threads, since they can lead to invalid data and incorrect execution when improperly handled. A variety of programming models and debug tools have hence been proposed to augment locking, a conventional form of managing potential conflicts. We summarize those systems that apply Bloom filters for conflict detection.

Progress has been made in tools for finding, replaying, and avoiding concurrency bugs [32] that may result when a programmer: (i) fails to synchronize accesses to a mutable shared variable (i.e., a data race) [45]; or (ii) incorrectly reasons about atomicity, failing to enclose a set of memory accesses in a critical section (i.e., an atomicity violation) [32,33]. Debugging in a concurrent environment is made even more challenging as the manifestation of these bugs depends on the non-deterministic interleavings of threads. Several debugging systems thus focus on deterministically replaying concurrency bugs to find their source [23,42,61]. Despite thorough testing, some bugs still make it to deployment, motivating dynamic avoidance of concurrency bugs [33].

Beyond debugging, Transactional Memory (TM) [22] and Thread-Level Speculation (TLS) [19,29,59] have emerged as methods of more automatically managing data access conflicts for the programmer. TM allows programmers to wrap critical sections in *transactions*, which the underlying system speculatively executes in parallel. The runtime system tracks accesses to shared memory, and upon discovering a conflict, rolls back memory to a previously consistent state, otherwise committing the results of conflict-free transactions. TM ideally provides the illusion of fine-grained atomicity and isolation, with the programming effort of coarse-grained locks. TLS divides a legacy sequential program into ordered speculative threads that are executed optimistically in parallel, also via an underlying system of detecting and recovering from data access conflicts.

2.4 Summary

In this chapter we provided a foundation in Bloom filter fundamentals, related work in the literature, and details on Bloom filter applications in parallelization tools. The insertion and query operations of a Bloom filter were described and illustrated, including the conditions for and probability of a false positive. The use and statistical properties of Bloom filters for approximating set intersection were also explained. We briefly surveyed the literature on (i) improving Bloom filters for eager parallelization tools, (ii) using collection-of-filters data structures or prefiltered Bloom filters, and (iii) prior applications of Bloom filter intersection.

In the next chapter we build upon the existing theory of query false positives and approximate set intersection (provided in Lemmas 1 and 2), and modify these ideas to study the Bloom filter configurations used, in the field, for testing null-intersection between address-sets. In Chapter 4 we propose a new compromise null-intersection test, inspired by earlier prefiltering or collection-of-filters designs.

Chapter 3

A Theory of Bloom Filters for Null-Intersection Tests

Despite the popularity of Bloom filters in research architectures and tools, no probability distributions have previously been proposed to model their use in testing sets for pairwise null-intersection (a.k.a., set disambiguation or disjointness [26]). Additionally, no prior work has conclusively compared the statistical behaviors of the three null-intersection tests, precisely demonstrating the degradation of space-efficiency when replacing querying with bitwise intersection. In this chapter we provide the missing theory: we (i) describe how Bloom filters are used to test for null-intersection between address-sets, and when they flag false conflicts among execution epochs; (ii) state and derive the probability distributions representing these unfortunate events; (iii) compare the probability distributions through proven inequalities of probability and space; (iv) empirically validate the distributions; and (v) identify the implications of these findings.

3.1 Methods of Testing for Null-Intersection

This section details the intersection operations and conditions for false set-overlap for each of the three existing Bloom filter null-intersection tests: queue-of-queries, unpartitioned

intersection, and partitioned intersection. We begin by motivating a definition of *false set-overlap*, when two disjoint sets appear to share some overlap due to Bloom filter operations. In line with Bloom's original motivation, systems implementing eager conflict detection use Bloom filter membership queries for runtime address-set comparison. At the time of accessing address y, the address is tested for membership in the read- or write-filters ($BF(R)$ or $BF(W)$) of other epochs (e.g., by querying incoming coherence requests). There is a probability of a false positive on each query (unnecessarily indicating an address conflict), which is modeled by Lemma 1. However, since address conflicts impact execution at the granularity of epochs, it becomes apparent that the probability of individual false conflicts is not of interest in parallel programming tools. Instead we wish to know the probability that entire epochs will falsely conflict, such as for lazy conflict detection schemes where the read- and write-sets are nontrivial. In the following subsections, we define two predicates which relate system epoch failures to false set-overlaps, and illustrate the operations of the three null-intersection tests.

3.1.1 Queue-of-Queries

Consider the lazy conflict detection scheme of SigTM [38], which maintains a write buffer (W) and read and write Bloom filters ($BF(R)$ and $BF(W)$) for each thread. These sets are finalized at the end of an epoch and otherwise grow monotonically. To detect conflicts at the end of a transaction, the system verifies that every member of the write-set W is not a member of all other threads' read-sets by performing membership queries into the read filters via coherence broadcasts. If any address in the write-set conflicts with the read filter of a remote transaction, the latter transaction is aborted. We use SigTM as a sample model of what we denote as the conventional approach to lazy null-intersection tests—executing a *queue-of-queries* into a Bloom filter. Figure 3.1a illustrates this idea, where the queue of elements is the aforementioned write buffer. Each element of the write buffer (queue) is queried into the Bloom filter of some other epoch, until a conflict is found;

otherwise the sets are disjoint. Supposing the two epochs did in fact access independent memory, we say that a false set-overlap occurred if one of the epochs unnecessarily aborted. The following definition formalizes false set-overlap by a queue-of-queries.

Definition 2. *Let $S_1, S_2 \subset U$ be two fixed, disjoint sets, and choose S_1 to be represented by a Bloom filter of m bits and hash k-tuple $h \in (\mathcal{H}_m \cup \mathcal{H}_{\frac{m}{k}})$. We define the* false set-overlap by queries *predicate* $\mathrm{FSO}_{\in}(S_1, S_2, h)$ *to be true if, for some $x \in S_2$,* $\mathrm{FP}_{\in}(S_1, x, h)$ *is true.*

This definition describes when two sets would be incorrectly reported as overlapping by the conventional method of using Bloom filters for membership queries. The predicate is defined for either type of bit-indexing by hash functions, since $\mathrm{FP}_{\in}$ of Section 2.1 is defined for both. In later sections, we will condition on the bit-indexing scheme as necessary.

3.1.2 Intersection: Partitioned and Unpartitioned

Lazy conflict detection must determine whether particular address-sets are disjoint—i.e., to ask "is their intersection empty?" Some researchers have astutely avoided the linear time required for a queue-of-queries by applying Bloom filter intersection to approximate this underlying set intersection task. Independent of the bit-indexing scheme, the bitwise `AND` of two bit-vectors is performed—the time-complexity of which is determined by the amount of available hardware (some researchers [58] reasonably argue that it is constant time).

On the other hand, determining set *emptiness* depends on the bit-indexing scheme. An unpartitioned Bloom filter represents an empty set if and only if all m bits of the bit-vector are set to zero. Consider that if a single bit is set, it is possible (though unlikely) that some element is mapped to that same bit by all k hash values, making the filter non-empty. In contrast, partitioned Bloom filters represent an empty set if and only if

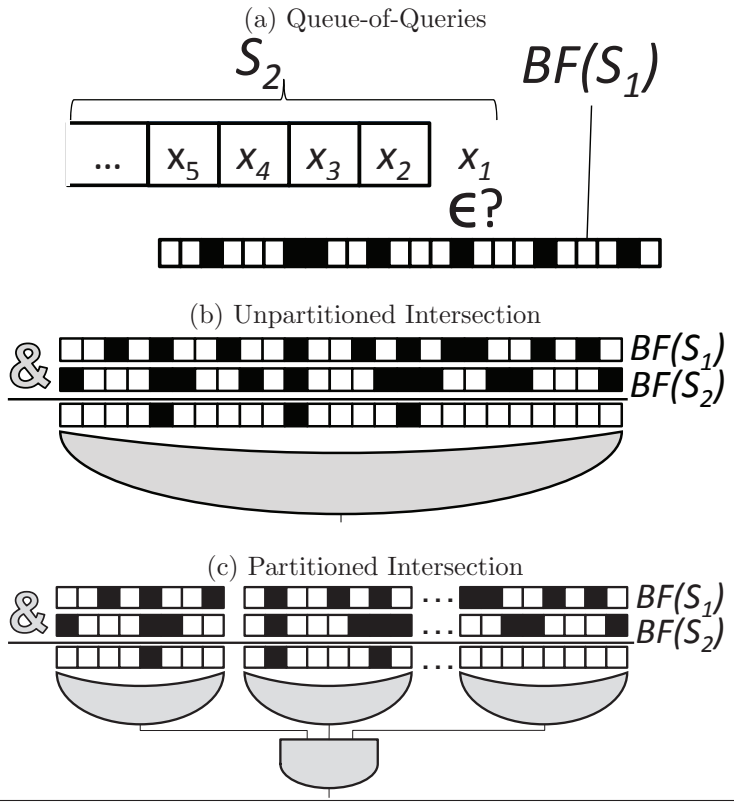


Figure 3.1: Three methods of testing for null-intersection between sets S_1 and S_2: (a) by a queue-of-queries into the Bloom filter of S_1, such that if any element of S_2 matches in $BF(S_1)$, the sets are reported to share some overlap; (b) by intersecting two unpartitioned Bloom filters by bitwise `AND`, where any resulting asserted bits indicate a set-overlap; (c) by intersecting two partitioned Bloom filters, where an intersection-result consisting of at least one empty partition indicates that the input sets are disjoint.

at least one partition is empty, with all m/k bits set to zero [13]. For sufficiency, note that an empty set asserts no bits, such that all k partitions remain zero. For necessity, since inserting one element requires asserting one bit in all partitions, then if at least one partition is empty, it must be that no combination of elements can be represented by that filter—i.e., the filter is empty. Figures 3.1b and 3.1c use logic gates to illustrate the use of Bloom filter intersection for null-intersection tests.

The following definition introduces a predicate that identifies false set-overlap via Bloom filter intersection. Due to the difference in empty-set representation, partitioned and unpartitioned filters have differing statistical properties; we condition on the choice of hash indexing scheme in the next section.

Definition 3. *Let $S_1, S_2 \subset U$ be two fixed, disjoint sets, each represented by Bloom filters of m bits and hash k-tuple $h \in (\mathcal{H}_m \cup \mathcal{H}_{\frac{m}{k}})$. We define the* false set-overlap by Bloom intersection *predicate* $\mathrm{FSO}_{\cap}(S_1, S_2, h)$ *to be true, if $BF(S_1) \cap BF(S_2) \neq \emptyset$, even though $S_1 \cap S_2 = \emptyset$.*

3.2 Probability of False Set-Overlap

Having defined the conditions for three types of Bloom filter false set-overlap events, we now model their probability distributions. We begin with the probability of false set-overlap by queue-of-queries—using filters as Bloom "intended." Concerning the following theorem, fix two disjoint sets $S_1, S_2 \subset U$. The filter $BF(S_1)$ is m bits long using a truly random hash tuple of the partitioned bit-indexing scheme: $h \in \mathcal{H}_{\frac{m}{k}}$.

Theorem 1. *A false set-overlap by queries of S_2 into partitioned $BF(S_1)$ is reported with probability*[1]

$$\Pr\left[\mathrm{FSO}_{\in}(S_1, S_2, h) \mid h \in \mathcal{H}_{\frac{m}{k}}\right] = 1 - \left(1 - \left(1 - \left(1 - \frac{k}{m}\right)^{|S_1|}\right)^{k}\right)^{|S_2|}. \tag{3.1}$$

Proof. Consider the contrary, between the two disjoint sets S_1 and S_2, when will a false set-overlap be avoided? Using the Bloom filter representation, the sets are correctly reported disjoint iff $(\forall x \in S_2)(x \notin BF(S_1))$, when every one of the $|S_2|$ unique queries into $BF(S_1)$ does *not* return a false positive. Model these unique queries as a sequence of up to $|S_2|$ Bernoulli trials, where a trial "success" implies a false positive on an individual query. Let random variable N be the number of unique membership queries from S_2 before one is reported a false positive. Thus N follows a geometric distribution, the number of Bernoulli failures before the first success, with probability of success $p = \Pr[\mathrm{FP}_{\in}(S_1, x, h)]$

[1] Building on Lemma 1, this distribution can also be approximated by $1 - \left(1 - \left(1 - e^{-\frac{k|S_1|}{m}}\right)^{k}\right)^{|S_2|}$.

for any $x \in S_2$. The two sets are deemed disjoint by Bloom filter queries if all $|S_2|$ trials fail, or if $N \geq |S_2|$. A false set-overlap results if the latter is not true. Thus,

$$\Pr[\text{FSO}_{\in}(S_1, S_2, h)] = \Pr[N < |S_2|] \tag{3.2}$$

$$= \Pr[N \leq |S_2| - 1] \tag{3.3}$$

$$= 1 - (1 - p)^{|S_2|-1+1} \tag{3.4}$$

$$= 1 - (1 - \Pr[\text{FP}_{\in}(S_1, x, h)])^{|S_2|}, x \in S_2 \tag{3.5}$$

where Eq. (3.4) substitutes the geometric cumulative distribution function. Conditioning Eq. (3.5) on $h \in \mathcal{H}_{\frac{m}{k}}$ and substituting Eq. (2.1) gives (3.1). □

We now state and prove the probability that Bloom filter intersection will flag a false set-overlap,[2] for both unpartitioned and partitioned bit-indexing. Let $S_1, S_2 \subset U$ be disjoint sets. Both are represented by Bloom filters with length m bits, using the same truly random hash function tuple h. The bit-indexing scheme is conditioned in the theorem.

Theorem 2. *A false set-overlap by Bloom filter intersection of unpartitioned $BF(S_1)$ and $BF(S_2)$ is reported with probability*

$$\Pr[\text{FSO}_{\cap} \mid h \in \mathcal{H}_m] = 1 - \left(1 - \frac{1}{m}\right)^{k^2|S_1||S_2|}. \tag{3.6}$$

For partitioned Bloom filters, a false set-overlap is reported with probability

$$\Pr\left[\text{FSO}_{\cap} \mid h \in \mathcal{H}_{\frac{m}{k}}\right] = \left(1 - \left(1 - \frac{k}{m}\right)^{|S_1||S_2|}\right)^k. \tag{3.7}$$

[2]Some readers may note that a membership query of y into a Bloom filter $BF(S)$ is akin to creating a new filter from y, $BF(y)$, and determining whether $BF(S) \cap BF(y)$ is empty. It is straightforward to show that Theorem 2 represents this idea, as Eq. (3.7) reduces to the false positive probability of Lemma 1 for $|S_2| = 1$.

Proof. Concerning Eq. (3.6), Section 3.1.2 argues that intersection of unpartitioned Bloom filters induces a false set-overlap when any bit in the resulting bit-vector is non-zero. An all-zero Bloom filter can only be created from an empty set (i.e., $BF(S) = \emptyset \iff S = \emptyset$). Therefore,

$$\begin{aligned}\Pr\left[\text{FSO}_\cap \mid h \in \mathcal{H}_m\right] &= \Pr\left[\neg(BF(S_1) \cap BF(S_2) = \emptyset) \mid S_1 \cap S_2 = \emptyset, h \in \mathcal{H}_m\right] \\ &= 1 - \Pr\left[BF(S_1) \cap BF(S_2) = \emptyset \mid S_1 \cap S_2 = \emptyset, h \in \mathcal{H}_m\right] \end{aligned} \tag{3.8}$$

We use Lemma 2 by Guo *et al.* [18],

$$\Pr\left[BF(S_1) \cap BF(S_2) = BF(S_1 \cap S_2) \mid h \in \mathcal{H}_m\right] = (1 - 1/m)^{k^2 \times |S_1 - S_1 \cap S_2| \times |S_2 - S_1 \cap S_2|}$$

but assume that the sets are disjoint, $S_1 \cap S_2 = \emptyset$:

$$\Pr\left[BF(S_1) \cap BF(S_2) = \emptyset \mid S_1 \cap S_2 = \emptyset, h \in \mathcal{H}_m\right] = (1 - 1/m)^{k^2 |S_1||S_2|}. \tag{3.9}$$

Substituting (3.9) into (3.8) shows (3.6).

Regarding Eq. (3.7), to avoid a false set-overlap, the intersect partitioned Bloom filter requires at least one partition to be empty, with all m/k bits set to zero. Thus the negation of this statement, a false set-overlap, results when all k partitions are non-empty. Consider the intersection of any one single partition: note that it operates identically to an unpartitioned Bloom filter with length m/k bits and a single hash function indexing the sub-vector. Eq. (3.6) of this theorem therefore suggests that a single Bloom filter partition of length m/k bits with one hash function is non-empty with probability

$$1 - \left(1 - \frac{1}{m/k}\right)^{(1)^2 |S_1||S_2|}. \tag{3.10}$$

Looking at the entire partitioned Bloom filter, we assume that the "emptiness" of all

k partitions is mutually independent. Then the probability that all k partitions are non-empty is the product of (3.10) k times,

$$\left(1 - \left(1 - \frac{k}{m}\right)^{|S_1||S_2|}\right)^k,$$

completing the proof of Eq. (3.7). □

3.3 Analytical Comparisons of Null-Intersection Tests

We wish to establish a better understanding of the three Bloom filter configurations for null-intersection tests—particularly how they compare in generating false conflicts. It is difficult to compare the probability distributions derived in the previous section by inspection, hence we apply them to analytically compare the statistical properties and space requirements of Bloom filter intersection and querying in this context. Specifically, we demonstrate (i) that partitioned intersection always outperforms unpartitioned intersection for the same k; and (ii) that for equivalent probability of false set-overlap (FSO), partitioned Bloom filter intersection requires a factor $\Omega(\sqrt{|S_2|})$ more space than performing a queue-of-queries (of set S_2) into a Bloom filter.

3.3.1 Preliminary Inequalities

We state elementary inequalities from Mitrinović *et al.* [39, 40] used to prove the main results of the section.

Lemma 3. [40] *Bernoulli's Inequality. If* $-1 < x < \frac{1}{n-1}, x \neq 0$, *and integer* $n = 2, 3, \ldots$, *then*

$$1 + nx < (1 + x)^n < 1 + \frac{nx}{1 + (1 - n)x}.$$

Lemma 4. [39] *Generalization of Bernoulli's Inequality.*

If $0 < q < p$ *and* $-q < x < 0$*, then*

$$\left(1 + \frac{x}{q}\right)^q \geq \left(1 + \frac{x}{p}\right)^p.$$

Lemma 5. *If real* x *is such that* $0 < x < 1$ *and integer* $n > 1$*, then*

$1 - x^n > 1 - x > (1 - x)^n$.

Proof. $x \in (0, 1) \Rightarrow x^n < x$, so evidently $1 - x^n > 1 - x$.

Also, $(1 - x) \in (0, 1) \Rightarrow 1 - x > (1 - x)^n$. □

3.3.2 Partitioned vs Unpartitioned Bloom Filter Intersection

The following theorem asserts that partitioned Bloom filter intersection has a lower probability of false set-overlap than unpartitioned. It concerns two disjoint sets $S_1, S_2 \subset U$, that are represented by Bloom filters of the same length m, with the same hash function tuple. In each case, the k hash functions are truly random and independent, and we consider $m > k > 1$, since for a single hash function, partitioned Bloom filters are effectively unpartitioned.

Theorem 3. *Concerning false set-overlap by Bloom filter intersection, the partitioned bit-indexing scheme follows a probability distribution that is strictly less than that of an unpartitioned Bloom filter. That is,* $(\forall h_{m/k} \in \mathcal{H}_{\frac{m}{k}})(\forall h_m \in \mathcal{H}_m)$*,*

$$\Pr\left[\mathrm{FSO}_{\cap}(S_1, S_2, h_{m/k})\right] < \Pr\left[\mathrm{FSO}_{\cap}(S_1, S_2, h_m)\right].$$

Proof. We begin by using Lemma 4, substituting $x = -1/m$, $q = 1/k$, and $p = 1$, which satisfies $0 < q < p$ and $-q < x < 0$, to see that $\left(1 - \frac{k}{m}\right)^{\frac{1}{k}} \geq \left(1 - \frac{1}{m}\right)$. Additionally, since

$m > k > 1$, then by Lemma 5, $\left(1 - \frac{1}{m}\right) > \left(1 - \frac{1}{m}\right)^k$. Combining these observations,

$$\left(1 - \frac{k}{m}\right)^{\frac{1}{k}} > \left(1 - \frac{1}{m}\right)^k \quad \Rightarrow \left(1 - \frac{k}{m}\right)^{|S_1||S_2|} > \left(1 - \frac{1}{m}\right)^{k^2|S_1||S_2|}, \tag{3.11}$$

where the implication follows since $m > k > 1$, and we raise each side to the power of $(k|S_1||S_2|) > 0$. Using (3.11), we show the main result,

$$\begin{aligned}
\Pr\left[\mathrm{FSO}_\cap(S_1, S_2, h_m)\right] &= 1 - \left(1 - \frac{1}{m}\right)^{k^2|S_1||S_2|} \\
&> 1 - \left(1 - \frac{k}{m}\right)^{|S_1||S_2|} && (3.12) \\
&> \left(1 - \left(1 - \frac{k}{m}\right)^{|S_1||S_2|}\right)^k && (3.13) \\
&= \Pr\left[\mathrm{FSO}_\cap(S_1, S_2, h_{m/k})\right], && (3.14)
\end{aligned}$$

where Eq. (3.13) follows from Lemma 5. □

3.3.3 Space Requirements of Intersection and Queue-of-Queries

Given that partitioned intersection outperforms unpartitioned intersection, the following theorem thus compares the methods queue-of-queries and partitioned Bloom filter intersection for testing null-intersection. The metric of consideration is more concrete than that in the previous theorem: for equivalent level of imprecision, we will compare the space requirements of QoQ and intersection. Specifically, we will show that the bit-vector space savings of QoQ is at least a factor of the square root of set cardinality, relative to partitioned Bloom filter intersection, when the respective probabilities of FSO are equal, under reasonable conditions.

Consider the disjoint sets $S_1, S_2 \subset U$. Let $BF_q(S_1)$ be a partitioned Bloom filter of length m_q bits with a truly random k-tuple hash function $h_q \in \mathcal{H}_{\frac{m_q}{k}}$, for use in a queue-of-queries. Let partitioned Bloom filters $BF_i(S_1), BF_i(S_2)$ have length m_i bits and

be indexed by the truly random k-tuple hash function $h_i \in \mathcal{H}_{\frac{m_i}{k}}$, for use in Bloom filter intersection. Assume more than one hash tuple value, $k > 1$, nontrivial sets, $|S_1|, |S_2| > 1$, and assume bit-vector lengths $m_q > k$ and $m_i > d|S_1||S_2| > k$, for some constant $d \geq 1$. For the following theorem, we will equate the FSO probability distributions of QoQ and partitioned intersection (i.e., assign them the same level of imprecision), and then compare the bit-vector lengths m_q and m_i. The constant $d \geq 1$ forces us to consider "reasonable" bit-vector lengths.[3]

Theorem 4. *Assuming the preceding system and conditions, the bit-vector space requirement of partitioned Bloom filter intersection is a factor $\Omega(\sqrt{|S_2|})$ larger than the queue-of-queries method, for equivalent probability of FSO. Specifically, if $k > 1$ and*

$$\Pr[\mathrm{FSO}_{\cap}(S_1, S_2, h_i)] = \Pr[\mathrm{FSO}_{\in}(S_1, S_2, h_q)], \tag{3.15}$$

then

$$m_i > m_q \frac{|S_2|^{\left(1-\frac{1}{k}\right)}}{1+\frac{k}{d}}. \tag{3.16}$$

Proof. Using the theorems of Section 3.2, we first equate the stated probabilities, then show the inequality between m_i and m_q using the lemmas of Section 3.3.1. When inequality (3.16) is proven, we prove the use of big omega notation. The following equality restates Eq. (3.15):

$$\left(1-\left(1-\frac{k}{m_i}\right)^{|S_1||S_2|}\right)^k = 1-\left(1-\left(1-\left(1-\frac{k}{m_q}\right)^{|S_1|}\right)^k\right)^{|S_2|}.$$

For clarity, let $a = |S_1|$, $b = |S_2|$, and $c_i = \left(1-\frac{k}{m_i}\right)^a$, and likewise for c_q. Applying these

[3] The probability of false set-overlap (for both QoQ and intersection) is prohibitively high until the Bloom filter length exceeds some inflection point (e.g., see Figure 3.2 of the following section). Without proof, the condition $m_i > d|S_1||S_2|$ captures the "useful" convex behavior of the probability distribution following the inflection point; it is indeed simpler, and more interesting, to compare bit-vector space requirements under this condition.

substitutions, we have

$$\left(1 - {c_i}^b\right)^k = 1 - \left(1 - (1 - c_q)^k\right)^b$$

which is rearranged into

$$1 - \left(1 - {c_i}^b\right)^k = \left(1 - (1 - c_q)^k\right)^b. \tag{3.17}$$

Observe that $c_i, c_q \in (0, 1)$ since $m_i, m_q > k$ and $a > 1$. Therefore $(1 - c_q)^k \in (0, 1)$, so with integer $b > 1$, we may apply the left side of Bernoulli's inequality (Lemma 3) to the right hand side of Eq. (3.17) and have

$$1 - \left(1 - {c_i}^b\right)^k = \left(1 - (1 - c_q)^k\right)^b > 1 - b\,(1 - c_q)^k\,.$$

Rearranging and simplifying terms,

$$b\,(1 - c_q)^k > \left(1 - {c_i}^b\right)^k.$$

Isolate b on the left hand side, take the k'th root, then expand c_i and c_q:

$$\begin{aligned} b^{\frac{1}{k}} &> \frac{1 - {c_i}^b}{1 - c_q} \\ &= \frac{1 - \left(1 - \frac{k}{m_i}\right)^{ab}}{1 - \left(1 - \frac{k}{m_q}\right)^a}. \end{aligned} \tag{3.18}$$

These steps are valid as $c_i, c_q \in (0, 1)$. Now consider the denominator of (3.18). Apply the left side of Bernoulli's inequality (Lemma 3), since integer $a > 1$, and $m_q > k$. Then

$$1 - \left(1 - \frac{k}{m_q}\right)^a < \frac{ak}{m_q} \Rightarrow \frac{1}{\left(1 - \left(1 - \frac{k}{m_q}\right)^a\right)} > \frac{m_q}{ak}. \tag{3.19}$$

Now focus on the numerator of (3.18), $1-\left(1-\frac{k}{m_i}\right)^{ab}$. Using the right side of Lemma 3, we let $x=-\frac{k}{m_i}$, and $n=ab$, which satisfies $-1<x<\frac{1}{n-1}$ since $m_i>k$. Thus,

$$\left(1-\frac{k}{m_i}\right)^{ab} < 1+\frac{ab\left(-\frac{k}{m_i}\right)}{1+(1-ab)\left(-\frac{k}{m_i}\right)}$$

$$\begin{aligned}\text{Rearranging,}\quad 1-\left(1-\frac{k}{m_i}\right)^{ab} &> \frac{\frac{kab}{m_i}}{1+\frac{kab}{m_i}-\frac{k}{m_i}}\\ &> \frac{\frac{kab}{m_i}}{1+\frac{kab}{m_i}}\\ &= \frac{kab}{m_i+kab}.\end{aligned} \tag{3.20}$$

Combining inequalities (3.19) and (3.20) into (3.18), we have

$$\begin{aligned}b^{\frac{1}{k}} > \frac{1-\left(1-\frac{k}{m_i}\right)^{ab}}{1-\left(1-\frac{k}{m_q}\right)^{a}} &> \frac{\frac{kab}{m_i+kab}}{\frac{ka}{m_q}}\\ &= b\frac{m_q}{m_i+kab}.\end{aligned}$$

Rearranging, we have shown thus far that

$$m_i+abk > m_q b^{1-\frac{1}{k}}.$$

For some constant $d\geq 1$, if designers choose $m_i > abd$, then $m_i\frac{k}{d} > abk$, thus $\left(1+\frac{k}{d}\right)m_i > m_i+abk$. Therefore, returning $b=|S_2|$,

$$m_i > m_q\frac{|S_2|^{1-\frac{1}{k}}}{1+\frac{k}{d}},$$

as desired.

Big Omega Notation Regarding asymptotic notation for the space comparison, we

claim that, $\forall k \geq 2$, and a fixed $d \geq 1$, $\frac{m_i}{m_q} = \Omega(\sqrt{|S_2|})$, or formally, $\forall k \geq 2$, $\exists d \geq 1$,

$$(\exists c, n_0 > 0)(\forall |S_2| \geq n_0) \quad \frac{m_i}{m_q} \geq c\sqrt{|S_2|}. \tag{3.21}$$

We must prove this statement to be true. Since $\frac{m_i}{m_q} > \frac{|S_2|^{1-\frac{1}{k}}}{1+\frac{k}{d}}$ from (3.16), we can simplify the statement; we substitute $n = |S_2|$, and observe that to prove (3.21), we can equivalently prove that $\forall k \geq 2$, and fixed $d \geq 1$,

$$(\exists c, n_0 > 0)(\forall |S_2| \geq n_0) \quad \frac{n^{1-\frac{1}{k}}}{1+\frac{k}{d}} \geq c\sqrt{n}. \tag{3.22}$$

Before proving this statement, we further transform the right hand side inequality to an equivalent one that is easier to work with, taking the logarithm of both sides; specifically the right hand side inequality in (3.22) is equivalent to the following statements:

$$\begin{aligned}
\frac{n^{1-\frac{1}{k}}}{1+\frac{k}{d}} \geq c\sqrt{n} \Leftrightarrow & \frac{n^{1-\frac{1}{k}}}{d+k} \geq \frac{c}{d}\sqrt{n} \\
& \Leftrightarrow \left(1-\frac{1}{k}\right)\lg n - \lg(d+k) \geq \lg c - \lg d + \frac{\lg n}{2} \\
& \Leftrightarrow \lg c \leq \frac{\lg n}{2} - \lg(d+k) - \frac{\lg n}{k} + \lg d.
\end{aligned}$$

We choose to work with the latter inequality, which was achieved by multiplying both sides by $\frac{1}{d}$, taking the logarithm, and rearranging.

Now to prove the correct use of asymptotic notation above, i.e., (3.21), we will prove by induction on k that $\forall k \geq 2$, and a fixed $d \geq 1$,

$$(\exists c, n_0 > 0)(\forall n \geq n_0) \quad \lg c \leq \frac{\lg n}{2} - \lg(d+k) - \frac{\lg n}{k}. \tag{3.23}$$

We dropped the term $\lg d$, as it simply scales the parameter c that we are free to choose.

For the base case, $k = 2$, we actually show (3.21) above. For fixed $d \geq 1$, we have

from inequality (3.16),

$$\frac{m_i}{m_q} > \frac{|S_2|^{1-\frac{1}{k}}}{1+\frac{k}{d}} = \frac{\sqrt{|S_2|}}{1+\frac{2}{d}}$$

which satisfies (3.21) by choosing $c = \frac{1}{1+\frac{2}{d}}$ and $n_0 > 0$.

For the inductive step we assume (3.23) to be true for k, and we show that it holds for $k+1$. We make the substitution $A = \frac{\lg n}{2} - \lg(d+k+1) - \frac{\lg n}{k+1}$, and thus seek to show

$$(\exists c, n_0 > 0)(\forall n \geq n_0) \quad \lg c \leq \frac{\lg n}{2} - \lg(d+k+1) - \frac{\lg n}{k+1},$$

or equivalently

$$(\exists c, n_0 > 0)(\forall n \geq n_0) \quad \lg c \leq A. \tag{3.24}$$

We begin from the inductive hypothesis (3.23), and manipulate to show that $\lg c \leq A$. We assume $(\exists c, n_0 > 0)(\forall n \geq n_0)$

$$\begin{aligned}
\lg c &\leq \frac{\lg n}{2} - \lg(d+k) - \frac{\lg n}{k} \quad \text{(from the inductive hypothesis)} \\
&= \frac{\lg n}{2} - \lg(d+k) - \frac{\lg n}{k} + \lg(d+k+1) - \lg(d+k+1) + \frac{\lg n}{k+1} - \frac{\lg n}{k+1} \\
&= \frac{\lg n}{2} - \lg(d+k+1) - \frac{\lg n}{k+1} - \lg(d+k) - \frac{\lg n}{k} + \lg(d+k+1) + \frac{\lg n}{k+1} \\
&= A + \lg\left(\frac{d+k+1}{d+k}\right) + \left(\frac{1}{k+1} - \frac{1}{k}\right)\lg n.
\end{aligned}$$

Since $\left(\frac{1}{k+1} - \frac{1}{k}\right) < 0$, then indeed we can find some $n_0 > 0$, such that for $n \geq n_0$, $\lg c \leq A$, satisfying the statement (3.24). Thus we have proven (3.23) by induction on k, and (recalling that $n = |S_2|$) have equivalently shown that $\forall k \geq 2$ and fixed $d \geq 1$, $(\exists c, n_0 > 0)(\forall |S_2| \geq n_0) \quad \frac{m_i}{m_q} \geq c\sqrt{|S_2|}$, or $\frac{m_i}{m_q} = \Omega(\sqrt{|S_2|})$. □

3.4 Empirical Validation

In this section we empirically validate the probability distributions derived in Section 3.2. Empirical *rates* of false set-overlap are gathered for each of the three null-intersection tests, in four discrete hash function tuple configurations. A simple experiment tests two disjoint address-sets for overlap, using the three methods discussed: queue-of-queries, and partitioned and unpartitioned Bloom filter intersection. Each method returns two possible outcomes: either a false set-overlap, or disjoint sets. The experiment is repeated over one million trials, and the relative frequency of false set-overlap is recorded as the empirical rate.

3.4.1 Methodology

For a single experiment, two disjoint sets are generated, S_1 and S_2, containing unique pseudorandom 32-bit integers (addresses), using the `C` standard library `rand` function. For a given bit-vector length, m, and hash function tuple size, k, partitioned and unpartitioned Bloom filters are constructed for each of the sets, $BF_p(S_1), BF_p(S_2), BF_u(S_1), BF_u(S_2)$. Random hash functions are selected from the H_3 family [10] that approximately match the performance of ideal hash functions, for an address stream with sufficient entropy [41]. To test the queue-of-queries outcome, each $x \in S_2$ is tested for membership in $BF_p(S_1)$. If at least one query returns true, the false set-overlap is recorded. Likewise, to test the Bloom filter intersection outcome, we separately intersect $BF_p(S_1) \cap BF_p(S_2)$ and $BF_u(S_1) \cap BF_u(S_2)$, and determine whether the remaining bits represent an empty set as described in Section 3.1; otherwise a false set-overlap is recorded.

3.4.2 Analysis

Figure 3.2 visualizes the false set-overlap rate as a function of Bloom filter length, m. The four plots differ only in the size of the hash function tuple: $k = 1, 2, 4, 8$. The

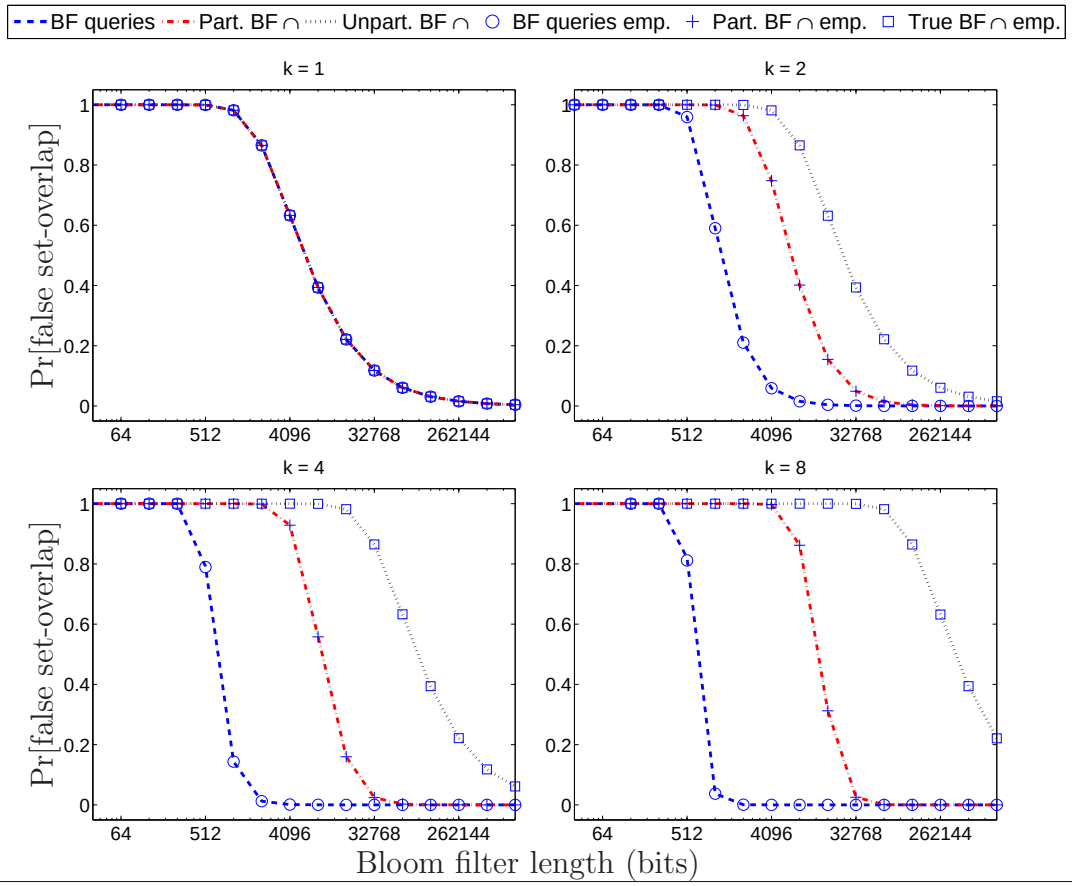


Figure 3.2: Probability and empirical rate of false set-overlap on the y-axis as they vary with increasing Bloom filter length on the x-axis (in log scale). Each plot represents a size k of the hash function tuple. The three curves plot the probabilistic models of false set-overlap, and are overlaid by sample points of the experimentally measured rate.

cardinalities of the address sets were fixed to $|S_1| = |S_2| = 64$ unique elements[4] for all trials. Each set of one million trials is represented by a point on the plot, having been assigned a fixed filter length. Filter lengths are shown in a log scale on the x-axis, and each length was sampled at a power-of-two to effectively show the trend of this roughly exponential decay. The derived theoretical distributions underlay the empirical sample points, visualizing the relationship between Bloom filter length and false set-

[4]Address-set cardinality is certainly application-specific; a number of earlier studies exhibited average read-set sizes of 26 to 67 addresses [12, 13, 51, 53, 68], but as few as 2 addresses [32, 67] and as many as 2000 addresses [51,68]. Write set sizes are typically smaller but still vary from 1 address [12,33,53,67,68] to over 1500 [51, 68]—we choose 64-element sets as a compromise within this large space. Varying the set cardinalities will not change the general trends observed in Figure 3.2.

overlap probability. It is apparent that the empirical sample points follow the theoretical distributions, validating the accuracy of our work.

3.5 Implications

Figure 3.2 illustrates that varying k has a different effect on each null-intersection test. For nontrivial hash function tuples (i.e., $k > 1$), the queue-of-queries method benefits from increasing k (up to a point), while the probability distribution for unpartitioned Bloom filter intersection only becomes worse. Upon close inspection of the distribution for partitioned Bloom filters, increasing k is beneficial only for filter lengths of at least 16k bits. These three patterns can also be shown analytically, by minimizing the probability distributions with respect to k. The reader is directed to Broder and Mitzenmacher [9] for an example of this process. Without proof, the optimal number of hash functions for partitioned intersection is $k^* = \frac{m}{|S_1||S_2|} \ln 2$, minimizing the false conflict probability to 2^{-k^*}.

Implication 1. *Issuing a series of Bloom filter membership queries permits testing null-set-intersection with significantly lower space overhead than Bloom filter intersection, for larger than one-tuple hash functions.*

Implication 2. *When intersection is unavoidable for null-intersection testing, partitioned intersection is preferable to unpartitioned as it provides the same service, with the same time-complexity, with lower (or equal) probability of false set-overlap.*

Surprisingly, for a single hash function, $k = 1$, all three methods share the same probability distribution (and empirical sample points)—this can be verified by substituting $k = 1$ for the theorems of Section 3, and the probability of FSO is $1 - (1 - \frac{1}{m})^{|S_1||S_2|}$. This fact may run counter to intuition, so consider the following explanation. Testing a single bit in a single-partitioned query of an element is also equivalent to intersecting

(AND) the Bloom filter of that element with the Bloom filter of a set. When issuing a queue-of-queries, a set-overlap is returned if *any* element of the queue matches in the filter; this is in turn equivalent to ORing all of the latter bitwise intersections, i.e., $\bigvee_{x \in S_1}(BF(x) \wedge BF(S_2))$. The $BF(S_2)$ can be factored out of the logical formula to give $BF(S_2) \wedge (\bigvee_{x \in S_1} BF(x))$, and with a single partition, $(\bigvee_{x \in S_1} BF(x)) = BF(S_1)$, thus single-partitioned QoQ and intersection are equivalent. Given this equivalence, a time-complexity comparison would be helpful; unfortunately the many hardware-dependent considerations make such a study beyond the scope of this dissertation.

Implication 3. *Remarkably, when restricted to encoding addresses using a single hash function, Bloom filter querying and intersection share equivalent probability of false set-overlap.*

3.6 Summary

In this chapter we developed a theory to model and compare Bloom filter null-intersection test configurations. The data structure operations were described, including the conditions necessary to induce false set-overlaps. The probability of false set-overlap was derived for queue-of-queries and partitioned/unpartitioned intersection. These probability distributions were then analytically compared and empirically validated. We concluded by summarizing the design implications of the developed theory, noting that in terms of probability of false set-overlap, the queue-of-queries approach is better than intersection (providing a factor $\Omega(\sqrt{|S_2|})$ space savings, i.e., the square root of set cardinality), and partitioned intersection is better than unpartitioned.

Chapter 4

The Batch-of-Bloom-Filters (BoB) Approach

Existing null-intersection tests present a difficult choice between time-consuming but accurate queues-of-queries, or fast but inaccurate Bloom filter intersection—hence we are motivated to develop a compromise between the two approaches. We do this by essentially maintaining a fixed-size queue of Bloom filters, and distributing addresses across these Bloom filters. This way we continue to exploit the fast bitwise `AND` operation of intersection, but also approach some of the accuracy benefit of making queries for individual elements—but in this case by maintaining a group or *batch-of-Bloom-filters* (BoB). Notice in Eq. (3.7) of Theorem 2 that the false conflict probability of partitioned intersection increases with the product of the set cardinalities; the basic idea of BoB is to reduce this product term by distributing the set elements over the batch of Bloom filters, with each filter representing a subset of the original address set.

This chapter introduces the structure and operations of a BoB, contributes an intuitive alternate design, and describes the obstacles to deriving a probability of false set-overlap. Lacking an analytical comparison, we thus empirically demonstrate the statistical compromise of BoB relative to the other null-intersection tests.

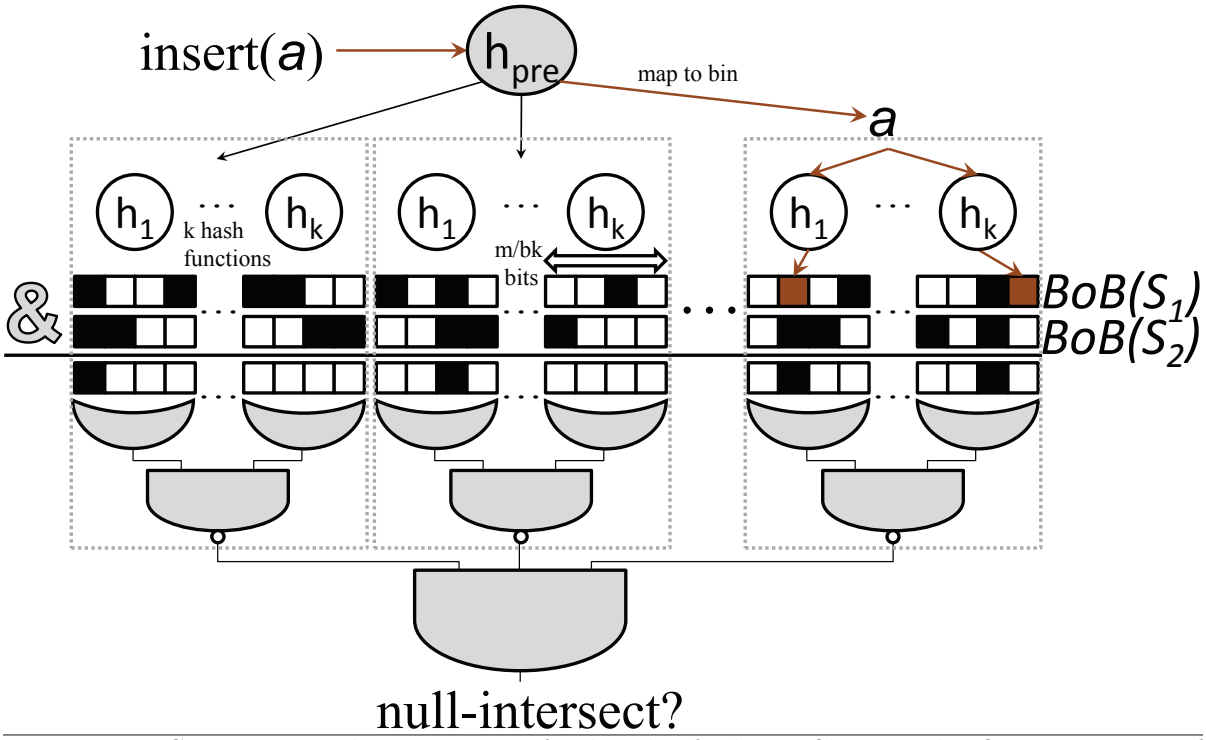


Figure 4.1: Structure and operations of a batch-of-Bloom-filters. The first two rows of boxes represent the BoB bit-vectors of sets S_1 and S_2. Address a is inserted into the BoB of S_1, with its h_{pre} mapping to the last bin shown in red. The dotted grey frames distinguish the internal partitioned Bloom filters (bins) of the batch, complete with k hash functions on top and the typical partitioned null-intersection test on the bottom. The sets S_1 and S_2 are tested for null-intersection, by pairwise partitioned intersection of the internal Bloom filters.

4.1 BoB Structure and Operation

To build a BoB, the m bits normally allotted to a single Bloom filter are segmented across a batch of b regular filters, each with k partitions of integer length $\frac{m}{bk}$ bits and k (ideally independent) hash functions.[1] An individual Bloom filter of the batch will be referred to as a *bin*, and b as the batch *bin count*. A "prefilter" hash function maps addresses to one bin, $h_{pre} : U \to \{0, \ldots, b-1\}$. Hao *et al.* [21] similarly prehash the set elements in their query-based Bloom filter, and visualize h_{pre} as partitioning the keys of the address space, U, into b disjoint subsets, $U = U_1 \cup U_2 \cup \cdots \cup U_b$. Therefore, each

[1]The set of hash functions does not have to be the same across filters, but we are so far not motivated to vary them and hence they are the same across filters for our implementations.

Bloom filter (or bin) in the batch can only represent elements from one of these address subsets, and null-intersection tests thus require the pairwise intersecting of equivalent subset bins. Partitioned Bloom filters ($k = 1, 2, \dots$) are used exclusively for the bins, as they outperform unpartitioned for bitwise intersection (see Theorem 3). Figure 4.1 illustrates the operations of the batch. We denote the bit-vector of the batch as B.

Insertion Given an address a, the $h_{pre}(a)$'th Bloom filter of the batch is selected, and a is inserted as usual. If the bit-vectors in the batch are instead thought of as a single bit-vector B, then the bits $B[h_i(a) + \frac{(i-1)m}{bk} + \frac{m}{b}h_{pre}(a)]$ are set to one for $1 \leq i \leq k$.

Null-Intersection Test The Bloom filters of equivalent address subsets are pairwise intersected by bitwise `AND`. If every one of the b intersected filters has at least one all-zero partition, then the intersection is reported empty, otherwise an overlap is reported.

4.2 Alternate Construction

The reader may recognize an increased risk of false set-overlap if the prefilter hash function introduces load imbalance across the batch.[2] To reduce the impact of poor prefilter hashing, one might instead consider time-multiplexing the addresses across the batch, for example by adding addresses to filters in a round-robin fashion. This strategy has two clear limitations, stemming from the fact that addresses can be mapped to any filter (depending on time). The first limitation is a lack of redundancy elimination (an address could simultaneously appear in multiple filters). In this case the filter density (the ratio of bits set to one) will be higher than for the BoB we previously described, and therefore false set-overlaps will be more frequent. The second limitation is that, for correctness, null-intersection tests must cross-intersect each filter with every other filter

[2] Choice of prefilter hash function is discussed in Chapter 5.

to prevent false negatives, increasing execution time immensely. Hence we dismiss this alternative construction.

4.3 Toward a Theoretical Analysis

Selecting the new parameter of BoB bin count, b, expands the already large Bloom filter intersection design space, thus an accurate model for probability of false set-overlap by BoB intersection would be extremely helpful for system designers. Such a derivation was pursued for this dissertation, depending in part on our developed theory of partitioned intersection, i.e., Eq. (3.7) of Theorem 2. Unfortunately, our closed-form probability distributions extend the "classic" analysis of Bloom filter false positives, which, as noted in Section 2.1.3, are good approximations for relatively large Bloom filters, but are overly optimistic for smaller filter lengths [16]. Since a modest-sized BoB contains a batch of distinct filters that are themselves each segmented into partitions, the resulting partitions are typically not large enough for the "classic" analysis or the closed form Eq. (3.7) to apply. An accurate model of BoB intersection must instead transform the false positive model proposed by Christensen *et al.* [16] to model bitwise intersections (similar to Lemma 2 by Guo *et al.* [18]), while accounting for address-to-bin mappings using a multinomial probability distribution. Such a derivation is beyond the scope of this work.

4.4 Rates of False Set-Overlap

Without an analytical model, we instead demonstrate that the BoB strategy is a good compromise between queue-of-queries and partitioned intersection by comparing synthetic rates of false set-overlap in Figure 4.2. Sensitivity to bit-vector length is observed along the horizontal axes. Each curve represents the FSO rate for a particular null-intersection test, and BoB intersection is evaluated with six curves representing

different bin counts. The experiments closely resemble the empirical validation in Section 3.4, but we omit unpartitioned intersection due to its inaccuracy, and study BoB intersection instead. The bit-vector length is the total allocated space used to represent one of the two sets—so a BoB of length 2048 bits with 2 bins and $k = 2$ has two internal Bloom filters with partition lengths $\frac{2048}{2\times2} = 512$ bits, and is intersected with a BoB of identical configuration. The four plots differ in the number of Bloom filter partitions and hash functions, $k = 1$ to 8.

4.4.1 Methodology

The empirical FSO rates are measured by recording the result of many synthetic null-intersection tests between two disjoint sets using one of QoQ, partitioned intersection, or BoB intersection. Each data point in the figure represents the relative frequency of observed false set-overlaps over 10 million trials. For a particular null-intersection trial, two disjoint sets of random 32-bit integers are generated, S_1 and S_2, using the C standard library `rand` function. For all trials, the sets have cardinalities of $|S_1| = 64$ and $|S_2| = 32$ unique integers, chosen to approximate the size disparity in typical read- and write-sets [52].

For this evaluation, Bloom filters are constructed from the integer sets using random hash functions from the H_3 [10] family, to approximately model uniform hash functions. With appropriately constructed Bloom filters, the null-intersection tests are executed as described in Sections 3.1 and 4.1. The QoQ is configured in two ways: querying integers of S_1 into $BF(S_2)$ (labeled QoQ $1 \in 2$), and vice versa (QoQ $2 \in 1$), to empirically demonstrate the difference between querying fewer elements into a denser filter, and querying more elements into a filter of lower occupancy.

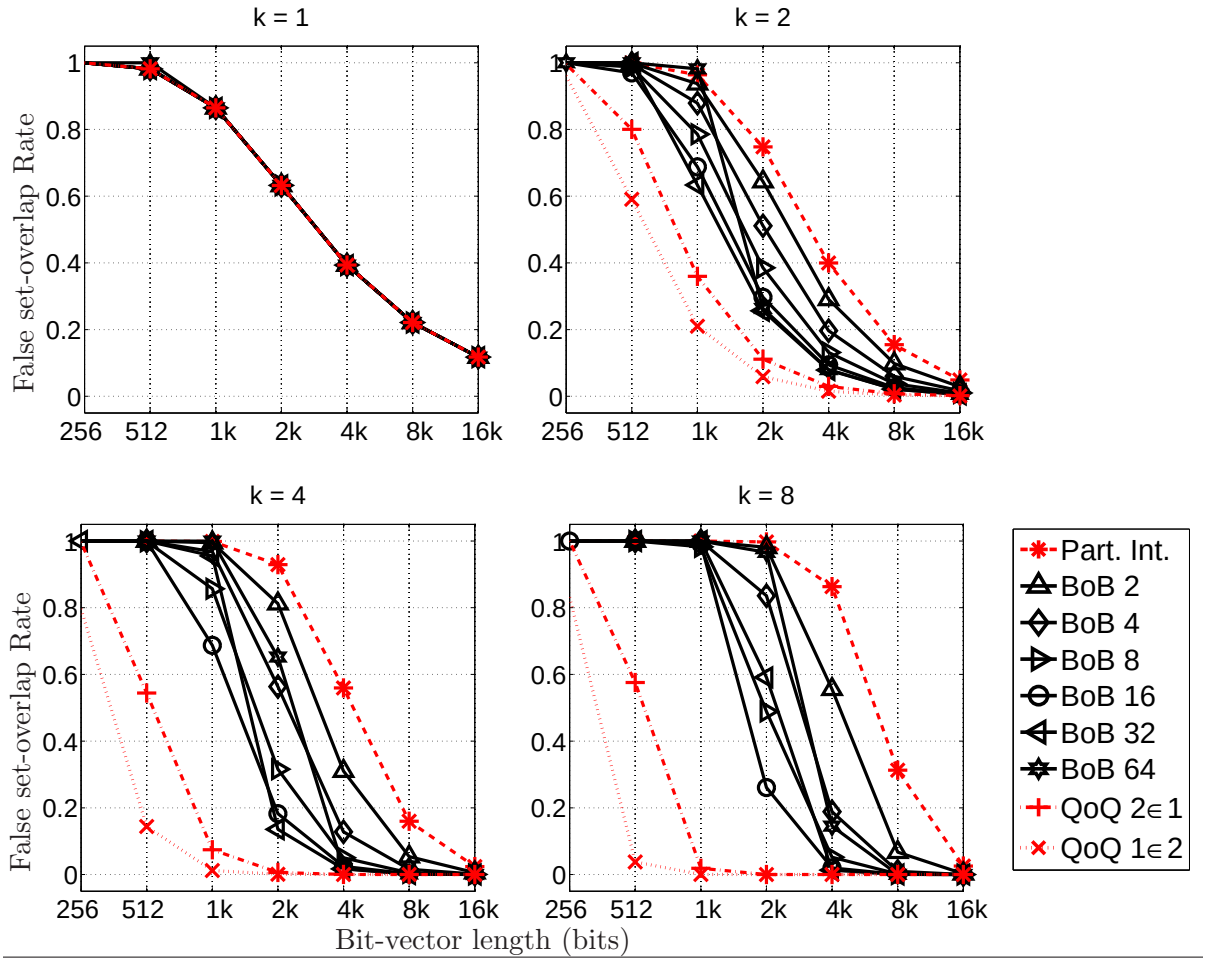


Figure 4.2: Rates of false set-overlap, varied with increasing bit-vector length on the horizontal axis. Each curve represents a particular null-intersection test design, where BoB is represented with several bin counts, b. A bin count of one is equivalent to partitioned intersection. Lower rate of FSO with smaller bit-vector length is better.

4.4.2 Single Partition

For the single-partitioned case of $k = 1$, all null-intersection tests appear to share equivalent probability of false set-overlap. With QoQ and partitioned intersection this fact was highlighted in Section 3.5, but remarkably, for BoB intersection every batch bin count appears to fit the same curve. Note that in a batch of single-partitioned Bloom filters, the prefilter function effectively perturbs the mapping of an address to the single bit set to one. In this case a BoB intersection closely resembles a regular unpartitioned intersection. All following subsections analyze the plots for $k = 2, 4, 8$.

4.4.3 Queue-of-Queries

Both configurations of QoQ outperform all other null-intersection tests when the filter uses more than one partition. It is apparent that querying elements of the larger set, S_1, into the Bloom filter of the smaller set (the 'x' marker) is the better strategy of the two. However, it is insightful to see that the alternative query method still outperforms all BoB configurations; the importance of this observation is revealed in Section 5. Increasing k further reduces the false set-overlap rate for both configurations, up to an optimum value that depends on the bit-vector length [60]. For example, for a Bloom filter length of 512 bits, querying elements of the larger set benefits from increasing k from four to eight, but the other configuration produces a slightly worse false set-overlap rate.

4.4.4 BoB Intersection

As intended by design, the BoB intersection curves lie between those of QoQ and partitioned intersection. Our proposed null-intersection test can indeed exploit the simplicity of `AND`-based intersection, while improving upon the accuracy of regular partitioned intersection. For the case $k = 2$, the false set-overlap rate is reduced with every increase in the number of filters in the batch (b) up to 32 (32 is the cardinality of the smaller of the two integer sets). We observe diminishing returns with every increase of b up to 32, and note that increasing b to 64 produces an equivalent or worse FSO rate.[3]

As with typical Bloom filter configurations, the effect of increasing the number of partitions depends on the bit-vector length. In these experiments, four-way partitioning is beneficial for BoBs of 8, 16, and 32 bins, at total length of 2k bits. At 4k bits, BoBs with bin count of 4 and 64 begin to benefit from four-way partitioning. Comparing eight to four partitions, all configurations of BoB intersection deteriorate at 2k and 4k bits, but show reduced false set-overlap rates at 8k bits and beyond.

[3]One data point of BoB $b = 64$ is even worse than partitioned intersection at bit-vector length of 1k bits. It would appear that too many bins can severely increase the rate of FSO.

From this empirical study, it seems that the batch bin count should not exceed the cardinality of the smaller of the two sets to be intersected. A bin count of half this cardinality performs nearly as well, and is at lower risk of very poor performance for small bit-vector lengths. Increased partitioning is beneficial depending on selected bit-vector length, and number of filters in the batch. Unfortunately, the ideal design parameters, bit length, number of partitions, and bin count, all depend on the cardinalities of the sets, which are rarely known or fixed for transactional memory systems. The designer must then choose the configuration that is most likely to degrade gracefully.

4.5 Summary

In this chapter, we introduced a new null-intersection test called batch-of-Bloom-filter intersection, that seeks to offer a compromise between the accuracy of QoQ and the simplicity of bitwise intersection. The structure and operations of a BoB were described and illustrated, followed by the dismissal of an intuitive alternate design. We explained the barriers to deriving a probability of false set-overlap for this test, and suggested the future steps to pursue such a model. We concluded by comparing the rates of false set-overlap of all null-intersection tests, synthetically demonstrating the statistical compromise of BoB intersection.

Chapter 5

Software Implementation of Null-Intersection Tests

We have described the motivation for and theoretical behavior of three conventional methods of using Bloom filters for testing null-intersection: unpartitioned intersection, partitioned intersection, and queue-of-queries. We have also introduced a compromise method, the batch-of-Bloom-filters. In this chapter, the focus is shifted from theory to practical considerations, as we discuss the generic software implementation of these four methods, including the use of SIMD optimization. In the next chapter we test the practical relevance of the theoretical implications with a performance evaluation; we thus conclude this chapter by describing and comparing the specific implementation of these techniques in RingSTM [58].

5.1 Unpartitioned Intersection

Single-partition (hereafter called unpartitioned) Bloom filter intersection has been implemented in recent systems [4, 47, 58] and is likely also present in other systems with unspecified Bloom filter configurations [5, 48]. It is also referred to as the baseline configuration, where those in the later sections are *alternates*. The Bloom filter hash

function ought to be of high quality with a near-uniform output distribution (e.g. H_3 [10]), however such implementations have high overheads. By default we use a simple least-significant contiguous bit-select hash function of 8-byte aligned words, requiring only bit-shift and `AND` instructions. A Bloom filter typically contains hundreds of bytes or more, and so set-overlap is computed via a series of bitwise `AND` operations in a loop. One very low-level implementation choice for implementing such an intersection is to short-circuit the loop at the first non-empty intersection result, saving the computation of further intersections but at the cost of an extra branch in the loop to check for this condition.

5.2 Partitioned Intersection

In a partitioned scheme, the partitions are sequentially tested for null-intersection. If at least one partition is found empty, a null-intersection is returned. To avoid redundant work, the remainder of the testing for a partition is skipped once the intersection of any of its words is found to be non-null.

In theory, the multiple hash functions of a Bloom filter are independent, otherwise correlation among bit indexes could reduce the efficacy of partitioning. As we are interested in low overhead operations, we resort to bit-select hash functions of disjoint bit-fields of the address (of length $\log_2\left(\frac{m}{k}\right)$ bits). Unfortunately, this constraint can impede the goal of independence as there are a finite number of address bits, and potentially more bits are required for indexing large filters. The sequentially-executed multiple hashes add overhead for every Bloom filter insertion (at every shared memory read and write).

Kirsch and Mitzenmacher [27] proposed a double-hashing optimization which simulates a large number of independent hash functions by linearly combining only two. This strategy was attempted with two H_3 functions, and it permitted the effective use of four or more partitions, but the overheads were still high relative to simple bit-select—hence

we do not discuss this method further. It would be most beneficial when high-quality hash functions can be parallelized, particularly in hardware.

5.3 Queue-of-Queries

We assume a TM that intersects only read- and write-sets (not write-write), so the first design decision is which of the sets will form the queue and which the Bloom filter. From the previous section, Figure 4.2 indicates that in a QoQ, the rate of false set-overlap is lower when issuing queries of the larger set into a Bloom filter of the smaller set.

Since transactions are typically composed of more reads than writes, a read-set queue would incur fewer false aborts, but the linear-time null-intersection test would take longer. Furthermore, every read would be inserted (at prohibitive total cost) into an associative data structure to avoid redundant queries during intersection. Assuming a write-buffered TM implementation, we can exploit the necessary existence of an associative write-set, and opt to double the write-buffer's use as the queue-of-queries. This also eliminates the need for all write filter operations (clears and insertions).

5.4 Batch-of-Bloom-Filters

The batch is a collection of conventional partitioned Bloom filters, so the remaining implementation details concern the intersection operation itself, as well as the design of the prefilter hash function. The filters of the two batches are sequentially tested for pairwise null-intersection; if any pair is found to be non-null, the routine is terminated and a set-overlap returned. Regarding the prefilter function, a non-uniform mapping of addresses to filters can have very detrimental effects on the rate of false-set overlap. Indeed, since the partitioned Bloom filters use simple bit-select hash functions, we find that the prefilter function must be of better quality. Several fast hash functions were

tested for this role: H_3, CRC, multiplicative, and `XOR` [62]; of these, H_3 and an `XOR` were the most promising, and hence are evaluated in Section 6.3.

5.5 SIMD Optimizations

Bloom filter operations are easily parallelized when implemented in hardware [53], so we are motivated to study the operations that are amenable to vector instructions in software. In this section we consider the parallelization of bit-vector operations by exploiting Intel ISA SSE4.1 and SSE2 instructions.

5.5.1 Bitwise Intersection

The obvious target for vectorization is the data-parallel bitwise `AND` intersection, with granularities of 128-bit vectors, rather than 32-bit words. As indicated in Section 5.1, null-intersection can be tested in two ways (and both are applicable to unpartitioned, partitioned, and BoB intersection). The first method accumulates intersections into a register, then tests for emptiness at the end of the loop. This is implemented with the SSE2 `PAND` and `POR` instructions, with a vector-to-word conversion at the end for the null test. Alternatively, SSE4.1 offers the `PTEST` instruction to intersect two 128-bit vectors, and it returns whether the result was zero. This instruction enables a vector version of the second intersection method, avoiding unnecessary work when a partition is found non-empty, but it requires a branch after each test. The baseline TM system optionally offers the first accumulator method for SSE2-enabled processors, and we compare it to the second branch-after-intersection method in the evaluation.

5.5.2 Queue-of-Queries

Querying each address of the queue is a data-parallel operation that applies the same (read-only) hashing and bit-testing operations on all addresses. SSE4.1 instructions

enable four 32-bit addresses to be queried in parallel, particularly parallelizing the hash function operations at the address granularity. The k hash functions of partitioned filters are evaluated sequentially, but for four addresses simultaneously. Unfortunately, the indexing and testing of bits present significant barriers to vectorization, as they require memory-gather and bit-shift instructions that do not yet exist.[1] Thus following the vectorized hashing, bits are tested sequentially, which incurs additional overhead to extract the 32-bit indexes from 128-bit vectors. The evaluation chapter compares this vectorized query with the sequential counterpart.

5.5.3 Insertion

Bloom filter insertions (unpartitioned, partitioned, and BoB) are not amenable to current Intel SSE instructions. Addresses are inserted individually at the time of access, so data-parallelism is not available, as it is for a queue-of-queries. Task-level parallelism would be the next target, particularly exploiting the use of multiple independent, data-partitioned hash functions and bit-setting. However, in addition to the currently non-existent memory-gather and vector-vector shift instructions required by QoQ, insertion also requires a memory-scatter instruction that does not even appear to be scheduled for release. Unfortunately, Bloom filter insertions cannot be vectorized.

5.5.4 Bit-Vector Reset and Copy

Bloom filters are reset and copied using vector move instructions. This functionality is provided in the baseline TM system.

[1] Memory-gather and vector-shift-by-vector-value instructions will be released in the Intel ISA AVX2 [25].

5.6 Implementation in RingSTM

In this dissertation we use RingSTM [58] to evaluate the performance impact of the various null-intersection tests. RingSTM is a lazy write-buffered STM that performs conflict detection by single-partitioned Bloom filter null-intersection tests—and hence an excellent candidate to validate alternate Bloom filter implementations. The algorithm replaces the linear-time read and commit validation stages of earlier STMs with a constant number of Bloom filter intersections, requiring a single atomic operation to commit. The authors introduce a global ring of Bloom filters to represent the logically-committed write-sets, with which concurrent transactions intersect their read-sets (filters) during read and commit validation.

The experiments in the following chapter build upon the optimized single-writer algorithm, released in the open source RSTMv6 (the most current version available at the start of this work). The new `AND`-based intersections (partitioned and BoB) are straightforward to swap into RingSTM with the implementations described in this chapter, but additional low-level details about the queue-of-queries method are insightful. Using QoQ, the write filters of the Ring are replaced with write buffers, so the buffer data must be quickly copied from thread-local to global memory. To avoid traversing the entire buffer to copy the data, we instead swap the 32 bytes of meta-data (including buffer length, pointer to data, pointer to tables, etc.) of the thread's write buffer with that of an unused write buffer in the ring. This memory swap is accelerated with the Intel SSE2 `MOVDQA` instruction, which moves 128-bit vectors to and from memory. The optimization performs best when the meta-data do not cross cache line boundaries, thus the thread class becomes 32-byte aligned.

5.7 Summary

This chapter marked the departure from a theoretical study of Bloom filter configurations to building practical implementations for the performance evaluation to follow. We described the generic software implementation for each of the four studied null-intersection tests: queue-of-queries, and unpartitioned, partitioned, and BoB intersection. Recognizing that Bloom filter operations are amenable to vectorization, we described the Intel SSE instructions used to vectorize bitwise intersection and issue queries in parallel. We concluded by summarizing RingSTM, the vehicle by which we evaluate the performance impact of the null-intersection tests in the next chapter.

Chapter 6

Performance Evaluation

Having described the motivation, theory, and implementation for the three alternate Bloom filter null-intersection tests, in this chapter we evaluate their utility in practice. We determine whether the predicted reduction in false conflicts for QoQ, BoB and partitioned intersection can overcome the overhead of increased implementation complexity. Each null-intersection test is implemented in the conflict detection unit of RingSTM, and these are evaluated based on their reduction of total execution time of the STAMP benchmarks relative to the original baseline unpartitioned intersection.

The experimental methodology is first explained, detailing the configuration of the STAMP benchmarks, and the two underlying system configurations. Performance results are then presented for the best implementations of the four null-intersection tests, including benchmark execution time and transaction abort rate; the performance impact of using more refined Bloom filter configurations is demonstrated. In the following section, we validate the choices of "best implementation," including use of SIMD, and BoB prefilter and bin count.

Table 6.1: STAMP benchmark input parameters.

Bench	Input
genome	g32768 s128 n16777216
intruder	a10 l128 n262144 s1
kmeansh	m15 n15 t0.00001 random-n65536-d32-c16
kmeansl	m40 n40 t0.00001 random-n65536-d32-c16
labyrinth	random-x512-y512-z7-n512
ssca2	s20 i1.0 u1.0 l3 p3
vacationh	n2 q90 u98 r1048576 t16777216
vacationl	n4 q60 u90 r1048576 t16777216

6.1 Methodology

Total execution time is measured for each benchmark of the Stanford Transactional Applications for Multi-Processing (STAMP) suite [37], stressing the null-intersection tests under a variety of transaction sizes, data set sizes, and memory access patterns.[1] Table 6.1 lists the input parameters to each benchmark; we distinguish both low- and high-contention inputs to kmeans and vacation, and exclude bayes and yada from study. Yada has known bugs [56] and would often run into segmentation faults or infinite loops, whereas bayes exhibits wildly varying execution times as the convergence to a solution is highly sensitive to transaction schedules [34]—no conclusions could be drawn from the results. The labyrinth benchmark suffices to represent the long transactions and mid to high contention of these two benchmarks.

Benchmarks are run on two system configurations: Section 6.2 and Figure 6.2 use eight threads on an 8-core Intel Xeon E5345,[2] [56] running 32-bit Linux 2.6.32-5-686, and benchmarks compiled with gcc 4.5.3. This hardware supports up to SSE2, so to compare the SIMD optimizations in Figure 6.3, including SSE4.1 instructions, benchmarks are run with four threads on an Intel Core2 quad Q9450 running 32-bit Linux 2.6.32-5-686-

[1] We use the version of STAMP that shipped with RSTMv6 source code.

[2] The RSTMv6 source code unfortunately causes some benchmarks to crash occasionally with segmentation faults on the 8-core Xeon. We discard all runs that crash or return invalid solutions.

bigmem, compiled with gcc 4.4.5. Both system configurations use compiler flags `-O3`, `-mtune=native`, and `-fvariable-expansion-in-unroller`.

All null-intersection tests are evaluated with bit-vector lengths of 256 to 16k bits to observe sensitivity to size. The reported performance measurements for a particular bit-vector length represent the arithmetic mean of ten repeated trials.

6.2 Evaluation of all Null-Intersection Tests

We present performance results for five null-intersection tests. These include the baseline unpartitioned (single-partition) intersection, partitioned intersection, BoB intersection, and two configurations for QoQ using both a partitioned and unpartitioned Bloom filter. The latter QoQ configuration is interesting because, as observed in Section 3.5, it will exhibit the same false abort rate as an unpartitioned intersection, and the operations for filter insertion are the same; this configuration thus isolates the empirical effects of issuing queries as opposed to intersecting a large bit-vector. Partitioned QoQ will share similar filter insertion overhead as partitioned intersection (adding one hash function), and exhibits twice the querying overhead.

The eight pairs of graphs in Figure 6.1 present results for the eight benchmarks, with increasing bit-vector length on the horizontal axis. For each benchmark there is a pair of graphs, with performance on top and aborts on the bottom. For performance we plot execution time, normalized to the best-performing unpartitioned intersection result (k=1) across all bit-vector lengths. Any normalized execution time of less than 1.0 thus indicates the speedup of an alternate null-intersection test relative to the best possible unpartitioned intersection configuration for that benchmark. For aborts we plot the percentage of started transactions that were aborted (or restarted). We would prefer to report the reduction of false aborts, but these cannot be measured directly in software without slowing the runtime system and affecting the level of contention and thus realism.

We compromise by noting that as Bloom filter resolution increases, the abort rate should ideally diminish into the true abort rate.

These results include the best configuration for each null-intersection test: SIMD optimization, numbers of hash functions and partitions, and BoB bin count and prefilter hash function type. We validate these choices in the next Section 6.3. SSE2 accumulator-based intersection is used for bitwise `AND`-based intersections, when Bloom filter partitions are at least 128-bits long—otherwise word-level intersection is used.[3] Serial word-level queries are used for QoQ configurations instead of the data-parallel SSE4.1 method. Filters for partitioned intersection, QoQ, and BoB do not exceed two partitions ($k = 2$), as we found that the overheads of additional hash functions are prohibitively high. We choose a BoB configuration with 8 bins that uses an `XOR`-based prefilter hash function. Note that BoB thus requires one more hash function than partitioned QoQ and intersection configurations.

6.2.1 Results for Bitwise Intersections

The bitwise `AND`-based intersections, including the baseline, show two general trends in execution time. In the first trend, for larger bit-vector lengths, execution time increases almost linearly for genome, intruder, kmeansh, kmeansl, and ssca2 as the bitwise intersection increasingly dominates execution time—likely due to frequent read and commit validations. For genome, the alternate null-intersection tests fail to outperform the baseline in execution time, despite increasingly-large reductions in the abort rate as size increases. Partitioned intersection introduces only slight overhead over the baseline (from 2% to 10%). The ssca2 benchmark spends very little time in transactions, and has very small read- and write-sets, and thus is insensitive to Bloom filter configuration. The alternate configurations reduce the abort rate below 1%, but show similar time

[3] Word-level intersection mainly applies to BoB intersection which can contain partitions smaller than 128-bits.

behavior to the baseline, although partitioned intersection performs slightly better. For intruder, kmeansh, and kmeansl: (i) either of BoB or partitioned intersection are able to reduce the execution time below the best baseline configuration; (ii) curiously, the abort rate moderately increases with increasing bit-vector length for all `AND`-based intersections. Note that in intruder, BoB intersection reduces the abort rate by up to 10% below the baseline, permitting a time reduction of about 5% under the best baseline configuration.

The second trend is seen for the remaining benchmarks, labyrinth, vacationh, and vacationl: execution time generally decreases or stabilizes with increased bit-vector length, as the time saved in reducing false aborts exceeds the overhead of longer bitwise intersection. The labyrinth benchmark contains long transactions with large address-sets under high contention. Under these conditions, the greatly-reduced abort rates of BoB and partitioned intersection lead to reduced execution time at nearly all bit-vector lengths; e.g., at 512 bits, BoB intersection provides a 25% speedup relative to the baseline of same length. The best configurations of BoB and partitioned intersection reduce execution time by 15% relative to the best baseline. In contrast, for the vacationh and vacationl benchmarks, the hashing overheads of BoB and partitioned intersection outweigh the moderate reductions in their abort rates.[4] BoB intersection shows a consistently reduced abort rate, whereas partitioned intersection actually increases the abort rate for large filters.

6.2.2 Results for Queue-of-Queries

We focus particularly on the partitioned QoQ configuration, as we would expect it to reduce the abort rate to be lower than that of even BoB intersection; the unpartitioned configuration is included only to bring attention to other empirical effects. At a high

[4]The vacationl and vacationh abort rates are considerably higher in the original unmodified STAMP suite, but are not available for RSTMv6.

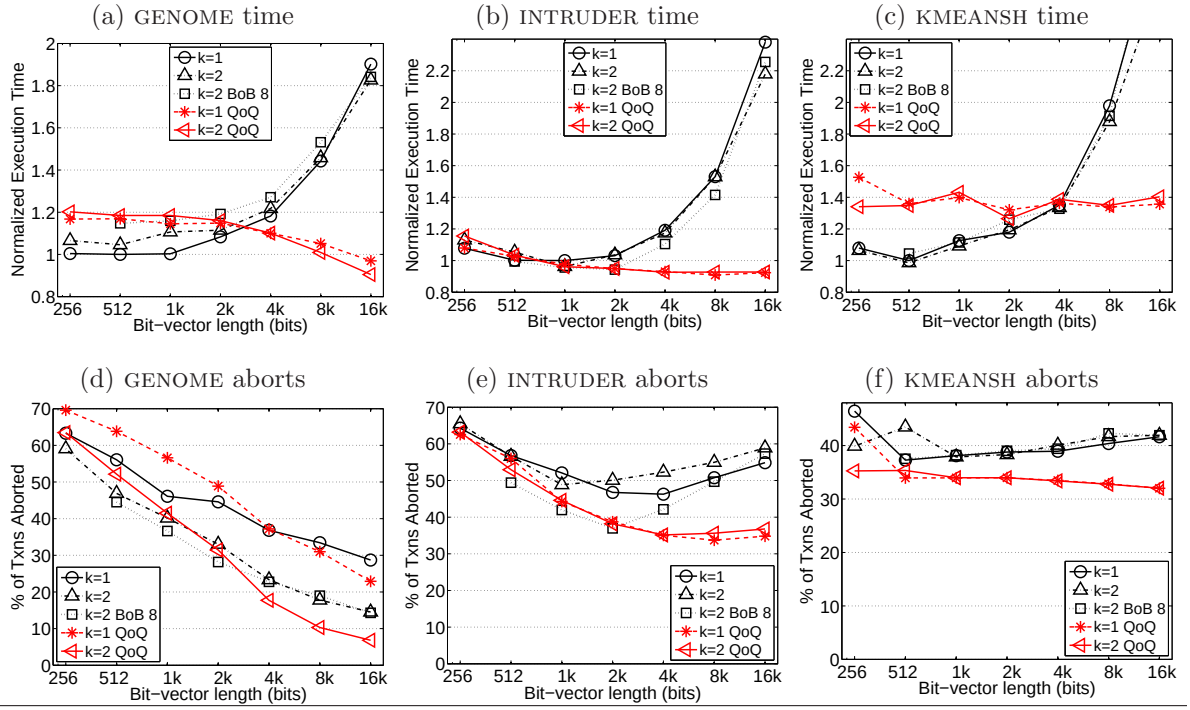


Figure 6.1: Eight pairs of graphs present STAMP execution time and transaction aborts for all null-intersection tests, with increasing bit-vector length on the horizontal axis. The top of each pair shows execution time, normalized to the baseline ($k = 1$) intersection data point with lowest execution time of all bit-vector lengths; the bottom shows the percentage of started transactions that were aborted or restarted. In both cases, a lower value is better.

level, we notice that the execution time when using either of the QoQ configurations scales better with increasing bit-vector length than any of the bitwise intersections—the intersection time is linear in the relatively small write-set cardinalities, rather than linear in the potentially large bit-vector lengths. In many benchmarks, this good scaling leads to partitioned QoQ outperforming even the best configuration of the baseline with up to a 21% speedup.

For GENOME, at small filter lengths (below 2k bits) both QoQ approaches show significant slowdown (up to 20%) relative to the baseline, but at 16k bits, partitioned QoQ yields a 10% speedup relative to the 256 bit baseline. At the small lengths, abort

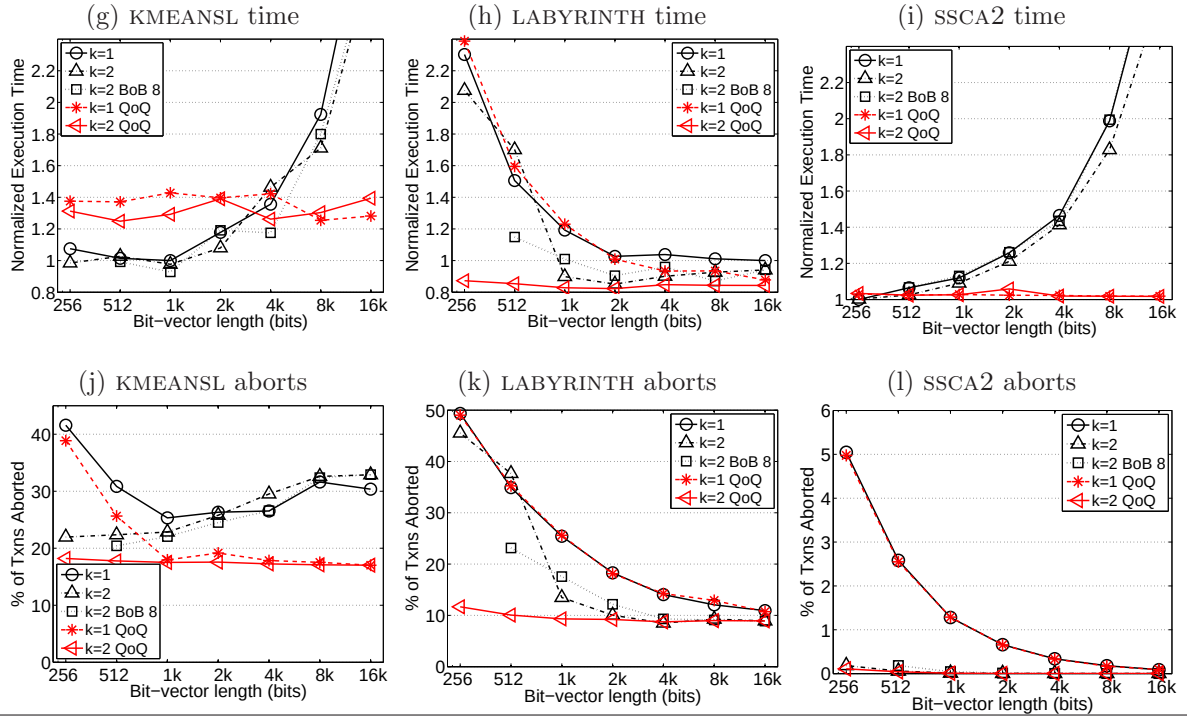


Figure 6.1: Eight pairs of graphs present STAMP execution time and transaction aborts for all null-intersection tests, with increasing bit-vector length on the horizontal axis. The top of each pair shows execution time, normalized to the baseline ($k = 1$) intersection data point with lowest execution time of all bit-vector lengths; the bottom shows the percentage of started transactions that were aborted or restarted. In both cases, a lower value is better.

rates are very high as read-sets are very large;[5] since unpartitioned QoQ performs nearly as badly as partitioned, the slowdown must indicate that the overhead of issuing multiple queries is higher than intersecting small filters. As the filter length increases to 4k and beyond, all abort rates drop, but partitioned QoQ shows the most reduction, down to less than 10%, and this strategy thus overcomes the overhead of issuing queries.

For INTRUDER, both configurations of QoQ exhibit speedup relative to the best baseline point for filter lengths of 1k bits or longer. Curiously, the two configurations equally reduce the abort rate relative to the bitwise intersections. Recall that unpartitioned QoQ and intersection should theoretically induce equivalent abort rates. It appears that

[5]Recall that QoQ would show fewer false aborts if the larger read-set were queried into the Bloom filter of the smaller write-set.

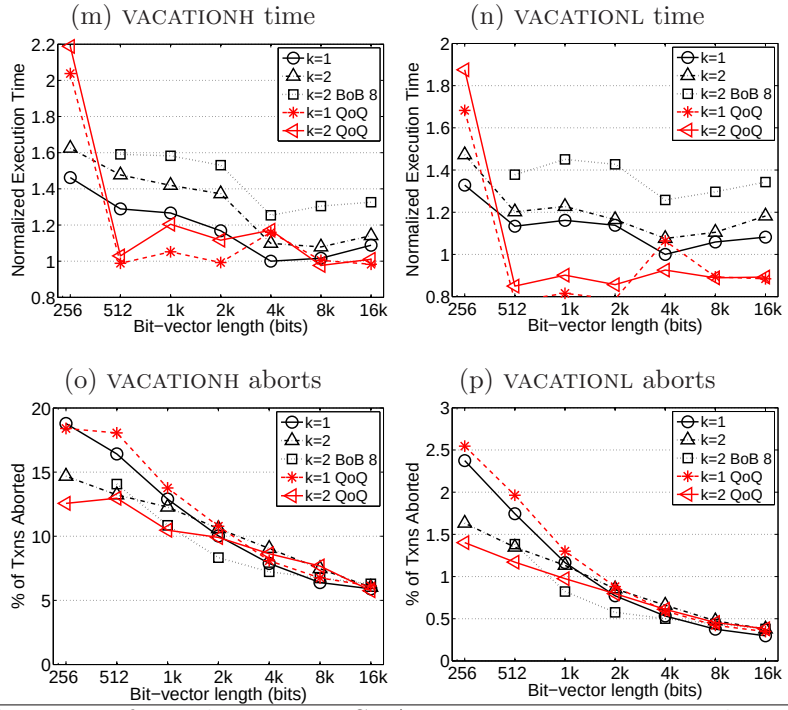


Figure 6.1: Eight pairs of graphs present STAMP execution time and transaction aborts for all null-intersection tests, with increasing bit-vector length on the horizontal axis. The top of each pair shows execution time, normalized to the baseline ($k = 1$) intersection data point with lowest execution time of all bit-vector lengths; the bottom shows the percentage of started transactions that were aborted or restarted. In both cases, a lower value is better.

partitioning does not have a positive effect on reducing aborts in this benchmark, and some other application/system characteristics must be forcing more aborts in the bitwise intersection schemes.

The QoQ approach performs poorly for KMEANSH and KMEANSL, with 25-40% slowdown relative to the best baseline configuration. Similar to INTRUDER, since unpartitioned QoQ shows nearly equivalent execution time as partitioned QoQ (with some variances), this slowdown must be attributed to querying multiple addresses as opposed to hashing overhead. Notice that partitioned QoQ yields the lowest abort rate of all methods, but that partitioning has minimal effect, as the unpartitioned flavor quickly catches up; this benchmark has little room for reducing false aborts.

As stated earlier, SSCA2 spends very little time in transactions, and the address-

sets are small, so there is little room for reducing aborts. The QoQ approaches show equivalent, constant overhead of less than 3% over the best baseline, caused mainly by querying instead of bitwise intersecting. Unpartitioned QoQ and intersection show identical abort rate curves, and partitioned QoQ has the lowest abort rate (nearly zero).

The execution time and abort rate of labyrinth perfectly demonstrate the intended benefit of the partitioned QoQ approach, with a consistent speedup of 14-21% over the best baseline filter length. The benchmark has long transactions with contentious, large address-sets. The execution time and abort rate using QoQ are the lowest of all approaches, and at 256 bits, the abort rate is reduced by nearly 40% relative to the baseline.

With the vacationh and vacationl benchmarks, QoQ shows excellent execution time for different reasons than in labyrinth. Notice that the abort rate for partitioned QoQ is not particularly lower than the other approaches, other than for small filters. Since both QoQ configurations perform well, the reduction in time must instead be a function of query-based intersection. In vacationh, both QoQ outperform all bitwise intersections except at 256 and 4k bits, where caching effects degrade performance. For vacationl, partitioned and unpartitioned QoQ reduce execution time relative to the best baseline by 15% and more than 20%, respectively.

6.2.3 Summary

We find that the QoQ approach is generally the best, aligning with theory but contrary to intuition about implementation overheads. The overheads for QoQ could even be further reduced when future SIMD instructions (AVX2 [25]) include memory gather and vector shift by vector values. We observe that partitioned intersection tends to yield execution times that are slightly worse or slightly better than the baseline, and typically reduces the number of aborts. Finally, we find that the performance impact of BoB intersection can be summarized in a similar way, but in many cases provides even lower abort rates, but

at the cost of higher overhead due to the additional hashing. Hence the BoB approach is likely best suited to hardware, where the hashing overheads can be better parallelized.

6.3 Configuring Null-Intersection Tests

In the previous section, we presented results for the best configuration of each null-intersection test, including the BoB prefilter hash function and number of bins, and choice of SIMD instructions. In this section we empirically validate these decisions.

For all figures in this section, speedups are calculated in the same way as follows. To summarize the behavior of a single configuration option across bit-vector lengths for one benchmark, the bar height is the harmonic mean of average speedups over all bit-vector lengths, from 256 to 4k bits. Each of these average speedups is the ratio (between two competing configurations) of arithmetic means of execution times at one particular bit-vector length. Results for 8k and 16k bits are excluded as those sizes are less common, and also typically slow down execution as seen in Section 6.2.[6]

6.3.1 BoB Prefilter Hash Function

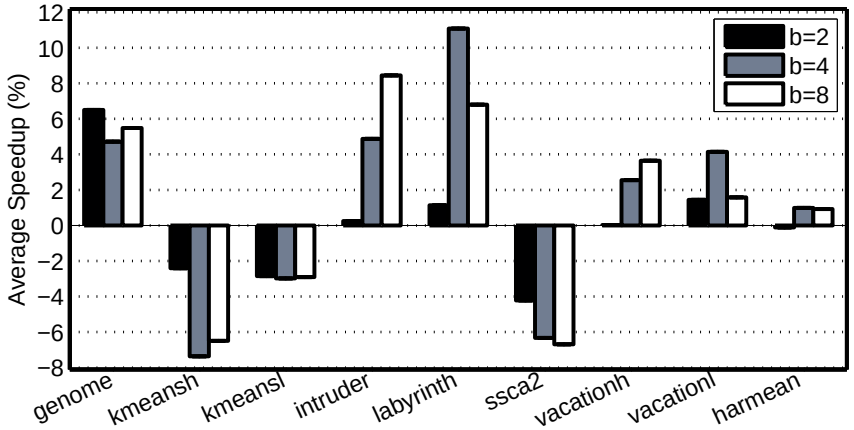
Figure 6.2a shows the speedup for each benchmark, of an `XOR` prefilter relative to H_3 prefilters.[7] Each bar represents the percent speedup for a particular batch bin count (2, 4, or 8 bins). The SSE2 accumulator method of Section 5.5 intersects vectors when individual Bloom filter partitions have length at least 128 bits, otherwise short-circuit word-level comparisons are performed. This SIMD operation is validated later with Figure 6.3.

The H_3 function tends to be cache-bound, with irregularly-indexed table lookups, but is more likely to yield a uniform output mapping. In contrast, the `XOR` function is

[6] Additionally, for small bit-vector lengths and higher BoB bin counts, partitions are less than 32-bits long—hence these bit-vector lengths are also excluded from the mean.

[7] A different H_3 matrix was randomly selected from the family of hash functions for each trial.

(a) Prefilter Hash Functions: XOR relative to H_3



(b) Number of Bins in Batch, relative to two

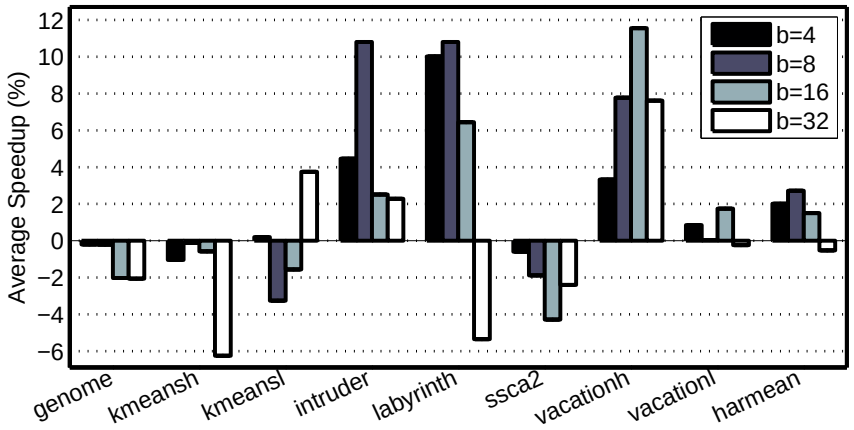


Figure 6.2: Evaluation of BoB configuration options. (a) Average speedup of XOR-based relative to H_3-based prefilter hash functions across the STAMP benchmarks. Each bar indicates the average percent speedup for a particular BoB bin count. The harmonic mean (harmean) speedup is computed across all benchmarks. (b) Performance impact of increasing the bin count for BoB intersection, only using the XOR prefilter hash function. Each bar represents a different bin count b, from 4 to 32. The vertical axis shows average percent speedup of the BoB bin count, relative to a BoB containing two filters.

CPU-bound, consisting of bit manipulations, but does not include as many address bits in the computation.

The suitability of the `XOR` function varies by benchmark, from over 6% slowdowns relative to H_3, for kmeansh and ssca2, to a speedup of at least 6% for the best bin counts in genome, intruder, and labyrinth. For the vacations, the choice of `XOR` vs H_3 is less significant, with speedups under 4%. We choose the `XOR` function as the better of the two options due to its slight harmonic mean speedup (far right), and noting that it performs worse for only those benchmarks with little room for reducing false aborts anyway. Hence the `XOR` prefilter is used in the remaining experiments that measure BoB.

6.3.2 Number of Bins

In Figure 6.2b the number of bins (filters) in a batch are varied from 2 to 32. The average execution speedup with each bin count is shown relative to a BoB containing just two filters. Looking at the harmonic mean speedup of all benchmarks (harmean), BoB intersection appears to benefit when batches contain more than two filters, up to and including sixteen filters. There is no ideal number of bins for all benchmarks: two bins is best for genome, kmeansh, and ssca2, eight bins for intruder and labyrinth, 16 bins for both vacations, and 32 bins for kmeansl. As with all alternate null-intersection tests, two aspects of execution are competing: reducing the rate of false aborts (which shows diminishing returns with increases in the BoB bin count), and implementation inefficiencies—which for increasing bin count involves increased control flow during intersection. We selected the eight-filter BoB as the best configuration as it exhibits the best overall speedup, performs best for labyrinth, and is a good runner-up to b=16 in vacationh.

6.3.3 SIMD Bitwise Intersection

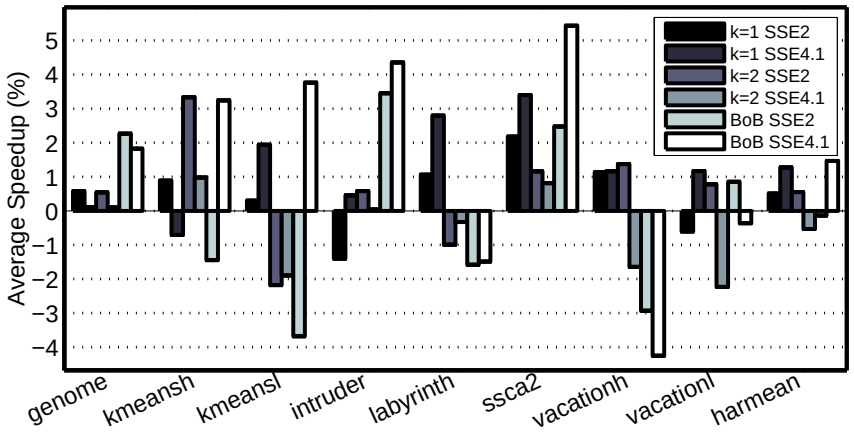
Figure 6.3a presents a performance comparison of the three implementations of `AND`-based intersections that are described in Section 5: (i) the basic 32-bit word comparison, (ii) the SSE2 accumulator method, or (iii) the SSE4.1 short-circuit method. The three strategies are each evaluated using unpartitioned, partitioned, and BoB intersection, labeled k=1, k=2, and BoB, respectively.

Looking at the harmonic mean speedup across benchmarks, it seems that use of SIMD instructions does not consistently reduce execution time: less than 2% speedup, or even 0.5% slowdown relative to word-level comparison. For unpartitioned intersection, SSE4.1 is marginally the best intersection approach, with over 3% speedup in ssca2, down to 1.5% slowdown in intruder. The SSE2 method typically has a positive effect for partitioned intersection. With BoB intersection, SSE4.1 is typically beneficial, with a harmonic mean speedup of 1.5%, up to over 5% speedup for ssca2. The short-circuit SSE4.1 method avoids unnecessary work, but is accompanied by repeated branch instructions. It shows the most promise with BoB intersection, likely because this null-intersection test is already burdened with more control flow, so there is relatively less additional overhead.

6.3.4 Queue of SIMD Queries

Figure 6.3b shows the speedup of using SSE4.1 instructions to parallelize queries for unpartitioned ($k = 1$) and partitioned ($k = 2$) approaches, relative to querying sequentially. The ssca2 benchmark is excluded as the write-set never accumulates four addresses, and we do not parallelize any fewer. For unpartitioned filters (only a single hash function), the SIMD instructions introduce overhead, except in the case of kmeansl, which exhibits a 5% speedup. With two hash functions, the overheads begin to disappear, and some benefit to parallelization is seen, but on average the SIMD operations

(a) SIMD Bitwise Intersection: word-level, SSE2, or SSE4.1



(b) SIMD Queries: word-level or SSE4.1

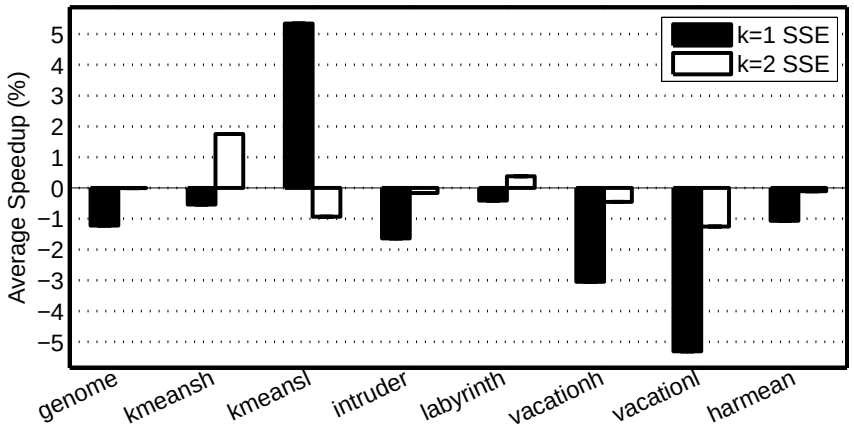


Figure 6.3: Impact of SIMD instructions. (a) Speedup when intersecting 128-bit vectors relative to 32-bit words, for bitwise intersections using SSE2 or SSE4.1. (b) Speedup when issuing four queries in parallel using SSE4.1 instructions, relative to sequential queries.

do not improve performance significantly. This strategy falls victim to Amdahl's law, as the bit-indexing and testing instructions must be serialized. Furthermore, additional overhead is imposed to move 32-bit words into and out of the 128-bit vectors. Bloom filter querying is an application that will certainly benefit from the future Intel AVX2 gather and vector-vector shift instructions.

6.4 Summary

This chapter evaluated the performance impact of our theoretical recommendations, making a compelling case for replacing bitwise intersection approaches in software, with the complex but accurate queue-of-queries approach. We implemented all null-intersection tests in the conflict detection unit of RingSTM, and evaluated performance based on execution time of STAMP benchmarks, augmenting the analysis with transaction abort rates. The best implementation of each null-intersection test was empirically determined, including the use of SIMD instructions, and BoB prefilter and bin count.

Except for the kmeans and ssca2 benchmarks, some bit-vector length of the QoQ approach generally improves execution time relative to the baseline. Our proposed BoB intersection successfully reduced execution time for some benchmarks, and always reduced or maintained the abort rate relative to unpartitioned intersection. Although Bloom filter configuration (and bit-vector length) appear to be application-specific, these results suggest that dynamically-configured Bloom filters will be of great importance in adaptive STM systems [56,64]. The software overhead of the BoB prefilter function often outweighed the elimination of some false aborts, but with its increased accuracy relative to partitioned intersection, our new approach will likely be well-suited to hardware parallelization systems that can better hide the time complexity of additional hash functions.

Chapter 7

Conclusions and Future Work

In many lazy parallelization tools and runtime systems, Bloom filters have gained popularity for the unconventional task of comparing entire address-sets for access overlap, or testing for null-intersection. We identify the three Bloom filter configurations that are in wide use, queue-of-queries (QoQ), and partitioned and unpartitioned intersection, but observe that they have never been theoretically modeled nor conclusively compared. This lack of theory forces designers to perform time-consuming empirical design-space explorations, or to use Bloom filters less efficiently. In this dissertation we thus introduced and analytically compared probabilistic models of false set-overlaps (FSOs) for Bloom filter null-intersection tests. We proved that partitioned intersection is better than unpartitioned intersection, and that QoQ provides a constant factor space savings, relative to partitioned intersection, that is of order square root of set cardinality. To compromise between the accuracy of QoQ and the simplicity of bitwise intersection, we proposed a new null-intersection test that we called batch-of-Bloom-filter (BoB) intersection.

The proven theoretical implications run contrary to practical intuition that refined Bloom filter operations will incur more overhead. We thus implemented the null-intersection tests in RingSTM and evaluated the impact on performance of the accurate

QoQ and BoB approaches relative to the bitwise intersections, showing that in fact, theory and practice can combine successfully to improve performance relative to a baseline single-partitioned intersection. We conclude that designers of future STMs should strongly consider implementing QoQ-based partitioned Bloom filter implementations, and for HTMs, they should consider and compare the BoB approach with alternatives, as it may prove to be superior.

7.1 Contributions

This dissertation makes the following contributions:

1. the probability distributions of false set-overlaps between two address-sets, for each of the three prior Bloom filter null-intersection tests, queue-of-queries (QoQ), partitioned intersection, and unpartitioned intersection;

2. a proof that the partitioned Bloom filter configuration statistically induces fewer false set-overlaps than the unpartitioned configuration;

3. a proof that, for equivalent probability of false set-overlap, partitioned intersection requires a bit-vector length that is larger than that of QoQ by at least a factor of the square root of set cardinality;

4. the design and theory-sketch of Batch-of-Bloom-filters (BoB) intersection, a new Bloom filter configuration for performing null-intersection tests;

5. an empirical demonstration that the BoB strategy is a good statistical compromise between queue-of-queries and partitioned intersection;

6. an observation that, for the special case of single-partitioned Bloom filters, querying and intersection remarkably share an equivalent probability of false set-overlap, and BoB intersection exhibits an equivalent rate of false set-overlap;

7. a performance evaluation using RingSTM of the three alternate Bloom-filter-based null-intersection tests: queue-of-queries (QoQ), and partitioned and BoB intersection;

8. an evaluation of the benefits of accelerating Bloom filter operations with SIMD instructions from SSE2 and SSE4.1.

7.2 Future Work

This work can be extended through a number of avenues, with directions in both theory and implementation.

7.2.1 Theory

The theory of Bloom filters for null-intersection tests can certainly be expanded. Arguably the simplest future contribution is a theorem proving what was observed in the rates of FSO in Section 4.4: that for two sets of differing cardinalities, querying elements from a larger set into the Bloom filter of a smaller set exhibits a lower probability of false set-overlap than vice versa. This fact was shown empirically for a set of fixed parameters (hash functions, set cardinalities), but should be proven generally for all input parameters.

Further contributions are significantly more challenging. In Section 2.1.3, we noted that our probability distributions are derived from the inaccurate "classic" analysis of Bloom filter false positives. The work of Christensen *et al.* [16] should be ported to accurately model the accuracy of Bloom filter intersection (as in Lemma 2), and subsequently substituted into our own probability distributions for null-intersection tests. Building on this idea, a probability of false set-overlap can then be pursued for BoB intersection, as we outlined in Section 4.3, since it typically exhibits far smaller Bloom filter partitions where the classic analysis does not hold. Throughout this dissertation, we

did not consider the effect of more than two in-flight epochs (e.g. one transaction/core). Our proposed models of false set-overlap suffice to provide a comparison of the three null-intersection tests, but for both eager and lazy systems, an accurate system-level epoch conflict rate would be very beneficial, considering the effects of many (> 2) interacting epochs.

7.2.2 Implementation

The merits of QoQ and BoB intersection were demonstrated in a single STM system, but future work should study whether our performance conclusions hold in other STMs or HTMs. Considering software, our work could be ported to the most current RSTMv7 source code to investigate the impact of null-intersection tests in InvalSTM [17]. More interestingly, we would like to see how BoB intersection would perform in the hardware Bulk [61] architecture, and whether QoQ could be ported to this hardware system as it was for SigTM [38]. Alternatively, it would be interesting to convert the QoQ-based conflict detection of SigTM to use BoB intersection, and whether this would permit an unbounded conflict detection strategy in hardware.

Bibliography

[1] Rishi Agarwal and Josep Torrellas. Flexbulk: intelligently forming atomic blocks in blocked-execution multiprocessors to minimize squashes. In *Proceeding of the 38th annual International Symposium on Computer Architecture*, pages 33–44, 2011.

[2] Paulo Srgio Almeida, Carlos Baquero, Nuno Preguia, and David Hutchison. Scalable Bloom filters. *Information Processing Letters*, 101(6):255 – 261, 2007.

[3] Arkaprava Basu, Jayaram Bobba, and Mark D. Hill. Karma: scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, pages 359–368, 2011.

[4] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 640–652, 2011.

[5] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 156–167, 2009.

[6] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Bloom filter guided transaction scheduling. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 75–86, 2011.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[8] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. On the false-positive rate of Bloom filters. *Information Processing Letters*, 108(4):210–213, 2008.

[9] Andrei Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, January 2004.

[10] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979.

[11] Jared Casper, Tayo Oguntebi, Sungpack Hong, Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Hardware acceleration of transactional memory on commodity systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 27–38, 2011.

[12] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 278–289, 2007.

[13] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, 2006.

[14] F. Chang, Wu chang Feng, and Kang Li. Approximate caches for packet classification. In *Joint Conference of the IEEE Computer and Communications Societies*, 2004.

[15] Woojin Choi and J. Draper. Locality-aware adaptive grain signatures for transactional memories. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, 2010.

[16] Ken Christensen, Allen Roginsky, and Miguel Jimeno. A new analysis of the false positive rate of a Bloom filter. *Information Processing Letters*, 110(21):944 – 949, 2010.

[17] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010.

[18] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. The dynamic Bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, 22(1):120–133, 2010.

[19] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, October 1998.

[20] Liang Han, Wei Liu, and James M. Tuck. Speculative parallelization of partial reduction variables. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010.

[21] Fang Hao, Murali Kodialam, and T. V. Lakshman. Building high accuracy Bloom filters using partitioned hashing. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 277–288, 2007.

[22] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300, 1993.

[23] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.

[24] Derek R. Hower, Pablo Montesinos, Luis Ceze, Mark D. Hill, and Josep Torrellas. Two hardware-based approaches for deterministic multiprocessor replay. *Commun. ACM*, 52(6):93–100, June 2009.

[25] Intel Corporation. *Intel Advanced Vector Extensions Programming Reference*, June 2011.

[26] Mark C. Jeffrey and J. Gregory Steffan. Understanding Bloom filter intersection for lazy address-set disambiguation. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, 2011.

[27] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. *Random Struct. Algorithms*, 33(2):187–218, September 2008.

[28] Eric Koskinen and Maurice Herlihy. Dreadlocks: efficient deadlock detection. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 297–303, 2008.

[29] Venkata Krishnan and Josep Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

[30] Martin Labrecque, Mark C. Jeffrey, and J. Gregory Steffan. Application-specific signatures for transactional memory in soft processors. *ACM Trans. Reconfigurable Technol. Syst.*, 4(3):21:1–21:14, August 2011.

[31] Y. Lu, B. Prabhakar, and F. Bonomi. Bloom filters: Design innovations and novel applications. In *Proceedings of the Forty-Third Annual Allerton Conference*, 2005.

[32] B. Lucia, J. Devietti, L. Ceze, and K. Strauss. Atom-aid: Detecting and surviving atomicity violations. *IEEE Micro*, 29(1):73 –83, Jan.-Feb. 2009.

[33] Brandon Lucia, Luis Ceze, and Karin Strauss. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 222–233, 2010.

[34] Austen McDonald. *Architectures for Transactional Memory*. PhD thesis, Stanford University, June 2009.

[35] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2009.

[36] Loizos Michael, Wolfgang Nejdl, Odysseas Papapetrou, and Wolf Siberski. Improving distributed join efficiency with extended Bloom filter operations. In *Proceedings of the 21st International Conference on Advanced Networking and Applications*.

[37] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2008.

[38] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[39] Dragoslav S. Mitrinović and Josip E. Pečarić. Bernoulli's inequality. *Rendiconti del Circolo Matematico di Palermo*, 42(3):317–337, 1993.

[40] Dragoslav S. Mitrinović and Petar M. Vasić. *Analytic Inequalities*. Springer-Verlag, Berlin, 1970.

[41] Michael Mitzenmacher and Salil Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, 2008.

[42] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.

[43] James K. Mullin. Estimating the size of a relational join. *Information Systems*, 18(3):189 – 196, 1993.

[44] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. AtomTracker: A comprehensive approach to atomic region inference and violation detection. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 287–297, 2010.

[45] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. SigRace: signature-based data race detection. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 337–348, 2009.

[46] Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. Cardinality estimation and dynamic length adaptation for Bloom filters. *Distributed and Parallel Databases*, 28(2):119–156, 2010.

[47] Lin Peng, Lun guo Xie, Xiao qiang Zhang, and Xin yan Xie. Conflict detection via adaptive signature for software transactional memory. In *Proceedings of the 2nd International Conference on Computer Engineering and Technology*, 2010.

[48] Lin Peng, Lun-Guo Xie, Xiao-Qiang Zhang, and Xin-Yan Xie. VectorSTM: Software transactional memory without atomic instructions. In *Proceedings of the third international Joint Conference on Computational Science and Optimization*, pages 278–282, 2010.

[49] Gilles Pokam, Cristiano Pereira, Klaus Danne, Rolf Kassa, and Ali-Reza Adl-Tabatabai. Architecting a chunk-based memory race recorder in modern cmps. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[50] Xuehai Qian, Wonsun Ahn, and Josep Torrellas. Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 447–458, 2010.

[51] Ricardo Quislant, Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. Improving signatures by locality exploitation for transactional memory. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[52] Ricardo Quislant, Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. Multiset signatures for transactional memory. In *Proceedings of the International Conference on Supercomputing*, pages 43–52, 2011.

[53] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[54] Kulesh Shanmugasundaram, Hervé Brönnimann, and Nasir Memon. Payload attribution via hierarchical bloom filters. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 31–41, 2004.

[55] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Implementation tradeoffs in the design of flexible transactional memory support. *J. Parallel Distrib. Comput.*, 70(10), October 2010.

[56] Michael F. Spear. Lightweight, robust adaptivity for software transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 273–283, 2010.

[57] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150, 2009.

[58] Michael F. Spear, Maged M. Michael, and Christoph von Praun. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, 2008.

[59] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[60] Sasu Tarkoma, Christian Esteve Rothenburg, and Eemil Lagerspetz. Theory and practice of Bloom filters for distributed systems. *IEEE Communications Surveys and Tutorials*, 14(2), 2012.

[61] Josep Torrellas, Luis Ceze, James Tuck, Calin Cascaval, Pablo Montesinos, Wonsun Ahn, and Milos Prvulovic. The Bulk multicore architecture for improved programmability. *Commun. ACM*, 52(12):58–65, 2009.

[62] Hans Vandierendonck and Koen De Bosschere. Xor-based hash functions. *IEEE Trans. Comput.*, 54(7):800–812, July 2005.

[63] M.M. Waliullah and P. Stenstrom. Efficient management of speculative data in hardware transactional memory systems. In *Proceedings of the international Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, July 2008.

[64] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. Towards applying machine learning to adaptive transactional memory. In *Workshop on Transactional Computing*, 2011.

[65] Shaogang Wang, Dan Wu, Zhengbin Pang, and Xiaodong Yang. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*.

[66] M. Xiao, Y. Dai, and X. Li. Split Bloom Filter. *Acta Electronica Sinica*, 32(2):241–245, 2004.

[67] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[68] Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware techniques to enhance signatures. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[69] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*.