Adaptive Approximate State Storage

A dissertation presented by

Peter C. Dillinger

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts

December, 2010

Acknowledgments

I can only begin to thank all who have helped me in completing this Ph.D.

I especially thank my advisor, Panagiotis (Pete) Manolios, for his patience, insight, encouragement, constructive criticism, and perspective on computer science, working as a scientist, and everything. After my Bachelor's and Master's degrees at Georgia Tech, I decided to continue there for a Ph.D. because of an awesome professor I had recently started working with, Pete Manolios. I was able to transition to Ph.D. work with the confidence that I had an advisor who would help me find the best ways to apply my strengths and who would work with me on my weaknesses. Without Pete, I would not have been able to distinguish the things that only I would care about from the things that would become widely read and respected. For such successes, first thanks goes to Pete.

Although I transferred to Northeastern University to continue working under Pete, I have many people to thank from my time at Georgia Tech. Perhaps at the top of the list is the great teacher and wise computer scientist, Prof. Olin Shivers, because I took three great classes from him while at Georgia Tech and he moved to Northeastern University a little before Pete and I did. I value the time and many interesting interactions I had with Professors Yannis Smaragdakis and Mary Jean Harrold. I thank Professors Spencer Rugaber and Dick Lipton for serving on my qualifying exam committee, and also Santosh Pande, Alex Orso, Jim Xu, and H. Venkateswaran for many useful discussions over the years. Fellow Ph.D. students were always great for bouncing ideas off of and helped me to keep a shred of sanity. They included

Daron Vroon, Sudarshan Srinivasan, Yimin Zhang, G.J. Halfond, Jim Clause, Matt Might, David Fisher (who also moved to Northeastern!), Gayatri Subramanian, Roma Kane, Saswat Anand, David Dagon, David Hilley, and many others.

At Northeastern University, I would like to thank those who welcomed Pete and me into collaborative relationships. Prof. Gene Cooperman and his students in particular were great about welcoming us into a forum to throw around and vet ideas relating to data structures, systems, and enumeration algorithms. On my thesis committee, Gene was active, thoughtful, and always constructive, and thanks to his input, my dissertation is better. After proving my skill in his beautifully-constructed algorithms class, Prof. Jay Aslam agreed to be on my committee, and I thank him for his insightful input and cheerful service. Thanks to other professors and students who sent ideas and feedback my way. Once again, I did not go completely insane, thanks to fellow students including Stevie Strickland (an old friend), David Herman, Dimitris Vardoulakis, Richard Cobbe, David Fisher (again!), Carl Eastlund, Dan Kunkle, Felix Klock, Christine Hang (Chambers), Ben Chambers, and heir to ACL2s, Harsh Raju Chamarthi.

Others to thank include those I worked with during internships, and others who have made important contributions and have been willing to correspond on technical points. I thank Gerard Holzmann for all his work on SPIN, and for the opportunity to work with him and Rajeev Joshi as an intern at NASA/JPL. I thank Willem Visser, Corina Pasareanu, Peter Mehlitz and others for the opportunity to work with them at NASA/Ames. I particularly thank Willem, now a professor at Stellenbosch, for serving on my thesis committee and, specifically, striking down an erroneous claim I tried to make regarding transitive omissions. My thesis builds on important contributions by Gerard Holzmann, Michael Mitzenmacher, Rasmus Pagh, John G. Cleary, and others, but I thank these in particular for taking time to answer my technical queries.

I would also like to thank the National Science Foundation, for funding a grant that supported me during much of my time as a Ph.D. student. For that grant, I was the primary developer of ACL2s, "The ACL2 Sedan," which has helped make computer-aided theorem proving accessible even to college freshmen [1]. Though it took time away from my unrelated thesis work, my work on ACL2s was unquestionably valuable in boosting and expanding my reputation and experience. And it kept the stipend coming. And it gave me a chance to collaborate with some great people at the University of Texas at Austin, most notably Matt Kaufmann and my grand-advisor, J Moore. I thank them for all the ways in which they have helped and supported me over the years. In particular, I thank Matt for co-authoring a paper with me [22] and, thus, lowering my Erdős number to three, and for offering a job reference that seems compelling enough to land me any job anywhere. Ever.

Olin Shivers and Matt Might hold a special place in my heart. According to my recollection, each of them said that by taking a full time job before finishing my dissertation, I would not finish my Ph.D. I assume they were merely giving me the perfect encouragement to finish: an opportunity to prove them wrong. I thank them for that extra motivation. It is likewise apropos that I thank my employer, Coverity, for allowing a leave of absence in which to finish. Proving Matt and Olin wrong—it still counts, right?

I proudly also thank my favorite distraction from Ph.D. work. No, I'm not talking about Portal[2]. In fact, "distraction" is not the right word, because it might be more accurate to say that the Ph.D. work was a distraction while I was waiting to meet this one. I'm talking about Miss Natasha Herman, my fiancée. She has enriched my life immensely and has helped me to become more the person I want to be, and for that I will always thank her.

[1] I would not have believed it if I had not seen it with my own eyes—and my own red pen. I thank the Northeastern College of Computer and Information Science for allowing me to teach the course one semester, and supporting me during teaching.

[2] Registered trademark of the Valve Corporation.

Earlier steps in making this achievement possible are thanks to my high quality public education in Thomasville, Georgia, at the Georgia Governor's Honors Program, and at Georgia Tech. I am grateful to all the teachers who put in an extra effort to challenge an arrogant kid like me.

Final thanks go to those who took the earliest steps in making this achievement possible, my parents Charles and Sissy Dillinger. Their dedication, moral support, and logistical support cannot be overstated. At an early age, they taught me to love knowledge, to love computers and programming, and to love a noble challenge. From that point, they made their best possible move: to let me explore my passion on my own terms and according to my own motivation, but to diligently smooth out any structural or logistical barriers to my learning. Perhaps the best way I know to thank my parents is with the observation, "I am pleased with my accomplishment, proving you should be with yours."

Thesis Statement

In an explicit-state model checker, no knowledge of the reachable state space size is needed for the speed and the possibility of overlooking errors to be near optimal for available memory.

Abstract

Efficiently storing and matching visited states is key to the effectiveness of explicit-state model checkers such as SPIN. Models of concurrent programs often have too many reachable states to enumerate easily in main memory, and an efficient model checker can exhaust main memory in minutes by storing state descriptors exactly. A popular alternative is to over-approximate the set of visited states using a randomized, probabilistic data structure, such as a Bloom filter. Because the approximation is sound and does not slow down the search with revisitation of states, it tends to find errors quickly. Because it is probabilistically complete, the approach can also convincingly demonstrate lack of errors.

In this dissertation, I analyze the approximate state storage problem in unprecedented detail, improve upon standard solutions, and demonstrate a novel approach that solves a configuration dilemma facing users of the standard solutions. Especially with my improvements, the best Bloom filter or hash compaction configuration for a given situation is quite good, but choosing the best configuration depends on a good estimate of the number of reachable states. Such an estimate is usually only available *after* checking a model. I solve this dilemma with an efficient storage scheme that is not tied to a particular estimate, because it is adaptive. Regardless of the number of states encountered at run time, its accuracy is near the information-theoretic optimal. It is also competitively fast, thanks to a novel in-place adaptation algorithm and a favorable access pattern to memory.

Contents

List of Figures

Outline of Contributions

- Chapter 3 – I coin terminology and derive formulas for analyzing the accuracy of over-approximate storage of visited states in explicit-state model checkers.
- Chapter 4 – In a unified way, I derive space lower bounds for both approximate and exact representations of sets of states, regardless of universe size.
- Chapter 6 – I demonstrate and analyze techniques for drastically reducing the hashing requirements and increasing the speed of Bloom filters, which are commonly used for approximate visited set storage.
- Chapter 7 – I describe the nature of Stern/Dill's remarkably compact hash table for approximating a set, including showing how it can be made more accurate if all the elements to be added are known in advance.
- Chapter 8 – I analyze two ways of using an exact set representation to implement an approximation, and show that despite them being indistinguishable in most cases, there are some cases in which one is noticeably better.
- Chapter 9 – Regarding a compact hash table design by Cleary, I describe and analyze variants and improve the encoding of metadata.
- Chapter 10 – I show how the representation of the Cleary table permits fast, in-place adaptation to accommodate more elements (states) with lower accuracy, as a valuable alternative to starting over because of overflow.
- Chapter 11 – Using Cleary tables with adaptation and a Bloom filter with limited hashing, I describe a state storage scheme that adapts to the number of states encountered at runtime, and show that its speed and accuracy are never far from optimal.

CHAPTER 1

Motivation and Scope

The primary motivation for this dissertation is in storing visited states in explicit-state model checking of asynchronous/non-deterministic programs. When searching for errors in the state graph induced by such a program, remembering the set of visited states (vertices) is needed to avoid repeated exploration where there are confluences in the graph. Verifying complex programs can quickly exhaust main memory if storing visited states exactly—a consequence of the *state explosion problem*. Resorting to out-of-core storage or forgetting some visited states usually slows down the rate at which new states are explored. Heuristic compression of states in memory is time consuming and dependent on the program structure.

This dissertation focuses on non-heuristic, potentially over-approximate storage of visited states, based on hashing. Over-approximating the visited set with a structure such as a Bloom filter is popular because, compared to full storage, it is roughly as fast per unique state but can explore orders of magnitude more unique states using a given amount of memory. There is a possibility of overlooking erroneous states, but this can be quite small and has little impact on the ability to find errors in large models quickly. This dissertation extends the state-of-the-art in understanding, evaluating, and choosing over-approximate storage of visited states, and presents a new scheme that is, in an unprecedented way, never a bad choice.

1.1 Verification problems

Explicit-state model checking is most often applied to models of asynchronous software or hardware systems. It can be applied to any non-deterministic program, but asynchrony is probably the most interesting source of nondeterminism. Thus, parallel and distributed systems are common targets for model checking. Model checking is usually only terminating and complete for finite programs—those with a finite state space—but that does not prevent bug-hunting in infinite or intractable state spaces. A more sophisticated approach to verifying a design is to work with models that remove features and/or hide implementation details. Models can be written from a high-level design specification or extracted from an implementation [50, 45]. In either case, the goal is a more tractable state space that "exercises" the features to be verified.

Writing and/or extracting models for verification is beyond the scope of this dissertation. Good resources on the subject include Gerard Holzmann's book [42] and methodology papers in collaboration with others [50, 45]. I focus on how the model is verified once given to an explicit-state model checker.

1.2 Explicit-state vs. alternatives

The relative usefulness of explicit-state model checking to particular problems compared to alternatives is not a topic of this dissertation either, but I briefly offer evidence that it is useful in many cases. SPIN won the ACM Software System Award in 2001 for being "a highly successful and widely used software model-checking system ... applicable to large and highly complex software systems." SPIN has been used to verify phone switch software [49], flood control software [54], and spacecraft software [44]. Murφ has proven valuable in verifying cache coherence protocols [76] and still has an active user community. Java PathFinder and Bogor are used to find errors in con-

current object-oriented programs. All of these tools use explicit-state search.

It is worth mentioning how symbolic model checking methods differ from explicit-state. Explicit-state computes one successor of one state at a time. These can stand for more than themselves using reductions, but each transition is computed on only one state at a time. Symbolic methods consider many states simultaneously by applying the transition relation to a specially-encoded set of states. These special encodings include binary decision diagrams (BDDs) [12] and logical formulae [4]. SAT solvers are used to check for counterexamples to logical formulae. Symbolic methods are more heuristic in that efficient verification depends on the right kind of "regularity" in the program structure.

1.3 State enumeration

This dissertation focuses on explicit-state verification algorithms based primarily on state enumeration, because these are most affected by the performance of the visited set. More sophisticated algorithms, such as nested depth-first search [47], do not visit as many states as quickly due to deliberate re-visitation of states but enable checking of more sophisticated properties. Enumerating all reachable states (vertices) in a program (implicit graph) requires exploring all transitions (edges) using a graph search algorithm, so enumeration allows us to check any properties local to each state or transition, such as invariants and deadlock-freedom.

To find errors quickly and reduce memory requirements, checking properties by state enumeration is done "on-the-fly." Rather than constructing the entire state graph in memory and then checking properties, properties are checked as the graph is explored, and only the set of visited states (vertices) must be stored in memory. Depth-first search is a common search algorithm, but breadth-first search has the advantage that counterexamples are automatically minimized with respect to the number of transitions from a start

state. Because they are accessed linearly, the bulk of the depth-first stack or breadth-first queue can be stored on disk with little impact on speed.

Symmetry and partial-order reductions play crucial roles in reducing the set of states that must be visited to preclude the existence of errors in all reachable states. Symmetry reduction is almost transparent to the rest of the search procedure because all it requires is (possibly heuristic) canonicalization of the order of symmetric state elements before each state is queried against the visited set [15]. This would only have implications for how to represent the set of visited states if that representation were dependent on the structure of state descriptors, and I explain soon that such representations are outside the scope of this dissertation.

Unlike symmetry reductions, partial-order reductions often have a significant impact on the rest of the search algorithm. The most common implementations of partial-order reduction utilize a *cycle proviso* that must determine whether a state is *active*—on the depth-first stack [47] or breadth-first queue [8]. This requires random access to those states, and providing such random access is likely to benefit from being combined with random access to the visited states—since each active state has also been visited. This dissertation considers and addresses this need.

Despite reductions, abstraction, etc., concurrent programs still suffer from the *state explosion problem*, in which the number of states tends to be exponential in the number of features or entities in the program. For more on how the properties to be checked influence the algorithms to check them, see a survey by Valmari [81].

1.4 Out-of-core storage and caching

Verification algorithms that do not represent all visited states in core memory have not been widely adopted, probably because they have not proven competitive in finding errors quickly. When *fast falsification* is a goal, one

should use a technique that is always able to explore an enormous number of unique states quickly. Except for this section, this dissertation focuses only on algorithms that represent all visited states in memory.

Storing some or all visited states on disk inhibits random access, which tends to slow down the search. Matching visited states on disk will typically require numerous scans of visited states and duplicates. By contrast, storing the bulk of the search stack or queue on disk involves writing and reading each visited state no more than once. The difference is especially significant because state descriptors can be hundreds or thousands of bytes and it is possible for a million or more to be computed per second. Using hashing and other tricks can reduce these problems but does not eliminate them [3, 71].

Another approach is to use main memory as just a cache of many of the visited states [32, 30, 71]. In such a scheme, uncached visited states can be revisited, and running times can easily succumb to the deleterious worst-case complexity. Specifically, as the proportion of visited states that are cached gets small, the rate at which truly new states are visited slows greatly.

Both approaches (disk and caching) are useful for "high-assurance verification" of large problems but tend to be poor at "fast falsification." Discrediting these approaches is not important to my thesis, however. The problem of representing *some* visited states in main memory is closely related to the problem of representing *all* visited states in main memory. Thus, development of my thesis is likely to benefit these approaches; in fact, the main technique promoted by my thesis is more easily compatible with caching than many alternatives, because entries can be replaced or even deleted.

1.5 Heuristic storage

Some ways of storing visited states are engineered to take advantage of "regularity" in state descriptors; these are outside the scope of this dissertation.

Usually the set of reachable states [1] in a program is minuscule compared to the set of describable states [2]—assuming the latter is even bounded. Models of software or hardware often have a fixed set of finite variables; full programs often have unbounded allocation potential. In either case, it is common for most bit vectors interpretable as states not to be reachable in the program. Furthermore, the reachable states tend to have some predictable structure and redundancy, within themselves and among other reachable states. This tendency, which makes symbolic model checking effective in many cases, can also be leveraged to reduce memory requirements in full explicit-state verification.

In the simplest kind of heuristic storage, each state is losslessly compressed before being stored in a hash table of visited states. SPIN, for example, does some simple compression by default (disabled with -DNOCOMP) for full/lossless verification. This will usually slow execution somewhat, but it is possible for savings in hashing time to exceed compression time. A more elaborate scheme, *collapse compression*, tracks all values actually taken by pieces of each state and stores only indexes into pools of those values [41]. This is available with -DCOLLAPSE in SPIN, and has proven critical to efficient operation of Java PathFinder, which uses collapse compression for active states.

Perhaps the most elaborate known heuristic state storage is with a minimized/canonical automaton, such as an ordered binary decision diagram (OBDD). Even using explicit state enumeration, the set of visited states can be stored in an OBDD [83]. Holzmann and Puri found that a byte-based automaton tends to perform better than a binary/bit-based one [48]; the byte-based design is available in SPIN with -DMA. Dramatic memory savings are possible but are problem-specific. Speed can be competitive but is more likely to be rather slow.

[1] Reachable states are those that can be encountered in a valid execution of the program.

[2] Describable states simply have an assignment of the correct type to each live variable and memory location.

Elaborate heuristic storage is useful in high-assurance verification, but has weaknesses for fast falsification. Quite simply, better compression requires more time, and more time slows the rate at which new states are explored. Heuristic storage also does not have a good solution to the problem of what to do when memory has been exhausted; the best known automatic lossless compression might still require a sizable amount of memory per state.

1.6 Non-heuristic, potentially over-approximate storage

Non-heuristic, over-approximate storage of visited states is popular for enabling an enormous number of unique states to be visited quickly, and this dissertation analyzes and extends this class of storage techniques. A basic example is to store only a small hash of each state visited. This is non-heuristic because a good hash function will mask any regularity among states. This over-approximates the visited set because there might be states that have not been visited whose hash is the same as one visited, meaning the structure would falsely consider such states as visited. This introduces the possibility of the search skipping over errors, but I describe a metric we can use to choose the storage technique with the smallest such possibility.

Despite the possibility of omitting errors, over-approximate storage is quite useful. First of all, if the search discovers an error—which is common in the development process—it does not matter if the search was inexact in this way. Any error found is a reachable error in the program/model. Secondly, the key to finding errors quickly is exploring as many unique states as possible as quickly as possible. If memory is exhausted by simple exact storage, the previous alternatives (disk storage, throwing away states, or elaborate lossless compression) slow down the rate at which new states are visited. Over-approximate storage can visit a number of unique states on

order with the number of bits of memory available, with no loss of speed. This makes it ideal for fast falsification.

Over-approximate storage can also provide high-assurance verification using much less memory than exact storage. For example, consider a problem with state descriptors of 1000 bytes. A good over-approximate storage scheme would be highly accurate storing just 10 bytes per state. The probability of that scheme omitting *any* states from the search would be around 2^{-50}. The hardware running the algorithm is probably not this reliable, and the memory required is two orders of magnitude smaller than exact storage.

A significant weakness of known techniques is dynamic flexibility: the ability to adapt to the number of states encountered at run time. Using a scheme that is good at fast falsification is far from optimal if the state space is small enough for a more accurate search. Using a scheme that is good at high-assurance verification requires good knowledge of the state space size. The problem is that the user knows little about the (reachable) state space size until a model is searched. In this dissertation, I examine these shortcomings in detail and describe an alternative that offers very competitive fast falsification and high-assurance verification no matter how big the state space is. That same technique can also store states exactly when enough memory is available. This offers the psychological benefit of knowing no approximations were used. This dissertation only examines exact storage in the case of fixed-size state descriptors, because (as explained in this dissertation) the problem of storing state descriptors of unbounded, variable size has fundamental differences that make it incompatible with our notion of optimality.

1.7 Hash functions

A key aspect of any non-heuristic storage is the hash function, but others' work allows hashing to be largely factored out of this dissertation. I be-

lieve the problem of quickly computing effectively random hash values of states is well solved, most notably by Bob Jenkins [51, 52]. In many years of testing the probabilistic behavior of model checkers using Jenkins' hash functions, any deviation from the expectation for a truly random hash function has been negligible, even when extracting more information from the hash function than the designer intended. Jenkins' hash functions are also among the fastest available—much faster than MD5 for example. Jenkins' functions, however, make no attempt to be cryptographically secure, so it might be easy for a well-informed adversary to manipulate its output.

For problems with especially large state descriptors, heuristically fast hash functions can be faster than Jenkins [25]. These hash functions take advantage of the tendency of nearby states to share a lot of common data in their descriptors, by computing hash values incrementally/differentially based on what parts have changed from the previous state. Such a scheme was introduced by Cousin and Hélary in Murφ [18], which I implemented in 3SPIN, a modified version of SPIN. Even when using a heuristically fast hash function, the compactness and accuracy of over-approximations such as Bloom filters are non-heuristic.

In this dissertation, I use Jenkins' hash functions almost exclusively, and when important, I validate that they behave like random hash functions. This will allow me to argue that regular structure in state descriptors is not affecting and will not affect the compactness and accuracy of non-heuristic storage.

CHAPTER 2

Overview of Dissertation

This dissertation bounds the possible accuracy of non-heuristic data structures that represent or over-approximate a set of states, shows how the best known structures are likely to fall well short of this ideal the first time verification of a program is attempted, and describes a new scheme that is never too far from this ideal. My new scheme uses known structures, Cleary's compact hash tables and Bloom filters, but adds the ability to adapt efficiently to the number of states encountered at run time. I make other contributions to the understanding, analysis, and implementation of these and other important solutions along the way.

2.1 Understanding the problem

To be used as a visited set, a set data structure only needs to support ADD and QUERY operations. As discussed, I am not considering structures that might "throw away" elements, by returning negative to a QUERY of something already ADDed. I am considering structures that might give false positive queries, as a result of over-approximation.

The number of *omissions* from a search is the number of states not fully explored because of false positive queries from the visited list. These can either be *hash omissions*, which are those states that were erroneously considered already visited by the visited set, or *transitive omissions*, which are

states never queried against the visited set because they were made unreachable due to other omissions. In some tools, only transitive omissions lead to the possibility of overlooking errors.

In any case, the key to maximizing the accuracy (error-finding ability) of a search is minimizing expected hash omissions. The relationship between transitive omissions and hash omissions varies from problem to problem. We call this the *transitive omission problem*. There is an intuitive and empirical correlation of transitive and hash omissions, but the only simple hard truth of the relationship is that a search with zero hash omissions has zero transitive omissions. Because none of the evidence suggests a better approach, the best known approach to minimizing total omissions is minimizing hash omissions.

Since the number of hash omissions for a given setup is a random variable, the "best" setup is the one with the smallest expected value for hash omissions. This will be our metric for inaccuracy. In the inverse, it measures accuracy. When polarity is not important, I will simply call this our "accuracy metric."

Accuracy metrics others have used cannot be predicted reliably or have limited applicability. It would be nice to have one number which is the probability of finding an assumed single error in the state space, but it is not known how to determine such in the presence of the transitive omission problem. *Coverage*, the proportion of reachable states actually visited, would seem to offer that, but it cannot be predicted reliably also because of transitive omissions. Another common metric is the *probability of any omissions*, which is fine when the search is expected to be highly accurate, but is not helpful for comparing less accurate searches. Note that the expected hash omissions roughly equals the probability of any omissions when closer to zero than to one. Unlike these other metrics, expected hash omissions is measurable and informative for all accuracies.

Computing expected hash omissions after a search can be precise, but

predicting the accuracy of a search is more complicated. After a search, we know how many states were recognized as unique, and we should be able to compute the false positive rate after each such addition. This is sufficient information to compute the *post facto* expected hash omissions. To predict the expected hash omissions *a priori*, we would need to know how many states will be seen, which requires knowledge of the state space size and expected transitive omissions. This information is usually not available. My formula predicting hash omissions does, however, allow us to examine the space of expected accuracies based on hypothetical numbers of states seen.

I take a different approach than others have in choosing the best structure when the state space size is unknown. Essentially the only other work examining this case is Holzmann's work on bitstate hashing (e.g. [40]). Holzmann focuses on optimizing the accuracy of the worst case: when the state space size is so large that we would not be able to explore more states in the same amount of memory even if the state space were larger (assuming non-heuristic, over-approximate techniques of course). In effect, this leaves a wide range of cases with much worse accuracy than is possible. In that sense, his solution is specialized for the case of extremely tight memory. I believe a truly general solution should be judged by the "farthest" from optimal its expected accuracy can be, among all possible numbers of states that might be encountered at run time while remaining in a fixed memory space. This requires notions of optimality and distance from optimal.

Space lower-bounds for the non-heuristic, potentially over-approximate visited set problem give us a notion of optimal compactness. Except in restricted cases, classical solutions are asymptotically far from optimal. The lower bounds point to two important aspects of solutions that are asymptotically near optimal. First, information must be encoded in the location of data within the structure, meaning any solution in which each full descriptor appears in memory is not asymptotically compact. Second, pointers cannot be used, because in that case the metadata could asymptotically dominate

the payload data. I have found references for most of the space bounds I derive for this problem (e.g. [13]), but I put them into a more unified framework.

Distance from optimality should be based on memory required, not directly on expected accuracy. Motivating this choice is the challenge of comparing perfect accuracy to imperfect accuracy. Conceptually, the "distance" between 0 expected hash omissions and 0.01 expected hash omissions is much greater than between 0.01 and 0.02. 0.02 is about twice as likely to omit any states as 0.01, but 0.01 is infinitely times as likely as 0. I believe the right way to bridge this divide is instead to examine the theoretical minimum memory required to achieve the expected accuracy and compare that to the memory used/available. I call the ratio of these two *competitive accuracy* and use this as my notion of how "close to optimal" a solution is for each number of states.

Between "flexible" structures, which excel in worst case competitive accuracy for an unknown number of states, and "inflexible" structures, which excel in best case competitive accuracy for a known number of states, is a spectrum of solutions that vary in what I call *dynamic flexibility*. This suggests that if an estimate of the reachable state space size is available, the best choice might depend on how precise we believe that estimate to be. I give a formal definition of *dynamic flexibility* as a strict partial order on data structure configurations, based on one yielding acceptable competitive accuracy in a superset of the cases of the other. Different configurations of the same structure are typically incomparable in this way because configuration usually shifts around the range of acceptability rather than narrowing or widening it. Different structures, however, often have inherently different dynamic flexibilities. I show that my adaptive Cleary tables are more flexible than Bloom filters (bitstate hashing), which are more flexible than compacted hash tables (hash compaction).

2.2 Bloom filters (bitstate hashing)

As a visited set, the Bloom filter [5] is unmatched in enabling fast exploration of most of an enormous state space. In explicit-state model checking, the approach was popularized by Gerard Holzmann under the names "supertrace" [37] and "bitstate hashing" [40]. The idea is that the visited set memory is treated as a bit vector, each state gets hashed to some preset number (k) of indices, and the bit at each index gets set to "1" when that state is visited. If any of the bits at indices associated with a state are still "0", it has not been visited, but some states that have not been visited will have all of their bits set to "1" from other additions. Thus, the structure is an over-approximation of the visited states.

2.2.1 Optimization

In this dissertation, I show how to choose k to minimize the expected hash omissions, assuming the number of states to be encountered is known. Others have shown how to choose k to minimize the final false positive rate, which is more appropriate in applications that do not use the Bloom filter as a visited set. I observe that optimizing according to the less appropriate criteria can result in up to 32% higher expected hash omissions.

2.2.2 Speed

I also describe a Bloom filter construction that virtually eliminates the cost of computing more hash values for k larger than two or three. Before our work on the subject [24, 23], it was widely believed that hash computation had to be linear in k. Since hash computation is a dominant cost of explicit-state model checking, users avoided Bloom filters with k greater than two or three even if it would give better accuracy per run [40].

One can compute two or three indices using independent hash functions and derive the rest using simple, fast arithmetic on these. Double hashing is

such an algorithm, but that scheme has weaknesses when applied to Bloom filters. I describe, analyze, and quantify the accuracy problems associated with double hashing and describe two main alternatives. My enhanced double hashing scheme for Bloom filters corrects the issues with double hashing by making near-optimal use of two indices of hash information. My triple hashing scheme technically has the same problems as double hashing, but is much closer to the accuracy of a standard Bloom filter simply by using three indices worth of hash information, which is often more costly to compute.

A critical aspect of such schemes is the limited independence/entropy among the computed indices, and I analyze the impact of that by analyzing what I call the *fingerprinting Bloom filter*. My analysis shows that the amount of entropy in the computed indices can be reduced greatly, to reduce hash computation, with negligible impact on accuracy. In fact, testing shows that the accuracy of the enhanced double hashing scheme is predicted rather precisely by the fingerprinting Bloom filter analysis.

Even though the fingerprinting Bloom filter seems to be the ideal for working with limited hash information, naive approaches using simple arithmetic can have noticeably *better* accuracy in some specific cases. I explain this with better models for those kinds of Bloom filters, namely the *hash-extending* and *hash-reusing* Bloom filters, with accompanying analysis. The hash-reusing Bloom filter is actually the kind of Bloom filter utilized by my adaptive storage scheme, so its analysis is useful there.

2.2.3 $k = 3$ usefulness

From the standpoint of coverage (proportion of states visited), a Bloom filter setting three bits per state is usually close to best. Notice how the $k = 3$ Bloom filter in Figure 2.1 rarely omits a large portion of its state space—say more than 10%. This is the default for SPIN's BITSTATE mode. Using more than three can cause the Bloom filter to become excessively saturated in the

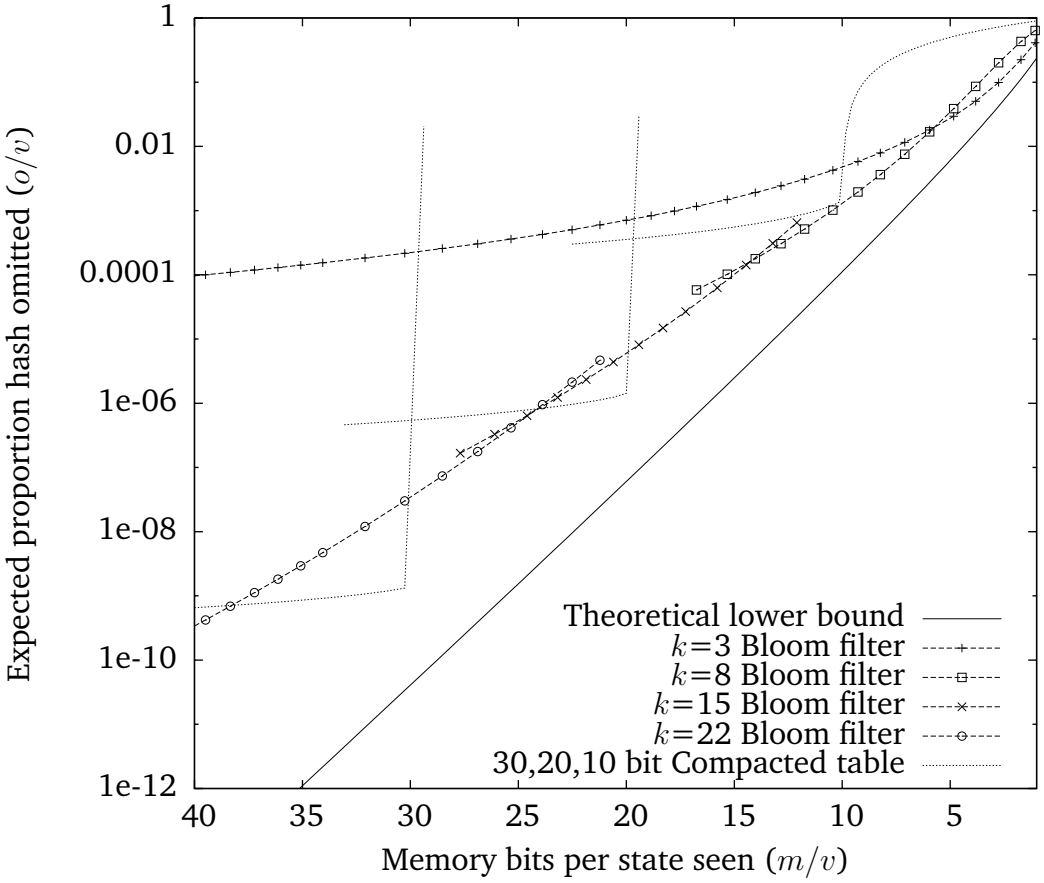

Figure 2.1: *Comparison of inaccuracy of Bloom filter and compacted table configurations with the theoretical optimal. Lower is better. This essentially shows the expected hash omissions for various numbers of states seen, using various data structure configurations, but the axes are chosen to be independent of the actual magnitude of memory or states. Looking instead at the proportion of states expected to be hash omissions vs. the ratio of memory to states seen, these "asymptotically compact" solutions quickly converge to these results. "Theoretical lower bound" assumes states are taken from a universe large enough not to permit exact storage near the domain of this graph. "$k = 3$ Bloom filter" shows the accuracy of the standard bitstate approach over the whole span. Other results are truncated to the area around where they are best. $k = 22$ is the best Bloom filter configuration when 30 bits is the best compacted table configuration, $k = 15$ for 20 bits, and $k = 8$ for 10 bits. The Y axis spans twelve orders of magnitude. The X spans about two orders of magnitude.*

case of many states. Using fewer can starve the search before reaching the number of states the Bloom filter is optimized for. I call using a $k = 3$ Bloom filter the "standard bitstate approach," because it is hard to beat in terms of fast falsification—finding errors quickly.

A problem with this notion of "close to best" is that 99% coverage is "close to" 99.99% coverage. In terms of finding errors, 99% is about 99% as effective as 99.99%. In terms of demonstrating error-freedom, 99% is about a hundred times more likely to overlook an error than 99.99%. This is roughly the difference between using a $k = 3$ Bloom filter and 16-bit hash compaction when there are about 16 bits per state in the visited set (assuming some transitive omissions). The difference is more profound if more memory is available per state; at 32 bits, for example, the difference in expected omissions is a factor of $100\ 000$, even though the $k = 3$ Bloom filter would usually cover more than 99.9% of the state space.

The most important reason for recognizing accuracy differences that tend to be minute in terms of coverage is that, in practice, we do not know the actual coverage of a search. If we knew that every $k = 3$ bitstate search that terminated having visited one state for every 16 bits of the Bloom filter had coverage greater than 99% (with high probability), then one should be rather convinced of error-freedom after a few runs of that. Without a solution to the transitive omission problem, however, we cannot be sure of that. Less than 1% hash omissions among encountered states may have led to omission of 90% of the reachable state space, and we do not know the likelihood of such a case for a given model, especially in the presence of reductions (partial order and/or symmetry). We *do* know that reducing the 1% hash omissions to 0.01% will (in expectation) reduce total omissions and increase coverage.

2.2.4 Usefulness of other configurations

When a rough estimate of the reachable state space size is available, a Bloom filter optimized for that estimate is a reasonable choice, because of its reasonably good dynamic flexibility. Notice how the $k = 22$, $k = 15$, and $k = 8$ curves in Figure 2.1 (left to right, each truncated) are each moderately close to optimal over a wide range. If the estimate is off by a sizable factor, though, the best configuration is much better.

It is also the best known practical choice when the hash factor (visited set memory per visited state) is less than about 12 bits per state. Notice how the 10-bit compacted hash table in Figure 2.1 is never better than the $k = 8$ Bloom filter. If the estimate is very good and the hash factor is expected to be greater than 12 bits per state, then the accuracy of a compacted hash table is likely to be higher.

2.3 Compacted hash tables (hash compaction)

In contrast to their low dynamic flexibility, compacted hash tables have remarkable peak competitive accuracy, thanks to a clever way of probabilistically encoding information into the location of values in the table. These hash tables are like typical open-addressed hash tables except that information used to build them is lost while building them, but only a probabilistically small amount of that information. In particular, the starting probe location is computed independently of the value to store; thus, the starting probe location of a stored value cannot be determined definitively just by looking at the table. It can, however, be narrowed down to a limited number of possibilities that would not have placed that value elsewhere first. Though that line of reasoning is complicated, the implementation of Stern & Dill's "hash compaction" scheme [77, 78] is almost indistinguishable from an open-addressed hash table using double hashing [34] and (preferably) ordered hashing [2].

There are some subtleties in combining double hashing, ordered hashing, and independent computation of starting location that I believe I describe more clearly than others have. In particular, the double hashing is noticeably less effective than random probing in this case because we have to make the distance between probe locations a function of the value stored. If we do not, we cannot relocate stored values and ordered hashing is essentially precluded. An exception is knowing in advance all the elements that will be added; I show that in that case, a static compacted hash table with a lower false positive rate can be constructed. Despite the probing issue, there is a significant accuracy benefit to using ordered hashing in the dynamic structure.

I show that Stern/Dill hash compaction schemes are asymptotically near optimal if they are filled only to some constant occupancy less than 100%. Experiments show that filling to between 98% and 99.8% is most effective, meaning the size of each cell should be slightly smaller than available memory divided by number of cells to be occupied (if known) to maximize accuracy. Even when filling up, the expected, amortized speed of a compacted hash table as a visited set is quite good. As in Figure 2.1, the structure is closest to optimal just before it fills up and the omissions spike.

The structure is dynamically inflexible because (a) it is not that competitively accurate when far from full and (b) it is practically useless once it is full. This is evident in Figure 2.1 because each compacted hash table is only better than the competing Bloom filter in a small range of cases, if at all. After the structure fills up, we assume that all remaining states are omitted, because the only possible alternative is to eject existing states, which is outside the scope of this dissertation because it can drastically slow the search.

Nevertheless, if one has a close estimate of the state space size and there are more than about 12 bits of memory per state available, a compacted hash table is likely the best choice because of its peak competitive accuracy.

At 30 bits per state, its expected hash omissions are more than ten times smaller than the best Bloom filter.

2.4 Adaptive Cleary tables

The central practical contribution of this dissertation is a uniquely capable solution to the visited set problem when the state space size is completely unknown. I have extended a compact hash table by John G. Cleary [14] with an algorithm that adapts it quickly and in-place to accommodate more elements, by throwing away information about each visited state. The result is the only known design that actually has a worst-case competitive accuracy (among all practical cases). I describe two variants: a "fast" variant that can be made to stay within a memory factor of 2.5 from optimal, and an "accurate" variant that is within a factor of two.

Because of its favorable access pattern to memory, the "fast" variant is faster than popular alternatives in many cases, even after adapting the entire visited set several times. Any access to a Cleary table requires only one random access to memory; the rest is local to that area of memory. This imparts a significant advantage when the structure occupies most of main memory and several processor cores are contending for access to main memory.

2.4.1 Exact reduction

A Cleary table is fundamentally an exact representation for a subset of some finite universe, but it is simple to use it to over-approximate a set from a larger universe. It is just a matter of storing a set of hashes. Others have established that this general approach to over-approximating a set can be asymptotically near optimal, assuming the underlying exact set is near optimal and the elements are drawn from an infinite or practically infinite universe [13, 65]. Over-approximating a subset of a finite universe is more nuanced, and I examine closely two reductions that arise naturally to solve

that case. One approach is observably more accurate in some cases and provably at least as accurate in almost all practical cases. In fact, the more accurate reduction is needed for my adaptive storage scheme to meet the claimed accuracy bound, and only because of some very rare cases. Both approaches are, however, within a constant number of bits per added element of optimal for the general over-approximation problem, which makes them "asymptotically compact" for my purposes.

2.4.2 Cleary tables

Cleary tables are similar to compacted hash tables except that metadata and the arrangement of entries enable the starting probe location, the *home address*, of each to be determined. This means that the descriptor of each added element can be reconstructed by combining the stored part with its home address. Cleary's inception of the tables implicitly required three bits of metadata per cell to match entries with their home addresses, but I show how to reduce that to two bits. When using the table as an over-approximation, my one bit of savings in metadata enables one more bit of hash data to be stored for each element, which cuts the expected hash omissions in half.

Cleary tables use a special bidirectional linear probing that keeps all elements in the table in unsigned numerical order of their descriptors. This has several implications. First, look-ups can quickly degrade to linear search if the descriptors are not uniformly distributed. For over-approximate storage, hashes are already uniformly distributed. For exact storage, descriptors can be passed through a randomization function (1:1 hash function). Even with uniform descriptors, the structure is subject to clustering; beyond about 90% occupancy, operations are unacceptably slow. An advantage of this design is that only one random access to memory is required per operation; the rest are streaming accesses near that one. Also, the choice of maximum occupancy allows almost limitless trading between space and time.

2.4.3 Adaptation

Because of the exact information it contains and the convenient encoding, the Cleary table has enormous potential flexibility. As-is, the Cleary table has an accuracy curve that makes it similarly as inflexible as the compacted hash table. Unlike the compacted hash table, a Cleary table has all the information needed to construct another configuration that accommodates more elements in the same amount of memory, by storing a smaller hash of each element. Better still, because the values are stored in order, it is possible to convert from one configuration to the other in place, with an elaborated scan through the structure. If we only remove some of the final bits of each hash, the order of elements in the old and new structure are the same.

The practical challenges of implementing such conversions are big. Even the simplest case, cutting the size of each cell in half, is complex. This doubles the number of home addresses, meaning that one bit that was stored for each entry becomes implicit in its new home address. A key to such conversions is a traversal that guarantees all entries between an entry and its home address will be processed before that entry is processed. This means that when we convert each element, its new location is available, because it can only be closer to the home address. The resulting conversion algorithm requires only $O(1)$ auxiliary space and makes no random accesses to memory that are not expected to be cached.

Essentially the same algorithm can convert a Cleary table into a kind of Bloom filter. This is helpful because Bloom filters are more accurate than over-approximate Cleary tables when available memory is below about 9 bits per state. The resulting Bloom filter must have locality among its indices, which diminishes its accuracy somewhat. Conveniently, this type of Bloom filter is a hash-reusing Bloom filter, described above. The hash-reusing analysis is applicable here.

2.4.4 Designs

My "fast" adaptive design depends only on splitting cell sizes in half and converting to a special $k = 2$ Bloom filter. It is at least as accurate as the optimal for 40% as much memory, as shown in Figure 2.2. Basically, we start out using a Cleary table with a power-of-two cell size that allows entire state descriptors to be stored. As each configuration reaches 85% occupancy, we cut its cell size in half. After the 8-bit Cleary table, we convert to a Bloom filter setting two bits in adjacent bytes for each state. This design is consistently faster than the standard bitstate approach before the first adaption and after adapting to the special Bloom filter, and not much slower in between.

My "accurate" adaptive design has an intermediate step between each power-of-two cell size for the Cleary tables. These intermediate configurations pack three entries in the space that was previously taken up by two cells. That hack is sufficient to be at least as accurate as is optimal for 50% as much memory, as shown in Figure 2.2. This design is usually no more than 10% slower than the "fast" design.

In practice, it would require lots of hashing to implement these schemes completely, which would make them slow. Using at most two machine words per cell has observably the same effect with low hashing requirements. The probability of any omissions for a Cleary table using two words per cell (at least 64 bits) is at worst one in billions. To some people, however, the psychological benefit of exact storage is important when possible. Fortunately, we can also store states exactly with low hashing requirements; only the configurations between exact and 64 bits per cell require lots of hashing.

My claims of being always within a certain memory factor of optimal cover all practical cases but are not, in theory, universal. One assumption is that the available memory is not close to being able to represent the universe explicitly with a bit table. Another is that the available memory is not trivial

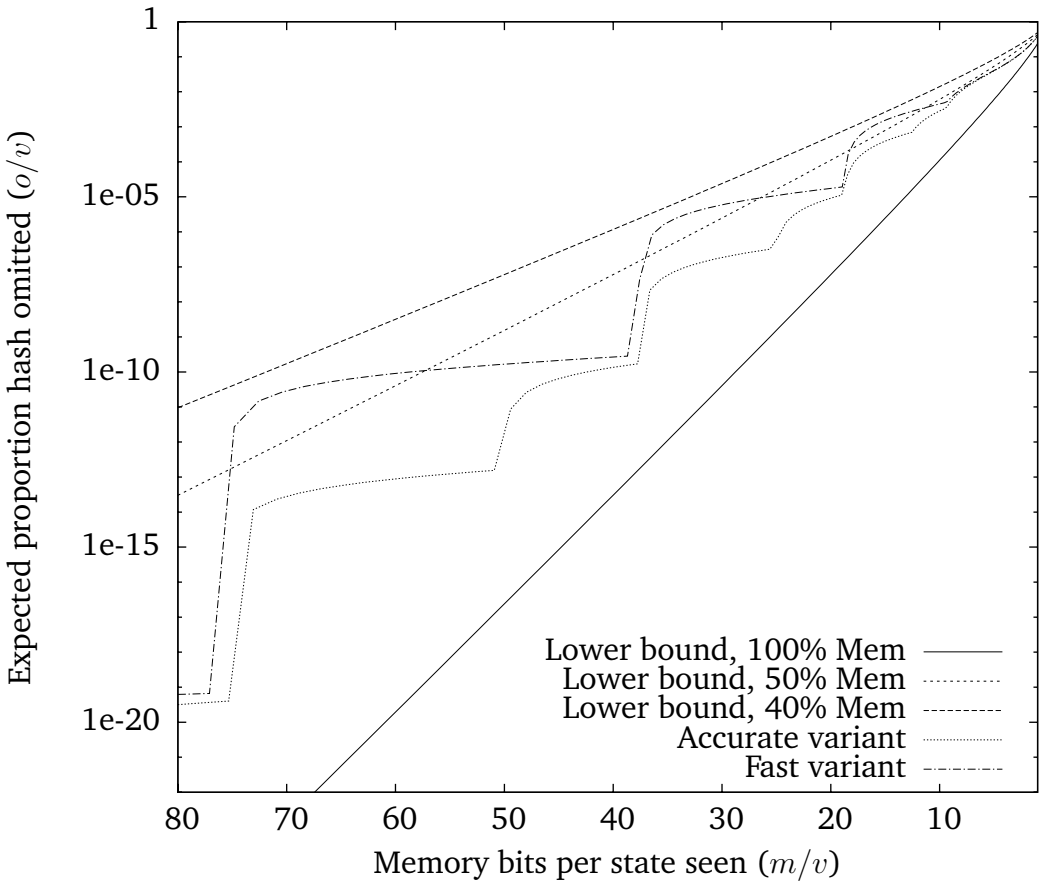

Figure 2.2: *Comparison of inaccuracy of adaptive storage designs with lower bound. Lower is better. This uses the same axes as Figure 2.1, but over greater spans, to compare the expected accuracy of my adaptive visited set designs to the optimal for various memory factors. Expected omissions for my "accurate" design are always below the minimum for 50% as much memory. Expected omissions for my "fast" design are below the minimum for 40% as much memory. It is easy to see where the cell size conversions take place. For example, the "fast" converts from 64 to 32 bits per cell at* $64/0.85 \approx 75$ *bits per state. It later converts to 16, to 8, and then to a special* $k = 2$ *Bloom filter at around 10 bits per state. Zooming in on the tail end would show the designs staying below their claimed bounds.*

in size, at least 8KB. Finally, the number of states reached should be less than the number of bits of memory available. As I describe in detail, these assumptions are very reasonable in practice. For example, Holzmann has repeatedly found that setting two bits per state in a Bloom filter ($k = 2$) is more accurate than setting just one ($k = 1$) even if the model has many more states than there are bits of memory available. This seems counter-intuitive but is a consequence of the search starving itself of new states to explore before it has seen enough states for $k = 1$ to be best.

I validate my claims on the adaptive storage designs as follows: First, using numerical computation, I show that the claims hold in many specific cases, based on the formulas for expected performance (as in Figure 2.2). Second, I empirically validate my implementation against those formulas, in many specific cases. Third, I give mathematical arguments that those specific cases generalize to all cases at least as large.

2.4.5 Active state matching

Cleary tables, and all structures based on cells, are well-suited to tracking which visited states are *active*: those states currently on the DFS stack or BFS queue. This knowledge is required for typical implementations of partial-order reduction. Using a bit of each cell to indicate whether an associated state is active eliminates the need to represent them in a separate structure, which can constitute significant additional costs in time and space. Getting this to work with adaptation, including adaptation into a Bloom filter, has a few complications, but I design around them effectively. Integrated stack matching improves the speed of my adaptive approach compared to the standard bitstate approach, and often improves space utilization/requirements as well.

CHAPTER 3

The State Storage Problem

Here I describe a basic problem that is an abstraction of the over-approximate state storage problem for explicit-state model checking. The basic problem is easier to characterize and analyze, and forms the basis for optimizing solutions to the motivating problem.

Bibliographic Notes and Contributions As my citations throughout indicate, the essential ideas and conclusions of this chapter have previously been identified and reached in Holzmann's analyses of bitstate hashing (Bloom filters) [40, 42]. My main contributions are vocabulary to talk about the problem in precise terms, and the precise analyses they facilitate. For example, I first used the terms "hash omission" and "transitive omission" in a 2004 SPIN Workshop paper [24], and they enable me to write exact formulas for concepts like "*a priori* expected hash omissions."

My treatment of the problem is related to the field of online algorithms, in which input is discovered over time [6, 1]. By only considering non-heuristic approaches to state storage, the particular states are not important, only the number reachable. My notion of "competitive accuracy" is naturally related to competitive analysis of online algorithms.

3.1 Definition

The basic problem is that of a dynamic set representation, parameterized over a universe of elements to draw from, U, and a memory limit, m bits. Here is more detail:

Problem 3.1. *Use up to m bits of memory to represent a subset of U to which one can* ADD *and* QUERY *elements, where*

- QUERY *must return* positive *for all elements of U that have been* ADD*ed.*
- QUERY *preferably returns* negative *for any element of U that has not been* ADD*ed.*
- ADD *and* QUERY *operations should be fast.*

This is an optimization problem because of the "preferably" and "should" aspects. A technically valid solution is to keep track of no information and always return positive from QUERYs. This is not likely to be a very useful solution, but that could depend on the value placed on the various performance dimensions (see below).

Because some elements that have not been added might return positive from a QUERY, the set actually represented by the data structure might be larger than the set of elements added. If the added set is V and the represented set is W, then $V \subseteq W \subseteq U$. Elements in $W \setminus V$ are called **false positives**, because they return positive from a QUERY even though they have not been added.

Note that the problem does not require support for removing elements, as the motivating problem does not.

3.2 Usage patterns

It will be important in analyzing solutions to Problem 3.1 to draw a distinction between two usage patterns:

Definition 3.2. *The **visited set usage pattern** for solutions to Problem 3.1 is followed if all* negative QUERY*s are followed immediately by an* ADD *of the same element, meaning V includes every element ever* QUERY*ed. The **general usage pattern** has no such restriction; elements can be* QUERY*ed without* ADD*ing.*

The visited set usage pattern is important to my thesis, because it calls for different optimization criteria, and the motivating problem (from explicit-state model checking) follows that usage pattern. However, the optimization criteria for the visited set usage pattern depend on those for the general usage pattern.

3.3 Single case performance dimensions

At a particular point in time, we evaluate a particular instance of a data structure solving Problem 3.1 based on these parameters/performance dimensions. (Each is listed with the "higher is better/more difficult" name, possibly with a "lower is better/more difficult" name in parentheses.)

Magnitude refers to how many items have been ADDed to the structure, but this definition is ambiguous. When we need to be precise, we will use one of two distinct interpretations:

Definition 3.3. *Let $v = |V|$ be the number of **unique additions** to the structure, the number of unique items either explicitly* ADD*ed to the structure or that would have been* ADD*ed if not for a false* positive QUERY*. Let n be the number of **affecting additions** to the structure, the number of additions that would have* QUERY*ed to* negative *immediately before being* ADD*ed; thus, the structure was modified so that* QUERY*s of each would return* positive*.*

Compactness (Space) refers inversely to the memory footprint. It could refer to the maximum allowed memory, m bits, or to the amount actually used, which might be smaller. The solutions we look at use virtually all available space immediately.

Universe Size refers to $|U|$, which must be finite or countably infinite. This is a static parameter that mostly serves to determine the compactnesses and magnitudes for which perfect accuracy is achievable. When accuracy is less than perfect, this parameter is often presumed infinite, with negligible effect on information-theoretic optimality. (See Chapter 4.)

Accuracy (Inaccuracy) refers to how closely W approximates V, or how close that approximation has been over the structure's lifetime. By the assumption that $V \subseteq W$, we are only considering cases of over-approximating or exactly representing the set of visited states. For brevity, I often refer to representations that strictly over-approximate the visited set ($V \subset W$) as **approximate storage** or **inexact storage**. The term "inexact storage" also suggests the dichotomy with **exact storage** ($V = W$).

Specifically, we use two related indicators of inaccuracy: we use the **false positive rate** for the general usage pattern and **omissions** for the visited set usage pattern. Detailed definitions of these are in Section 3.6.

In addition to speaking about these accuracy metrics in absolute terms, we are also interested in how close to optimal the accuracy is for a given magnitude, compactness, and universe size. (See Chapter 4 for derivation of relevant lower-bounds.) We run into trouble, however, if we base this on accuracy ratios, because the ratio between perfect accuracy and imperfect is infinite. However, the memory required to achieve any non-zero accuracy is always finite and non-zero. This gives us a useful metric for differences in accuracy: the ratio in memory required to achieve them. We can therefore use memory ratios to gauge a solution's "distance" from the theoretical optimal:

Definition 3.4. *Let m be the memory required by a structure and $\breve{m}$ be the information-theoretic minimum for a structure with the same magnitude, accuracy, and universe size. The **competitive accuracy** is $\breve{m}/m$. **Competitive inaccuracy** is the inverse, $m/\breve{m}$.*

That definition depends on an accuracy metric, such as false positive rate or number of omissions, and a magnitude metric, such as number of affecting additions or unique additions.

Speed (Time) we can talk about several ways: we can talk about the absolute time of a QUERY or ADD operation in a particular implementation on a particular machine, or we can talk about the asymptotic time complexity. In either case, we can consider worst case, mean, median, or sometimes amortized time cost in either a worst case structure, a median structure, or averaged over all structures.

3.4 Single configuration performance

The existence of space lower-bounds for given magnitude and accuracy (see Chapter 4) points to an inherent trade-off between these performance variables, so optimizing one of these based on the other two seems like the best way to get the most competitive accuracy. But this is only sufficient if we know rather precisely what the magnitude will be at run time. In many applications, the number of elements added to the structure cannot be predicted well, and restarting with a different configuration is likely to be costly.

Thus, another way of evaluating a configuration of a set data structure is by its dynamic flexibility, or competitive accuracy over a range of magnitudes. It is difficult to define precisely what it means for one data structure configuration to be more flexible than another, but here I present the most general definition I have been able to come up with, which is only parameterized by the accuracy metric and magnitude metric.

Definition 3.5. *Let $a(A, i)$ be the expected competitive accuracy*[1] *of data structure configuration A after i additions. Let $R(A, q) = \{i : a(A, i) \geq q\}$. $R(A, q)$ is therefore the region in which the data structure is expected to achieve com-*

[1] Because competitive accuracy is not a linear function of the accuracy, computing its expected value is difficult. Using instead the competitive accuracy for the expected accuracy, which is easier to compute but more difficult to say, would not be tangibly different here.

petitive accuracy of at least q. Data structure configuration A is more ***dynamically flexible*** *than B iff there exists competitive accuracy q such that $R(A, q) \supset R(B, q)$ and for all $q' \leq q$, $R(A, q') \supseteq R(B, q')$.*

In other words, A is more dynamically flexible than B iff there is a competitive accuracy that A reaches at least whenever B does, and the same is true for every competitive accuracy worse than that. Also, in at least one of those cases, A must reach that competitive accuracy in more cases.

This means that if we lower our competitive accuracy standards low enough, A is expected to be acceptable in at least as many cases as B would be, and potentially more.

Observe that this definition is a strict partial ordering of data structure configurations solving Problem 3.1, since it is irreflexive, asymmetric, and transitive. This means that in many cases, dynamic flexibilities will be incomparable. This is often the case for different configurations of the same data structure; they tend to have a remarkably similar competitive accuracy curve scaled differently in magnitude space.

Dynamic flexibility implicitly includes **overflow potential**, the probability distribution of magnitudes at which the data structure is not able to ADD another element or is at a 100% false positive rate. These two cases have different meanings: the former means that the data structure is unable to adapt so that a QUERY of the requested element returns positive; the latter need not adapt to satisfy the ADD requirement. We group these two together, however, because one can easily adapt a data structure that could encounter the possibility of not being able to ADD another element to one that has a 100% false positive rate simply by adding an "overflowed" bit to the structure, which indicates all QUERYs should return positive.

The counterpart to dynamic flexibility is **peak competitive accuracy**, which is simply the highest competitive accuracy reached by a data structure configuration over all magnitudes. Note that these two are not opposites; an ideal structure would have universally very high competitive accuracy.

Instead, there seems to be a design trade-off between these two (a thesis of this dissertation), and different applications could demand one much more than the other, or call for a balance of the two.

On occasion, I will use "flexibility" to refer to "dynamic flexibility" for brevity. I prefer "dynamic flexibility," however, to emphasize that it is a property of individual configurations of a data structure, and does *not* refer to the static configurability of the data structure.

3.5 Data structure performance

We will also compare data structures based on their **configurability**, which generally refers to how well the static configuration parameters of a data structure allow it to satisfy various needs. Here are some examples:

- **Speed vs. Compactness/Accuracy/Magnitude**. Some data structures have a natural way of trading speed for higher compactness/accuracy/magnitude, while others only improve negligibly given more time. But if a data structure has both very high speed and very high compactness/accuracy/magnitude, such as a software implementation of hash compaction (see Chapter 7), this type of configurability is irrelevant.

- **Satellite information**. This represents an addendum to Problem 3.1 by generalizing from sets to partial maps. A structure supports satellite information if each element added is also associated with a fixed-size piece of information that can be recalled and updated *ad infinitum*. Some structures, including most Bloom filters, have no natural way to add satellite information, but the feature is important in many applications, including computation of ample sets for partial-order reduction of an explicit-state search (see Section 11.2.3).

  We will make the simplifying assumption that satellite data is completely random (uniformly distributed) and, thus, the space required

for representing it cannot be optimized beyond the naive approach of associating the appropriate amount of space with each element to be added. This keeps consideration of satellite information from polluting the analysis of space lower bounds for Problem 3.1.

3.6 Accuracy details

3.6.1 False positive rate

The **false positive rate** is, informally, the proportion of unADDed elements that will QUERY to positive (as a false positive). When U is finite, this is easy to formalize:

$$\text{If } |U| < \infty, \quad f \stackrel{(def)}{=} \frac{|W \setminus V|}{|U \setminus V|} \tag{3.1}$$

The false positive rate is undefined if $U \setminus V = \emptyset$, and rightly so.

Potentially probabilistic sets, however, are quite often used over a countably infinite universe, and authors report the false positive rate over that universe. Mathematically, this seems dubious because there is no single concept of proportion over countably infinite sets. In particular, there is no non-zero, finite, and uniform (permutation-invariant) measure over the natural numbers, just as there is no uniform probability distribution over the natural numbers.

We can, however, define the false positive rate if we're given an ordering of the universe for which the false positive rates of finite prefixes converge. We shall assume from now on that countably infinite universes come with an ordering for determining the false positive rate. In practice, the natural "smallest-first" ordering will suffice. We define $U_0, U_1, U_2, \ldots$ to be prefixes of that ordering. Specifically, $|U_i| = i$ and $U_i \subset U_{i+1}$ for $i \in \mathbb{N}$, and $\bigcup_{i=0}^{\infty} U_i = U$. We can use those prefixes to define the false positive rate:

$$\text{If } |U| = \infty, \quad f \overset{(def)}{=} \lim_{i \to \infty} \frac{|(W \setminus V) \cap U_i|}{|(U \setminus V) \cap U_i|} = \lim_{i \to \infty} \frac{|W \cap U_i|}{i} \tag{3.2}$$

The equality is justified by the finiteness of V and our assumptions about the prefixes.

Note that in theory, the order matters in the resulting false positive rate, as long as $|W| = \infty$ and $|U \setminus W| = \infty$. In these cases, for example, we could modify an ordering to yield half the false positive rate by spreading out occurrences of elements in W so that they appear only half as often as previously. In practice, however, a natural "smallest-first" ordering feeds standard hash functions in such a way as to get the anticipated limit behavior.

It is possible for the limit not to exist for a particular ordering, which would make the false positive rate undefined for that ordering. Consider $U = \mathbb{N}$ with the natural ordering and $W = \{v : \lfloor \log_{10} v \rfloor \equiv 1 \pmod 2\}$. In this case, there is no limit proportion of W to U, because it fluctuates between roughly 0.9 and 0.1 *ad infinitum*. This would be absurd behavior for a hash function feeding a data structure, though, so it is not really a practical concern.

The point here has been to have a mathematical characterization of what it means for a particular data structure instance to have a particular false positive rate, and we have done that for both the finite and infinite cases. Note that lots of previous literature is loose with the term "false positive rate", using it to refer to the median false positive rate over a space of cases. In fact, the false positive rate will almost always follow a non-trivial probability distribution rather than be some constant value for some number of unique additions. The variance, however, is typically small enough that the median is accurate enough for most purposes.

3.6.2 Omissions

The visited set usage paradigm calls for a different inaccuracy metric, because the final false positive rate does not tell us much about how accurate the search was overall. Ideally, we would be able to determine the probability that an assumed error was overlooked by the search. There are problems with computing that, including computing even the proportion of states that were omitted from the search. Nevertheless, we can justly compare data structures using accurate estimates of "hash omissions," which are closely related to the potential for overlooking errors.

A **hash omission** occurs when an element/state that has not yet been added/visited is presumed previously added/visited because of a false positive QUERY. Without some kind of oracle, we don't know exactly how many hash omissions a data structure has had, for if the data structure knew when it was making an omission, we would have it avoid doing so! But we can usually compute the expected number of hash omissions and the probability of any hash omissions.

The rest of the omissions from a lossy search are **transitive omissions**, states that are never reached because they are only reachable through hash omissions. The total **omissions** from a search, therefore, is the sum of the hash omissions and the transitive omissions. If there are no hash omissions, there cannot be any transitive omissions; thus, probability of any hash omissions is also the probability of any omissions overall.

Implementation note: Some model checkers only check properties of states (vertices) and not of the intervening transitions (edges). Many of those checkers validate each state as it is **seen**, before checking against the visited set, rather than as it is **visited**, after the visited set indicates it is new. In such checkers, because the hash omissions are seen, only the transitive omissions represent the possibility of overlooked errors. In any case, our goal will be to minimize the possibility or quantity of hash omissions, as

soon explained.

3.6.3 The Transitive Omission Problem

The complete relationship between these two types of omissions is complicated because it depends on the structure of the graph being explored. Figure 3.6.2 has an example graph that demonstrates how uncertain the number of transitive omissions is, even given statistics such as number of states visited, number of hash omissions, and branching factor. There is a chain of a million nodes connecting Subgraph A to Subgraph B; these are "critical" because each must be along any path from Subgraph A to Subgraph B. Thus, if we expect just one hash omission per million visited states, we could expect to transitively omit all of Subgraph B, which could be enormous compared to Subgraph A. Also, the chain is only 1% the size of Subgraph A, so it doesn't contribute significantly to many of Subgraph A's structural statistics, such as branching factor.

To reliably estimate the **coverage** of a search, the proportion of the reachable state space visited, we would need to solve the problem of estimating transitive omissions. We will soon see why this is not needed for this dissertation, but it is worth mentioning some anecdotal knowledge on the subject. Before widespread use of partial-order reduction [33, 47], Holzmann observed that a general rough estimate was one transitive omission per hash omission [40], and that matches our observations in using SPIN [42]. We can expect more for poorly connected problems (less than two edges per state) and fewer for highly connected problems (greater than three or four edges per state). We incorporated some heuristics like this into 3SPIN for rough coverage estimation. A shortcoming is that we have also observed outlying cases with orders of magnitude more transitive omissions than these heuristics expect or most random trials produce.

We also do not know exactly how partial-order reductions affect these

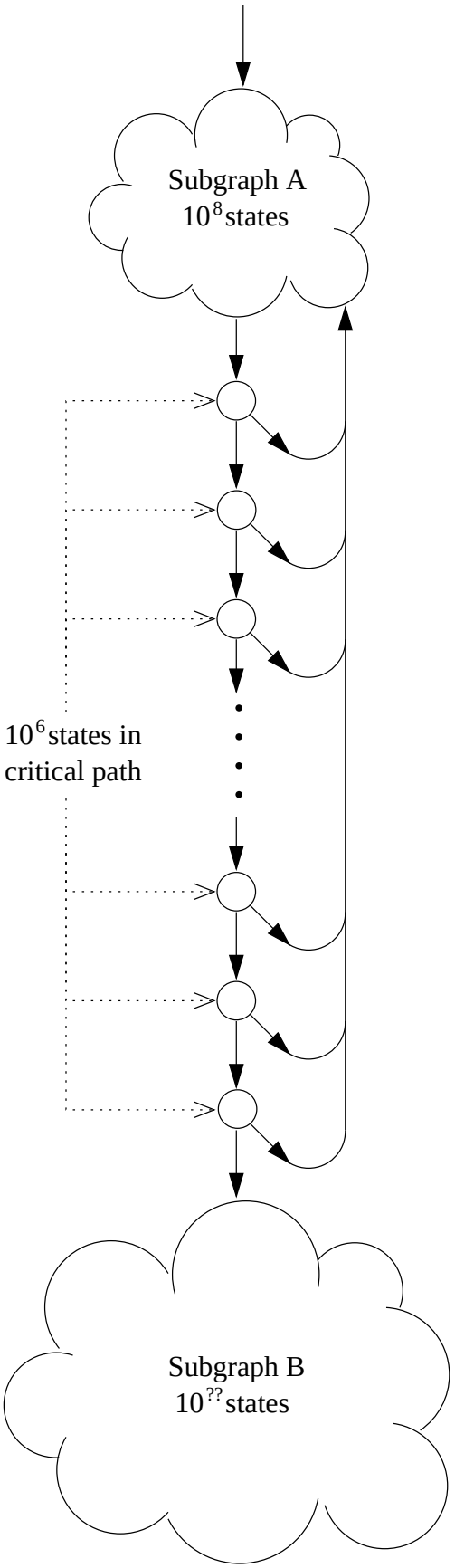

Figure 3.1: *State graph severely affected by the transitive omission problem. If any of the* 10^6 *states critical to reaching the bottom subgraph is a hash omission, that entire subgraph will be omitted transitively. Note: the back edges from the critical states make the example more portable because some checkers only perform state matching at points with non-determinism.*

heuristics. With P.O.R. enabled, we have observed cases in which a search with omissions explores *more* states than a lossless search of the same graph. Since aspects of the reduction are "on-the-fly," it is conceivable that such a perturbation could rightfully result in more states being visited. With P.O.R., therefore, we cannot attribute all of the difference between states visited with lossy search and states visited with lossless search to omissions, and we do not know how to measure how much is contributed by each. A partial-order reduced state graph is less connected, which, anecdotally, should increase transitive omissions per hash omission, but we have not verified this.

3.6.4 Error omission bound

A paper by Stern and Dill reports to derive a bound on the probability of omitting an error, taking into account the possibility of transitive omissions [78]. This would seem to better capture the effectiveness of a probabilistic verification run, since the possibility of an error being in the place you happen to omit a few states is usually quite small.

The problem is that the analysis depends on the diameter (maximum breadth-first depth) of the graph, which one knows only with the same certainty that one has explored the state space to its maximum diameter. In other words, we can really only trust the computed bound on error omission to extent we are certain we have not omitted *any* states, in which case it is not that helpful!

Though it would be nice to be able to use this metric in reporting to users, we do not know how to bound reliably the probability of omitting a single error. For this reason, and because for a given graph it is proportional to the expected omissions, we do not explore this metric any further. Instead we focus on minimizing omissions.

3.6.5 Accuracy optimization criteria

Minimizing hash omissions is the best known approach to minimizing the possibility of overlooking errors, because it is the best known approach to minimizing total omissions. Others have come to this conclusion also, namely Holzmann [40], but not so explicitly.

The basic truth that justifies this approach is that every transitive omission has a hash omission as its root cause. This means that if there are zero hash omissions from a search, then there are zero transitive omissions. Also, as observed earlier in the discussion of transitive omissions, the relationship between hash omissions and transitive omissions tends to be linear.

Even though there might be a better way to ensure errors are found than to minimize total omissions, until such a method is discovered, it is best to assume every state has equal potential for revealing an error. This suggests the approach of minimizing total omissions to minimize the possibility of overlooking errors.

Even though there might be a better way to minimize total omissions than to minimize hash omissions, until an efficient such method is discovered, it is best to assume that every state has equal potential for causing transitive omissions if omitted. This suggests the approach of minimizing hash omissions to minimize total omissions–and the possibility of overlooking errors.

It is worth mentioning a hypothesis regarding the relationship between hash omissions and transitive omissions that has failed to show validity in our testing. The hypothesis was that hash omissions earlier in the search might lead to more transitive omissions than hash omissions later in the search. Virtually any conceivable state storage technique concentrates hash omissions toward the end of the search, but they can vary somewhat in the degree of that concentration. In terms of choosing one structure over another, this hypothesis has shown no validity. One could construct a synthetic

test to get a better answer, but my testing has convinced me that among reasonable structures, the winner will be the one that minimizes the number of hash omissions.

Our "best known" metric for inaccuracy is, therefore, expected number of hash omissions.

3.6.6 More definitions and analysis

Recall that in the visited set paradigm, we consider V to be all $v = |V|$ unique elements QUERYed, and n is how many of those were recognized as "new". Thus, we can formally define the number of hash omissions, o, as the difference between our two notions of magnitude:

$$o \stackrel{(def)}{=} v - n \tag{3.3}$$

But as mentioned, this is not a practical formula because we would need some kind of oracle to know both v and n.

In a specific data structure, however, we can keep track of n and use that to estimate v and, thus, o. Using a "hat" (as in $\hat{v}$) to signify the expected value (or some other estimator) of a variable, we define the ***post facto* expected hash omissions** by this practical variant of Equation 3.3:

$$\hat{o} = \hat{v} - n \tag{3.4}$$

To estimate v from n and the history of actual false positive rates, we simply add for each affecting addition the (real-valued) expected number of unique additions to generate exactly one affecting addition at the given false positive rate ($f_{n \leftarrow i}$ is the false positive rate after i affecting additions; i.e. replace n by i):

$$\hat{v} = \sum_{i=0}^{n-1} \frac{1}{1 - f_{n \leftarrow i}} \tag{3.5}$$

For example, when the false positive rate is 50%, we expect it to take 2 more

unique additions to have 1 more affecting addition.

Combining the previous two equations, we have

$$\hat{o} = \sum_{i=0}^{n-1} \frac{f_{n \leftarrow i}}{1 - f_{n \leftarrow i}} \tag{3.6}$$

For example, when the false positive rate is 50%, we expect one omission per affecting addition, because each is expected to be half of any forthcoming unique additions.

We can also consider predicting the ***a priori* expected hash omissions**, which is the difference between a known v and the expected n that entails:

$$\hat{o} = v - \hat{n} \tag{3.7}$$

For each unique addition, the contribution to the total expected affecting additions is the probability the addition is not an omission, or one minus the expected false positive rate at that point:

$$\hat{n} = \sum_{i=0}^{v-1} 1 - \hat{f}_{v \leftarrow i} \tag{3.8}$$

Note that linearity of expectations makes the right hand side the exact expected value of the number of affecting additions if the expected false positive rates are exact.

Combining the previous two equations, we have

$$\hat{o} = \sum_{i=0}^{v-1} \hat{f}_{v \leftarrow i} \tag{3.9}$$

Another inaccuracy performance measure based on omissions is the probability of any omissions, which is the opposite of (one minus) the probability of no omissions. (Recall that there are no omissions overall iff there are no hash omissions.) The probability of no omissions is the probability that the

two types of magnitude are actually equal. The ***post facto* probability of no omissions** is simple, just "the probability I got to these n affecting additions with just $v = n$ unique additions," or, "the probability that nothing added would have QUERYed as a false positive:"

$$P(o = 0) = \prod_{i=0}^{n-1} 1 - f_{n \leftarrow i} \tag{3.10}$$

For the ***a priori* probability of no omissions** case, given some number of states to visit v, we must assume at each step that $n = v$ and factor in the probability that it stays that way:

$$P(o = 0) = \prod_{i=0}^{v-1} 1 - f_{n \leftarrow i} \tag{3.11}$$

That works for structures for which we know the exact false positive rates that will arise at runtime given $v = n$ (such as probabilistic sets based on exact sets; see Chapter 8), but for many structures (such as Bloom filters, Chapter 6), there is non-zero variance in the false positive rates even assuming $v = n$. So in those cases, we can only approximate the *a priori* probability of no omissions, using the *expected* false positive rates:

$$P(o = 0) \approx \prod_{i=0}^{v-1} 1 - \hat{f}_{n \leftarrow i} \tag{3.12}$$

This is an approximation because it's the product of expectations rather than the expectation of the product and the events are *not* independent, assuming covariance among false positive rates while adding to the structure. (It's hard to imagine a structure that would have variance in false positive rates but no covariance from addition to addition.) The approximation is very good, however.

Observe that when close to zero, the probability of any omissions and the expected number of omissions approximate each other, based on the

property that for $\varepsilon_1, \varepsilon_2, \ldots$ close to 0,

$$1 - (1 - \varepsilon_1)(1 - \varepsilon_2) \ldots \approx \varepsilon_1 + \varepsilon_2 + \ldots$$

Based on an asymptotic approximation and relaxing certain assumptions, there is an approximate relationship between the probability of no omissions and expected hash omissions:

$$P(o = 0) \approx 1 - e^{-\hat{o}} \tag{3.13}$$

Finally, observe that another piece of evidence can qualify the above predicted numbers of omissions/unique false positives: the number of positive queries. Specifically, the number of unique false positive queries cannot exceed the number of positive queries, but this is typically not helpful due to "revisitation" of elements/states, which entail positive queries.

CHAPTER 4

Lower Bounds for State Storage

Here I derive and cite information-theoretic lower bounds for memory required to solve the state storage problem. These give us an optimal to compare against and, in some cases, help us to understand the nature of near-optimal solutions. Recall that I am only considering non-heuristic solutions.

Bibliographic Notes and Contributions The results of Corollary 4.3 and Corollary 4.4 agree with others' results [13, 65, 11], but my approach is more unified and, I believe, more elegant. A similar analysis appears in [19, Appendix C], in which bounds are derived in terms of u (called u), v (called n), and f (called ε). I find it simpler to use the represented set size, w, instead of the false positive rate, f, when analyzing the case of finite u.

4.1 Most cases

We now determine by information theory how much space is required to guarantee being able to represent a set of a particular magnitude with a particular accuracy from a particular universe. Here we use the false positive rate as the accuracy metric, so if $|U| < \infty$, the accuracy is determined by the trio $|U|$, $|V|$, and $|W|$: the universe size, the visited set size, and the

represented set size respectively. We will address the infinite universe case later.

So that we can use the variables V and W in quantification, let us use u, v, and w for the universe size, visited set size, and represented set size respectively. Recall, therefore, that $v \leq w \leq u$.

We consider approximating all possible V for which $|V| = v$ and $V \subseteq U$, but the data structure can use any W for which $|W| = w$ and $V \subseteq W \subseteq U$. Therefore, the amount of space required depends on how many subsets of U of size w are required for each subset of U of size v to be a subset of (at least) one of those w-sized subsets.

More symbolically, find a smallest set S such that

$$\forall W \in S,\ |W| = w \wedge W \subseteq U$$

$$\forall V \subseteq U \text{ where } |V| = v, \exists W \in S \text{ where } V \subseteq W$$

S then contains the minimum number of possibilities to represent any size-v subset of U with an over-approximation of size w. Thus, $\lg |S|$ bits of information are required to pick any one of those possibilities. (I will use $\lg$ to refer to the base-2 logarithm.)

Lemma 4.1. *Letting S be constrained as above,*

$$|S| \geq \frac{\binom{u}{v}}{\binom{w}{v}}$$

$$|S| \leq \left(1 + \ln \tbinom{w}{v}\right) \frac{\binom{u}{v}}{\binom{w}{v}}$$

$$|S| \leq \tbinom{uv/w}{v}$$

Proof The lower bound is simple and well-known: the numerator is the number of size-v subsets of U. S must contain a size-w subset that covers each of those size-v subsets. The most size-v subsets that can be covered by

a size-w subset is the denominator. Thus the quotient is a lower bound for how many size-w subsets are needed to cover all possible size-v subsets of U.

The first upper bound was proven by Erdős and Spencer [27], and the second upper bound comes from a construction by Carter et al [13], who also observe that neither upper bound dominates the other in all cases [13, Section 5], so both are useful. □

Theorem 4.2. *Let $\breve{m}_{v,u,f}$ bits be the minimum space required to represent any size-v subset of U ($|U| = u < \infty$) with a false-positive rate of $f = \frac{w-v}{u-v}$ (implying represented set size $w = (u-v)f + v$). The following bounds hold:*

$$\breve{m}_{v,u,f} \geq \lg \frac{\binom{u}{v}}{\binom{w}{v}}$$

$$\breve{m}_{v,u,f} \leq \lg \left(1 + \ln \tbinom{w}{v}\right) + \lg \frac{\binom{u}{v}}{\binom{w}{v}}$$

$$\breve{m}_{v,u,f} \leq \lg \tbinom{uv/w}{v}$$

The reason we use f as a parameter to $\breve{m}$ rather than w is to support the case of infinite U and non-zero f:

Corollary 4.3. *Let $\breve{m}_{v,\infty,f} = \lim_{u\to\infty} \breve{m}_{v,u,f}$ be the minimum space required to represent any size-v subset of an infinite universe U with non-zero false positive rate f. The following bounds hold:*

$$\breve{m}_{v,\infty,f} \geq \lg f^{-v}$$

$$\breve{m}_{v,\infty,f} \leq \lg \tbinom{v/f}{v}$$

Proof First, observe that

$$\lim_{u\to\infty} \frac{w}{u} = \lim_{u\to\infty} \frac{(u-v)f + v}{u} = f$$

From Theorem 4.2, this gives us rather directly the upper bound. The lower bound is not as obvious, but is well-known (written "$n \log(1/\epsilon)$" in [65]):

$$\lim_{u\to\infty} 2^{\breve{m}_{v,u,f}} \geq \lim_{u\to\infty} \frac{\binom{u}{v}}{\binom{w}{v}} = \lim_{u\to\infty} \frac{\binom{u}{v}}{\binom{uf}{v}} = \lim_{u\to\infty} \prod_{i=0}^{v-1} \frac{(u-i)}{(uf-i)} = \lim_{u\to\infty} \left(\frac{u}{uf}\right)^{v} = f^{-v}$$

□

Another special case of Theorem 4.2 is when the false-positive rate is zero, when the set is *exact*:

Corollary 4.4. *A lower-bound estimate of the space required to represent exactly any size-v subset of a finite universe U ($|U| = u$) is $\breve{m}_{v,u,0}$ bits, defined as*

$$\breve{m}_{v,u,0} = \lg \binom{u}{v}$$

Proof When $f = 0$, $w = v$. Thus, the $\binom{w}{v}$ terms in Theorem 4.2 are 1, making the upper and lower bounds equal. □

4.2 Various magnitudes

For these cases above, we only analyzed the space requirements given some fixed number of elements, but we are more interested in structures that can represent *up to* some number of elements. Here I show the distinction is usually insignificant. The exact lower bound for up to v given lower bounds for fixed sizes is

$$\breve{m}_{0..v,u,f} \stackrel{(def)}{=} \lg \sum_{i=0}^{v} 2^{\breve{m}_{i,u,f}} \tag{4.1}$$

That formula comes directly from counting the possibilities for up to v. We can generate upper and lower bounds for that based on the maximum value of the summation:

Theorem 4.5.

$$\overset{v}{\underset{i=0}{\mathsf{MAX}}}\, \breve{m}_{i,u,f} \;\leq,\approx\; \breve{m}_{0..v,u,f} \;\leq,\approx\; \left(\overset{v}{\underset{i=0}{\mathsf{MAX}}}\, \breve{m}_{i,u,f}\right) + \lg v$$

Proof The lower bound is a simple consequence of the non-negativity of the $\breve{m}_{i,u,f}$. The upper bound is also simple:

$$\lg \sum_{i=0}^{v} 2^{\breve{m}_{i,u,f}} \leq \lg \left(v \overset{v}{\underset{i=0}{\mathsf{MAX}}}\, 2^{\breve{m}_{i,u,f}} \right) = \left(\overset{v}{\underset{i=0}{\mathsf{MAX}}}\, \breve{m}_{i,u,f}\right) + \lg v$$

Finally, because the bounds only differ by $\lg v$, both bounds are good approximations. □

And in most cases, the largest term is going to be the last one, which simplifies the bounds:

Corollary 4.6.

$$\textit{When}\;\; \breve{m}_{v,u,f} = \overset{v}{\underset{i=0}{\mathsf{MAX}}}\, \breve{m}_{i,u,f}, \quad \breve{m}_{v,u,f} \;\leq,\approx\; \breve{m}_{0..v,u,f} \;\leq,\approx\; \breve{m}_{v,u,f} + \lg v$$

For example, when $u = \infty$, we could use a structure to which we have added v elements with false positive rate f to represent any $i < v$ elements with the same false positive rate. Thus, $\breve{m}_{v,\infty,f}$ is the maximum.

Also, when $v = w$ (equivalently $f = 0$) and $v \leq \frac{u}{2}$, properties of binomial coefficients tell us that the last term is once again the largest. Otherwise, it is likely to also hold when $w \leq \frac{u}{2}$, but this is a difficult case to analyze. We clearly run into trouble if $v > \frac{u}{2}$, and probably so if $w > \frac{u}{2}$.

Someone else is welcome to resolve more exactly when these bounds hold, but it is clear that in many cases, we do not need to draw a distinction between the space required for representing some exact number of elements and representing up to that number of elements.

4.3 Simpler bounds

It is easy to see the per-element memory lower bound for the infinite universe, non-zero false positive rate case,

$$\breve{m}_{v,\infty,f} \geq v \lg f^{-1} \tag{4.2}$$

That is, a minimum of $\lg f^{-1}$ bits per added element are needed for the over-approximation to have a false positive rate of f.

However, it is not easy to understand the per-v contribution to the bounds involving binomial coefficients, such as $\breve{m}_{v,u,0} = \lg \binom{u}{v}$. But we can use bounds on the binomial coefficient to generate bounds that enable us to think more concretely about what the bounds mean for the implementation of a data structure. The bounds $\left(\frac{u}{v}\right)^v \leq \binom{u}{v} \leq \left(e\frac{u}{v}\right)^v$, where e is the base of the natural logarithm, are well known [17, Section C.1] and give us

$$v\left(\lg u - \lg v\right) \leq \breve{m}_{v,u,0} \leq v\left(\lg u - \lg v + \lg e\right) \tag{4.3}$$

$$v\left(\lg u - \lg w - \lg e\right) \leq \breve{m}_{v,u,f} \leq v\left(\lg u - \lg w + \lg e\right) \tag{4.4}$$

where $w = (u - v)f + v$

$$v \lg f^{-1} \leq \breve{m}_{v,\infty,f} \leq v(\lg f^{-1} + \lg e) \tag{4.5}$$

We can state these using asymptotic notation:

$$\breve{m}_{v,u,0} = v\left(\lg u - \lg v + O(1)\right) \tag{4.6}$$

$$\breve{m}_{v,u,f} = v\left(\lg u - \lg w \pm O(1)\right) \tag{4.7}$$

where $w = (u - v)f + v$

$$\breve{m}_{v,\infty,f} = v(\lg f^{-1} + O(1)) \tag{4.8}$$

When $w \ll u$ (and thus $v \ll u$), it is difficult for an implementation to fall within these asymptotic characterizations of memory lower bounds. The only practical structure I have found to meet these bounds is the Cleary table, and that's only in the case of letting the structure fill up completely. [1]

The formulas do not tell us much when w is close to u, but that is fine for this dissertation. I consider the Bloom filter to handle well the cases in which a large portion of the universe of elements is represented, and I propose no replacement for the Bloom filter in those cases. To the contrary, my new adaptive solution in Chapter 11 uses Bloom filters when w must get close to u because of memory constraints.

In the context of these asymptotic characterizations of "near minimal" memory, we can simplify things further by the following observation:

Corollary 4.7.

$$\breve{m}_{v,u,f} = \min(\breve{m}_{v,u,0}, \breve{m}_{v,\infty,f}) - O(v)$$

Proof If $u = \infty$, then it is clearly true, so we assume from here that $u < \infty$.

Observe that, with respect to f, $\breve{m}_{v,u,0}$ is constant and both $\breve{m}_{v,\infty,f}$ and $\breve{m}_{v,u,f}$ are non-increasing, with $\breve{m}_{v,u,f} \leq \breve{m}_{v,\infty,f}$. Trivially, when $f = 0$, $\breve{m}_{v,u,f} = \breve{m}_{v,u,0}$, and when $f = 1$, $\breve{m}_{v,u,f} = \breve{m}_{v,\infty,f} = 0$. Also, $\breve{m}_{v,\infty,f}$ and $\breve{m}_{v,u,f}$ are concave-up with respect to f, based on graphing our best lower bounds (Theorem 4.2) or positivity of the second derivatives of simpler bounds in Equations 4.4 and 4.5.

See Figure 4.1 for an example depicting these constraints.

Under that set of constraints, the maximum difference between $\breve{m}_{v,u,f}$ and $\min(\breve{m}_{v,u,0}, \breve{m}_{v,\infty,f})$ must occur when $\breve{m}_{v,u,0} = \breve{m}_{v,\infty,f}$. When that is the case,

$$v\,(\lg u - \lg v + O(1)) = v(\lg f^{-1} + O(1))$$

[1] The standard Cleary table has no random access when it is full. See Chapter 9.

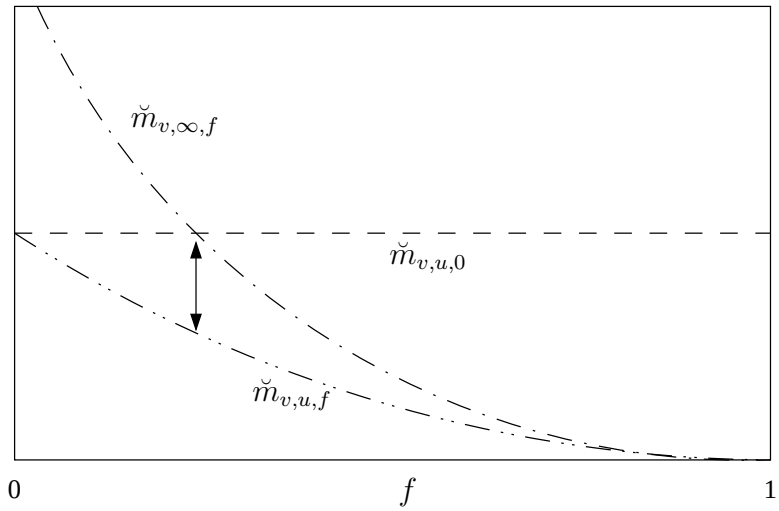

Figure 4.1: *Graphical depiction of various memory lower bounds vs. false positive rate. This depicts the properties needed for the proof of Corollary 4.7.*

$\Rightarrow$ ($v > 0$, definition of f)

$$\lg u - \lg v = \lg \frac{u - v}{w - v} \pm O(1))$$

$\Leftrightarrow$

$$\lg(w - v) = \lg \frac{v(u - v)}{u} \pm O(1))$$

$\Leftrightarrow$

$$w - v = \frac{v(u - v)}{u}\ 2^{\pm O(1)}$$

$\Leftrightarrow$

$$w = v\left(\frac{u - v}{u}\ 2^{\pm O(1)} + 1\right)$$

$\Rightarrow$ ($0 < v \leq u$)

$$w = \Theta(v)$$

$\Leftrightarrow$

$$\lg w = \lg v \pm O(1)$$

Plugging this into Equation 4.7 and combining with Equation 4.6 completes the proof, because the $O(v)$ bound on the difference holds in the max-

imum case. □

I suspect this has been proven before in perhaps a slightly different form, but it points to why literature on representations of sets tends to ignore the detailed analysis of over-approximating a subset of a finite universe: it is usually okay because your imprecision is only $O(v)$ bits. This is no less precise than my asymptotic form of the lower bound, Equation 4.7.

4.4 "Asymptotically compact" litmus test

Using variants of Equations 4.6 and 4.8, we can come up with a "litmus test" for structures that are within a constant factor of optimal, which I call "asymptotically compact."

Consider this variant of Equation 4.6:

$$\frac{\breve{m}_{v,u,0}}{v} = \lg \frac{u}{v} + O(1) \tag{4.9}$$

The suggests a litmus test for exact storage from a finite universe: the memory required per added element should stay essentially the same if we scale v and u by some constant c.

A similar test arises out of Equation 4.8:

$$\frac{\breve{m}_{v,\infty,f}}{v} = \lg f^{-1} + O(1) \tag{4.10}$$

For inexact storage from an infinite universe, the litmus test is this: the false positive rate should stay essentially the same if we scale v and m by some constant c.

Finally, we can do the same for finite universes generally, based on Equation 4.7:

$$\frac{\breve{m}_{v,u,f}}{v} = \lg \frac{u}{w} + O(1) = \lg \frac{u}{(u-v)f + v} + O(1) \tag{4.11}$$

For inexact storage from a finite universe, the litmus test is this: the false

positive rate should stay essentially the same if we scale u, v, and m by some constant c.

These litmus tests are not sufficient for establishing that solutions are close to optimal, but they do establish that the asymptotic behavior of the problem allows us to generalize good performance on moderately large problems to good performance on arbitrarily large problems.

4.5 Exact representation, infinite universe

What is left is the case of exactly representing a subset of an infinite universe. This is fundamentally different from the cases above, in which we could assume we extract some fixed, finite amount of information from each element for adding it to the structure. More specifically, the amount of information needed about each element is bounded if either our universe size is bounded or our accuracy is bounded/imperfect, but now we consider the case in which neither is bounded. In other words, we have to store elements of variable size.

In the bounded case, we were agnostic to the input distribution by considering the worst case: assuming the input contained as much information as possible, which for finite ranges of integers means they have uniform distribution.

The unbounded case requires some finesse because we can no longer assume the elements are uniformly distributed: there is no such thing as a uniform distribution over a countably infinite domain. It is clear that because the elements are of variable size, the amount of memory required will depend not just on the number of elements, but also on how much information they contain. But we cannot assign a unique, finite piece of information to each element of an infinite, countable set and still be distribution-agnostic. Some elements will be assigned more information than others.

Despite there not being a distribution-agnostic lower bound, we can still

come up with minimal encodings of these sets that can be used as standards of optimality. In this case, a minimal encoding is a bijective function from $\mathcal{F}(\mathbb{N})$ to $\mathbb{N}$, where $\mathcal{F}(\mathbb{N})$ is the set of all finite subsets of $\mathbb{N}$. In theory, all minimal encodings are equally valid as standards for optimality, because they are just permutations of each other, but in practice, some are more reasonable and practical than others.

These observations serve to contrast this variant of the problem with the ones already discussed. In this dissertation, I will focus on the cases in which the amount of information needed from each element is bounded, and I will leave to others the problem of examining the optimality of structures for exactly representing elements from an infinite universe.

CHAPTER 5

Classical Solutions

Here I present important classical data structures solving Problem 3.1. These are exact solutions, meaning false positive queries are not allowed/possible, but I will describe in Chapter 8 how exact solutions can be used as the basis for inexact solutions. Note that most "textbook" or "standard library" implementations of dictionaries, maps, or sets are far less space-efficient than the structures in this chapter, which are only near-optimal in relatively restricted cases.

Contributions This chapter is mostly to put other chapters in context, but I describe a simple and reasonably effective design for a compacted chaining hash table, "2/3rds chaining."

5.1 Open-addressed table

One of the simplest classical structures that seems very compact is an open-addressed table of all the elements. If it takes $b = \lceil \lg |U| \rceil$ bits to identify each element, then an open-addressed table accommodating up to v elements is an array of v entries of b bits each. It is well-known how to use hashing to compute probe sequences efficiently, such as with double hashing [34].

Given $v = |V|$ and $u = |U|$, the structure requires at least $v \lg u$ bits overall. However, the optimal, $\breve{m}_{v,u,0}$, is close to $v(\lg u - \lg v)$ bits. Thus for small V, the open-addressed table is close to optimal, but for larger V, it requires $\Theta(v \lg v)$ more bits than optimal.

Implementation notes:

- There is no need for a bit for each cell indicating whether it is occupied; we can reserve the entry of all zeros to indicate that a cell is unoccupied and use one extra bit overall (negligible) to indicate whether the entry of all zeros is in our structure.

- Double hashing is very attractive in that a reasonably small amount of hash information is needed, there is almost no observable clustering, and it is easy to make sure your probe sequence is a permutation of the cell indices [34]. The last is accomplished by either using a prime number of cells with non-zero increments, or a power-of-2 number of cells with only odd increments.

- The above analysis presumes we allow the structure to fill up and in practice, it does not seem that slow. In theory, however, the practice does not scale. The time to fill the structure completely (as a visited set) is actually log-linear in the number of cells in the structure, while the time to fill to some constant occupancy less than 100%, such as 99.5%, is linear. See Section 7.1.5.

- Ordered hashing [2] is not useful when this structure is used as a visited set, because in that case all negative QUERYs lead to an ADD.

5.2 Bit table

Probably the simplest approach to representing a subset of a finite universe is to allocate a bit to each element of the universe to indicate whether that element is in the subset. We shall call this approach a "bit table". Assuming

we have an easily-computable bijection between U and $0, \ldots, u-1$, we can represent any subset V of U as a bit vector of length u, in which each bit indicates whether the corresponding element of U is in V.

This structure always uses u bits, so it is only competitively compact when v is a sizeable fraction of u. For example, if $v = \frac{u}{2}$ then $\breve{m}_{v,u,0} = \lg \binom{u}{v} \lessapprox u$ (because $\lim_{u\to\infty} u^{-1} \lg \binom{u}{u/2} = 1$). Thus, this representation is practically optimal when $v = \frac{u}{2}$ and, thus, practically optimal for representing sets *up to* sizes $\frac{u}{2} \leq v \leq u$ (see Section 4.2). But the bit table is far from optimal when v is much less than $\frac{u}{2}$.

Observe that the open-addressed table and the bit table are near optimal at opposite ends of the spectrum of visited set sizes: the former for small, sparse sets and the latter for dense sets. A **sparse** set is one that contains a negligible portion of the universe, and usually even a small portion when viewed on a log scale, such as less than the fourth root of the universe size. A **dense** set is one that contains a non-negligible portion of the universe, probably more than one hundredth of the universe. Sets that are larger than the square root of the universe size but still negligible in absolute terms I consider to be **moderately dense**. Note that sparse does not necessarily imply small, nor does large necessarily imply dense.

5.3 Compacted chaining

Using "compacted" representations of external chaining, which I consider a reasonably intuitive twist on a classical approach (Knuth agrees [58, Sect. 6.4, exer. 13]), one can construct tables that represent large, moderately dense sets reasonably compactly. Geldenhuys and Valmari demonstrate such a structure in SPIN [31] and Valmari also uses the approach in representing the state space of a 2x2x2 Rubik's cube [82]. However, the asymptotic memory usage of the structure inherently dominates the lower bound, which points to what is special about structures that are within a constant factor of

the lower bound.

5.3.1 Description

The key to this structure is that collisions are resolved in chains (linked lists) of entries that hash to the same "home address". If we run each descriptor through a randomization function (one-to-one hash) and use part of that result as the "home address", that part does not need to be stored, because that information is implied by what chain the entry is in. Thus, the location and the remainder of the randomized descriptor uniquely identify an element from our universe. Contrast this to the open-addressed table, in which the location of an entry within the structure gives us no *definitive* information[1] about the element stored there; any element could end up in any cell.

A second key is not to use machine pointers or a general memory manager for cells; instead, cells should be in an array and "allocated" in order to the linked lists, which should use bit-packed array indices as pointers. In fact, it is best to use one array whose indices are home addresses and another for the dynamically allocated cells, whose indices are pointers in the linked lists. See [82] for more detail.

Even assuming we know how many elements to accommodate (v), there are numerous design trade-offs:

- Number of "home" addresses, which is how many distinct chains the structure will have. More homes enable more information to be encoded in the location of each entry rather than stored with it, but more homes also means more chains are expected to be empty.

- Chain preallocation, which is how many entries are pre-allocated for each chain. In a classical chaining hash table, there would be just a pointer for each chain, zero preallocation. Some preallocation means fewer pointers are required to represent short chains, but too much

[1] It does give us *probabilistic* information. See Chapter 7.

means lots of wasted space for chains shorter than the preallocation length.

- Entries per overflow cell. In a typical linked list, there would be only one entry and one pointer in a cell, but putting more than one entry in each cell–with just one pointer–can reduce the memory required, or waste space with unused entries. Heterogeneous cell sizes could also provide benefit if there is not too much overhead in managing them.

5.3.2 Analysis

The full size of each element, as in an open-addressed table, is $\lg u$ rounded up. In a compacted chaining table, however, there will be a number of home addresses equal to some reasonable proportion of v, say v/c home addresses. Consequently, $\lg(v/c) = \lg v - \lg c = \lg v - O(1)$ bits of each entry need not be stored. The memory dedicated to storing the entries themselves then is $v(\lg u - \lg v + O(1))$, which is roughly equal to the memory lower bound given in Equation 4.3—even asymptotically.

The overhead comes with the pointers and unoccupied cells. The number of dynamically allocated cells will be some reasonable proportion of v, say v/c' pointer addresses. The size of each pointer is then $\lg(v/c') = \lg v - \lg c' = \lg v - O(1)$ bits. If we have one pointer per entry, then the total memory usage is similar to an open-addressed table: $v(\lg u \pm O(1))$. For the structure to beat the open-addressed table, therefore, there must be multiple entries for each pointer, say m entries per pointer on average. Now it is clear that there is enormous potential for savings over the open-addressed table: $v(\lg u - \lg v + \frac{1}{m}\lg v \pm O(1))$, but as long as there are $\Theta(v)$ pointers of size $\lg \Theta(v)$ each, the memory usage of a compacted chaining table asymptotically dominates the lower bound, $v(\lg u - \lg v + O(1))$

Coupling multiple entries per pointer almost necessarily implies some entries will be left unoccupied, even if the structure overflows. When there

are entries left unoccupied, the memory usage increases by $\lg u - \lg v + O(1)$ for each one beyond the v that are occupied. This means the overall memory required is really $\Theta(v(\lg u - \frac{m-1}{m} \lg v))$.

Implementation notes:

- Unless you know your data will be uniformly distributed, it is very important to use a randomization function on the input elements so that they are effectively uniformly distributed. Valmari discusses such a function that works well [82].

- The number of home addresses should be a power of two, so that a whole number of bits can be used and removed from the randomized descriptor. The number of dynamically allocated cells can easily be a non-power of two, with pointer sizes being rounded up to the next whole number of bits.

- I did not mention OCCUPIED bits in the high-level discussion. It would clearly take more memory to add another bit to each entry to indicate whether it is occupied. Some or all of these can be eliminated with some tricks or special circumstances. The first entry in dynamically-allocated cells, for example, can be assumed to contain valid entries; otherwise they would not have been allocated. The trick of reserving the entry of all zeros to indicate "unoccupied" is not as simple as in the case of the open-addressed table: rather than there being only one possible entry that is stored as all zeros, there is a unique such possible entry per home address/chain. A natural solution, therefore, is to have a special bit for each home address indicating whether the "all zeros" entry for that home address is in the set. With that taken care of, the all zeros trick can be used, and the cost is just one bit per home address/chain rather than probably several per chain. In the analysis above, this overhead is subsumed by unknown constant addends.

- In case it was not obvious, optimizing the design of a compacted chaining table depends heavily on the relative size of entries to pointers. Smaller, more sparse sets will have relatively large entries, while larger, more dense sets will have relatively large pointers. The former case favors filling allocated entries, while the latter favors coupling multiple entries per pointer.

5.3.3 A clever design: 2/3rds chaining

One approach to saving space in a compacted chaining table is to allow the last part of a cell to store either a pointer or an entry. Consider employing this approach with preallocated and dynamically-allocated cells that can store either [three entries] or [two entries and a pointer], as depicted in Figure 5.1. A flag bit in the cell indicates whether the final data is an entry or a pointer. The space should be made large enough to accommodate either, but the bit of metadata is required to distinguish the two cases.

The nice property is that in any chain with at least two elements, all cells in that chain have at least two-thirds of their entries storing elements, as depicted in Figure 5.2. This property is preserved even when a new cell must be added to the end of the chain, because that case presupposes three elements occupying the last cell and combining that with the element to be added gives four elements to be split between the last two cells, leaving each of them 2/3rds full. This, of course, is the source of the name I have given the approach.

5.4 Summary

For a comparison of the compactness of these structures over a range of densities, see Figure 5.3.

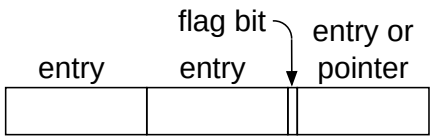

Figure 5.1: *Allocation unit in a 2/3rds chaining table. Each contains either two entries and a pointer or up to three entries, as indicated by the flag bit.*

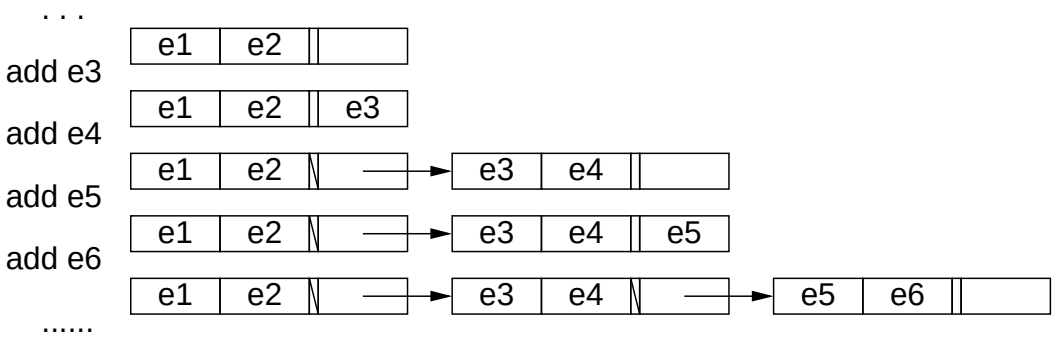

Figure 5.2: *Elements being added to one chain of a 2/3rds chaining table. Observe that in every case, at least roughly 2/3rds of the space is used to store entry data.*

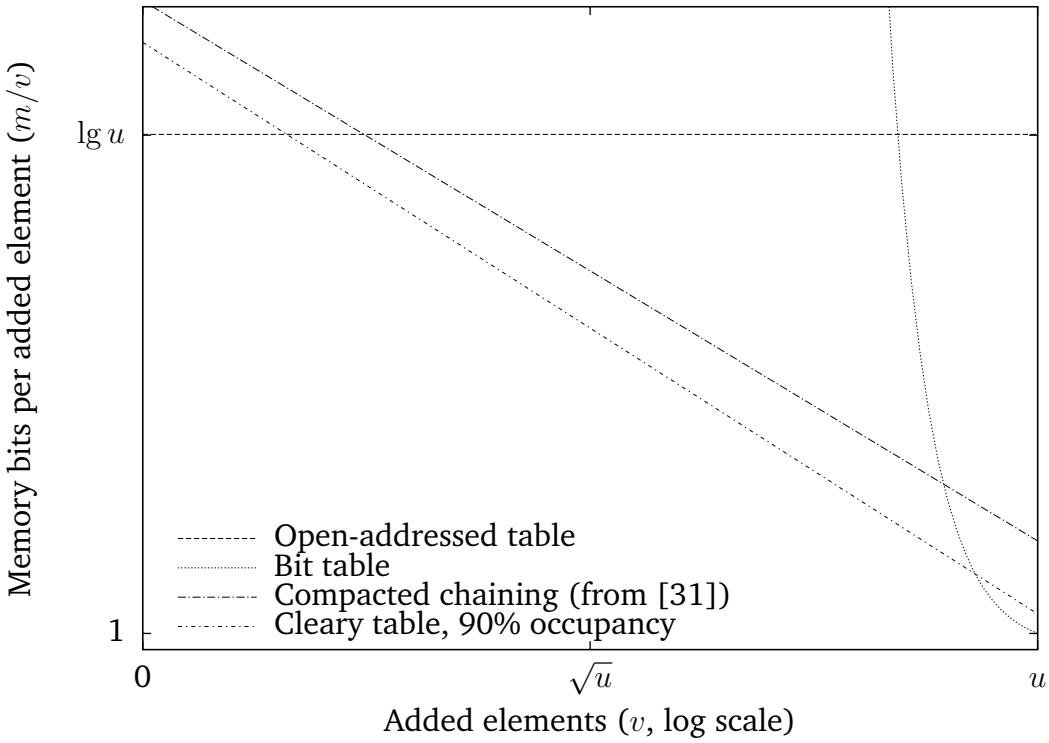

Figure 5.3: *Comparison of the compactness of classical structures for various densities. These numbers come from analyses of these structures, using $u = 2^{32}$, but only the general shape regardless of u is intended to be conveyed. For compacted chaining, I use the expected compactness for a design by Geldenhuys and Valmari [31]. I do not consider the Cleary table a "classical structure" but it is included here for comparison purposes (see Chapter 9).*

CHAPTER 6

Bloom filters (Bitstate hashing)

6.1 Introduction

A **Bloom filter** [5] represents an over-approximation W of a set V by setting bits in a bit vector for each element of V. The structure is implemented as a bit vector of m bits, all initially "0". To add an element, we set all the bits associated with that element to "1". An element is in W iff the all the bits associated with it are set to "1".

A **standard Bloom filter** uses some number, call it k, of independent hash functions, $h_1, \ldots, h_k : U \to \{0, \ldots, m-1\}$, to determine the indices of bits to be set for each element added. Adding an element x entails setting the bits at positions $h_1(x), h_2(x), \ldots h_k(x)$ to "1". An element x is in the set represented by the structure (regardless of whether x was added) iff the bits at positions $h_1(x), h_2(x), \ldots,$ and $h_k(x)$ are all "1".

Before looking closely at the accuracy of the structure, we can make several observations. First, there is no traditional collision resolution; there is no conditional probing; only a fixed number of addresses are probed to see if an element has not been added. Second, the structure does not overflow in the traditional sense; a Bloom filter can always "add" more elements because it can always adapt its representation to ensure its represented set includes any given elements; any Bloom filter can represent the entire universe of elements, by having all its bits set to "1". Finally, it is not possible to reverse-

engineer much information about the elements added to a Bloom filter; we can't tell which bits came from the same element, which hash function(s) set a particular bit, or even exactly how many elements have been added.

Bibliographic and Historical Notes "Bitstate hashing" was a name coined by Gerard Holzmann to refer to a method of storing visited states of a graph by setting bits in a table [39, 40]. It was later discovered that this approach was congruent to using a Bloom filter, in time for publication of Holzmann's 2003 book on SPIN [42].

Holzmann also used the term "supertrace" for the same approach [36, 37, 38], but typically only for the specific case of setting very few bits per state in anticipation of memory being severely constrained compared to the state graph size.

Crediting Holzmann's contribution, I call using a $k = 3$ Bloom filter in a lossy state-space search the "standard bitstate approach." The justification for this configuration is best described in Holzmann's 2003 SPIN book [42], but I echo that justification in Section 6.3.3.

Contributions My key contributions to Bloom filter theory and practice are the design and analysis of Bloom filters whose indices are computed from less hash information than would normally be required. This improvement can drastically reduce the dependence of access times on the number of indices computed per state. I also show how to opimize Bloom filters for use as a visited set, because the existing analysis of optimizing Bloom filters is based on a different usage paradigm. I use citations throughout indicate and acknowledge the work of others.

6.2 Accuracy analysis

A Bloom filter is inherently inexact unless used as a bit table ($k = 1$ and no hashing; see Section 5.2). Consequently, its expected false positive rate does

not depend on the size of the input universe.

The false positive rate of a standard Bloom filter is easily determined at runtime by the proportion of bits that are still "0", z, and the number of indices associated with each element, k. This formula is exact assuming the hash functions are independent, uniform, and random and that the input universe is infinite, because under those assumptions it corresponds to the probability that all k indices for an element refer to bits that are "1":

$$f_{\mathsf{BF}z}(z,k) \stackrel{(def)}{=} (1-z)^k \tag{6.1}$$

If the universe of elements is finite, the actual false positive rate varies, because random chance determines how much the indices associated with the unadded elements of the universe overlap with those of the added elements. In practice, this is rarely significant or important.

But even with an infinite universe, the false positive rate is not statically determined. Given some number of additions (call it v), indices per element (k), and bit vector length (m), the proportion of "0" bits can vary based on the extent to which the additions' bits overlap, due to random chance. Assuming the additions are chosen without regard for the hash functions, z follows a probability distribution. The distribution itself is not simple, but it's expected value is easy to find: if we know the probability of each bit being "0" then linearity of expectations tells us that that is equal to the expected proportion of "0"s. That probability is just the probability that each of the vk times a bit has been chosen to set, the bit chosen was not the one in question:

$$\hat{z}_{\mathsf{BF}}(m,v,k) \stackrel{(def)}{=} \left(1-\frac{1}{m}\right)^{vk} \tag{6.2}$$

Now if we plug $\hat{z}$ into Equation 6.1, we get what is actually an approximation:

$$\tilde{f}_{\mathsf{BF}}(m,v,k) \stackrel{(def)}{=} \left(1-\left(1-\frac{1}{m}\right)^{vk}\right)^k \approx \hat{f}_{\mathsf{BF}} \tag{6.3}$$

Numerous papers have implicitly characterized this formula as the exact false positive rate of a Bloom filter [5, 73, 9, 62, 16]. Researchers at Carleton University, however, pointed out that this formula is "incorrect" [7]; it does not in fact correspond exactly to the expected false positive rate of a Bloom filter, because the false positive rate for the expected proportion of bits set is not exactly equal to the expected false positive rate. Equation 6.3 essentially assumes that a new Bloom filter is constructed independently for each of the k probes. Details are in the paper [7], but some of their claims are technically false for failing to recognize that there is variance in the false positive rates of randomly-seeded Bloom filters. Many places where they say "false positive rate," they mean "expected false positive rate."

However, Mitzenmacher is likely the most thorough, for showing that the probability of the actual false positive rate deviating significantly from that induced by these false assumptions is small and proportional to $\sqrt{m}/e^m$ [63, Section 5.5.3]. Equation 6.3 is more than accurate enough for this dissertation, and after this point, I will often say just "false positive rate" or "accuracy" when I really mean the approximate expected value of such.

In fact, since I am mostly concerned with large Bloom filters, an asymptotic approximation (below) of Equation 6.3 that depends only on m/v and k is accurate enough for our purposes. Also, Equation 6.3 is inconvenient to compute since floating-point arithmetic is poor at representing values close to 1.

The following approximation of $\hat{z}$ approaches the actual expectation in Equation 6.2 as $m \to \infty$:

$$\tilde{z}_{\mathsf{BF}}(m, v, k) \stackrel{(def)}{=} e^{-kv/m} \approx \hat{z}_{\mathsf{BF}}(m, v, k) \tag{6.4}$$

We can plug that into Equation 6.1 to get the most convenient approximation

of the false positive rate of a standard Bloom filter:

$$\tilde{f}_{\mathsf{BF}'}(m/v, k) \overset{(def)}{=} \left(1 - e^{-kv/m}\right)^k \approx \tilde{f}_{\mathsf{BF}}(m, v, k) \tag{6.5}$$

With this approximation, the Bloom filter passes our litmus test for an inherently inexact structure being asymptotically compact (see Section 4.4). In fact, it has been shown that a Bloom filter at its best is a memory factor of $\lg e \approx 1.44$ from optimal [65], as shown in Figure 6.1. Thus, its peak competitive accuracy is roughly $1/\lg e \approx 0.693$ (see Definition 3.4).

As shown in Section 3.6.2, accuracy metrics relevant to the visited list usage paradigm can be determined given false positive rate data.

6.3 Optimization

Optimizing the expected accuracy of a Bloom filter depends on which accuracy metric is most appropriate, but some changes always improve expected accuracy, by any conceivable measure. Such changes are often underappreciated, and include these: (1) A Bloom filter using more memory (all other parameters being equal) is always more accurate (in expectation). (2) A Bloom filter to which fewer elements have been added (all other parameters being equal) is always more accurate (in expectation).

6.3.1 False positive rate, known v and m

Picking k to minimize the false positive rate is a well-studied problem [62], but simple solutions are not as effective as they might appear. The main observation is that the false positive rate approximation of Equation 6.5 is minimized when $k = \frac{m}{v} \ln 2$, which also happens to be when $z = 0.5$, when the representation is most entropic.

This suggests that if we know ahead of time m and v, choosing k to maximize accuracy is a matter of plugging it into the formula and round-

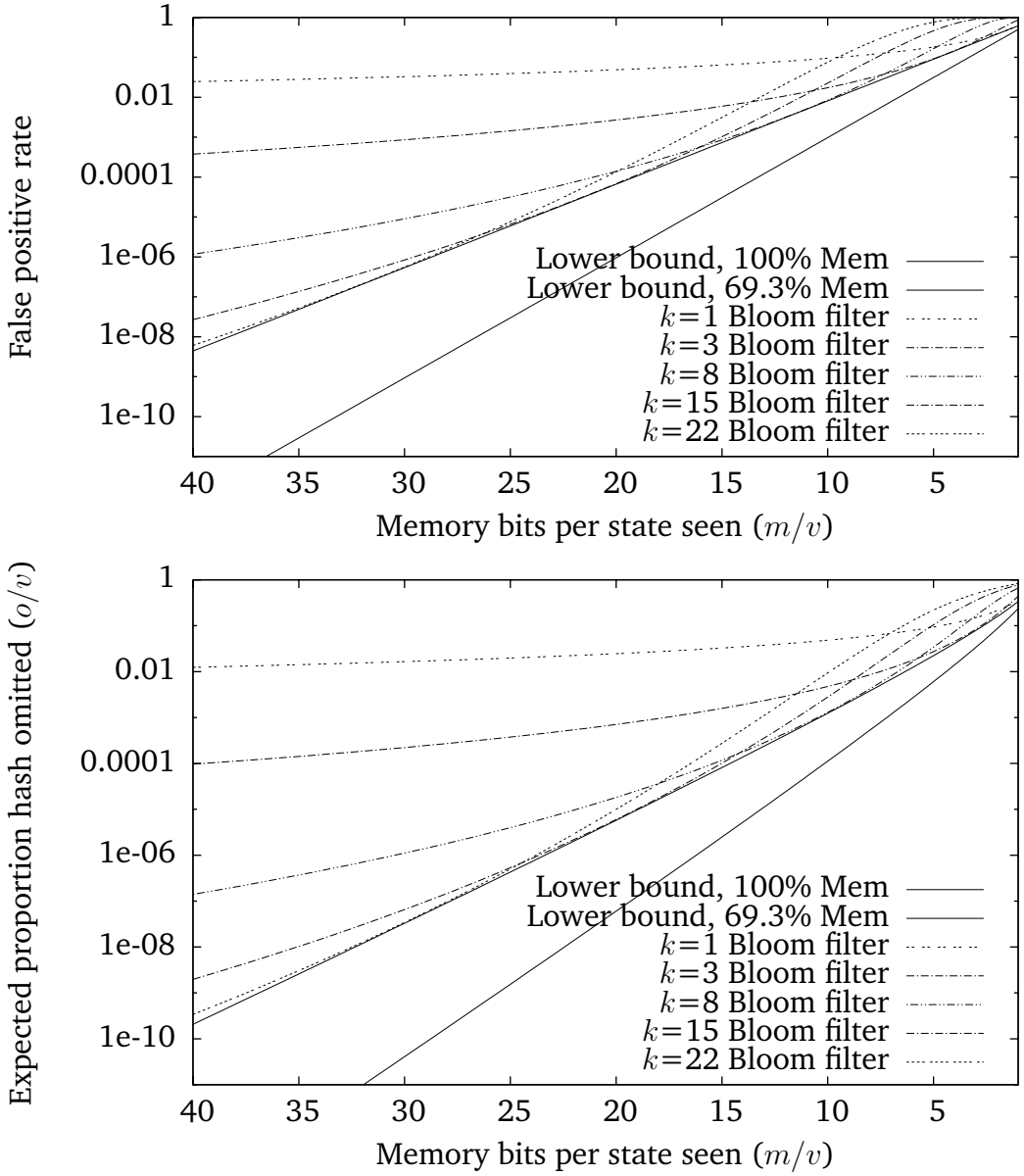

Figure 6.1: *Comparison of inaccuracy of Bloom filter configurations with information-theoretic lower bounds. It is known that a Bloom filter at its best is a memory factor* $\lg e \approx 1.44$ *from optimal [65], or, equivalently, as accurate as the optimal for* $1/\lg e = \ln 2 \approx 0.693$ *as much memory. That result applies to the false positive rate, as shown in the top graph, for several Bloom filter configurations. The bottom graph shows essentially the same bound holds for the expected hash omissions (or expected proportion of states hash omitted) also.*

ing to the nearest positive integer. Even if the formula were not based on an approximation, this method is flawed. The method finds the best non-discrete value of k and suggests the nearest discrete value of k should be the best, but the best discrete k is not always the one nearest the non-discrete k that minimizes the false positive rate formula. For example, according to Equation 6.5, $k = 1$ and $k = 2$ yield the same false positive rate when $m/v = 2.078$, but $\frac{m}{v} \ln 2 = 1.5$ when $m/v = 2.164$. If m/v is between those two values, the answer given by rounding the non-discrete k is likely to be suboptimal.

What about a generalization of the standard Bloom filter that implements non-discrete k? We do not know of such a structure that actually offers improvement over the standard Bloom filter. Suppose we implement a structure that supports non-discrete k by setting $\lfloor k \rfloor$ indices for $\lfloor k+1 \rfloor - k$ proportion of the additions and $\lfloor k+1 \rfloor$ for the remaining $k - \lfloor k \rfloor$ proportion. This makes the expected proportion of bits being "0" after v additions satisfy Equation 6.2 for real-valued $k \geq 1$. The false positive rate is very close to $\tilde{f} = (\lfloor k+1 \rfloor - k)(1-\hat{z})^{\lfloor k \rfloor} + (k - \lfloor k \rfloor)(1-\hat{z})^{\lfloor k+1 \rfloor}$. Consider once again the example of $m/v = 2.078$, when $k = 1$ and $k = 2$ yield approximately the same false positive rate. Using the above scheme for non-discrete k, it is easy to see graphically that the false positive rate is *worse* for every k between 1 and 2 than it is for either 1 or 2, as in Figure 6.2. Although I have not confirmed it formally, it is easy to be convinced that in any case, the minimum false positive rate for this structure will occur for a discrete value of k, unlike Equation 6.5.

An efficient way of implementing non-discrete k for Bloom filters would be an interesting and impressive piece of work.

Stuck in the discrete world, a better way to pick the best k would be to precompute the m/v values which are the boundaries between the regions of a particular k minimizing the false positive rate given by Equation 6.5. This approach is strikingly simple and has been done in the left side of Figure 6.3

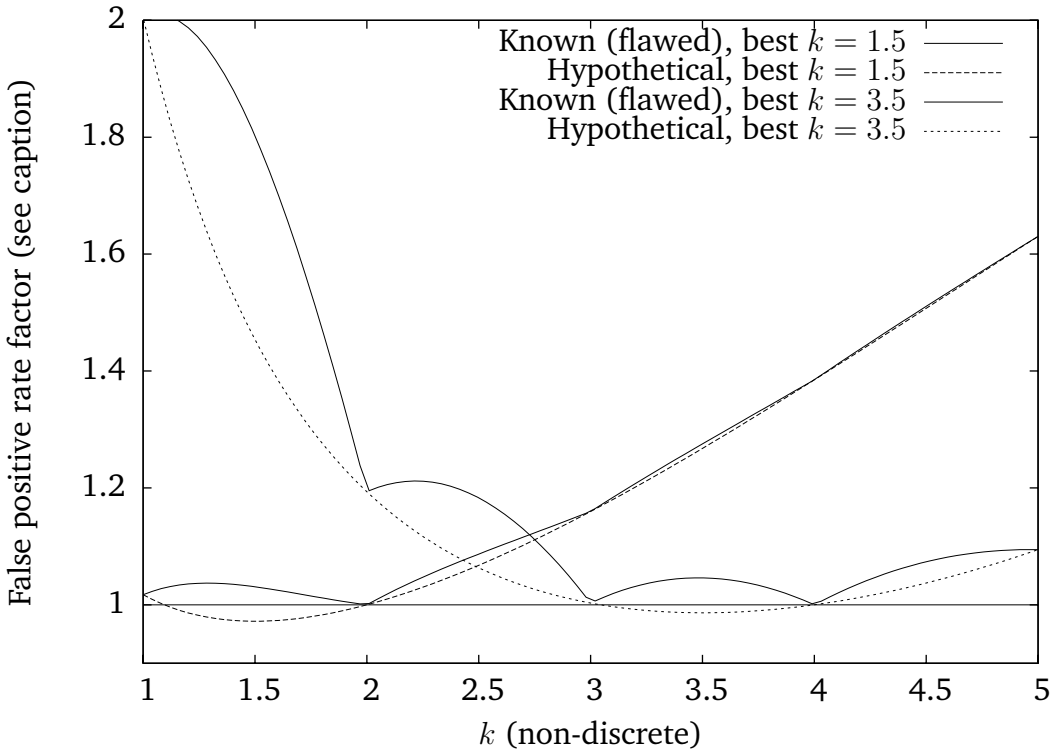

Figure 6.2: *Accuracy of a flawed method for using non-discrete k in a Bloom filter compared to optimistic expectations. The expected false positive rates are computed from formulas and then each curve is normalized against the false positive rate of the best known implementation, which is to use the best discrete k. The "Known (flawed)" curves are based on the expected false positive rates for a flawed approach to setting a non-discrete number of indices per element in a Bloom filter. The "Hypothetical" curves are based on using Equation 6.5 with non-discrete k. Using $k = \frac{m}{v} \ln 2$, $\frac{m}{v} = 2.164$ makes the best non-discrete $k = 1.5$, and $\frac{m}{v} = 5.049$ makes the best non-discrete $k = 3.5$.*

Minimize false positive rate			Minimize expected hash omissions			
$k =$	m/v	$\frac{m}{v}\ln 2$ err†	$k =$	m/v	$\frac{m}{v}\ln 2$ err‡	Eqn 6.6 err‡
1 and 2	2.07809	1.718%	1 and 2	1.13459	32.31%	0.315%
2 and 3	3.55619	0.600%	2 and 3	2.34809	20.72%	0.082%
3 and 4	5.01353	0.303%	3 and 4	3.64409	15.37%	0.073%
4 and 5	6.46426	0.183%	4 and 5	4.98501	12.24%	0.042%
5 and 6	7.91206	0.122%	5 and 6	6.35288	10.18%	0.022%
6 and 7	9.35826	0.087%	6 and 7	7.73819	8.717%	0.012%
7 and 8	10.8035	0.066%	7 and 8	9.13545	7.625%	0.006%
8 and 9	12.2482	0.051%	8 and 9	10.5413	6.777%	0.004%
9 and 10	13.6924	0.041%	9 and 10	11.9534	6.099%	0.003%
10 and 11	15.1364	0.033%	10 and 11	13.3703	5.545%	0.003%
11 and 12	16.5801	0.028%	11 and 12	14.7910	5.084%	0.004%
12 and 13	18.0237	0.024%	12 and 13	16.2147	4.693%	0.004%
13 and 14	19.4671	0.020%	13 and 14	17.6409	4.359%	0.005%
14 and 15	20.9105	0.018%	14 and 15	19.0689	4.069%	0.006%
15 and 16	22.3537	0.015%	15 and 16	20.4987	3.815%	0.006%
16 and 17	23.7969	0.014%	16 and 17	21.9298	3.592%	0.007%
17 and 18	25.2400	0.012%	17 and 18	23.3621	3.393%	0.007%
18 and 19	26.6831	0.011%	18 and 19	24.7954	3.215%	0.008%
19 and 20	28.1261	0.010%	19 and 20	26.2295	3.054%	0.008%
20 and 21	29.5692	0.009%	20 and 21	27.6645	2.909%	0.009%
21 and 22	31.0121	0.008%	21 and 22	29.1999	2.777%	0.009%
22 and 23	32.4551	0.007%	22 and 23	30.5361	2.657%	0.009%
23 and 24	33.8981	0.007%	23 and 24	31.9728	2.547%	0.009%
24 and 25	35.3409	0.006%	24 and 25	33.4099	2.445%	0.009%
25 and 26	36.7839	0.006%	25 and 26	34.8474	2.351%	0.009%
26 and 27	38.2267	0.005%	26 and 27	36.2852	2.264%	0.010%
27 and 28	39.6696	0.005%	27 and 28	37.7234	2.184%	0.010%
28 and 29	41.1124	0.005%	28 and 29	39.1619	2.109%	0.010%
29 and 30	42.5553	0.004%	29 and 30	40.6006	2.039%	0.010%
30 and 31	43.9981	0.004%	30 and 31	42.0396	1.973%	0.010%
31 and 32	45.4410	0.004%	31 and 32	43.4787	1.911%	0.010%
32 and 33	46.8838	0.003%	32 and 33	44.9181	1.854%	0.010%

Error percentages are maximum relative error in †false positive rate or ‡expected hash omissions due to choosing the best k with the given approximation scheme.

Figure 6.3: *Comparison of methods for choosing best Bloom filter k. On the left are the m/v values which are the boundaries between regions for which a particular k minimizes the false positive rate of a Bloom filter. For example, $k = 3$ is optimal if $3.55619 \leq m/v \leq 5.01353$. The "$\frac{m}{v}\ln 2$ err" tells how much the false positive rate could deviate if choosing k by rounding $\frac{m}{v}\ln 2$ to the nearest positive integer. The right is similar, except that expected hash omissions are minimized and the relative error in expected hash omissions for two approximation schemes is shown. m/v values are given with six significant digits.*

for k up to 32. Aside from limited significant digits, this approach gives the best answer possible using only Equation 6.5, which is quite accurate for Bloom filters larger than, say, a kilobyte. The $\frac{m}{v} \ln 2$ result is based on Equation 6.5 as well but can choose a k with a false positive rate up to 1.718% higher than the optimal, as demonstrated in Figure 6.3.

The most accurate way of picking k to minimize the false positive rate, however, still involves picking the one for which Equation 6.3 is minimized. Since the formula is considered undefined for non-discrete k, it cannot be minimized with traditional symbolic methods. We do know, however, that any local minimum in the false positive rate versus k is a global minimum, and we also know that the best k will be either $\lfloor \frac{m}{v} \ln 2 \rfloor$ or $\lceil \frac{m}{v} \ln 2 \rceil$ (except perhaps for extremely small m). Thus, instead of rounding $\frac{m}{v} \ln 2$, one could use it to determine which two discrete k to pick between.

6.3.2 Expected hash omissions, known v and m

For a visited set, one should pick k to minimize the *a priori* expected hash omissions, since this is the best way we know to minimize total omissions and, thus, maximize coverage as well. We do not have a closed form formula for expected hash omissions from a Bloom filter, so minimization might seem to require a lot of computation. In reality, we can do one set of computations and reuse those.

As with the false positive rate, the k that minimizes expected hash omissions depends only on the ratio between m and v (for large m). Equation 6.5 helps us out here, but we should confirm this result analytically. First, the k that minimizes $\hat{o}$ also minimizes $\hat{o}/v$, the expected *proportion* of additions hash omitted. Based on that reduction, it is sufficient to confirm that minimizing $\hat{o}/v$ for a given m and v is very close to minimizing $\hat{o}/(cv)$ for cm and cv, which has been scaled by some constant c but has the same memory per

added element:

$$\begin{aligned}
& \frac{1}{v}\hat{o}_{\mathsf{BF}}(m, v, k) \\
\approx\ & \frac{1}{v}\sum_{i=0}^{v-1}(1 - e^{-ki/m})^k \\
\approx\ & \frac{1}{v}\int_0^{v}(1 - e^{-ki/m})^k di \\
=\ & \frac{1}{cv}\int_0^{cv}(1 - e^{-ki/cm})^k di \\
\approx\ & \frac{1}{cv}\sum_{i=0}^{cv-1}(1 - e^{-ki/cm})^k \\
\approx\ & \frac{1}{cv}\hat{o}_{\mathsf{BF}}(cm, cv, k)
\end{aligned}$$

Consequently, we can precompute the m/v values that are boundaries between regions for which a particular k minimizes expected hash omissions, which we have done in the right side of Figure 6.3. We also see there that the scheme for approximating the best k to minimize the false positive rate does not transfer well to minimizing expected omissions, since it can result in up to 32.31% higher expected hash omissions. We have constructed a reasonably simple function that fits these boundary values much more closely:

$$\text{Choose } \ k = \left\lceil 3.8^{(m/v+4.2)^{-1}} \frac{m}{v} \ln 2 \right\rceil \tag{6.6}$$

As shown in the right-most column of Figure 6.3, choosing k based on this formula can increase expected hash omissions by no more than a third of one percent versus using the actual best k.

6.3.3 Unknown v

If we do not know how many elements will be added to a Bloom filter, optimizing it is tricky. Any k we pick is going to favor some ratios of m and

v over others. One way to resolve this is if there is a "threshold" accuracy, above which is generally "good" and below which is generally "bad".

For example, we might have an application in which we reconstruct the Bloom filter with more memory each time the false positive rate gets above 1%. In this case, we want to maximize the cases in which the structure has below 1% false positive rate. Given m, this is not too hard.

When using the Bloom filter as a visited set, we have a similar conundrum. If, for example, we want the best chance of exhaustive exploration of the state space, we would choose k such that the most cases as possible expect less than one hash omission. Some computation has shown that k to be close to $\ln m$.

Holzmann's approach to the problem of unknown v is optimized for finding bugs quickly. By optimizing for the worst case scenario—when there is very little memory for each state—Holzmann guarantees that the number of states visited is going to be near the maximum possible, even when there is moderate memory per state. The number of states omitted might be orders of magnitude higher than would be possible with a different configuration, but that difference is small in comparison to the number of states visited. This is shown in Chapter 2, including Figure 2.1.

The interesting thing about the Holzmann approach is that the best configuration to use when memory is most constrained is usually *not* $k = 1$. It is sometimes not even $k = 2$ or $k = 3$. The actual best configuration in this case depends on the connectivity of the state space being explored, because the search usually starves for known new states before the Bloom filter is saturated enough for $k = 1$ to be the best. Holzmann used to advocate using $k = 2$ by default but later switched to advocating $k = 3$. I suspect the change was associated with common use of partial order reductions, which seem to reduce the connectivity of a graph and, thus, hasten starvation of the inexact search. See Holzmann's papers [38, 40] and 2003 book [42] for more details.

6.4 Speed and fingerprinting

6.4.1 History

There has been a long-standing assumption that Bloom filters could not be simultaneously compact, accurate, and fast, because of the large amount of hash computation required for highly accurate, compact configurations, which have large k. In a 2001 paper, Mitzenmacher made this observation [62]:

> [I]t is worth noting that there are *three* fundamental performance metrics for Bloom filters that can be traded off: computation time (corresponding to the number of hash functions k), size (corresponding to the array size m), and the probability of error (corresponding to the false positive rate f).

In his 2003 book, Holzmann states the impact on a model checker [42]:

> In a well-tuned model checker, the run-time requirements of the search depend linearly on k: computing hash values is the single most expensive operation that the model checker must perform. The larger the value of k, therefore, the longer the search for errors will take. In the model checker SPIN, for instance, a run with $k = 90$ would take approximately 45 times longer than a run with $k = 2$. ... The question is then how much quality we sacrifice if we select a smaller than optimal value of k.

It was widely assumed that the k indexes could only come from independent hash functions without a significant impact on accuracy. Beginning with a 2004 paper in the SPIN Workshop, we showed that once two or three indices are computed from hash functions, the rest can be computed with simple arithmetic on those, removing most of the time cost associated with

larger values of k [24, 23]. Our basis for these approaches, largely for analytical purposes, is the fingerprinting Bloom filter.

6.4.2 Fingerprinting Bloom filter

Rather than computing each index using an independent hash function on the input element, a **fingerprinting Bloom filter** computes an intermediate hash fingerprint from which all the indices are computed. Having these two levels of hashing can reduce quite significantly the hash computation required. If the input element is $|x|$ words long, the hash fingerprint is $|\phi|$ words long, and each index is one word, then a fingerprinting Bloom filter requires $|x| \cdot |\phi| + |\phi| \cdot k$ units of hash computation where the standard Bloom filter requires $|x| \cdot k$ units of hash computation. Thus, if $|x| = 20$ words, $|\phi| = 2$ words, and $k = 12$, then the fingerprinting Bloom filter requires just 64 units of time hashing where the standard requires 240.

The remaining question is whether it is reasonable to base many Bloom filter indices on a fingerprint that only has two indices worth of entropy in it. To answer this question, we analyze what can lead to false positives in a fingerprinting Bloom filter. If we assume that the indices are computed using independent hash functions that operate on the fingerprint, then there are just two sources of false positive queries: a **fingerprint false positive** occurs when all indices probed are set to "1" because an element that happened to have the same hash fingerprint was added previously; a **filter false positive** occurs when all indices probed are set to "1" by elements with different hash fingerprints. Assuming the indices are computed using independent, random hash functions on the fingerprint, the filter false positive rate is exactly as in a regular Bloom filter with the same m, v, and k. The fingerprint false positive rate depends on s, the number of possible hash fingerprints:

$$\hat{f}_{\mathsf{FP}}(v, s) \stackrel{(def)}{=} 1 - \left(1 - \frac{1}{s}\right)^{v} \tag{6.7}$$

But when v is large, the fingerprint false positive rate essentially only depends on v/s:

$$\tilde{f}_{\mathsf{FP}'}(\tfrac{v}{s}) \stackrel{(def)}{=} 1 - e^{-v/s} \approx \hat{f}_{\mathsf{FP}}(v, s) \tag{6.8}$$

The overall false positive rate of a fingerprinting Bloom filter is the probability of either:

$$\tilde{f}_{\mathsf{FPBF}}(m, v, k, s) \stackrel{(def)}{=} \hat{f}_{\mathsf{FP}}(v, s) \oplus \tilde{f}_{\mathsf{BF}}(m, v, k) \tag{6.9}$$

where we use $\oplus$ as shorthand for the probability of the union of independent probabilities:

$$p \oplus q = p + q - pq = 1 - (1 - p)(1 - q)$$

Equation 6.9 is an approximation because $\tilde{f}_{\mathsf{BF}}$ is an approximation and because the expected number of unique fingerprints added to the underlying Bloom filter is less than v due to potential overlapping. The latter effect is usually negligible and could only make the fingerprinting Bloom filter look worse analytically.

We can use a simpler approximation to show how the fingerprinting Bloom filter scales. In Equation 6.5, we used m/v as a parameter instead of m and v individually, because m/v is scale-independent. The scale-independent variant of s is s/m, which is better understood in terms of its logarithm: $\lg(s/m) = \lg s - \lg m$ is how many more bits are in a fingerprint than in a Bloom filter index.

$$\begin{aligned} \tilde{f}_{\mathsf{FPBF}'}(\tfrac{m}{v}, k, \tfrac{s}{m}) &\stackrel{(def)}{=} \tilde{f}_{\mathsf{FP}'}(\tfrac{v}{\cancel{m}}, \tfrac{\cancel{m}}{s}) \oplus \tilde{f}_{\mathsf{BF}'}(\tfrac{m}{v}, k) \\ &\approx \tilde{f}_{\mathsf{FPBF}}(m, v, k, s) \end{aligned} \tag{6.10}$$

From this approximation, it is obvious that if we scale m, v, and s each by a constant c, the resulting structure should have approximately the same false positive rate. If we assume the size of a fingerprint is the size of an

index plus some fixed amount, then the false positive rate depends only on the ratio of m and v. As a corollary, if the fingerprint is the size of two or more indices, the false positive rate due to fingerprinting approaches zero if we scale up m and v while maintaining their ratio.

Figure 6.4 shows the interaction between $\tilde{f}_{\mathsf{FP}}$ and $\tilde{f}_{\mathsf{BF}}$ in the false positive rate of a fingerprinting Bloom filter. Basically, when few elements have been added, false positives due to fingerprinting are most likely. When many elements have been added, false positives due to the underlying Bloom filter are most likely. The v/m at which fingerprinting begins to have a negligible impact depends on s/m, or how much larger than an index a fingerprint is ($\lg(s/m)$ bits).

For those interested in using a Bloom filter at a particular v/m, the fingerprint can be made large enough to have a negligible impact. For example, $k = 3$ is optimized for v/m near 0.2; Figure 6.4 shows that using just $\lg(s/m) = 6$ extra bits for the fingerprint puts $v/m \approx 0.2$ well into the territory of negligible impact.

For those interested in a Bloom filter remaining below a certain false positive rate, the fingerprint can (similarly) be made large enough to have a negligible impact. That same example remains below a 1% false positive rate for nearly the same set of v/m values.

In practice, with stock hash functions, we make sure the fingerprint is big enough, but utilize *all* final hash information computed by the hash function(s). Basically, more fingerprint is more accurate, and fast index computation techniques in Section 6.5 can compute indices quickly from a reasonably large fingerprint.

6.4.3 Hash-extending Bloom filter

An optimization of the fingerprinting Bloom filter is to use the fingerprint as-is in computing one or more indices. Strangely, this optimization im-

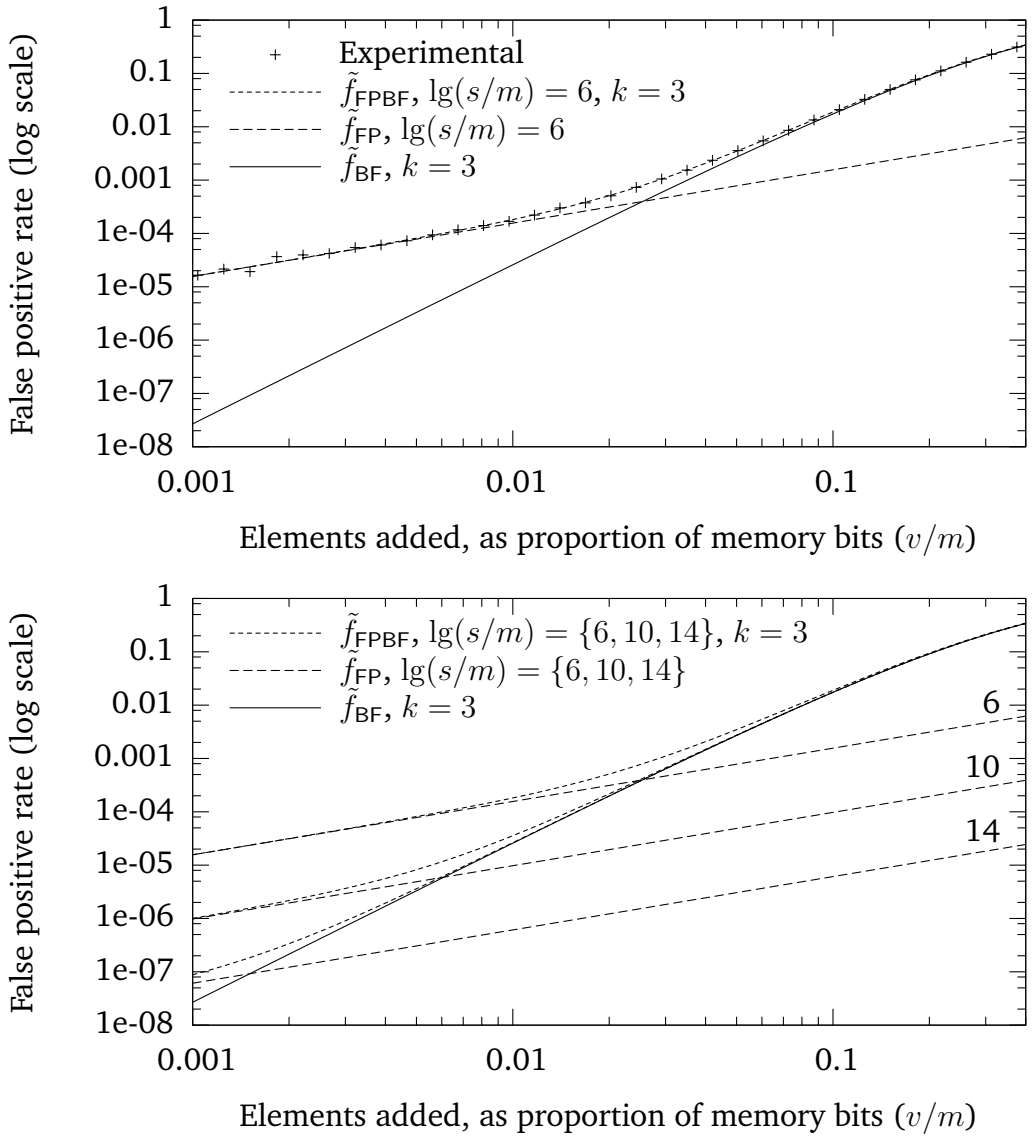

Figure 6.4: *How fingerprinting affects the false positive rate of a Bloom filter. The expected false positive rate of a fingerprinting Bloom filter ($\tilde{f}_{\mathsf{FPBF}}$) is roughly the sum of the false positive rate due to fingerprinting ($\tilde{f}_{\mathsf{FP}}$) and the false positive rate due to the underlying Bloom filter ($\tilde{f}_{\mathsf{BF}}$). Both axes use a logarithmic scale. In all these examples, we set $k = 3$ bits per element and used $m = 2^{16}$ bits of memory, though the graphs look the same for any large m. The fingerprint was $\lg(s/m)$ = 6, 10, or 14 bits larger than one index. Each point of "Experimental" came from simulating $4\,000\,000$ queries against a fingerprinting Bloom filter by querying that many random fingerprints. Indices were computed from each fingerprint using a Jenkins hash.*

proves not only speed, but also accuracy. I call this construction the "hash-extending Bloom filter," illustrated in Figure 6.5. The only difference from a fingerprinting Bloom filter is that we use the initial hash value (or "fingerprint") directly to get indices before computing other hash values from it. If the fingerprint is not much larger than one index and k is small, this approach has a distinguishably *lower* false positive rate than the ordinary fingerprinting Bloom filter, despite using less hashing.

The analysis is affected because the two false positive probabilities (fingerprinting and Bloom filter) are not *as* independent. When we consider false positives in the underlying Bloom filter, we are assuming the fingerprint did not match a previously added fingerprint. A non-randomized relationship between the fingerprint and the first index means that that assumption makes the first index less likely to collide with other first indices and, thus, less likely to collide with other indices overall.

To think about this formulaically, let us first consider an **a**lternate parameterization of the false positive rate of a Bloom filter:

$$\tilde{f}_{\mathsf{BF}a}(m,a,k) \stackrel{(def)}{=} \left(1-\left(1-\frac{1}{m}\right)^{a}\right)^{k} \tag{6.11}$$

$$\tilde{f}_{\mathsf{BF}a}(m,vk,k) = \tilde{f}_{\mathsf{BF}}(m,v,k)$$

vk is the number of bits, potentially overlapping, that have been set to "1" because of the v additions.

Now consider that we can divide up the false positive rate of a Bloom filter as the probability the first index colliding (already set to "1") and the probability of the rest colliding:

$$\tilde{f}_{\mathsf{BF}a}(m,vk,k) = \tilde{f}_{\mathsf{BF}a}(m,vk,1)\tilde{f}_{\mathsf{BF}a}(m,vk,k-1) \tag{6.12}$$

In each case, any one (or more) of the vk bits previously set to "1" might cause the collision.

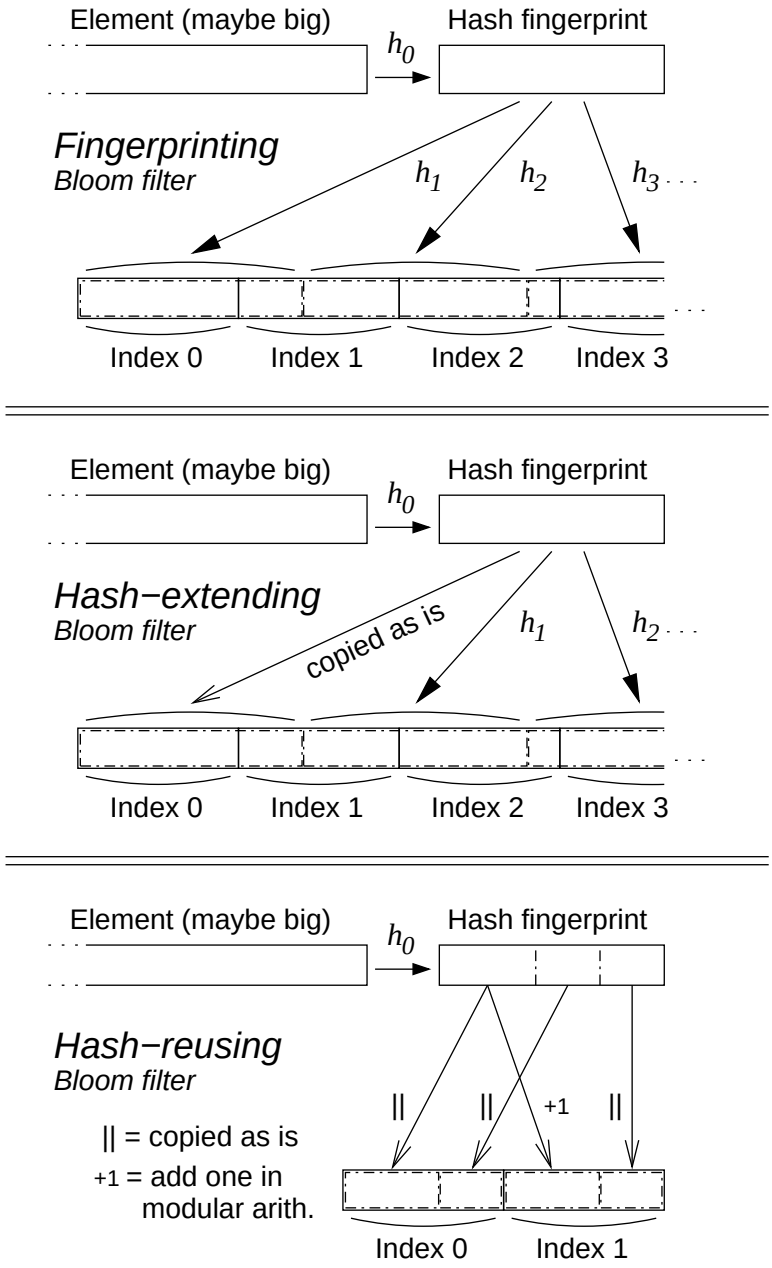

Figure 6.5: *Comparison of index computation in "fingerprinting," "hash-extending," and "hash-reusing" Bloom filters.* $h_0, h_1, \ldots$ *are independent hash functions. In this depiction, the hash value returned by each hash function is a little bigger than one index into the Bloom filter.*

The probability of the first index colliding can be divided between the probability of it colliding with another first index or with a non-first index:

$$\tilde{f}_{\mathsf{BF}a}(m, vk, 1) = \tilde{f}_{\mathsf{BF}a}(m, v, 1) \oplus \tilde{f}_{\mathsf{BF}a}(m, v(k-1), 1) \tag{6.13}$$

When considering the underlying Bloom filter false positive in a hash-extending Bloom filter, the assumption of the uniqueness of the fingerprint makes the first index less likely to have collided with previous first indexes. In particular, this index only has $s/m - 1$ fingerprints associated with it that are different from the current fingerprint, while all the other indices have s/m. The effect on the likelihood of a first index to first index collision is similar to adding only $\frac{s/m-1}{s/m} = 1 - m/s$ as many elements.

Thus, to get the false positive rate for a hash-extending Bloom filter, we replace $\tilde{f}_{\mathsf{BF}a}(m, v, 1)$ with $\tilde{f}_{\mathsf{BF}a}(m, v[1 - m/s], 1)$. Then using Equation 6.13 as a lemma, we get

$$\begin{aligned}&\tilde{f}_{\mathsf{BF}a}(m, v[1 - m/s], 1) \oplus \tilde{f}_{\mathsf{BF}a}(m, v[k-1], 1)\\ &\quad = \tilde{f}_{\mathsf{BF}a}(m, v[k - m/s], 1)\end{aligned}$$

Thus, the false positive rate of the hash-extending Bloom filter is approximated like so:

$$\begin{aligned}\tilde{f}_{\mathsf{HEBF}}(m, v, k, s) &\overset{(def)}{=} \tilde{f}_{\mathsf{FP}}(v, s) \oplus\\ &\tilde{f}_{\mathsf{BF}a}(m, v[k - m/s], 1)\tilde{f}_{\mathsf{BF}a}(m, vk, k-1)\end{aligned} \tag{6.14}$$

We only consider the first index here because if the fingerprint is significantly larger than one index, such as the size of two indices, then the overall effect is negligible (m/s is tiny) and the fact that more than one index is affected does not matter. Note that our Bloom filter hashing schemes based on double hashing (Section 6.5) are actually more like hash-extending than plain fingerprinting, but the difference is negligible assuming the fingerprint is the size of two or more indices. ($\lg s = 2 \lg m \Rightarrow m/s = 1/m.\ k - 1/m \approx k$)

6.4.4 Hash-reusing Bloom filter

Even cheaper than using the fingerprint literally as *some* index information is to reuse parts of it to get *all* of the index information, as shown in Figure 6.5. We call this approach "hash-reusing." We really only consider the $k = 2$ case because there are unexplored nuances for larger k, and $k = 2$ is the only case needed for my adaptive state storage scheme, in Chapter 11.

Here is how it works. Like the hash-extending Bloom filter, the first index is exactly some "prefix" of the fingerprint. To get the second index, we take what remains of the fingerprint and replace that much information at the "end" of the first index. That is the basic idea, but it requires a tweak because of a tremendous flaw: the second index will often overlap with the first! All it takes is for the suffixes to be the same; thus, this can dramatically raise the false positive rate. This problem can be rectified by guaranteeing uniqueness between the two indices, by adding 1 to the prefix re-used for the second index (mod the range of values). Figure 6.5 depicts this better solution.

Note that none of the Bloom filters previously discussed guarantee that the k indices associated with an index are unique. For the kinds of large Bloom filters of primary interest here, uniqueness of the indices is insignificant—except in the hash-reusing Bloom filter. This design is an exception because it uses/assumes no additional hashing to compute indices from a fingerprint. Basically, the easiest pit to fall into with pseudo-random Bloom filter indices is to have an abnormally high probability of overlap within the k indices for an element. An easy way to avoid this pitfall is to build uniqueness into the design, which also proves useful for double hashing in Bloom filters (see Section 6.5.1). Interestingly, the easiest method of guaranteeing uniqueness in a standard Bloom filter, confining the ith index of each element to the ith region of a k-partition of the bit vector, reduces the false positive rate slightly [9]. Because the hash-reusing Bloom filter is

already working with reduced entropy, guaranteeing uniqueness in a similar way seems to confer an advantage in this case.

Analytically, this is a simple extension of the hash-extending Bloom filter. Basically, both indices computed by this scheme benefit from the assumption of fingerprint uniqueness, because both the first and the second index are effectively a literal piece of the fingerprint. Thus, each of the two indices has the same favorable collision probability that the first index had in the hash-extending Bloom filter. (The second index is less likely to collide with a previous second index, just as a first index is less likely to collide with a previous first index, because of the assumption of fingerprint uniqueness.) Here is the equation for $k = 2$ (using Equation 6.12 as a lemma):

$$\tilde{f}_{\mathsf{HRBF}}(m, v, 2, s) \stackrel{(def)}{=} \tilde{f}_{\mathsf{FP}}(v, s) \oplus \tilde{f}_{\mathsf{BF}a}(m, v[2 - m/s], 2) \tag{6.15}$$

If only a small number of bits are replaced to get the second index, the two indices are guaranteed to be close to each other. This normally has an adverse affect on the false positive rate of a Bloom filter (see e.g. [72]), but under the pretense of limited hash information being available, the hash-reusing approach is actually superior. The speed advantage of this locality is discussed in Section 11.2.4.

6.4.5 Empirical validation

Accuracy For purposes of comparison with each other and with formula predictions, we have implemented the above schemes in a way that allows us to simulate a series of ADD and QUERY operations simply using random values as fingerprints.

A simple validation of the fingerprinting Bloom filter's accuracy is already shown in Figure 6.4.

Next, we examine the case in our adaptive storage scheme (from Chap-

ter 11) when fingerprinting or hash-reusing will have the largest impact on accuracy. We know fingerprinting has the largest relative impact on accuracy when the number of additions is small relative to memory; thus, we need to examine the Bloom filter's accuracy right after creating it from the 8-bit-per-cell Cleary table. Assuming we convert after the table is 80% full, the number of elements in the Bloom filter will already be about 1/10th the number of memory bits. In that case we get these false positive rates:

Technique	Theoretical	Experimental
Standard†	0.03286	0.03284
Fingerprinting	0.04488	0.04450
Hash-extending	0.04303	0.04279
Hash-reusing	0.04129	0.04130

† Not compatible with our adaptive storage scheme.

These results are actually the "1/10" results from Figure 6.6, which has additional results and a different way of presenting the false positive rates. The experimental setup is described in the caption. These results are just representative of many runs and trials not recorded here. I was also careful to validate the "random" input, which XORs results from two pseudorandom generators and a large entropy pool.

The empirical results confirm the strange analytical result: when the hash information available is limited, using no additional hashing to compute the indices ("hash-reusing") can be better than additional hashing ("fingerprinting"). Using limited additional hashing ("hash-extending") gives results between those two. The standard Bloom filter has the best accuracy, but requires the most hashing (and is not compatible with my adaptive storage scheme in Chapter 11).

Speed The speed advantages of using fingerprinting instead of an independent hash function for each index should be obvious, and are well docu-

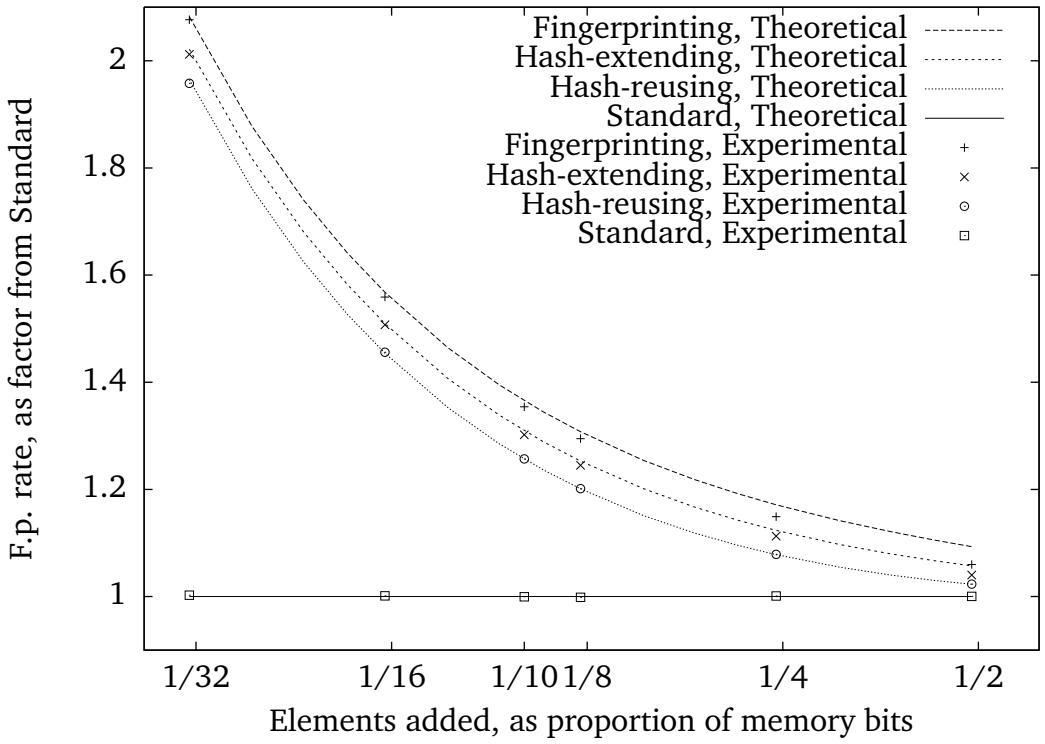

Figure 6.6: *Comparison of false positive rates of three kinds of Bloom filters based on limited hash information with a standard Bloom filter. Lower is better. "Theoretical" results come from formulas and "experimental" results represent 20 million random queries on at least 625 different structures. All use $k = 2$ and $m = 2^{16}$, though m was chosen simply to be large enough that the results generalize to larger structures, confirmed by re-running the experiments with $m = 2^{18}$. The "1/10" results represent the smallest v/m the hash-reusing Bloom filter will encounter as a part of the adaptive storage scheme, meaning the special $k = 2$ Bloom filter has at worst a 30% higher false positive rate in that scheme compared to a standard $k = 2$ Bloom filter.*

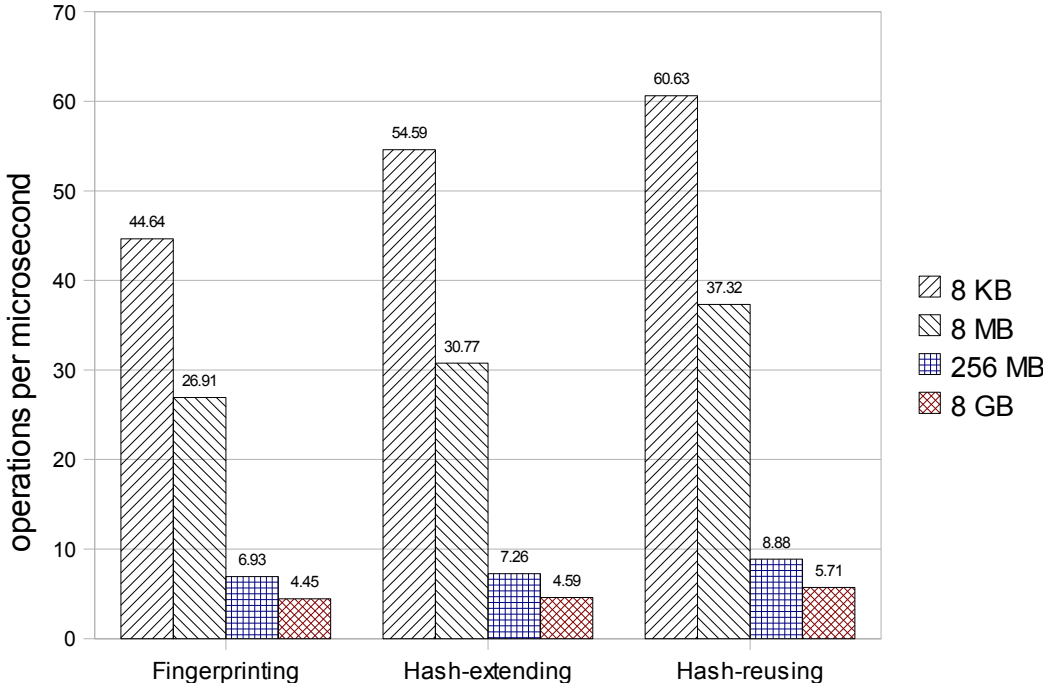

Figure 6.7: *Comparison of the speed of "fingerprinting," "hash-extending," and "hash-reusing" Bloom filters. Higher is better. The y-axis is the average speed of each add and query operation among tens of millions of random queries against a pre-populated structure ($z \approx 0.5$) and the same number of random additions to that structure. All the cases use $k = 2$, because that is what the hash-reusing Bloom filter supports. Tests were compiled with gcc 4.4.3 (-O3) and run on a 64-bit Linux system with Intel Xeon X5677 CPUs (3.47GHz, 32KB L1 data cache, 12MB L3 cache). See text for more information on the implementation and analysis of the results.*

mented in my publications [24, 23].

The more interesting speed issues have to do with comparing the three special kinds of Bloom filters described in this section. Figure 6.7 tells us a lot about the speed characteristics of the various approaches by comparing their essentially ideal speed over wildly different memory sizes. I say the results are essentially "ideal" speed because the cost of obtaining a fingerprint from hypothetical input is near minimal, because they are obtained from glibc's random() function, which is quite fast and adequately random for speed simulations. Also, the Jenkins-derived hash function [52] for computing

indices from the fingerprint is also quite fast, because it hashes just one 64-bit word into another, with no multiplication, division, or branching.

In analyzing the results in Figure 6.7, keep in mind these statistics:

$k = 2$ Bloom filter	Hashed indices	Trivial indices	Index locality?
Fingerprinting	2	0	no
Hash-extending	1	1	no
Hash-reusing	0	2	yes

In other words, the fingerprinting Bloom filter is slowest because it has to compute both indices using hash functions on the fingerprint, and the two indices are not guaranteed to be near each other. The hash-extending Bloom filter does not have locality between indices either, so its faster speed is due to replacing one hash function call with direct use of the fingerprint. Note that when main memory must be consulted for the operation (256MB and 8GB), and to a lesser extent L3 cache (8MB), the savings in hashing time are small compared to memory access time.

By contrast, the hash-reusing Bloom filter shows its biggest advantage when memory accesses are most expensive, because its indices are in adjacent words. (Reusing is more than 20% faster than extending for 8GB but only about 10% faster for 8KB.) Everything from the DRAM to the TLB to the on-chip cache is designed to take advantage of such locality.

It is interesting to note the difference in speed between 256MB and 8GB. This cannot be explained by a 12MB L3 processor cache, which would only hit about 1 in 20 accesses. Even if half the accesses to the 8GB space were instantaneous, it would not be as fast on average as the accesses in the 256MB space. This must be due to more TLB misses due to a much larger working set size, which increases the latency of random accesses. (The TLB is responsible for mapping virtual memory addresses to physical addresses, using page tables, which might be more extensive than the TLB is able cache.)

6.5 Fast index computation

Although the $k = 2$ hash-reusing Bloom filter is quite useful as-is, the hash-extending and fingerprinting Bloom filters have so far assumed that a sophisticated hash function is required to compute each additional index from the fingerprint. Here I describe several techniques requiring negligible computation time for each additional index but whose accuracy is almost indistinguishable from a hash-extending or fingerprinting Bloom filter.

For describing these techniques, we say that a Bloom filter uses k **index functions**, $g_1, \ldots, g_k$, to determine the indices associated with an element. In a standard Bloom filter, these are independent hash functions: $g_1(x) = h_1(x), \ldots, g_k(x) = h_k(x)$. In these Bloom filter variants, index functions will not be just hash functions.

Bibliographic Notes Double hashing is a well-known technique for computing probe sequences in open-addressed hash tables [17, Section 11.4]. For more detail, refer to Knuth [58] or Gonnet [34].

Our 2004 SPIN Workshop paper [24] appears to be the first to apply double hashing to Bloom filters, and introduced a variant called "triple hashing" to make better use of available hash information. Our FMCAD follow-up [23] introduced "enhanced double hashing" to address some of the shortcomings of using double hashing with Bloom filters, without requiring more hash information.

Kirsch and Mitzenmacher followed up on our work with a more formal analysis of index computation schemes like double hashing in Bloom filters [55, 56]. Others have also cited our Bloom filter work.

This section includes more unpublished ideas, details, and analyses of index computation in Bloom filters.

6.5.1 Double hashing

We can apply the "double hashing" method of computing probe sequences in hash tables to computing Bloom filter indices. It does not work especially well in this new application, but its weaknesses do motivate the "enhanced double hashing" and "triple hashing" techniques that follow.

To minimize the expected number of probes to locate an entry in an open-addressed hash table, one ideally uses a sequence of independent hash functions to choose the sequence of table locations to probe. Double hashing is an alternative that is nearly as effective but requires only two hash functions to compute a sequence of probe locations. The first hash function computes starting location (call it $a = h_1(x)$). The second hash function computes the difference between subsequent locations (call it $b = h_2(x)$). Specifically, any next location in the sequence is the current location plus b, modulo the number of locations.

We can also express the ith index in the sequence in closed mathematical form:

$$g_i = a + ib \pmod{m} \tag{6.16}$$

It is simple to apply this idea to computing Bloom filter indices. See Figure 6.8 (excluding parts in boxes) for the algorithm. We will call this the "naive" double hashing algorithm because it does not address Issue 1 below, which also has to be addressed when double hashing is used in open addressed hash tables. Issues 2 and 3 seem to be unique to use in Bloom filters, and are only addressed by "enhanced double hashing."

Issue 1 The standard issue with naive double hashing is that some possibilities for $b = h_2(x)$ can result in many repetitions of the same index in the computed sequence. Obviously, if $b = 0$, all the indices are the same, but if b and m have non-trivial common factors, then there might also be repetitions among the k computed indices. Consider, for example, $b = m/2$, in which

```
algorithm " [ Enhanced ] Double/[ Triple ] Hashing"
input x : "the element being added or queried"
input m : "number of bits in bit vector"
input k : "number of indices to compute"
output g[0 .. k - 1] : "array of indices"
uses h1, h2[, h3] : "hash functions"
begin
   a := h1(x)
   b := h2(x)
   [ c := h3(x) ]
   g[0] := a
   for i := 1 .. k - 1
   begin
      a := (a + b) MOD m
      [ b := (b + c) MOD m ]
      [ [ b := (b + i) MOD m ] ]
      g[i] := a
   end
end
```

Figure 6.8: *Algorithm for double, enhanced double, and triple hashing in Bloom filters. For double hashing, include no parts in boxes. For triple hashing, include the parts* [*in single boxes.*] *For enhanced double hashing, include the parts* [[*in double boxes.*]] *Note that the double and triple hashing algorithms are "naive" (see text).*

case the sequence of indices alternates between only two possibilities. Even thought these cases might seem rare, they can have a severe impact on the accuracy of a Bloom filter.

Specifically, because of linearity in i, if $g_i = g_{i+j}$, then also $g_i = g_{i+cj}$ for any integral c.

There are two standard ways of dealing with this first issue, both of which ensure that b is greater than zero and relatively prime to m. The first solution is to choose m to be prime and compute the hash b such that $0 < b < m$. The second solution is to make m a power of two and compute b to be odd. These concessions reduce the range of values for b, which reduces the entropy flowing into index computation. For example, ensuring b is odd cuts

the possibilities in half. In terms of a fingerprinting Bloom filter, this reduces the size of the fingerprint by one bit.

Implementation note: Despite a common assumption that a single machine instruction is always very fast, division/remainder operations are very slow on modern machines. My testing has shown that if division can be replaced by a conditional branch and some other ALU logic, it is worth the replacement [24, Section 5.2].

Issue 2 The second issue with double hashing, specific to Bloom filters, is that there are two ways to specify any set of indices generated by double hashing, one "forward" and one "backward." A key idea here is that unlike the open-addressed hash table, the Bloom filter does not really care about the order of the indices [1], treating them as a *set* rather than a *sequence*.

Given the pair $\langle a, b\rangle$, there is another pair $\langle a', b'\rangle$ generating the same set of indices, specifically,

$$a' = a + (k-1)b \pmod m$$

$$b' = m - b = -b \pmod m$$

(Note that the constraints to fix Issue 1 guarantee that $b' \neq b$.)

In effect, this issue erases one bit of the fingerprint, because there are only half as many sets of indices as there are $\langle a, b\rangle$ pairs. Eliminating the redundancy by (further) limiting the range of b is similar; it actually reduces the fingerprint size by one bit.

Issue 3 Finally, the double hashing Bloom filter is unusually prone to partial overlapping of the k indices among two elements. In an ideal fingerprinting Bloom filter, two fingerprints that are close to the same are no more likely to have indices overlapping than completely different fingerprints [2].

[1] Order matters only in the speed of negative queries, which can return upon encountering the first bit set to "0".

[2] The paradoxically superior hash-extending or hash-reusing Bloom filters are not as

By contrast, in a double hashing Bloom filter, two fingerprints that are close to the same are likely to have many indices overlapping.

Consider the pairs $\langle a, b\rangle$ and $\langle a', b'\rangle$ in which $b = b'$. In addition to the a' for which all k indices overlap with those for $\langle a, b\rangle$ ($a' = a$), there are are two a' for which $k-1$ indices overlap ($a' = a - b$ and $a' = a + b$), and two a' for which $k-2$ indices overlap ($a' = a - 2b$ and $a' = a + 2b$), and two a' for which $k-3$ indices overlap ($a' = a - 3b$ and $a' = a + 3b$), ...

Just as using fingerprinting in Bloom filter introduces an "anomalous" case in which all indices overlap with higher probably than normal, the double hashing design introduces "anomalous" cases in which three or more indices overlap with higher probably than normal. In a standard Bloom filter, the probability of two sets of indices overlapping by i indices would diminish exponentially in i, but in a double hashing Bloom filter, the probability of overlapping by any quantity from three to $k-1$ has virtually the same non-negligible probability.

It is only considered anomalous partial overlap if at least three indices are overlapping among two sets. The reason it needs to be three is tricky. There is nothing about the structure of double hashing that makes overlapping by two indices more likely than it is for a standard Bloom filter. Consider set of indices $i_1, i_2, \ldots, i_k$ and another set $j_1, j_2, \ldots, j_k$. Suppose i_1 overlaps with j_x. What are the chances of i_2 overlapping with j_y for some $y \neq x$? Well, for each such y there is a double hashing "b" that makes j_y overlap with i_2 (assuming m is prime; the argument is not exact if m is a power of two):

$$b = (i_2 - i_1)(y - x)^{-1} \pmod{m}$$

And the chances of b being the right one are the same as the chances of an independent hash function generating the index directly. Thus, there is nothing anomalous about overlapping by two indices with a double hashing

easy to use in this argument.

Bloom filter versus a standard Bloom filter. To overlap by three, the double hashing Bloom filter must use the same b, which greatly increases the probability of another overlap compared to an independent hash function.

Partial overlap must also be less than k indices of overlap, because the case of all k overlapping has been covered by the fingerprinting false positive rate and Issues 1 and 2. A consequence of these constraints is that Issue 3 only applies if $k \geq 4$. If $k = 3$, the only overlap by at least three is complete overlap.

The impact of this potential for partial overlap can be analyzed in terms of the probability of all indices overlapping with a single previous addition (call that probability g) and the proportion of bits set to "1" in the Bloom filter (call that proportion p). Working from Equation 6.9, the false positive rate should be approximately

$$\tilde{f}_{\mathsf{BF}}(m, v, k) \oplus \left(g + \sum_{i=1}^{k-3} 2gp^i \right)$$

In other words, other than the standard false positive rate of a Bloom filter, we have the probability of all indices overlapping with a previous addition and the false positive probability due to partial overlapping. Partial overlapping is relevant if 1 up to $k-3$ indices do *not* overlap with a previous addition. Each case is twice as likely as all indices overlapping with a previous addition ($2g$). The chances of a partial overlap by $k-i$ indices leading to a false positive is the probability that the i non-overlapping indices happen to have been set to "1" by other additions (p^i).

For example, if we are using a k close to the one that minimizes the false positive rate, p will be roughly $1/2$. This means that the probability of a false positive due to partial overlapping on the "left" is $1/2 + 1/4 + 1/8 + \ldots$ times g, just as on the "right" is $1/2 + 1/4 + 1/8 + \ldots$ times g. Thus, the chances of a false positive due to use of double hashing is, in a typical case, about three times higher than it would be if not for partial overlapping.

Note that this analysis does not include all the cases of partial overlap, but should include the dominant cases—those in which $b' = b$ (or $b' = m - b$ if allowed). For the less significant cases, consider $b' = b/2$, $b' = b'/3$, etc. These allow up to $k/2$, up to $k/3$, etc. indices to overlap. The impact on the false positive rate is not really significant compared to nearly all k overlapping, especially if $b' = b/2$ is precluded by requiring b to be odd for Issue 1.

Analysis Combining the impacts of Issues 1 through 3 results in the following approximate false positive rate for a Bloom filter using double hashing, which is the impact of Issue 1, times the impact of Issue 2, times the impact of Issue 3, times the probability of a two-index fingerprint false positive, "or" a standard filter false positive:

$$\begin{aligned} \tilde{f}_{\mathsf{DHBF}}(m, v, k) &\overset{(def)}{=} 2 \cdot 2 \cdot \left(1 + 2\sum_{i=1}^{k-3} p^i\right) \hat{f}_{\mathsf{FP}}(v, m^2) \ \oplus \ \tilde{f}_{\mathsf{BF}}(m, v, k) \\ &= \left(4 + 8\,\frac{p - p^{k-2}}{1 - p}\right) \hat{f}_{\mathsf{FP}}(v, m^2) \ \oplus \ \tilde{f}_{\mathsf{BF}}(m, v, k) \end{aligned} \tag{6.17}$$

where p is the proportion of "1" bits ($p \overset{(def)}{=} 1 - z \approx 1 - \hat{z}_{\mathsf{BF}}(m, v, k)$).

This assumes Issue 1 is resolved by ensuring b is odd; the corresponding factor of two can be removed if m is prime instead of a power of two. In that alternative case, simply ensuring b is non-zero has negligible impact.

Summary Computing Bloom filter indices with double hashing is extremely fast, as shown in Section 6.5.6, but its accuracy is a bit short of what we would expect from an ideal hash-extending Bloom filter with the same fingerprint size. Also, it can only utilize a fingerprint up to two indices in size. I address these limitations with the subsequent approaches.

6.5.2 Triple hashing

One way to counteract the issues associated with double hashing is to make use of a larger fingerprint, so that the loss of accuracy from those issues is less significant in absolute terms. There is a natural way to add another index-sized hash, c, to the index computation (see Figure 6.8), which adds a term of the next higher power to the equation describing the indices:

$$g_i = a + ib + \frac{(i)(i-1)}{2}c \pmod{m} \tag{6.18}$$

We call this natural generalization **triple hashing**. It is a generalization because it is the same as double hashing in the case of $c = 0$.

Triple hashing actually does very little to address the three issues with double hashing, aside from reducing their absolute impact on the false positive rate by starting with a larger fingerprint.

Issue 1 is certainly complicated by having a third variable in the equation. To have significant self-overlap in indices, the values of b and c have to work poorly together. For example, $c = 0$ is only a problem if b is 0, $m/2$, $m/3$, etc. Similarly, $b = 0$ is only a problem if c is 0, $m/2$, $m/3$, etc. Putting the same restrictions on either b or c as we placed on b in double hashing should keep repeated indices near normal levels, at the same relative cost: up to one bit of fingerprint.

Triple hashing does not fix Issue 2 either. It is just more difficult to see that, as before, there are two ways of specifying each set of indices. Distinct triples $\langle a, b, c\rangle$ and $\langle a', b', c'\rangle$ generate the same set of indices when

$$a' = a + (k-1)b + \frac{(k-1)(k-2)}{2}c \pmod{m}$$

$$b' = -b - (k-2)c \pmod{m}$$

$$c' = c$$

One can check by hand that as a transformation, this relationship is its own inverse. One can also push it through the algorithm to see that the same indices are generated, except in reverse order. It is interesting to note that $k-2$ is used in b' instead of $k-1$, which is because in the algorithm (see Figure 6.8), the reassignment to b comes after its use in the reassignment to a. The value assigned to b on the $i = k-1$ iteration is never actually used; the value on the $i = k-2$ iteration is the last one used to modify a. Also notice that c' is not the negation of c, which is essentially because the negative of a negative is a positive. (b' is considered "negative".)

Issue 3 is similarly not corrected by triple hashing. Starting with a $\langle a, b, c\rangle$ triple, if we iterate through the loop once, computing the first index, we get a triple $\langle a+b, b+c, c\rangle$ that as a starting triple would overlap by $k-1$ indices with the original. Like with double hashing, we can keep iterating to compute cases of $k-2$, $k-3$, ... indices overlapping.

Consequently, the approximate false positive rate for a triple-hashing Bloom filter is as the double-hashing Bloom filter, corrected for a fingerprint the size of three indices:

$$\tilde{f}_{\mathsf{THBF}}(m, v, k) \stackrel{(def)}{=} \left(4 + 8\,\frac{p - p^{k-2}}{1-p}\right) \hat{f}_{\mathsf{FP}}(v, m^3) \;\oplus\; \tilde{f}_{\mathsf{BF}}(m, v, k) \qquad (6.19)$$

where p is the proportion of "1" bits.

Summary Triple hashing was originally motivated by utilizing all available hash information when that's more than two indices worth [24, Section 3.4], and it is still not a bad choice when that is the case.

6.5.3 Improved double hashing

We can use the triple hashing algorithm to make a modest improvement to the double hashing algorithm. I call this **improved double hashing**, but it is mostly presented to motivate **enhanced double hashing**, which has

superior accuracy.

This algorithm is not directly in Figure 6.8, but is simply triple hashing modified so that c is a suitable constant, such as 1. The formula for indices is the same as for triple hashing, but with the constant for c substituted. This is a double hashing algorithm because only two indices of hash are needed.

The advantage this technique has over basic double hashing is that it eliminates Issue 1, assuming a suitable constant such as 1 is used. Basically, if we fix a non-problematic value for c, no values for b are problematic. For example, if b starts at zero, the first two indices are the same but almost certainly no others, and the probability of the first two indices being equal is the same as in a standard Bloom filter (1 in m). Mathematically, the improvement to double hashing makes it no longer true that $g_i = g_{i+j}$ implies $g_i = g_{i+cj}$, because the new scheme has non-linear dependence on i.

Like triple hashing, however, this scheme does not address Issues 2 and 3. We can show Issue 2 still is a problem using the same construction from triple hashing, because $c = c'$ is compatible with c being a constant. The demonstration of Issue 3 transfers just as simply.

The approximate false positive rate is similar to the double hashing Bloom filter, just without the factor of two due to Issue 1:

$$\tilde{f}_{\mathsf{IDHBF}}(m, v, k) \stackrel{(def)}{=} \left(2 + 4\,\frac{p - p^{k-1}}{1 - p}\right) \hat{f}_{\mathsf{FP}}(v, m^2) \;\oplus\; \tilde{f}_{\mathsf{BF}}(m, v, k) \qquad (6.20)$$

where p is the proportion of "1" bits.

Summary Improved double hashing is interesting from an analytical standpoint, but enhanced double hashing is better.

6.5.4 Enhanced double hashing

With a small tweak to the double hashing algorithm, all three issues identified for standard double hashing in Bloom filters are virtually eliminated.

We call this version **enhanced double hashing**.

The enhancement is that after computing the ith index, we adjust b by i in computing the next index. Thus, b varies by a different amount between each index. See Figure 6.8 for the exact algorithm. The computed indices satisfy the following equation:

$$g_i = a + ib + \frac{(i)(i^2-1)}{6} \pmod{m} \tag{6.21}$$

Enhanced double hashing does not suffer from Issue 1, for the same reasons that improved double hashing does not.

The advantage of enhanced double hashing is that it does not suffer from Issues 2 or 3, except when $k \leq 3$. For illustration purposes, suppose $h_1(x) = 0$ and $h_2(x) = \omega$, where ω is large ($\omega \gg k^3$). Suppose m is much larger than ω, such as $m \sim \omega^2$. Thus, each kind of double hashing starts with $a = 0$ and $b = \omega$, and we don't have to worry about any duplicate indices or even "wrapping around" with respect to m. The first five indices ($k = 5$) generated by plain double hashing on $\langle 0, \omega \rangle$ are

$$\vec{g} = [0, \omega, 2\omega, 3\omega, 4\omega]$$

Starting with $\langle 4\omega, m-\omega \rangle$ generates the same indices. The pair $\langle \omega, \omega \rangle$ overlaps by four, $\langle 2\omega, \omega \rangle$ overlaps by three, etc.

For improved double hashing ($c = 1$), the indices are

$$\vec{g} = [0, \omega, 2\omega + 1, 3\omega + 3, 4\omega + 6]$$

Starting with $\langle 4\omega+6, m-\omega-3 \rangle$ generates the same indices. The pair $\langle \omega, \omega+1 \rangle$ overlaps by four, $\langle 2\omega + 1, \omega + 2 \rangle$ overlaps by three, etc.

For enhanced double hashing,

$$
\begin{aligned}
g_0 &= a_0 = 0 & b_0 &= \omega \\
g_1 &= a_1 = \omega & b_1 &= \omega + 1 \\
g_2 &= a_2 = 2\omega + 1 & b_2 &= \omega + 3 \\
g_3 &= a_3 = 3\omega + 4 & b_3 &= \omega + 6 \\
g_4 &= a_4 = 4\omega + 10 & b_4 &= \omega + 10
\end{aligned}
$$

If we try to generate the same set of indices by working backwards, we fail. For it to work, we would start with the pair that generates the last two indices in reverse order:

$$
\begin{aligned}
g'_0 &= a'_0 = 4\omega + 10 = g_4 & b'_0 &= m - \omega - 6 \\
g'_1 &= a'_1 = 3\omega + 4 = g_3 & b'_1 &= m - \omega - 5 \\
g'_2 &= a'_2 = 2\omega - 1 \neq g_2 & b'_2 &= m - \omega - 3 \\
g'_3 &= a'_3 = \omega - 4 \neq g_1 & b'_3 &= m - \omega \\
g'_4 &= a'_4 = -4 \neq g_0 & b'_4 &= m - \omega + 4
\end{aligned}
$$

Thus, strict reverse order fails, and it is easy to see that all other strategies fail. First, we know we can't manipulate the small trailing constants to interfere with ω-scale differences. Similarly, we cannot manipulate the small factors of ω to interfere with m-scale differences. For example, if we try to choose numbers that overlap with the above g_0, g_2, g_4 in that order, it's easy to see that the next index cannot be g_1 or g_3 from above. Thus, enhanced double hashing does not seem to suffer from Issue 2.

If we try to overlap with most of the indices, we also fail. Consider trying to overlap with the g_1 through g_4 from $\langle 0, \omega \rangle$, in that order:

$$
\begin{aligned}
g''_0 &= a''_0 = \omega = g_1 & b''_0 &= \omega + 1 \\
g''_1 &= a''_1 = 2\omega + 1 = g_2 & b''_1 &= \omega + 2 \\
g''_2 &= a''_2 = 3\omega + 3 \neq g_3 & b''_2 &= \omega + 4 \\
g''_3 &= a''_3 = 4\omega + 7 \neq g_4 & b''_3 &= \omega + 7 \\
g''_4 &= a''_4 = 5\omega + 14 & b''_4 &= \omega + 11
\end{aligned}
$$

Based on the assumptions about ω and m, there are no better prospects for significant overlap. Enhanced double hashing does not seem to suffer from Issue 3.

Probably the only exception to these observations is overlapping by three indices. For example, if $k = 3$, enhanced double hashing is the same as improved double hashing (where $c = 1$), and it suffers from Issue 2 in than case. (Issue 3 only applies if $k \geq 4$.) For a similar reason, partial overlap by three indices (limited form of Issue 3) is possible for $k \geq 4$. These effects should only be noticeable when k is three, four, or maybe five, and only for rather small Bloom filters (see Figure 6.10).

Note that if $k \leq 2$, then any of these double hashing schemes is equivalent to a standard Bloom filter. The only difference is adding the two hash values to compute the second index, which has no effect on accuracy (assuming quality hashing) and minimal effect on speed.

The approximate false positive rate with enhanced double hashing is that of a fingerprinting (or hash-extending) Bloom filter with a fingerprint the size of two indices:

$$\tilde{f}_{\mathsf{EDHBF}}(m, v, k) \stackrel{(def)}{=} \hat{f}_{\mathsf{FP}}(v, m^2) \oplus \tilde{f}_{\mathsf{BF}}(m, v, k) \tag{6.22}$$

Summary Except for some small effects when k is around three or four, enhanced double hashing corrects all the issues identified with basic double hashing.

6.5.5 Related work: exponential double hashing

Smith, Heileman, and Abdallah propose two forms of exponential variants of double hashing, the first of which has this schema [74]:

$$g_i = a + b^i \pmod{m} \tag{6.23}$$

This suffers from Issue 1, because b needs to be relatively prime to m. The sequence also has repetitions, however, if $b^i = 1 \pmod m$ for some b and some i greater than zero and less than k. This is not uncommon; in particular, there is always $b = -1 \pmod m$ which only generates two unique indices. That could be specifically excluded, but there are likely to be others generating only three unique indices, which is a significant problem for reasonably accurate Bloom filters.

This scheme mostly passes our other tests, however. It does not suffer from Issue 2, and it only suffers from the secondary form of Issue 3, in which half or a third of indices overlap due to $b' = b^2$, etc. This form of Issue 3 is only an issue for extremely accurate Bloom filters, however.

To get full-length sequences, Luo and Heileman designed an "improved" exponential scheme [61]:

$$g_i = a + br^i \pmod m \tag{6.24}$$

where m is prime and r is a primitive root of m.

Applying this scheme to Bloom filters results in some accuracy problems. Though this "improved" version guarantees each sequence of k indices contains no repetitions, it does suffer from the full version of Issue 3, by letting $a' = a$ and $b' = br$, or $b' = br^2$, or $b' = br^{-1}$, etc. It also technically suffers from Issue 1 since b cannot be zero, but this is the insignificant form of Issue 1, where m is prime. As in the previous case, it does not suffer from Issue 2.

Summary The improved exponential scheme might have some advantages if the hash fingerprint quality is dubious, but otherwise, enhanced double hashing is better for computing Bloom filter indices. Using Issue 3, I have analytically bounded its accuracy to be worse than what enhanced double hashing seems to live up to (see validation). It will also tend to be slow, because it must use prime m and cannot utilize my optimization for addition

in modular arithmetic.

6.5.6 Empirical validation

Accuracy Figure 6.9 shows expected and observed false positive rates of Bloom filters using the various techniques for fast index computation, in an interesting example range. For each technique, the observed false positive rate is close to that predicted by the formula given, though there is a small unexplained divergence with triple hashing and smaller v. This could be an unknown issue with triple hashing that causes it to fall short of the ideal for a three-index fingerprint even sooner than would be explained by Issues 1 through 3. Nevertheless, if enough fingerprint is available, triple hashing offers a clear accuracy advantage over enhanced double hashing, which is fundamentally limited by working with a two-index fingerprint.

The enhanced double hashing Bloom filter is of particular interest, since I claim its accuracy should be similar to the ideal for a Bloom filter based on a two-index fingerprint. In the enhanced double hashing samples obtained for Figure 6.9, about half were within 1% of the prediction/ideal and none were more than 2.6% worse nor more than 2% better. About two thirds were worse than ideal.

For the results of Figure 6.9 to scale naturally, based only on the v/m ratio, the size of the fingerprint would need to be scaled as the size of one index plus 13 bits, rather than the size of two indices. As described in Section 6.4.2, the false positive potential due to a two-index fingerprint diminishes steadily as the size of the Bloom filter is increased. The impact of issues associated with fast index computation also diminish, since they are rooted in the possibility of a fingerprint false positive. To show a graph like Figure 6.9 for 1 MB Bloom filters instead of 1 KB Bloom filters would have taken about a thousand times as many Bloom filter queries, because the false positive rates at which the techniques are distinguishable are about

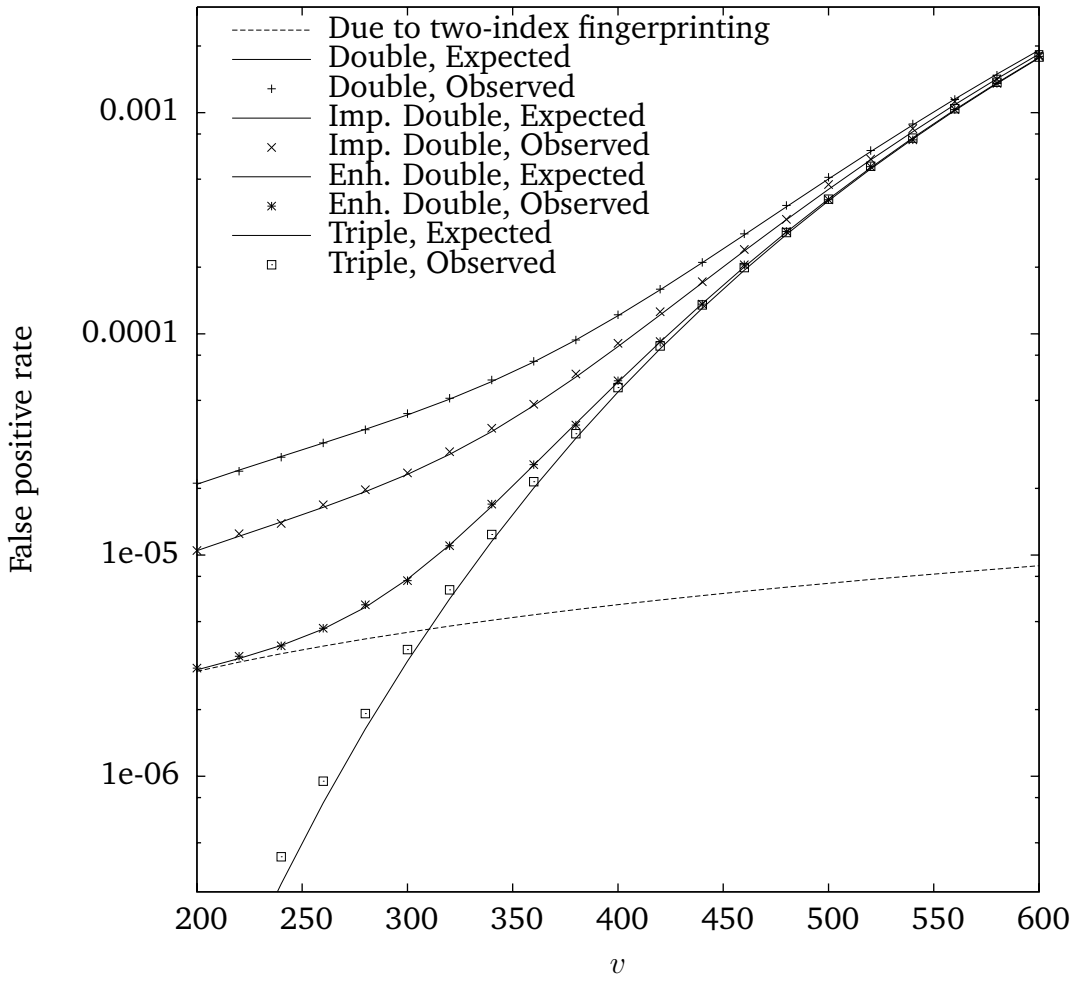

Figure 6.9: *Comparison of false positive rates of Bloom filters with fast index computation. Lower is better. For all data points,* $m = 8192$ *and* $k = 13$*, the best for* $v = 450$*. Each "Observed" data point represents enough queries to get* $10\,000$ *false positives. After the* v *additions, each structure was only used for* v *queries, and then a new one with the same configuration was built for additional queries. The additions and queries were simulated starting from pseudorandom fingerprints, derived by mixing two pseudorandom generators and a large, static pool of random data (compressed Linux kernel). "Double" refers to double hashing, in Section 6.5.1 and Equation 6.17. "Imp. Double" refers to improved double hashing, in Section 6.5.3 and Equation 6.20. "Enh. Double" refers to enhanced double hashing, in Section 6.5.4 and Equation 6.22. "Triple" refers to triple hashing, in Section 6.5.2 and Equation 6.19.*

a thousand times lower.

Figure 6.10 shows the relationships between problem scale, hash factor (m/v), and which techniques are appropriate. For example, if $m/v = 20$ (best $k = 14$) and $m = 2^{18}$, then that lies in the "enhanced double OK" region, which means that enhanced double hashing or triple hashing would have no significant impact on the false positive rate, but double hashing probably would.

Speed Figure 6.11 shows how little per-k computation is needed for Bloom filters utilizing one of my fast index computation techniques. The Bloom filter only needs to be larger than L2 cache for the time per operation to become dominated by waiting for access to the bits in the table. (Changing the memory size does not affect the complexity of what is computed, because indices are put in a machine word in each case.)

The top graph of Figure 6.11 shows some small differences in time among the various double and triple hashing techniques. However, these results are rather fragile to small changes in implementation and compiler options. Generally, the per-k computation time for double hashing is slightly less than enhanced double, which might be slightly less than triple hashing. In some implementations, though, their speed is indistinguishable.

To implement the hash-extending Bloom filter in those tests, a Jenkins 3x64-bit mix function was used to generate successive 3-index tuples starting from the first three, given by the fingerprint. It is fast enough to be cheaper than jumping to the next level of the memory hierarchy, by increasing Bloom filter size. Strangely, though, the absolute time difference between using basic hash-extending and enhanced double hashing increases for larger Bloom filters. For a purely serial processing core, this should not happen, but our cores are superscalar and able to execute instructions out of order. The extra hash computation in the hash-extending implementation must be interfering with efficient pipelining/parallelization of instructions

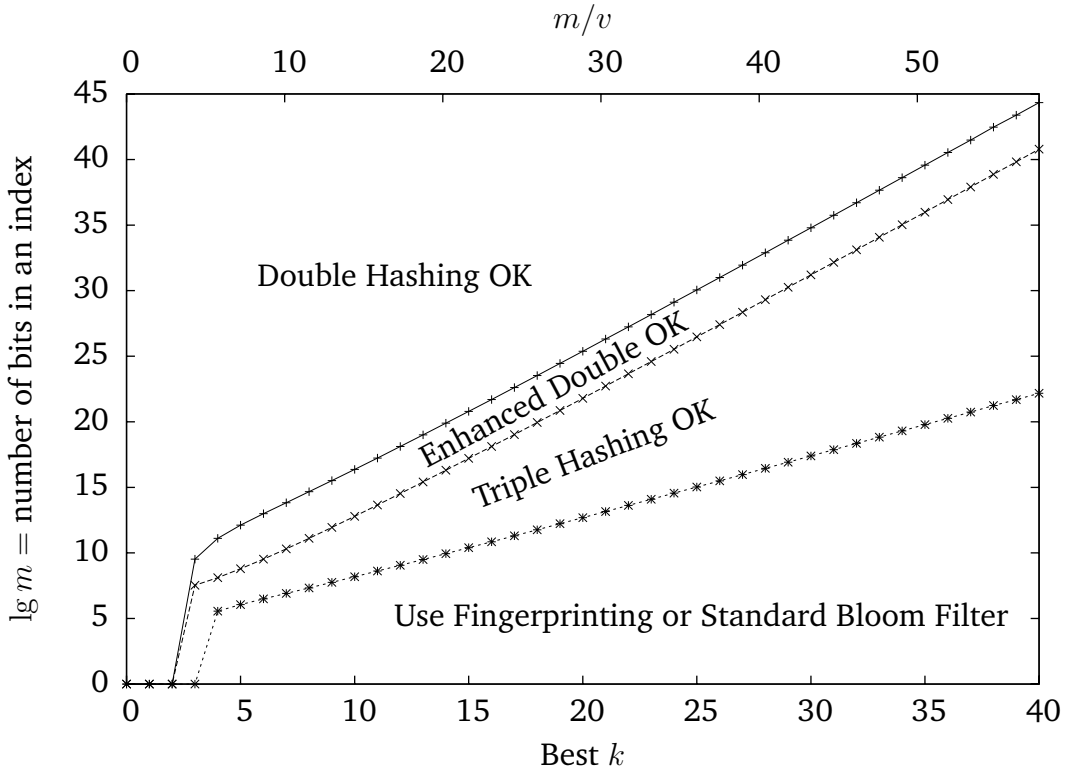

Figure 6.10: *Regions of configurations in which fast index computation techniques have low impact on false positive rate. This graph shows the "cheapest" technique among double, enhanced double, and triple hashing that has less than a 1% impact on the false positive rate for a given configuration, if there is one. We assume the k that minimizes the false positive rate is used in each case, which is shown across the bottom. The corresponding m/v is shown across the top. The boundaries were derived using the assumption that use of the best k will make the proportion of bits set to "1" approximately half. Using $z = p = 0.5$ and $m/v = k/\ln 2$ simplifies the formulas enough to set the "extra" false positive rate associated with each technique equal to 0.01 times the standard Bloom filter false positive rate, and solve for m. The same graph based on expected hash omissions instead of false positive rate would be similar but would have a different correspondence between best k and m/v; I would expect these results based on best k to match more closely for expected hash omissions than based on m/v.*

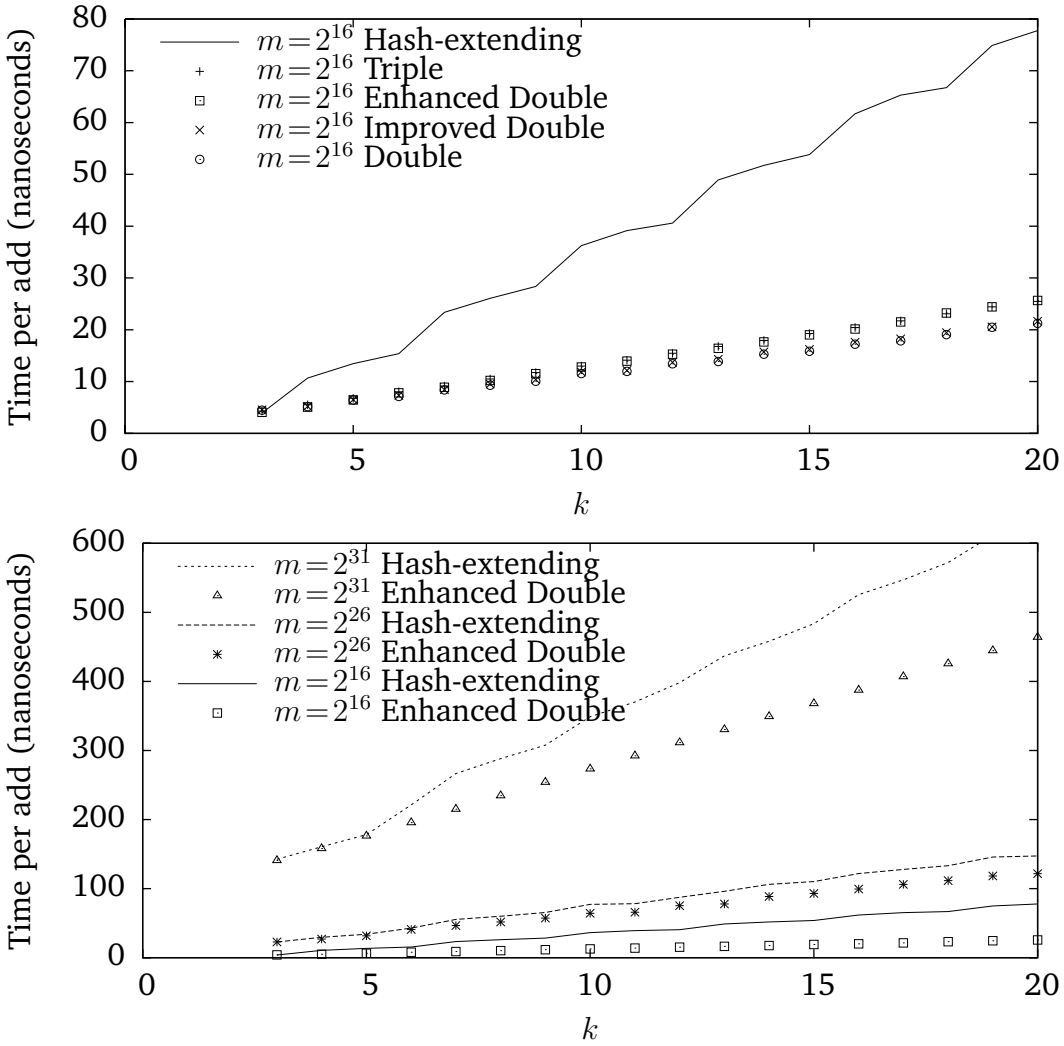

Figure 6.11: *Comparison of the* ADD *time for Bloom filters of various sizes with various index computation methods. Lower is better. All cases in the top graph use $m = 2^{16}$, which is small enough (8KB) to fit in the processor's closest cache (32KB L1 data cache). It should capture the minimum time required by each technique. The bottom graph uses various Bloom filter sizes, and includes a series from the top graph. It shows how the time just to access bits of the table dominate the time required to* ADD *to larger Bloom filters. Fingerprints to add were generated pseudorandomly, and the measured time for generating them has been removed from the times shown, even though it should take longer to compute larger fingerprints. Each point is the average of tens or hundreds of millions of operations. Tests were compiled with gcc 4.4.3 (-O3) and run on a 64-bit Linux system with Intel Xeon X5677 CPUs (3.47GHz, 32KB L1 data cache, 12MB L3 cache). More description is in referencing text.*

between multiple indices, and thus revealing more memory latency time. This seems to indicate another advantage of the tight loops in my fast index computation schemes.

I did not show random QUERY performance here because they level off around $k = 5$, because they can short-circuit when a "0" bit is encountered. Positive QUERYs must inspect all k bits, but they usually use conditional branches, which make them a little slower than "blind" ADD operations. Thus, the ADD is the best way to expose differences in speed for the various techniques.

Many more empirical results, including results using non-power-of-2 m, are in my relevant publications [24, 23].

6.5.7 In practice and future work

These techniques for fast index computation do not quite match how people should be using Bloom filters in practice. In fact, Figure 6.10 should not be seen as great practical advice—more like a sketch to help understand the techniques that have been presented. What usually happens in practice is that the hash fingerprint to work with comes from a stock hash function, which returns some number of words of output. An index is not likely to be exactly a word in length, nor half a word, etc., so the fingerprint is not likely to be a whole number of indices of length.

For the best accuracy, one should not be throwing away hash information in order to adhere to the techniques shown here. More research is needed in order to answer questions like, "What should I use if my hash fingerprint is the size of two and a half indices and I don't need it to be extremely fast?" The answer is likely to be something based on triple hashing, possibly with enhancement like enhanced double hashing. The design space opens up a lot when you consider fingerprints that are not a whole number of indices, and that should be explored, but not in this dissertation.

Also, the techniques I have described are optimized for implementation in software. I am sure the hardware logic required for implementing Bloom filter hash functions could be reduced by using fingerprinting and a simple bitwise scheme for deriving indices from it. There does not seem to be much demand for such an improvement, however, and it is outside my expertise.

6.6 Summary

Bloom filters have a rich history, in verification and elsewhere. Though there are more compact over-approximations of sets available, Bloom filters have properties that make them dynamically flexible, and I have highlighted those traits. I have also improved the implementation and analysis of Bloom filters in many ways. I have identified some issues with methods for optimizing Bloom filters by choosing the best k, and have described how to choose the best k when a Bloom filter is used as a visited set. I have shown how to reduce the hashing requirements of Bloom filters with minimal impact on the false positive rate. Despite the various improvements, there is no "best" Bloom filter design, because the needs of different applications are so varied. What I have presented, however, are well-described tools in the toolbox for implementing compact, fast, and accurate Bloom filters. For my adaptive storage scheme in Chapter 11, I draw from an unusual tool in that toolbox, the hash-reusing Bloom filter, to fulfill a need for a Bloom filter with access to limited hash information.

CHAPTER 7

Compacted tables (Hash compaction)

A **compacted hash table** (our name for the data structure underlying "**hash compaction**") is an open-addressed hash table of hash values in which the storage address is independent of the hash value to store there. In other words, whenever an element is ADDed or QUERYed, some hash information computed from that element is used to determine the first address to probe and other hash information is used as the value to store. Because of collision resolution, the value might not be stored in the first location probed. Thus, the starting address is not entirely encoded in the location of a value, so the structure is inherently inexact. No exact structure can have the same essential optimization, though the Cleary table is close (Chapter 9).

Bibliographic Notes Like the Bloom filter, the compacted hash table has probably been independently discovered/invented by several computer scientists over the years. Morris had the insight to use independent hashes for location and value in his "virtual scatter tables" [64], as Cleary does later in his compact hash table [14] (see Chapter 9). The "approximate membership tester 2" of Carter et al. [13] is a compacted hash table. For state storage in a model checker, Wolper and Leroy describe a "hashcompact" scheme that does not separate the hashing and, thus, has poor asymptotics [84]. Stern and Dill were first to apply the notion of separate computation of the value and location to state storage [77] and then made the under-appreciated im-

provement of adding Amble and Knuth's ordered hashing [78, 2]. Stern's dissertation features a detailed mathematical analysis [77, 75].

Contributions This chapter mostly serves to explain the interesting and nuanced behavior of this solution, by gathering analyses that explain certain aspects of the structure. Nevertheless, I describe how the false positive rate could be improved if all elements to be added were known in advance, and I mention a somewhat standard optimization of the compacted table metadata, which was not mentioned in Stern and Dill's work nor used in their implementation.

7.1 Description

7.1.1 Basics

The structure is composed of c cells, each storing a hash value of $b = \lfloor m/c \rfloor$ bits. Initially, all cells are unoccupied, and in the past, implementors have used an extra bit per cell to indicate whether it is occupied. This bit is not needed, of course, if we reserve the zero hash value to mean "unoccupied" and ensure the hash function giving values to store does not return that value. Mathematically, the hash function for values to store will be $h_{\mathsf{V}} : U \to \{1, \ldots, s\}$ where $s = 2^b - 1$. Freeing that bit to be used in the hash value cuts the false positive rates nearly in half.

Another hash function computes the first address to probe for storing the value: $h_{\mathsf{A}} : U \to \{0, \ldots, c-1\}$. That hash function needs to be independent of h_{V} for the structure to be asymptotically compact. Depending on how collision resolution is done, another hash function might be used, but the issue is complicated enough for detailed discussion (next).

7.1.2 Collision resolution

If ordered hashing is not used, any collision resolution scheme based on open-addressing that does not move elements after they have been placed (linear probing, quadratic probing, double hashing, etc.) can be used. The scheme used determines what is required of the hash functions that determine the probe sequence. For example, linear or quadratic probing only require a the starting address from h_A, while double hashing requires a starting location and an increment. (The increment should satisfy constraints given in Sections 5.1 and 7.1.4.) Nevertheless, double hashing will usually be the best choice thanks to negligible clustering [34]

But ordered hashing offers significant accuracy improvements, with the complication that we need to be able to move elements down in their probe sequence after they have been added. This is a significant complication because we cannot look at a value in some location and determine — definitively — the hash information that gives its probe sequence. However, we don't need to know what the whole probe sequence has been, just how to find the next location in that sequence. Linear probing is good for this, since the next probe location is always the subsequent address, but we should avoid the clustering problems of linear probing if we can. Quadratic probing, for example, would not work because it requires knowledge of how far an entry is in its probe sequence to determine how far away the next probe is.

One of the under-emphasized Stern and Dill insights in applying ordered hashing to hash compaction was that double hashing could be used if the increment between probes is determined by the hash value stored. Thus, we can determine the next probe location for a value stored in some location, despite not knowing its original probe location. Assuming some method of turning stored hash values into probe increments (see Section 7.1.4), this design is what we consider the standard design for a compacted hash table.

7.1.3 Ordered hashing

Now that we have a collision resolution scheme that allows stored values to be moved down their probe sequence, let us consider how ordered hashing uses this to increase accuracy. After all, ordered hashing was originally designed as an optimization for negative QUERYs, and in the visited set usage paradigm, negative QUERYs become ADDs, which might be slowed down by ordered hashing. Thus, ordered hashing confers no time benefit in the visited set usage paradigm.

Ordered hashing maintains the invariant that each location contains the "largest" value that was considered for storage there. Instead of always ADDing to the first empty cell in the probe sequence, the value is placed in the first cell that contains a "smaller" value if that is encountered before an empty cell. If replacing a smaller value, the ADD continues with that smaller value, proceeding to the next location in *its* probe sequence.

This invariant gives each probe to an occupied cell about a 50/50 chance of ruling out something being in the structure, because if the new element is larger than what is stored there, then the invariant says that it has not been added already. This can significantly reduce the number of probes required to resolve a negative query, which likewise reduces the number of opportunities for the hash value of a new element to collide due to random chance and yield a false positive.

Consider, for example, checking a new element against a compacted hash table that is 95% full. If it is not using ordered hashing, we expect to compare the hash value against about 20 others (effectively random) before encountering an empty cell, each with a $1/s$ chance of causing a false positive. With ordered hashing, the first one probed has a $1/s$ chance of matching randomly, but it also has about a 50% chance of being smaller and eliminating any remaining possibility of a false positive query. In the case of subsequent probes, they have the same chance of matching and a similar probability of

eliminating remaining possibilities of a false positive.

An Interesting Phenomenon An interesting behavior of hash compaction with ordered hashing, which no one else has described, is the case in which an element is added to the structure, it is recognized as new, but the number of cells occupied in the structure does not increase. Suppose the value to add is larger than the value currently occupying the target location. This definitively indicates the element being added has not been added before. To maintain the structural invariant of ordered hashing, the larger value replaces the one there, and the ADD procedure continues with the replaced value at its next probe location. Now suppose that at that next probe location, the hash value is the same as what we are now attempting to add. If that's the case, we can stop; there is no need to continue searching and replacing until an empty cell is encountered. We have altered the structure to include the element we wanted to add, but without occupying an additional cell. Instead, values were moved around the structure and two with the same value collided and were essentially merged.

In hash compaction *without* ordered hashing, the number of occupied cells equals the number of affecting additions. *With* ordered hashing, we have just seen how the number of affecting additions can be larger. In fact, for a given number of unique additions, the number of occupied cells is reasonably close, whether or not ordered hashing is used, but the lower false positive rate with ordered hashing causes a correspondingly higher number of the unique additions being recognized as new and affecting the structure. (More in Section 7.2.3.)

7.1.4 Implementation notes

- As discussed above, we can eliminate the need for a bit to indicate whether a cell is occupied by reserving the "0" hash value. This makes the range of our hash function $2^b - 1$ values. The easiest way to do

this is let its range be 2^b values and use a second function if the first returns "0". If the second returns "0," just use "1." Or one can skip the second function and just use "1" whenever the function returns "0." Since b needs to higher than 10 to have higher accuracy than a competing Bloom filter, the impact of such a hack is negligible.

- I discuss how to compute appropriate double hashing increments in Section 5.1, but now we have the potential complication of deriving these from the stored values. Tests indicate that using a hash function to derive these values confers no measurable advantage over using them almost directly. One should, however, ensure near minimal information loss in deriving an increment appropriate for the number of cells. For example, to guarantee an odd increment, shift left by one and add 1.

7.1.5 Maximum occupancy and configuration

Assuming we know the number elements to be added, v, a naive way to configure the structure is to use $b = \lfloor m/v \rfloor$ bits per hash value. There are two problems with that. First, the number of cells should be a prime or a power of two (Sections 5.1 and 7.1.4). Second, we should not allow the structure to fill beyond some constant occupancy less than 100%. This will be critical to the accuracy being asymptotically compact (Section 7.2.5), but there's also an issue with execution speed:

The expected number of probes per operation for a double-hashed table is known to be close to $(c+1)/(c+1-n)$ [58]. If we allow the structure to fill up, the average number of probes is

$$\frac{1}{c}\sum_{i=0}^{c-1}\frac{c+1}{c+1-i} \overset{c\gg 0}{\geq,\approx} \frac{1}{c}\int_0^{c-1}\frac{c+1}{c+1-i}di = \frac{c+1}{c}\ln\frac{c+1}{2} \overset{c\gg 0}{\geq,\approx} \ln\frac{c}{2}$$

Thus when a double-hashed open-addressed hash table (such as a com-

pacted table) is allowed to fill up, the average number of probes per addition is expected to be logarithmic in the number of cells. This makes the time to fill up super-linear in the number of cells. Filling up hurts accuracy as well, as discussed in Section 7.2.1.

I usually use a maximum occupancy of no more than 99.8%. That is easy to follow if we choose the amount of memory and number of cells based on other parameters, but configuring to use no more than some predetermined amount of memory (m bits) is more nuanced, even knowing the number of elements to be added (v). We can start by computing the minimum number of cells needed, c_0, which is the smallest prime or power of two not smaller than $v/0.998$. We can then chose the largest integer number of stored value bits $b = \lfloor m/c_0 \rfloor$ and use that to potentially boost the number of cells to its final value, the greatest prime or power of two not greater than m/b. Except in rare cases, this procedure will give the configuration with the best accuracy.

7.2 Accuracy analysis and validation

Ulrich Stern has analyzed this structure in some detail [75, Appendix A], but I consider more aspects of the structure and in more variations to aid in understanding the behavior of compacted hash tables.

7.2.1 By collisions

One of Stern's basic observations allows us to approximate the false positive rate of some compacted hash tables given the average number of collisions for a negative QUERY. A **collision** occurs each time the hash value being QUERYed is compared to one already in the table. To be most precise, we would use the probability distribution of collisions per negative QUERY to determine the false positive rate, but we can approximate that by using only the expected collisions per negative QUERY, which we call $\hat{x}$. With $s = 2^b - 1$

possible hash values, each collision results in a false positive with probability s^{-1}. We call this false positive rate approximation $\tilde{f}_{\mathsf{HC}\hat{x}}$:

$$\begin{aligned} \tilde{f}_{\mathsf{HC}\hat{x}}(\hat{x}, s) &\overset{(def)}{=} 1 - \left(1 - s^{-1}\right)^{\hat{x}} \\ &\overset{s \gg 0}{\geq, \approx} 1 - e^{-\hat{x}s^{-1}} \\ &\overset{s \gg \hat{x}}{\approx} \hat{x}s^{-1} \end{aligned} \tag{7.1}$$

Using the result from Section 7.1.5, we can now conclude that allowing the structure to fill up is bad for accuracy. Similarly, our asymptotic accuracy argument in Section 7.2.5 will depend on not filling up.

7.2.2 Unordered

When not using ordered hashing, the expected *collisions* per negative QUERY is one less than the expected *probes* per negative QUERY. Expected probes for double hashing is known to be close to $(c+1)/(c+1-n)$ [58], where c is the number of cells and n is the number occupied. (When not using ordered hashing, the number of occupied cells equals the number of affecting additions.) Thus, when not using ordered hashing,

$$\hat{x}_{\mathsf{UHC}} \approx \frac{c+1}{c+1-n} - 1 = \frac{n}{c+1-n} \tag{7.2}$$

Thus, in terms of the occupancy, $\alpha = n/c$, for large n,

$$\hat{x}_{\mathsf{UHC}} \overset{n \gg 0}{\approx} \left(\alpha^{-1} - 1\right)^{-1} \tag{7.3}$$

The validity and quality of Equation 7.2 is confirmed in the "Unordered, Theoretical" and "Unordered, Experimental" data of Figure 7.1.

Using the expected collisions, the approximate false positive rate for

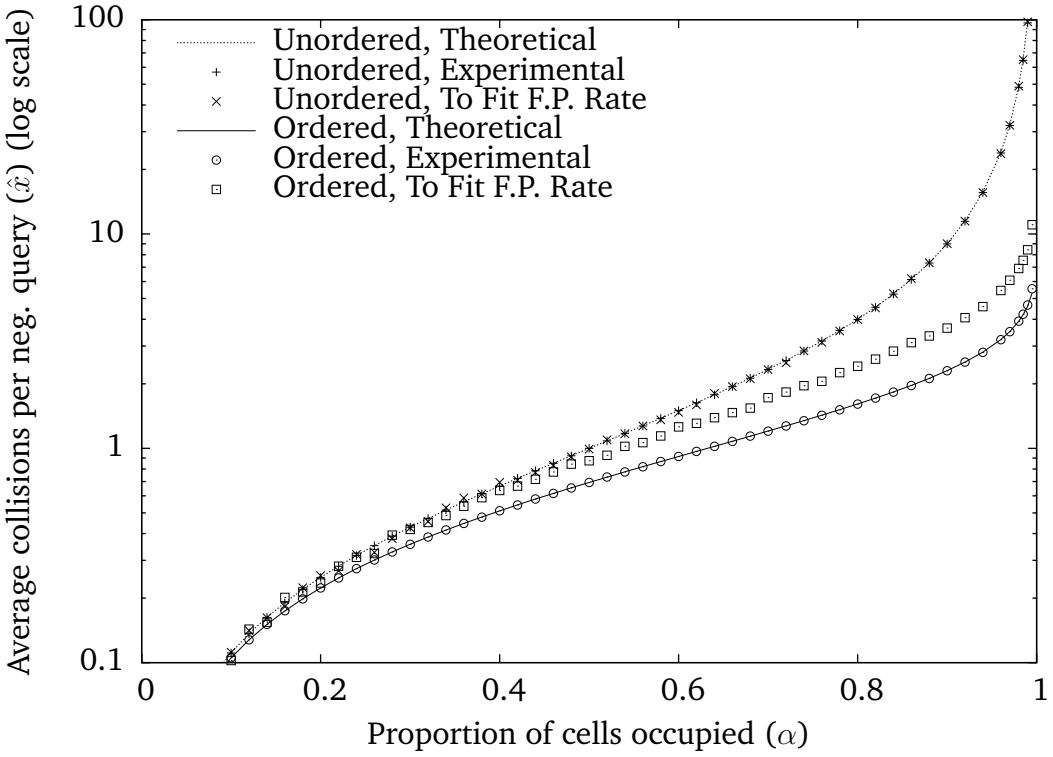

Figure 7.1: *Average collisions per negative query, expected and observed, for "unordered" and "ordered" compacted table designs, at various occupancies. Experimental values came from instrumenting our Java implementation to count collisions and performing 10 million new queries to a structure with each of the given occupancies. "To Fit F.P. Rate" values are based on false positive rate observations (see Figure 7.2) and inverting Equation 7.1. In each case,* $c = 2\,000\,003$ *cells and* $b = 12$ *bits (*$\Rightarrow s = 4095$*).*

unordered **h**ash **c**ompaction should be

$$\begin{aligned} \tilde{f}_{\mathsf{UHC}}(n, c, s) &\overset{(def)}{=} 1 - \left(1 - s^{-1}\right)^{n/(c+1-n)} \\ &\overset{s \gg 0}{\geq, \approx} 1 - e^{-ns^{-1}(c+1-n)^{-1}} \\ &\overset{s \gg n}{\approx} \frac{n}{s(c+1-n)} \end{aligned} \tag{7.4}$$

The validity and quality of this formula is confirmed in the "Unordered, Theoretical" and "Unordered, Experimental" data of Figure 7.2. In fact, if we use the observed false positive rates to deduce the expected number

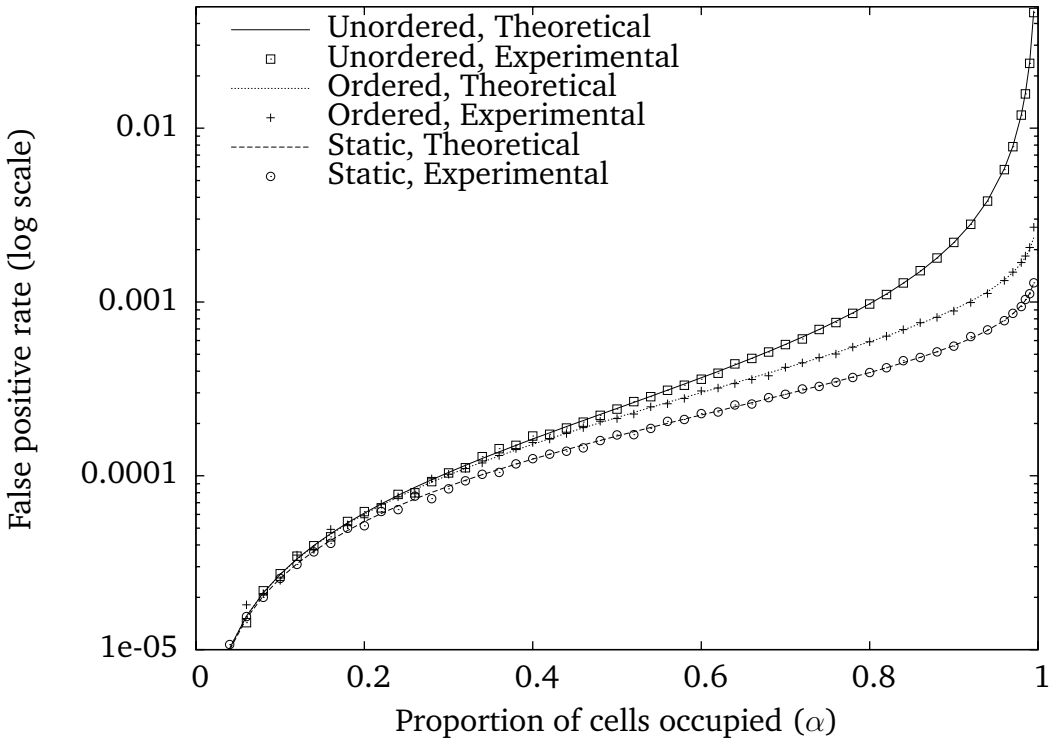

Figure 7.2: *False positive rates, expected and observed, of three compacted table designs, at various occupancies. Experimental results came from performing 10 million new queries to a structure with each of the given occupancies and recording the proportion that returned* positive*. In each case,* $c = 2\,000\,003$ *cells and* $b = 12$ *bits (*$\Rightarrow s = 4095$*).*

of collisions per negative QUERY (by inverting Equation 7.1), we get the "Unordered, To Fit F.P. Rate" data of Figure 7.1, which matches both the analytically expected and experimentally observed collisions per negative QUERY.

7.2.3 Ordered, false positive rate

Using ordered hashing can reduce the expected collisions and false positive rate dramatically, but the analysis is more complicated [75, Appendix A.3]. Here is Stern's approximation of the false positive rate for **o**rdered **h**ash **c**ompaction, when k out of c cells are occupied and there are s possible hash

values to store:

$$\tilde{f}_{\mathsf{OHC}}(k,c,s) \stackrel{(def)}{=} \frac{2}{s}\left(H_{c+1} - H_{c-k}\right) - \frac{2c + k(c-k)}{cs(c-k+1)} \tag{7.5}$$

where $H_i = \sum_{j=1}^{i} j^{-1}$ is the ith harmonic number.

The validity and quality of this formula is confirmed in the "Ordered, Theoretical" and "Ordered, Experimental" data of Figure 7.2. However, we introduced k to refer to number of cells occupied rather using n, because in an ordered compacted hash table, the number of affecting additions can exceed the number of occupied cells. (Recall the "interesting phenomenon" in Section 7.1.3.) This formula is fine for a *post facto* analysis since k can be counted at run time. For an *a priori* analysis, the *a priori* expected affecting additions to an *unordered* compacted table is a good overestimate of how many cells will be occupied in an *ordered* compacted table. This is confirmed by Figure 7.3, which compares these quantities after subtraction from the number of unique additions; observe that "Unordered, Omissions," both expected and observed, are below the "Ordered, Unique minus Occupied" observations—when the values dominate sampling error.

There is a reason we did not use $\tilde{f}_{\mathsf{HC}\hat{x}}$ to derive the false positive rate for ordered compacted tables. If we use the observed false positive rates to deduce the expected number of collisions per negative QUERY (by inverting Equation 7.1), we get the "Ordered, To Fit F.P. Rate" data of Figure 7.1, which does not match the "Ordered, Experimental" data. This implies that the false positive rate using ordered hashing is not explained just by the number of collisions during queries, as Stern's approximation (Equation 7.5) takes into account.

The reason the false positive rate using ordered hashing is worse than is explained by the average number of collisions is because the increments for double hashing are determined by the values to store. Recall that this dependency exists to allow values to be relocated down their probe sequence

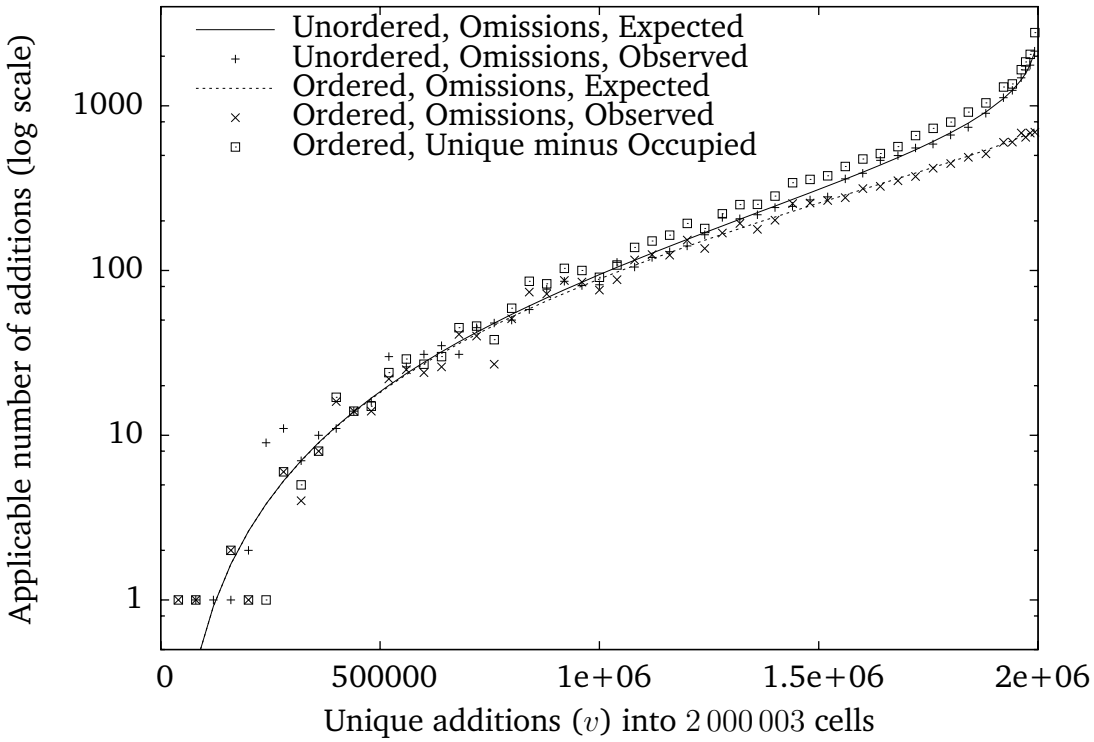

Figure 7.3: *Omissions and non-omissions, observed and expected, in two designs of compacted tables. The given number of unique elements was added once to the given structure and the number of affecting additions and number of occupied cells were tracked. Hash functions were given a new seed for each set of additions. "Omissions" is unique additions minus affecting additions. "Unique minus occupied" is unique additions minus occupied cells. By construction, "Unique minus occupied" for the unordered table would be the same as its "Omissions." In each case,* $c = 2\,000\,003$ *cells and* $b = 12$ *bits (*$\Rightarrow s = 4095$*).*

after they have been added. We could test this claim if we could remove this dependency and observe the resulting false positive rate. One way to do this is by constructing a **static compacted hash table**, in which every element to be added is known ahead of time and elements are added in (decreasing) order of stored hash values. Adding in this order precludes relocation of elements and allows double hashing increments to be determined independently of stored values. The "Static, Experimental" data in Figure 7.2 confirms that this new variation has noticeably lower false positive rates. The "Static, Experimental" and "Static, To Fit F.P. Rate" data in Figure 7.4 confirm that, with the dependency removed, the average collisions per negative QUERY explain the false positive rate.

The static compacted hash table cannot be used as a compact visited set, but could find applications in compact "summaries," which are often based on Bloom filters [28]. However, Bloom filters are only close to optimal accuracy for a given size if compressed for transit [62].

7.2.4 Ordered, collisions

The false positive rate of the static ordered table is explained by the expected collisions per negative QUERY, but I have not found a simple analysis that predicts this quantity. Here I present such an analysis: let $\alpha = k/c$ be the proportion of cells occupied and let p be the proportion of the hash values greater than the current one under consideration. In that case, the expected number of collisions, $\hat{x}_p$, should satisfy

$$\hat{x}_p = \alpha(1 + p\hat{x}_p)$$

On any given probe, there is a $1 - \alpha$ probability of hitting an empty cell, in which case there are zero expected collisions. With probability α, we probe an occupied cell, which guarantees at least one collision. With probability $1-p$, this is the last probe and there are no more collisions. With probability

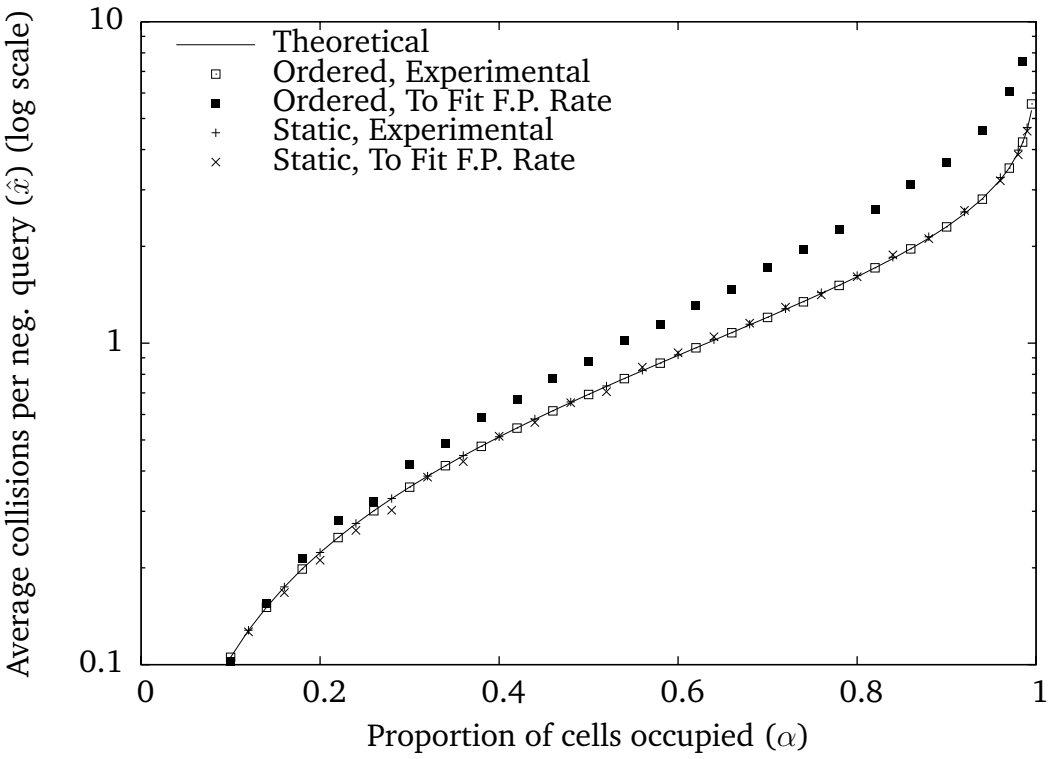

Figure 7.4: *Average collisions per negative query, expected and observed, for "ordered" and "static" compacted table designs, at various occupancies. Experimental results came from instrumenting our Java implementation to count collisions and performing 10 million new queries to a structure with each of the given occupancies. "To Fit F.P. Rate" values are based on false positive rate observations (see Figure 7.2) and inverting Equation 7.1. In each case,* $c = 2\,000\,003$ *cells and* $b = 12$ *bits (*$\Rightarrow s = 4095$*).*

p, we continue to the next probe.

Thus, solving for $\hat{x}_p$,

$$\hat{x}_p = \frac{1}{\alpha^{-1} - p}$$

Letting the uniformly distributed hash values go to infinity, we get the overall expected collisions by integration:

$$\hat{x}_{\mathsf{OHC}} \approx \int_0^1 \frac{1}{\alpha^{-1} - p} dp = -\ln(1-\alpha) = -\ln(1 - k/c) \tag{7.6}$$

This is validated in Figure 7.4, where it is the "Theoretical" line, match-

ing the observations for both of our ordered compacted tables. It also explains the false positive rates observed for the static table ("Static, To Fit F.P. Rate"). Combining with Equation 7.1 gives the "Static Theoretical" line of Figure 7.2, which is closely matched by the "Static Experimental" observations.

7.2.5 Asymptotics

To meet our criteria of "asymptotically compact" for inherently inexact structures, the false positive rates of compacted tables should effectively depend only on m/v, the memory per element added (see Section 4.4). In the worst case for a compacted table's final false positive rate, each unique element added occupies a new cell and $k = n = v$. If we show it is asymptotically compact for its worst false positive rates, which depend only on $m/v = m/n = m/k$, it should be easy to believe the structure is asymptotically compact in general.

If we pick a constant, non-full occupancy, $\alpha = k/c < 1$, to use in all cases, we can derive the appropriate number of cells, c, from k/α. The size of each stored value is then $b = \lfloor m/c \rfloor$. The number of possible values is $s = 2^b - 1$.

It should be sufficient to show that if we double m and v, the false positive rate is essentially the same. (See "litmus tests" in Section 4.4.) Doubling these doubles the number of cells, c, but keeps the stored hash values the same size (b and s do not change). For structures whose false positive rate depends only on the number of collisions per query, the false positive rate from Equation 7.1 will stay the same if $\hat{x}$ also stays the same (as s does). Equation 7.3 shows that $\hat{x}$ for the unordered structure depends only on α (for large n and $\alpha < 1$). Equation 7.6 shows that $\hat{x}$ for the static, ordered structure depends only on α (for large k and $\alpha < 1$).

The false positive rate for Stern and Dill's dynamic, ordered structure should lie between those two, but to have more confidence, we work from

Stern's approximation (Equation 7.5), rephrased:

$$\tilde{f}_{\mathsf{OHC}}(\alpha c, c, s) = \frac{2}{s}\left(H_{c+1} - H_{(1-\alpha)c}\right) - \frac{2c + \alpha c[(1-\alpha)c]}{cs[(1-\alpha)c+1]}$$

Using a property of harmonic numbers, that $\lim_{i\to\infty}[H_i - \ln i] = \gamma$ for a constant γ,

$$\begin{aligned}\lim_{c\to\infty} \tilde{f}_{\mathsf{OHC}}(\alpha c, c, s) &= \lim_{c\to\infty} \frac{1}{s}\left(2\ln\frac{\cancel{c+1}}{(1-\alpha)\cancel{c}} - \frac{\cancel{\alpha(1-\alpha)c^2 + 2\alpha c}}{\cancel{(1-\alpha)c^2 + c}}\right) \\ &= \frac{1}{s}\left(2\ln(1-\alpha)^{-1} - \alpha\right)\end{aligned}$$

From the limit, it is clear that if $0 < \alpha < 1$ and c is large enough, the false positive rate of the standard compacted table depends only on α and s, modulo Stern's approximation. α is constant and s has a lower bound based on m/v. Thus, the structure is asymptotically compact.

7.2.6 Negative result: reordered hashing

Suppose that instead of using the same ordering in each cell for ordered hashing, each cell uses a different ordering on stored values. (Equivalently, one could imagine using the same ordering in all cells and instead transforming the hash values between probes, but this would interfere with double hashing.) By using different orderings, a value could be near one end of the ordering for the first probe and anywhere else in the ordering for the next probe. This should remove dependence among the probes for the probability of ordered hashing precluding the element as previously added, bringing more equity to the accuracy afforded to different inputs. I had hypothesized that correcting that inequity would lower the false positive rate, closer to that of the static compacted table, but experiments showed no difference in false positive rate compared to the dynamic structure using traditional ordered hashing.

7.3 Summary

The compacted hash table ("hash compaction") has remarkable speed and peak competitive accuracy, but existing work had left holes in the analysis, understanding, and use of the structure. I believe I have filled many of those holes. I have shown how collisions fully explain the false positive rate for some variants of the structure but not for others. I described a new, "static" variant of the structure, which might have the lowest false positive rates of any known, practical structure supporting random-access reads. And I have shown that the structure's impressive accuracy scales perfectly, to arbitrarily large structures–but only if it is not filled completely. The compacted hash table is a valuable device in the verifier's toolbox.

CHAPTER 8

Inexact Storage Using Exact Storage

This chapter describes the basis for using exact data structures to solve the inexact visited set problem. I present two general approaches that arise naturally, show that one is usually better, and show that for each, the required beyond the unrestricted optimal is at most a constant number of bits per added element. The distinction between the two approaches is rarely significant, but it does affect the bound in Theorem 11.1 (see Figure 11.5). Formulas presented here are also useful for computing and displaying the approximate expected hash omissions in a model checker using an exact structure for inexact storage.

Bibliographic Notes The general approach of using an exact representation to implement an inexact structure traces back to a 1978 STOC paper by Carter et al. [13, Approximate Membership Tester 3], if not earlier. For "an optimal Bloom filter replacement," Pagh et al. extend that to describe the inherent space overhead in using an exact structure to implement an inexact structure [65, Section 2], but their analysis is limited to cases in which u is much too large for exact storage (with a given m and v). The same limitation applies to later work by Dietzfelbinger and Pagh [20, 19].

Contribution If m/v is close to $\lg u$, the bounds by Pagh et al. are not accurate, and such cases are important for adaptive storage—if it is to start with exact storage and remain close to the optimal accuracy for available

memory (Chapter 11). Here I fill in details, so that the analysis fits our needs in Chapter 11.

8.1 Introduction

Solving the visited set problem inexactly using an exact set is just a matter of hashing the input elements to elements of some smaller set and exactly storing a subset of the smaller set. One could think of this as fingerprinting all the input elements and storing a set of fingerprints.

Recall that the visited set should be a subset of the represented set, itself a subset of the universe in consideration: $V \subseteq W \subseteq U$. Let P be a partitioning of U. That is, for all $X \in P$, $X \subseteq U$; for all $X \in P$ and $Y \in P$, $X \neq Y \Rightarrow X \cap Y = \emptyset$; and $\bigcup_{X \in P} X = U$. Let $h(x) = \{X : x \in X \wedge X \in P\}$; in other words, h associates elements of the universe with what partition they are in. Mathematically, this is what the fingerprinting hash function does, since each fingerprint value is associated with some unique subset of the universe.

There seem to be two general approaches to computing the hashes, and it is not immediately clear that one is inherently better than the other, or that they can have significantly different accuracy. I consider both in detail and show the two approaches are indistinguishable when $|U| = \infty$, which is why work that assumes a relatively large universe (such as [65]) need not consider the distinction. When $|U| < \infty$, both methods arise quite naturally in practice. "Even" partitioning is detectably better (lower false positive rate for same memory) when the number of partitions is only a small factor from the universe size, and I show it is at least as good as "balls and bins" partitioning in almost all practical cases.

8.2 "Balls and bins" partitioning

The first approach ("**balls and bins**"), given a desired number of partitions p, is to place each element of the universe randomly and independently into one of the p partitions, as an ideal hash/fingerprinting function would. After adding v elements, the probability that any given partition has had none of its elements added is the probability that all v were assigned to other partitions: $(1 - 1/p)^v$. One minus that is also the probability that an element of the universe that has not been added maps to a partition which has had at least one of its elements added. The probability that an element that *has* been added maps to a partition which has had at least one of its elements added is, of course, 1. Thus, by linearity of expectations, if v elements have been added and $u - v$ have not, then the expected number of elements of the universe whose partition must be in the represented subset is

$$\hat{w}_{\mathsf{BB}v}(u, v, p) = v + (u - v)\left(1 - \left(1 - \frac{1}{p}\right)^v\right) \tag{8.1}$$

Recall that if $u = |U| < \infty$, $f = \frac{w-v}{u-v}$, so we can get an expected false positive rate by plugging in a $\hat{w}$ for w. In this case, that makes things simpler, even removing dependence on u:

$$\hat{f}_{\mathsf{BB}v}(v, p) = 1 - \left(1 - \frac{1}{p}\right)^v \tag{8.2}$$

From standard Bloom filter analysis (Equation 6.5), we know there is a good approximation that depends only on v/p:

$$\hat{f}_{\mathsf{BB}v}(v, p) \approx 1 - e^{-v/p} \tag{8.3}$$

The analysis is much simpler if we are given the number of affecting additions, n, such as when observing a data structure for which the number of unique additions is unknown. In that case, because of the independence

of assigning each element to a partition, the false positive rate is exactly the proportion of represented partitions:

$$f_{\mathsf{BB}n}(n,p) = \frac{n}{p} \tag{8.4}$$

This makes the *post facto* expected hash omissions from a search easy to compute (see Equation 3.6):

$$\hat{o}_{\mathsf{BB}n} = \sum_{i=0}^{n-1} \frac{n/p}{1-n/p} \;\leq,\approx\; n\frac{p}{n}\int_0^{n/p} \frac{x}{1-x}dx = -n - p\ln\left(1-\frac{n}{p}\right) \tag{8.5}$$

Note that floating-point arithmetic is likely to give inaccurate results for the last formula, because floating-point is not good at representing numbers very close to 1. That formula requires high-precision arithmetic. Here are simpler bounds that are good approximations when $n \ll p$:

$$\frac{n(n-1)}{2p} = \frac{n-1}{2}\cdot\frac{n/p}{1} \;\leq\; \hat{o}_{\mathsf{BB}n} \;\leq\; \frac{n-1}{2}\cdot\frac{n/p}{1-n/p} = \frac{n(n-1)}{2(p-n)} \tag{8.6}$$

Theorem 8.1. *A set over-approximation for v unique elements using an information-theoretic optimal exact representation based on p "balls and bins" partitions uses $O(v)$ more bits than an information-theoretic optimal representation giving the same false positive rate.*

Proof Let $\hat{n}$ be the expected number of unique partitions represented by the v unique elements. Observe $\hat{n} \leq v$, because $\hat{n}$ is also the expected number of affecting additions.

Let $\hat{f}$ be the false positive rate of the structure based on "balls and bins" partitioning, based on $\hat{n}$ and p. Observe that by Equation 4.4, $\hat{f} = \frac{\hat{n}}{p}$.

The proof obligation is that $\breve{m}_{\hat{n},p,0} = \breve{m}_{v,u,\hat{f}} + O(v)$, or

$$\breve{m}_{\hat{n},p,0}/v - \breve{m}_{v,u,\hat{f}}/v = O(1).$$

To prove this, I will replace $\breve{m}_{\hat{n},p,0}/v$ with things greater than or equal and re-

place $\breve{m}_{v,u,\hat{f}}/v$ with things less than or equal, until I can show the difference is $O(1)$.

Starting with the left operand, we use Equation 4.3 and $\hat{n} \leq v$ to bound from above ($\lg e$ is the base-2 logarithm of $2.71828\ldots$):

$$\begin{aligned}
\breve{m}_{\hat{n},p,0}/v &\leq \frac{\hat{n}}{v}\left(\lg p - \lg \hat{n} + \lg e\right) \\
&= \frac{\hat{n}}{v}\lg\frac{p}{\hat{n}} + \frac{\hat{n}}{v}\lg e \\
&\leq \lg\frac{p}{\hat{n}} + 1.5
\end{aligned}$$

For the right operand, we use $\hat{f} = \frac{\hat{n}}{p}$, and $\hat{w} = (u-v)\hat{f} + v$ to bound from below:

$$\begin{aligned}
\breve{m}_{v,u,\hat{f}}/v &\geq \lg u - \lg \hat{w} - \lg e \\
&\geq -\lg\frac{\hat{w}}{u} - 1.5 \\
&= -\lg\frac{(u-v)(\hat{n}/p) + v}{u} - 1.5 \\
&\geq -\lg\left(\frac{\hat{n}}{p} + \frac{v}{u}\right) - 1.5
\end{aligned}$$

Thus, we have reduced the proof to

$$\lg\frac{p}{\hat{n}} + 1.5 + \lg\left(\frac{\hat{n}}{p} + \frac{v}{u}\right) + 1.5 = O(1).$$

Or, using the fact that $\breve{m}_{\hat{n},p,0} \geq \breve{m}_{v,u,\hat{f}}$ by definition, it suffices to prove

$$\lg\frac{p}{\hat{n}} + \lg\left(\frac{\hat{n}}{p} + \frac{v}{u}\right) = \pm O(1).$$

From here, we case split on whether $\frac{\hat{n}}{p} \geq \frac{1}{2}$.

We first consider the case of $\frac{\hat{n}}{p} \geq \frac{1}{2}$. Recall that $\hat{n} \leq p$. Thus, $1 \leq \frac{p}{\hat{n}} \leq 2$. Using that and the by-definition bounds $0 \leq \frac{\hat{n}}{v} \leq 1$ and $0 \leq \frac{v}{u} \leq 1$, it is easy to see that all the terms are bounded by constants and, thus, equal to $\pm O(1)$. Conceptually, this makes sense because $\frac{n}{p} \geq \frac{1}{2}$ implies $\frac{v}{p} \geq \frac{1}{2}$, which

implies that using a p-bit bit table as the exact representation requires only a constant number of bits per added element.

We now have to consider the case of $\frac{\hat{n}}{p} < \frac{1}{2}$. For this case, we back up to expand one of our bounds, using $p \leq u$:

$$\begin{aligned} \breve{m}_{v,u,\hat{f}}/v &\geq -\lg\left(\frac{\hat{n}}{p} + \frac{v}{u}\right) - 1.5 \\ &\geq -\lg\left(\frac{\hat{n}}{p} + \frac{v}{p}\right) - 1.5 \end{aligned}$$

Combining the bounds as before, we get

$$\lg\frac{p}{\hat{n}} + \lg\left(\frac{\hat{n}}{p} + \frac{v}{p}\right) = \pm O(1).$$

$\Leftrightarrow$

$$\lg\left(1 + \frac{v}{\hat{n}}\right) = \pm O(1).$$

Thus, to finish the proof, we need to show that the number of added elements is not a significant factor larger than the expected number of affecting additions when $\frac{\hat{n}}{p} < \frac{1}{2}$. This seems quite reasonable, because when $n \ll p$, virtually every unique addition is going to map to a new/unvisited partition of U, so in that case, $\frac{v}{\hat{n}} \approx 1$. By the time we reach $\frac{n}{p} = \frac{1}{2}$, the next unique addition has a 50% probability of mapping to a new/unvisited partition.

Working it out shows that $\frac{v}{p} \approx \ln 2$ is where we expect $\frac{\hat{n}}{p} = \frac{1}{2}$. Thus, $\frac{v}{\hat{n}}$ is less than about 1.39 whenever $\frac{\hat{n}}{p} < \frac{1}{2}$. $\square$

8.3 "Even" partitioning

The other approach ("**even**") is to partition U as evenly as possible among the p partitions. This arises if the elements from our universe are already "randomized" or uniformly distributed and we do not want to incur the cost of more hashing to partition the elements. As a small example, if it takes

16 bits to describe each element ($u = 2^{16}$) and we divide the space into $p = 2^{12} = 4096$ partitions, we can use 12 bits of the (randomized) descriptor to choose the partition. Thus, exactly $u/p = 2^{16-12} = 2^4 = 16$ elements are assigned to each partition. In "balls and bins" partitioning, 16 would only be the average.

In general, each partition will have either $\lfloor u/p \rfloor$ or $\lceil u/p \rceil$ elements, but to simplify the analysis, we will assume u/p is an integer. Once again, after adding v elements, the probability that any given partition has had none of its elements added is the probability that all v were assigned to other partitions. In this case, that probability is captured precisely by the ratio of the number of ways of choosing v elements from the universe without that one partition, $\binom{u-u/p}{v}$, to the number of ways of choosing v elements from the whole universe, $\binom{u}{v}$. By linearity of expectations, the expected number of elements of the universe whose partition must be in the subset is

$$\hat{w}_{\mathsf{E}v}(u, v, p) = u\left(1 - \frac{\binom{u-u/p}{v}}{\binom{u}{v}}\right) \tag{8.7}$$

First observe that

$$\hat{w}_{\mathsf{E}v}(u, v, p) = u\left(1 - \prod_{i=0}^{v-1} \frac{u - u/p - i}{u - i}\right) = u\left(1 - \prod_{i=0}^{v-1}\left(1 - \frac{1}{p(1 - i/u)}\right)\right)$$

Thus,

$$\hat{w}_{\mathsf{E}v}(u, v, p) \;\geq, \approx\; u\left(1 - \left(1 - \frac{1}{p}\right)^v\right) \;\geq, \approx\; u\left(1 - e^{-v/p}\right) \tag{8.8}$$

Also observe that

$$\hat{w}_{\mathsf{E}v}(u, v, p) = u\left(1 - \prod_{i=0}^{u/p-1} \frac{u - v - i}{u - i}\right) = u\left(1 - \prod_{i=0}^{u/p-1}\left(1 - \frac{v}{u - i}\right)\right)$$

Thus,

$$\hat{w}_{\mathsf{E}v}(u,v,p) \;\leq,\approx\; u\left(1-\left(1-\frac{v}{u-u/p}\right)^{u/p}\right) \;\leq,\approx\; \frac{uv}{p-1} \tag{8.9}$$

The approximations are best when $u \gg p \gg v$.

If observing a data structure for which the number of unique additions is unknown, we can count the number of affecting additions, n and use that to determine the exact represented set size, which is n times the number of unique elements that map to each partition, $\frac{u}{p}$:

$$w_{\mathsf{E}n} = n\frac{u}{p} \tag{8.10}$$

Thus,

$$f_{\mathsf{E}} = \frac{w_{\mathsf{E}} - v}{u - v} = \frac{n/p - v/u}{1 - v/u} \tag{8.11}$$

Computing the exact false positive rate, therefore, requires knowledge of both v and n. Contrast with the the "balls and bins" scheme, in which we know f_{BB} precisely given only n but need v as well to determine w_{BB} precisely.

Theorem 8.2. *A set over-approximation for v unique elements using an information-theoretic optimal exact representation based on p "even" partitions uses $O(v)$ more bits than an information-theoretic optimal representation giving the same false positive rate.*

Proof This proof is set up analogously to the proof of Theorem 8.1. The proof obligation is

$$\breve{m}_{\hat{n},p,0}/v - \breve{m}_{v,u,\hat{f}}/v = O(1),$$

and we can assume from the previous proof that

$$\breve{m}_{\hat{n},p,0}/v \leq \lg\frac{p}{\hat{n}} + 1.5 \quad \text{and} \quad \breve{m}_{v,u,\hat{f}}/v \geq -\lg\frac{\hat{w}}{u} - 1.5.$$

Given $\hat{n}$, $\hat{w}$ is very simple for "even" partitioning: $\hat{w} = \hat{n}u/p$. Replacing that gives a nicely simple bound:

$$\breve{m}_{v,u,\hat{f}}/v \geq -\lg \frac{\hat{n}}{p} - 1.5.$$

Using these bounds, we can reduce the overall proof obligation to a triviality:

$$\lg \frac{p}{\hat{n}} + 1.5 + \lg \frac{\hat{n}}{p} + 1.5 = O(1)$$

$\Leftrightarrow$

$$3 = O(1)$$

□

8.4 Comparison

Now the question is, "Which approach is better?" First, let us observe that in the limit ($u \to \infty$) the two approaches are indistinguishable:

$$\lim_{u\to\infty} \hat{f}_{\mathsf{E}} = \lim_{u\to\infty} \frac{\hat{w}_{\mathsf{E}}(u,v,p)}{u} = 1 - \lim_{u\to\infty} \prod_{i=0}^{v-1} \left(1 - \frac{1}{p(1-i/u)}\right) = 1 - \left(1 - \frac{1}{p}\right)^{v} = \hat{f}_{\mathsf{BB}}$$

When u is finite, it might appear the "even" approach always has a lower false positive rate, because it is lower for a given number of affecting additions (n). The problem is that the number of affecting additions is expected to be higher (closer to the number of unique additions), precisely because of this lower false positive rate when we start adding to an "even" structure. It is possible the false positive rate for "even" does get worse than "balls & bins" after enough additions.

We now analyze the conditions under which the "even" approach would

have a lower false positive rate:

$$\text{“even” is better}$$

$$\Leftrightarrow$$

$$\hat{f}_{\mathsf{E}} < \hat{f}_{\mathsf{BB}}$$

$$\Leftrightarrow$$

$$\left(u - u\frac{\binom{u-u/p}{v}}{\binom{u}{v}} - v\right)(u-v)^{-1} \;<\; 1 - \left(1 - \frac{1}{p}\right)^{v}$$

$$\Leftrightarrow$$

$$\frac{u-v}{u-v} - \frac{u}{u-v}\prod_{i=0}^{v-1}\frac{u-u/p-i}{u-i} \;<\; 1 - \left(\frac{p-1}{p}\right)^{v}$$

$$\Leftrightarrow$$

$$\frac{u}{u-v}\prod_{i=0}^{v-1}\frac{u-u/p-i}{u-i} \;>\; \prod_{i=0}^{v-1}\frac{p-1}{p}$$

$$\Leftrightarrow$$

$$\frac{u}{u-v} \;>\; \prod_{i=0}^{v-1}\frac{pu-u-pi+i}{pu-u-pi}$$

$$\Leftrightarrow$$

$$\prod_{i=0}^{v-1}\frac{u-i}{u-i-1} \;>\; \prod_{i=0}^{v-1}\frac{pu-u-pi+i}{pu-u-pi}$$

$$\Leftrightarrow$$

$$1 \;>\; \prod_{i=0}^{v-1}\frac{[(u-i)-1][(pu-u-pi)+i]}{(u-i)(pu-u-pi)}$$

$$\Leftrightarrow$$

$$1 \;>\; \prod_{i=0}^{v-1}\left(1 + \frac{ui-i^2-pu+u+pi-i}{(u-i)(pu-u-pi)}\right)$$

$$\Leftrightarrow$$

$$1 \;>\; \prod_{i=0}^{v-1}\left(1 + \frac{(u-i)(i-p+1)}{(u-i)(pu-u-pi)}\right)$$

$$\Leftarrow$$

$$\bigwedge_{i=0}^{v-1}\left[\frac{i-p+1}{pu-u-pi} \;<\; 0\right]$$

$\Leftrightarrow$

$$\bigwedge_{i=0}^{v-1} \left[\frac{i-p+1}{u-u/p-i} \ < \ 0 \right]$$

$\Leftarrow$

$$v - p < 0 \bigwedge u - u/p - v + 1 > 0$$

$\Leftrightarrow$

$$v < p \bigwedge u/p < u - v + 1$$

$\Leftarrow$ {using $v < p$}

$$v < p \bigwedge u/p < u - p + 1$$

$\Leftarrow$

$$v < p \bigwedge p < u \bigwedge p > 1$$

"Even" is expected to have a better false positive rate in at least those cases, including virtually all reasonable cases, and likely many more. If $v \geq p$, then the false positive rate is quite high, at least $(0.63u - v)/(u - v)$. And there is no point in partitioning if $p \geq u$ (exact storage is possible) or $p \leq 1$ (false positive rate 100% after first addition).

"Even" is only noticeably better, however, if u/p is not big, meaning the size of stored hash values is close to the size of the original elements from U. Consider $u = 2^{16}$, $p = 2^{12}$, $v = 2^{10}$. "Balls and bins" gives a false positive rate of 0.2212, while "even" is between 0.2088 and 0.2105. If p is even closer to u, say $p = 2^{14}$, then the difference is even bigger: 0.061 vs. 0.046.

The advantage for "even" partitioning comes from the fact that unrepresented partitions are associated with more unadded elements than represented partitions, which have had at least one of their elements added. As u/p get larger, the proportion of unadded elements in the universe that are in represented partitions gets closer to the proportion of partitions that are represented and the advantage becomes negligible. In "balls and bins" partitioning, the two are always the same (in expectation, assuming random hashing).

These results also help to inform implementation of the inexact-exact reduction. Most importantly, if the descriptors are already uniformly distributed, the "even" approach offers top quality with no [additional] hashing; partitions can be chosen based on a prefix of the descriptor. But the practical difference in accuracy between the two approaches is rarely significant.

8.5 Summary

The point of this chapter is to verify that using an exact representation to implement an inexact representation is reasonable. By my criteria, it is reasonable, because if an exact representation falls within my characterization of "asymptotically compact," using a given reduction to implement an inexact structure results in an "asymptotically compact" solution to that problem. The two natural approaches to this reduction are usually indistinguishable, but the accuracy of "even" partitioning is noticeably better in rare cases—a fact utilized in demonstrating the bounds of Theorem 11.1.

CHAPTER 9

Cleary tables

John G. Cleary's "compact hash table" [14] is a clever twist on a classical design for hash tables, but Cleary tables are remarkably space efficient [65, Section 5]. Cleary tables represent subsets of finite U exactly, but as shown in Chapter 8, we can use such structures for inexact representation. This chapter describes the operation and use of Cleary tables without dynamic adaptation, described in Chapter 10.

The Cleary table is a remarkable structure. If speed is no concern, it can be made to use near minimal space. To maintain good speed, however, it should not exceed about 90% occupancy. Beyond that point, lookups quickly degrade to linear search due to clustering. Unfortunately, linear probing for collision resolution is critical to the design of the structure. On the other hand, the maximum allowed occupancy serves as a space vs. time "control knob".

Contributions I optimize the metadata of the structure to save one bit per cell. I also describe and analyze several variants of the structure, to aid in understanding and to increase the structure's configurability.

9.1 Description

Like the compacted chaining table, we ensure the elements are uniformly distributed, using a randomization function if needed, and then use part of the resulting descriptor as a location, a "home address," and store only the remaining data, an "entry," in a "cell" along with some metadata. The Cleary table consists of a single array of cells, and entries from the same home address are placed in succession in the array, in "runs" [1]. Metadata bits in the cells indicate where runs begin and end and match them up to home addresses. No matter the distribution of input elements, the Cleary table can represent them using just the entries plus two bits of metadata each. Efficient operation, however, depends on reasonably short runs and at least a small proportion cells throughout remaining empty. Figure 9.1 shows the logical structure of part of a Cleary table.

9.1.1 Representation

Now I describe three conceptual metadata bits per entry; later I show that one is redundant, something Cleary did not observe in his paper. (I have also changed some names from Cleary's paper.) There is a MAPPED bit for each home address. For now, we assume one home address for each cell, so we put a MAPPED bit in each cell. This bit is set iff there has been at least one element added with that home address. If the bit is set, that also means there is a run of entries somewhere for that home address. Each cell also has an OCCUPIED and a CHANGE bit, which relate to the entry stored in the cell. The OCCUPIED bit is set iff the cell stores an entry, which could be all zeros. The CHANGE bit, only relevant if the cell is occupied, is set iff the entry is the first in a run.

The nth CHANGE bit that is set to "1" begins the run of entries whose home address is where the nth MAPPED bit that is set to "1" is located. To

[1] In previous publications I have referred to runs as "chains."

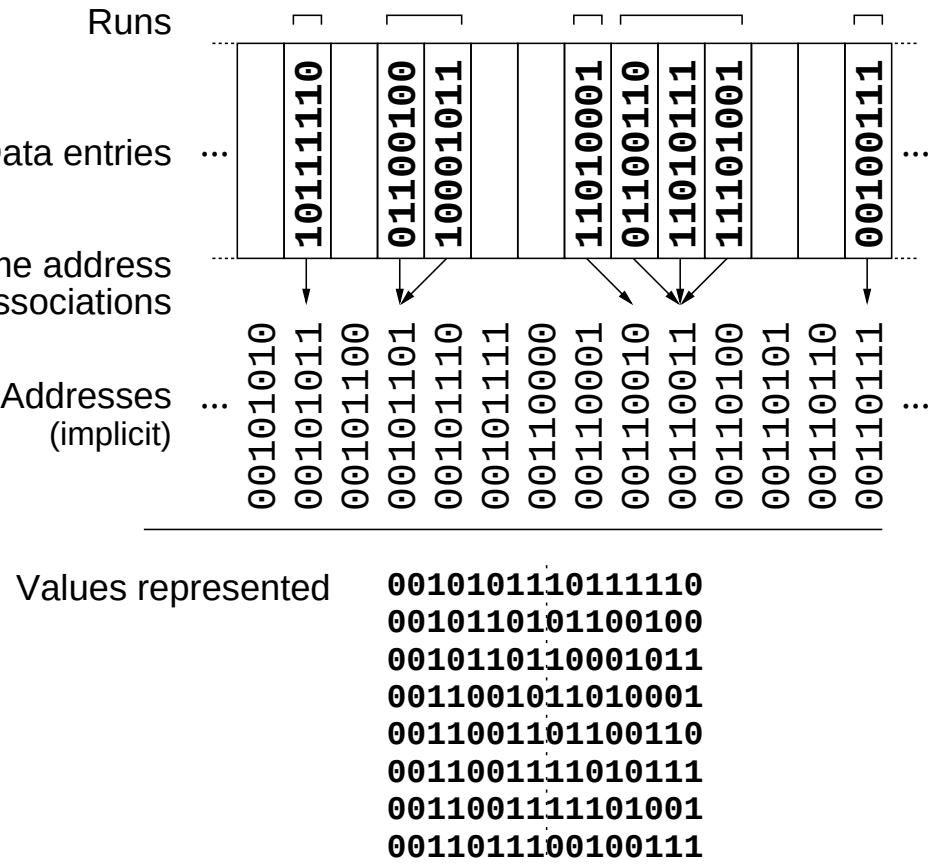

Figure 9.1: *Logical diagram of part of a Cleary table. In this example, the "elements" or "values" are sixteen bits long. The first eight bits of each are the home address, and the remaining eight are stored in the cell entry. The metadata bits are not shown. The logical structure given by the metadata is shown: grouping of occupied entries into runs, each associated with a home address.*

ensure that every OCCUPIED entry belongs to a run with a home address, a Cleary table maintains the following invariant:

Invariant 9.1. *In a Cleary table,* (a) *a cell with its* CHANGE *bit set must also be* OCCUPIED [2]*,* (b) *the first* OCCUPIED *cell (if there is one) has its* CHANGE *bit set, and* (c) *the number of* MAPPED *bits set is the same as the number of* CHANGE *bits set.*

The runs do not *have* to be near their homes for the representation to

[2]This part of the invariant is automatic when the OCCUPIED bits are optimized away.

work, but the order of the runs corresponds to the order of the set mapped bits.

Unlike the OCCUPIED and CHANGE bits, the MAPPED bit is not necessarily associated with the entry in the cell at that location. In fact, if we scrap having one MAPPED bit per cell, it is not necessary for the number of home addresses to be the same as the number of cells, but it seems to make sense for them to be similar in size. (When the structure is full, we expect, asymptotically, $1 - e^{-1} \approx 0.632$ proportion of the MAPPED and CHANGE bits to be set.) See Section 9.5.3.

9.1.2 Random access

For the structure to be fast, each home address must have a preferred location, a single cell, for its possible run of entries. Assuming a home address for each cell, the mapping is trivial; see Section 9.5.3 for discussion of other mappings.

Clearly, runs of two or more entries cannot occupy just their own preferred location. More generally, there are cases in which it is impossible for all runs to include their preferred location. As an example, consider three successive home addresses with three successive preferred locations and elements have been added with each of the three home addresses, with at least two added to the middle one. If the run for the middle one includes its preferred location, it will necessarily occupy the preferred location of one of the other mapped home addresses.

Nevertheless, this next invariant makes fast access the likely case when a portion of cells are left unoccupied:

Invariant 9.2. *In a Cleary table, all cells from where an element is stored through its preferred location (based on its home address) must be occupied.*

This basically says that runs of elements must not be interrupted by empty cells, and that there must not be any empty cells between a run and

its preferred location.

Consequently, when we go to add an element, if its preferred location is free/unoccupied, we store it in that cell, set its CHANGE bit, and, of course, set the MAPPED bit for its home address. We know by Invariant 9.2 that if the preferred location of an element is unoccupied, then no elements with that home address have been added. Consequently, the MAPPED bit is not set and there is no run associated with that home address. Note also that this most basic case of adding preserves both invariants given.

If we're trying to ADD an element whose preferred location is already occupied or QUERY an element whose home address MAPPED bit is set, we must find the run for that home address—or where it must go—in order to complete the operation. Recall that the element stored in the corresponding preferred location may or may not be in the run for the corresponding home address.

To match up runs with home addresses—to match up CHANGE bits with MAPPED bits—we need a "synchronization point". Without Invariant 9.2 the only synchronization points were the beginning and end of the array of cells. With Invariant 9.2, however, unoccupied cells are synchronization points. Thus, to find the nearest synchronization point, we perform a bidirectional search for an unoccupied cell from the preferred location. This is called `findEmpty` in the algorithm in Section 9.2. (Variants described in Section 9.5.4 do not treat the beginning and end as synchronization points.)

From an unoccupied, unmapped cell, we can backtrack, matching up MAPPED bits with CHANGE bits, until we reach the run corresponding to the home address we are interested in–or the point where the new run must be inserted to maintain the proper matching (`searchLeft` and `searchLeft` in the algorithm). If we are querying, we can just iterate through the run to complete the query. If we are adding and the element is found to be new, there must be space to add it. Luckily, we already found the empty cell nearest the preferred location of the run we're adding to (unless we

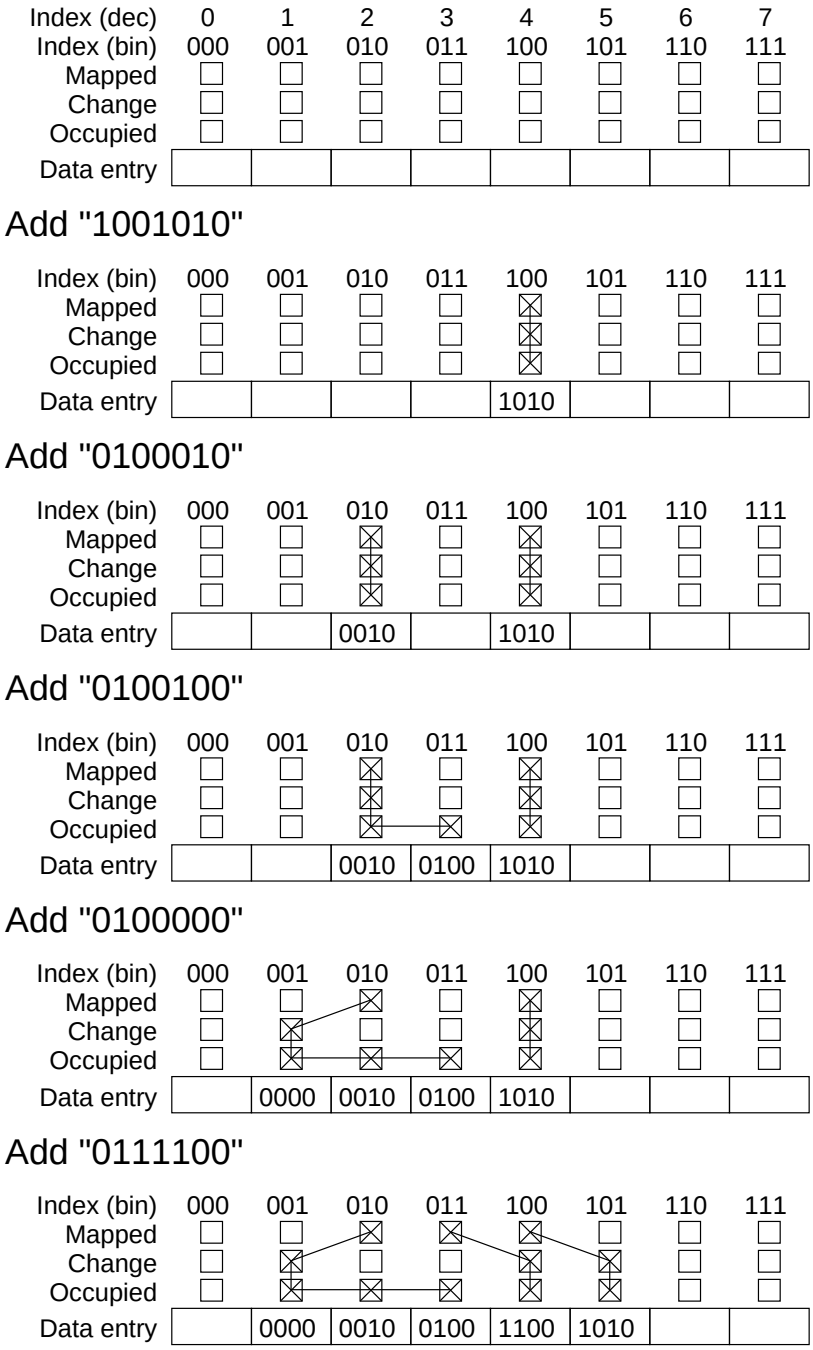

Figure 9.2: *Adding five elements to a Cleary table with eight cells. In this example, the elements are seven bits long, the home addresses are three bits long, and the cell data entries are, therefore, four bits long. Each cell is shown with three metadata bits (*MAPPED*,* CHANGE*, and* OCCUPIED*). The lines connecting various metadata bits depict how the metadata bits put the entries into runs associated with home addresses.*

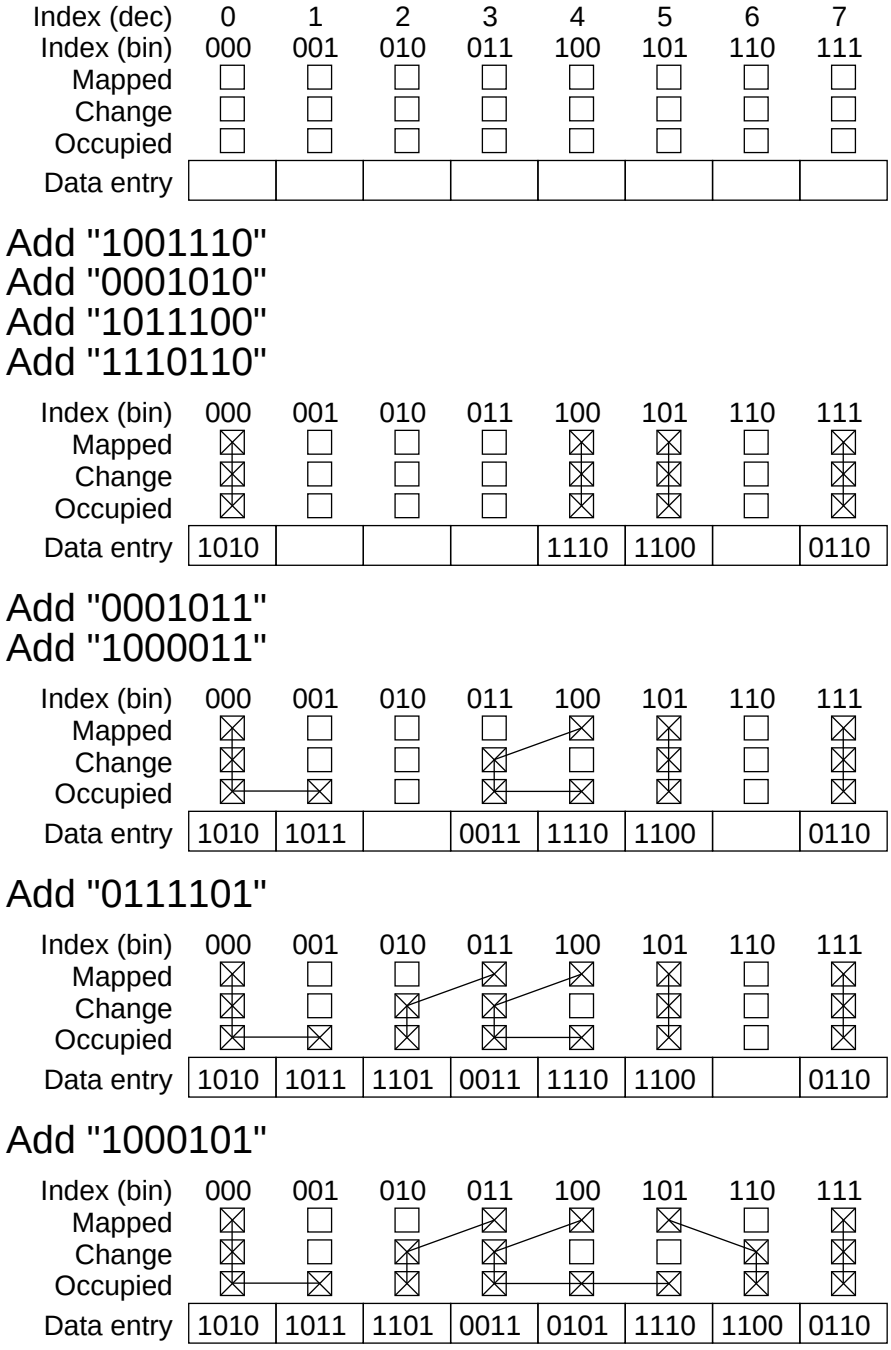

Figure 9.3: *A complex example, filling a Cleary table with eight cells. The table uses the same configuration as in Figure 9.2. I will re-emphasize that the lines connecting metadata bits depict for the reader how those bits are interpreted; the connecting lines do not add any information.*

used the beginning or end as a synchronization point). We simply shift all entries (and their OCCUPIED and CHANGE bits) toward and into the empty cell, opening up a space in the correct run—or where the new run must be added (`shiftForLeft` and `shiftForRight` in the algorithm). The rest is a matter of keeping the invariants on CHANGE and MAPPED bits straight. For details, see the algorithm in Section 9.2.

Figure 9.2 shows what happens to the data in a Cleary table as elements are added. The first two additions are simple, but each of the next three has unique, potentially tricky aspects to it. Figure 9.3 has additional examples.

9.1.3 An optimization

We add one more invariant, originally intended to speed up lookups, but I show how it can be used to eliminate the OCCUPIED bits:

Invariant 9.3. *In a Cleary table, all entries in a run are put in low-to-high unsigned numerical order.*

This makes adding slightly more complicated, and probably does not affect much the time per ADD (likely less searching, likely more shifting). Negative QUERYs can be a little faster, because an element's presence can be ruled out without scanning the whole run. Note that this optimization of negative QUERYs is insignificant in the visited set paradigm, in which negative QUERYs become ADDs.

With Invariant 9.3, the OCCUPIED bits are redundant,[3] because if we fill all unoccupied entries with all zeros and make sure their CHANGE bit is unset, then entries with all zeros are occupied iff their CHANGE bit is set. This is because entries with all zeros will always be first in their run, so they will always have their CHANGE bit set.

[3] In [65, Section 5], a Cleary table was claimed to have two bits of metadata per cell, but this was later admitted to be an oversight rather than an unsubstantiated new claim. In context, the oversight was minor.

This optimization precludes the encoding of multisets by repeating entries, because only one zero entry per run is possible under the optimized encoding.

9.2 ADD algorithm

Here I present a basic Cleary table implementation suitable for use as a visited set; it has only an ADD operation that returns a flag indicating whether the structure needed modification to include the given value.

Because it is reasonably complex and I want to be sure it is correct, I present it in a tested, executable form. It is written in Java, third edition [35], and tested using the Sun Microsystems/Oracle "Java SE 6.0."

The Cleary table is encapsulated in the class **ClearyTable**. The beginning of the class definition includes fields for configuration information and the underlying memory table, a constructor, and methods to abstract away the memory. The method definitions that follow, ending with `add`, use those methods to manipulate the structure only in terms of the metadata bits and cell entries. The **BitVector** class used by the **ClearyTable** should be understood in context. An implementation is available from the author.

This implementation should have the same computational complexity as the best known algorithm, but I focused on making this implementation easy to read. As such, this implementation makes some simplifying assumptions, such as that inputs are already uniformly distributed (hashed if needed), that represented values are no larger than 64 bits, and that the structure is less than about 256MB in size. My scalable, highly-tuned implementation is in 3SPIN, which is written in C.

```
public class ClearyTable {
    protected int addr_bits;  // # of bits in an address
    protected int entry_bits; // # of bits in an entry
    protected int num_cells;  // # of cells
    protected int cell_bits;  // # of bits in a cell
    protected BitVect table;  // the bit vector for the cells

    /** Constructor, called in creating new ClearyTables. */
    public ClearyTable(int addr_bits0, int entry_bits0) {
        addr_bits = addr_bits0;
        entry_bits = entry_bits0;
        num_cells = 1 << addr_bits; // 2 to the addr_bits
        cell_bits = entry_bits + 2; // two metadata bits
        // total bits needed = num_cells * cell_bits
        table = new BitVect(num_cells * cell_bits);
    }

    // Where things are in the bit layout of each cell
    static final int MAPPED_OFFSET = 0;
    static final int CHANGE_OFFSET = 1;
    static final int ENTRY_OFFSET = 2;
    protected boolean getMapped(int addr) {
        return table.getBit(addr * cell_bits + MAPPED_OFFSET);
    }
    protected void setMapped(int addr, boolean v) {
        table.putBit(addr * cell_bits + MAPPED_OFFSET, v);
    }
    protected boolean getChange(int loc) {
        return table.getBit(loc * cell_bits + CHANGE_OFFSET);
    }
    protected void setChange(int loc, boolean v) {
        table.putBit(loc * cell_bits + CHANGE_OFFSET, v);
    }
    protected long getEntry(int loc) {
        int bitPos = loc * cell_bits + ENTRY_OFFSET;
        return table.getUpto64(bitPos, entry_bits);
    }
    protected void setEntry(int loc, long v) {
        int bitPos = loc * cell_bits + ENTRY_OFFSET;
        table.putUpto64(bitPos, v, entry_bits);
    }

    protected boolean isOccupied(int loc) {
        return getEntry(loc) != 0 || getChange(loc) != false;
    }
```

```
protected int findEmpty(int startLoc) {
    int fwd = startLoc;
    int rev = startLoc - 1;
    for (;;) { // loop until return
        if (fwd < num_cells) {
            if (!isOccupied(fwd)) return fwd;
            fwd = fwd + 1;
        }
        if (rev >= 0) {
            if (!isOccupied(rev)) return rev;
            rev = rev - 1;
        }
    }
}

static enum SearchFlag {
    /** Value already in the table. */
    FOUND,
    /** Need to insert a run here for home address. */
    NEW_RUN,
    /** Insert here as beginning of run to the right. */
    BEGIN_RUN,
    /** Insert here into same run as is on the left. */
    CONTINUE_RUN
}

static class SearchResult {
    /** Whether value was found, and if not, how to add it. */
    SearchFlag flag;
    /** Where the value was found, or where to add it. */
    int loc;

    SearchResult(SearchFlag flag0, int loc0) {
        flag = flag0;
        loc = loc0;
    }
}

protected void copyEntryAndChange(int src, int dst) {
    setChange(dst, getChange(src));
    setEntry(dst, getEntry(src));
}
```

```
/**
 * Search for given entry with given home address to the left
 * of given empty location.
 * Precondition: targetHome < emptyCell
 *                  && !isOccupied(emptyCell)
 */
protected SearchResult searchLeft(long targetEntry,
        int targetHome, int emptyCell) {
    int curHome = emptyCell;
    int curLoc = emptyCell;
    for (;;) { // loop until break
        do {
            curHome = curHome - 1;
        } while (curHome > targetHome && !getMapped(curHome));
        if (curHome == targetHome) break;
        do {
            curLoc = curLoc - 1;
        } while (!getChange(curLoc));
    }
    if (!getMapped(curHome)) {
        // run should be inserted here
        return new SearchResult(NEW_RUN, curLoc);
    }
    // otherwise, curLoc is after existing run for home
    for (;;) { // loop until return
        curLoc = curLoc - 1;
        long curEntry = getEntry(curLoc);
        if (targetEntry > curEntry) {
            // belongs after the current position
            return new SearchResult(CONTINUE_RUN, curLoc + 1);
        }
        if (targetEntry == curEntry) {
            // found!
            return new SearchResult(FOUND, curLoc);
        }
        if (getChange(curLoc)) {
            // belongs at current position, after shifting,
            // as beginning of same run
            return new SearchResult(BEGIN_RUN, curLoc);
        }
    }
}
```

```
/**
 * Search for given entry with given home address to the right
 * of given empty location.
 * Precondition: emptyCell < targetHome
 *                  && !isOccupied(emptyCell)
 */
protected SearchResult searchRight(long targetEntry,
        int targetHome, int emptyCell) {
    int curHome = emptyCell;
    int curLoc = emptyCell;
    do {
        do {
            curHome = curHome + 1;
        } while (curHome < targetHome && !getMapped(curHome));
        do {
            curLoc = curLoc + 1;
        } while (curLoc < num_cells && isOccupied(curLoc) &&
                  !getChange(curLoc));
    } while (curHome < targetHome);
    // now curHome == targetHome
    if (!getMapped(curHome)) {
        // run should be inserted before curCell
        return new SearchResult(NEW_RUN, curLoc - 1);
    }
    // otherwise, curLoc is beginning of existing run for home
    long curEntry = getEntry(curLoc);
    if (targetEntry < curEntry) {
        // belongs before current position, beginning of the run
        return new SearchResult(BEGIN_RUN, curLoc-1);
    }
    for (;;) { // loop until return
        if (targetEntry == curEntry) {
            // found!
            return new SearchResult(FOUND, curLoc);
        }
        curLoc = curLoc + 1;
        if (curLoc >= num_cells || !isOccupied(curLoc) ||
              getChange(curLoc)) {
            // belongs before current, as end of preceding run
            return new SearchResult(CONTINUE_RUN, curLoc-1);
        }
        curEntry = getEntry(curLoc);
        if (targetEntry < curEntry) {
            // belongs before current position
            return new SearchResult(CONTINUE_RUN, curLoc-1);
        }
    }
}
```

```
protected void clearEntryAndChange(int loc) {
    setChange(loc, false);
    setEntry(loc, 0);
}

protected void shiftForLeft(int empty, int loc) {
    while (empty > loc) {
        copyEntryAndChange(empty - 1, empty);
        empty = empty - 1;
    }
    // for clarity, clear shifted "empty" cell here
    clearEntryAndChange(empty);
}

protected void shiftForRight(int empty, int loc) {
    while (empty < loc) {
        copyEntryAndChange(empty + 1, empty);
        empty = empty + 1;
    }
    // for clarity, clear shifted "empty" cell here
    clearEntryAndChange(empty);
}
```

9.3 Analysis

In a Cleary table with c cells and c home addresses, each data entry is $\lg u - \lg c$ bits before any rounding off, and with two bits of metadata per entry, the whole structure uses just $c(\lg u - \lg c + 2)$ bits.

To add v elements, we must have enough cells: $c \geq v$. If we use the minimum sufficient for the Cleary table to represent any such set, $c = v$, the space usage is (in bits)

$$v(\lg u - \lg v + 2) \leq \breve{m}_{v,u,0} + 2v$$

(see Equations 4.3 and 4.6). Note that Theorems 8.1 and 8.2 generalize the Cleary table's asymptotic near-optimality to the inexact set representation problem, using either reduction in Chapter 8.

The $2v$ "extra" bits in the representation, used for metadata, are modest,

```
/**
 * Add the given value if not already added.
 * Returns true iff the structure was modified.
 */
protected boolean add(long value) {
    // split up value into an entry and a home address
    int home = (int)(value >> entry_bits);
    long entry = value & ((1L << entry_bits) - 1);

    // find nearest empty cell
    int empty = findEmpty(home);

    // search for where entry should go, using empty
    // cell as synchronization point
    SearchResult r;
    if (home == empty) {
        r = new SearchResult(NEW_RUN, home);
    } else if (home < empty) {
        r = searchLeft(entry, home, empty);
        if (r.flag == FOUND) return false; // no modification
        shiftForLeft(empty, r.loc);
    } else { // (home > empty)
        r = searchRight(entry, home, empty);
        if (r.flag == FOUND) return false; // no modification
        shiftForRight(empty, r.loc);
    }

    // was not found, and empty cell was shifted to r.loc
    setEntry(r.loc, entry);
    assert getChange(r.loc) == false;
    if (r.flag == NEW_RUN) {
        assert getMapped(home) == false;
        setMapped(home, true);
        setChange(r.loc, true);
    } else if (r.flag == BEGIN_RUN) {
        assert getChange(r.loc + 1) == true;
        setChange(r.loc + 1, false);
        setChange(r.loc, true);
    } // nothing for r.flag == CONTINUE_RUN
    return true; // added
}

} // end of class definition, for Section 9.2
```

except when v is close to u. That overhead renders the bit table (see Section 5.2) superior to the Cleary table when v is close to u. If $c = u/4$, then each cell has two bits of entry data and two bits of metadata, for a total of four bits. But $4c = u$ is exactly how many bits would be used by a bit table, and the bit table has O(1) access time and cannot overflow. Thus a bit table should be used instead of a Cleary table when $c \geq u/4$. With 90% maximum occupancy, that is $v \geq 0.225u$.

Allowing the structure to fill up, however, does not allow for random access. To add v *and* preserve random access, some proportion of the cells need to be left unoccupied. Pagh et al. have analyzed this in detail [66], but our rule of thumb is that the structure becomes unacceptably slow beyond 90% occupancy (see Figure 9.4). Note that their analysis shows that a hash function with sufficient independence among sets of inputs is required for average access times to scale perfectly [66].

Thus, if $\alpha = \frac{v}{c}$ is the proportion of cells occupied, the memory usage in terms of v and α is (in bits)

$$\alpha^{-1}v(\lg u - \lg v + 2 - \lg \alpha^{-1})$$

which is a small constant factor (less than α^{-1}) from the near-optimal memory usage of the full Cleary table.

9.4 Validation

9.4.1 Speed

Because the primary motivator for my study of Cleary tables is explicit-state model checking, I examine how its speed for various final occupancies compares to standard Bloom filter configurations. Note that because the structure is used as a visited set, the effect the final occupancy has on the structure's overall "speed" is smaller than for most applications, because it is

accessed just as heavily for occupancies up to the final one.

Figure 9.4 shows an example in which 90% occupancy is about 10% slower than 50% occupancy. Specifically, the verification time for a protocol with little overhead increases by about 10% if we increase final occupancy from 50% to 90%. I manipulate final occupancy by controlling the number of cells in the Cleary table. (See Section 9.5.2 for how I implemented non-power-of-two numbers of cells.)

The Cleary table is the same speed as the $k = 3$ Bloom filter at about 85% occupancy. At about 50% occupancy, the Cleary table matches the $k = 2$ Bloom filter's speed. This should be less problem-specific than configuration-, system-, implementation-, and compiler-specific. Also, these comparisons are essentially for "cumulative speed" because explicit-state model checking follows the visited set usage paradigm.

9.4.2 Compactness

For exact storage With the help of my OCCUPIED bit optimization, the Cleary table is a remarkably compact, exact representation of a set. To demonstrate its relative compactness, consider using it to represent the state space of a 2x2x2 Rubik's Cube. I use Antti Valmari's analysis of the problem and compare to his solutions [82].

The problem has $3\,674\,160$ states of 31 bits each. Valmari also wishes to associate with each state the rotation at which it was first found, which is one of six values. This means we need to associate $\lg 6 \approx 2.58$ bits of "satellite info" with each entry, or 3 bits if it is not packed.

A simple open-addressed table (Section 5.1) would need $31 + 3$ bits per state, unless we pack the rotations. Valmari points out that standard C++ data structures require many times this because of structural overhead and internal fragmentation, and because less than 1% of representable states are reachable, a bit table (Section 5.2) requires hundreds of bits per state.

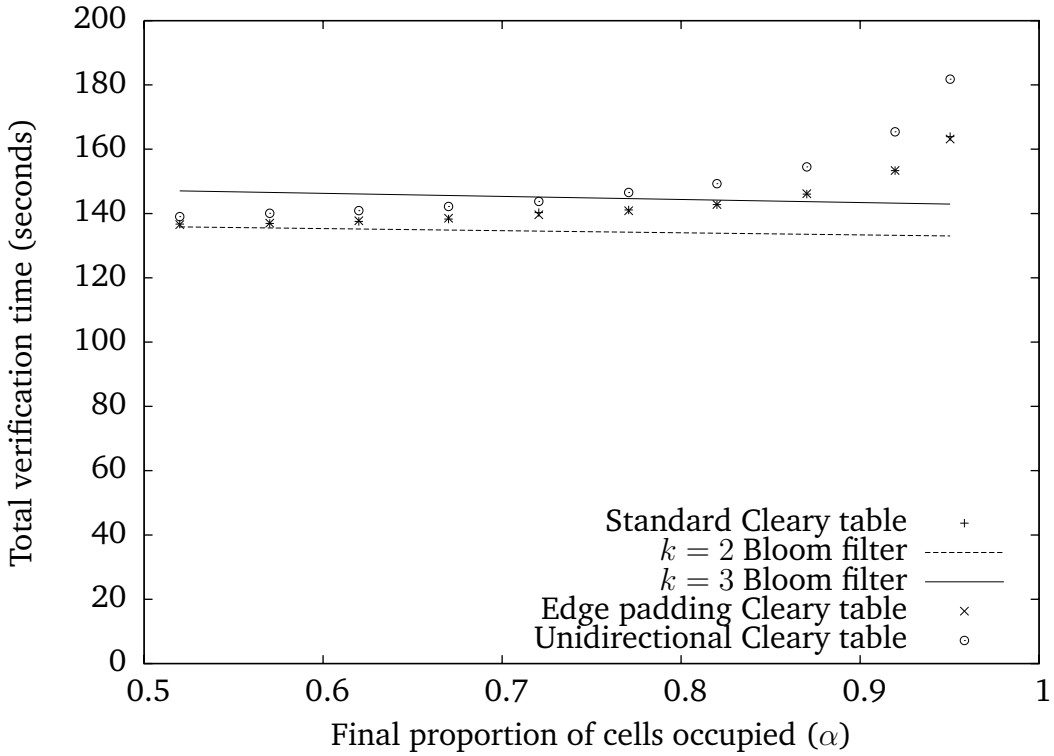

Figure 9.4: *Verification times using standard Bloom filters and Cleary tables of various final occupancies. Lower is better. This graph primarily shows how increases in Cleary table access times due to higher final occupancies increase overall verification time in an explicit-state model checker. In each case, I use 3SPIN (based on* SPIN *5.1.7) on an instance of the PFTP protocol with a 152-byte state vector and 104 million states. The Cleary table configurations used 64-bit cells, and I manipulated the final occupancy by manipulating the total memory for the table, from 837 MB (95%) to 1530 MB (52%). The difference in times for each Bloom filter configuration is due to using the different memory sizes used by the Cleary tables. To keep other overhead in the model checker near minimal, I disabled partial order reductions; thus, because* SPIN *is well optimized and the protocol is reasonably simple, these results represent about the largest difference in speed one would observe in practice. The non-standard Cleary tables are described in Section 9.5. Tests were compiled with gcc 4.4.3 (-O3) and run on a 64-bit Linux system with Intel Xeon X5677 CPUs.*

The "fast" variant of Valmari's compacted chaining structure (Section 5.3) requires 24.17 bits per state. The "slow" variant requires 19.37 bits per state.

For a Cleary table, we need at least $\lg 3\,674\,160 = 21.81$ address bits. Conveniently, if we round up to 22 address bits (2^{22} cells), we get $\alpha = 0.876$, which is acceptably fast. In this case, the Cleary table, with 3-bit rotation info, needs $31 - 22 + 2 + 3 = 14$ bits per cell, or $14\alpha^{-1} = 15.98$ bits per state. That's closer to the information-theoretic lower bound Valmari derives, 13.22 bits/state, than to his most optimized compacted chaining solution.

If we store the rotations in a separate table and pack groups of six of them in 16 bits, we have $31 - 22 + 2 + 2.67 = 13.67$ bits per cell, or 15.61 bits per state. (This is a fair comparison, because Valmari's tables the rotations in with metadata in the structure.)

The Cleary table is not always the most compact among the structures I have presented. The open-addressed table (Section 5.1) is more compact for very sparse sets, and the bit table (Section 5.2 is more compact for dense sets.

For inexact storage Using either reduction from Chapter 8 (the difference is usually negligible), we can use the Cleary table to over-approximate a set. Its compactness is usually a little worse than that of the compacted hash table (Chapter 7); the degree of the difference depends on the maximum occupancy used in the Cleary table, which also factors heavily into the speed.

Figure 9.5 shows such a comparison between the Cleary table and the other inherently inexact structures presented. Keep in mind that this figure does not incorporate what I consider to be the true strength of the Cleary table as an inexact representation: the ability to adapt it dynamically. See Chapter 10.

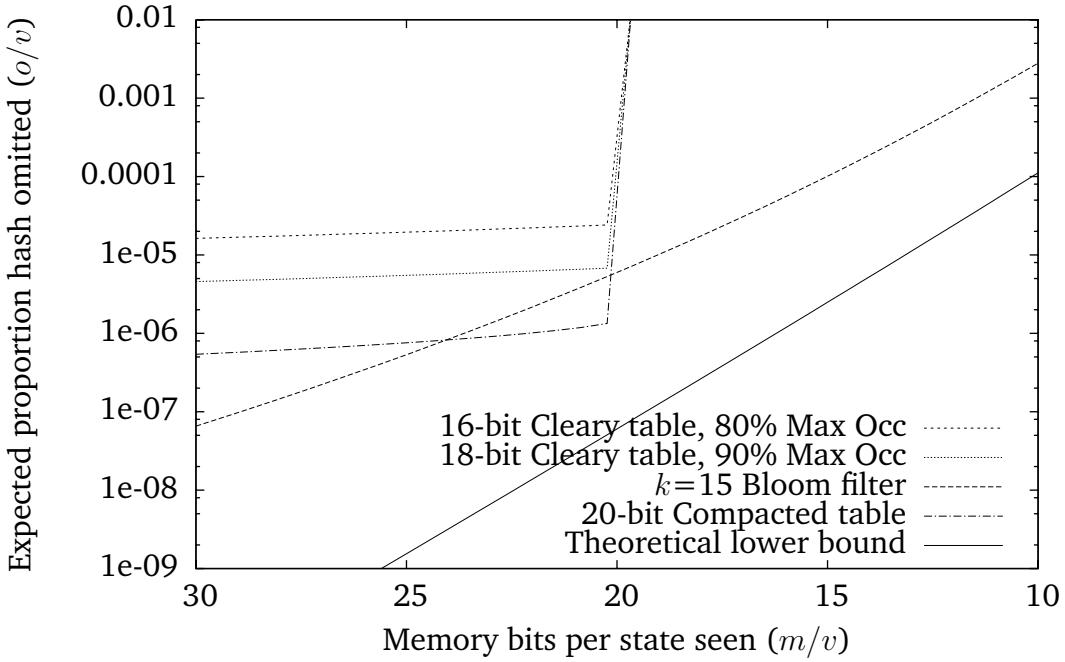

Figure 9.5: *Comparison of inaccuracy of comparable Bloom filter, compacted table, and Cleary table configurations. Lower is better. This essentially shows the expected hash omissions for various numbers of states seen, using various data structure configurations, but the axes are chosen to be independent of the actual magnitude of memory or states. They instead show the proportion of states expected to be hash omissions vs. the ratio of memory to states seen, "Theoretical lower bound" assumes states are taken from a universe large enough not to permit exact storage near the domain of this graph. All the configurations are chosen to optimize the $m/v \approx 20$ case. "16-bit Cleary table" refers to 16 bits per cell. The Y axis spans seven orders of magnitude.*

9.5 Variations

9.5.1 Mini-pointers (sometimes useful)

In Cleary's original paper, he describes a way of speeding up accesses by using a little more space per cell [14, Section III.B]. The idea is to speed up synchronization of MAPPED bits and CHANGE bits by storing with each MAPPED bit the location offset of the matching CHANGE bit, if it falls within the representable range of offsets. In most cases, the offset will be small

enough to be represented exactly and synchronization will be immediate. When the real offset is out of range, we must search for a synchronization point, such as an association whose offset is in the representable range.

However, this is mostly a speed improvement for QUERY, because ADD must still search for an empty cell to shift entries into. In the visited set usage paradigm, the improvement is mostly when QUERY returns positive, because each negative QUERY becomes an ADD. In explicit-state model checking with partial-order reduction, the number of ADD operations often exceeds the number of positive QUERY operations. Thus, this space-vs.-time trade-off is unlikely to be a good choice for the visited set paradigm. (Related work in Chapter 12 is similarly optimized for fast QUERY operations.)

Recall that there is already a space-vs.-time trade-off in the maximum allowed occupancy; thus, to be worthwhile, any space dedicated to these miniature pointers (location offsets) should be recouped in the increase in maximum occupancy made possible. Clearly, the payoff of these mini-pointers will depend on the absolute cell size without the mini-pointers. For example, an extra 4 bits per cell for mini-pointers increases the cell size of an 8-bit-per-cell Cleary table by 50%; therefore, they should make a 96% occupied structure as fast as a 64% occupied structure without mini-pointers to have a net benefit. For a 64-bit-per-cell table, the payoff is around 96% occupancy with mini-pointers being as fast as 90% occupancy without mini-pointers.

Even if there is a net payoff for large cell sizes in the visited set paradigm, mini-pointers add another layer of complexity to the implementation, particularly when using adaptation (Chapter 10). I did not feel the added complexity was worth the limited payoff to pursue this optimization.

9.5.2 Non-power-of-two number of cells (sometimes useful)

So far we have assumed that the number of cells (and home addresses) in a cleary table is a power of two. If we can control the range of values representable, then it is reasonably easy to accommodate any number of cells. In the common case of using a Cleary table to over-approximate a set (see the reduction in Chapter 8), if we can get a non-power of two range from our hash function then we can get a non-power of two number of cells.

For example, if we have 175MB of memory for the table and want at least 40 million cells, how big should the cells be and what should the range of the hash function be? $1\ 468\ 006\ 400$ bits for 40 million cells is 36.70 bits per proposed cell. Dealing with fractions of bits for the cells is too much of a hassle, so we round down to 36 bits per cell (2 metadata, 34 entry). In that case, we can have $40\ 777\ 955$ ($= 1468006400/36$) cells, or approximately 25.28 ($= \lg 40777955$) address bits. Combining the address and entry, each hash value should be a non-negative integer less than 700559932491038720 ($= 40777955 \times 2^{34}$), or approximately 59.28 ($= 34 + 25.28$) bits. If we use a stock hash function producing 64 bits of output, we can use 34 bits for the entry, take the remaining 30 bits as an unsigned integer and `mod` it by 40777955.

Another way to support a number of cells that is not a power of two is to allow the number of cells and home addresses to be different, as described next. This allows us to use a power-of-two number of home addresses with any number of cells.

9.5.3 Different number of cells and home addresses (sometimes useful)

The Cleary table is more adaptable without the requirement of exactly one MAPPED bit per cell. The algorithms require little adjustment to treat home

addresses (the nth MAPPED bit) as distinct from preferred locations (the mth cell), but a good mapping from home addresses to preferred locations is important. Also needed is a decoupling of cells and MAPPED bits in the physical layout, so that the numbers of each can be chosen freely.

This section supports these basic findings: the best home-address-to-cell ratios seem to be from 1.0 to 2.0, a good mapping is just a scaling from the domain of home addresses to the range of preferred locations, and putting the MAPPED bits in their own bit vector is the easiest layout.

9.5.3.1 The sweet spot

Starting simple, consider the following modification to the standard Cleary table: suppose there are twice as many home addresses (twice as many MAPPED bits) but the same number of cells (CHANGE bits and entries). In this case, the mapping from home addresses to preferred locations would be a single bit right shift, so that each cell is the preferred location for two (adjacent) home addresses. Because we have one more address bit, we don't need one of the old entry bits. This means we're storing the same information in the same amount of space, and negative queries are likely to be much faster, because the probability of the MAPPED bit being unset is nearly double.

Can we take that a step further? If we double the number of home addresses again we increase the MAPPED bits per cell from two to four, while removing the need for only one entry bit per cell. This means that taking another step further in the direction of more MAPPED bits makes the structure take more space.

Let us consider going in the other direction: modifying the standard Cleary table to have half as many MAPPED bits: one for every two cells. In that case, we save one MAPPED bit for every two cells at the cost of one more entry bit for every cell. This means that a Cleary table with a home-

address-to-cell ratio of 0.50 requires more memory than one with a ratio or 1.0 or 2.0. We also saw how 4.0 required more memory. This suggests that the 1.0 to 2.0 range is a "sweet spot".

9.5.3.2 In theory

We can actually set up this optimization problem as a formula and set the derivative equal to zero to find the configuration that minimizes memory required. Recall that if c cells are in the table, the memory required is $c(\lg u - \lg c + 2)$ bits, but since we are decoupling the number of cells and number of home addresses, it is really $c(\lg u - a + 1) + 2^a$, where a is the number of home address bits. (There are 2^a MAPPED bits and c CHANGE bits, and each entry is $\lg u - a$ bits.)

To minimize that with respect to a, we can set its derivative equal to zero, and get

$$a = \lg \frac{c}{\ln 2}$$

or an optimal home-address-to-cell ratio of

$$\frac{2^a}{c} = \frac{1}{\ln 2} \approx 1.44$$

This makes sense because it corresponds to maximizing the entropy in the MAPPED bits when the structure is completely full. If α is the proportion of cells occupied and r is the home-address-to-cell ratio, the expected proportion of MAPPED bits set is

$$1 - e^{-\alpha/r}$$

For $\alpha = 1$ and $r = 1/\ln 2$, this is $1/2$, which maximizes entropy in the MAPPED bits.

In practice, it is too awkward to compactly store entries that are not a whole number of bits, so using $r = 1.44$ in every case does not make sense.

Instead, one should use the whole number of entry bits that puts r between 1.0 and 2.0.

9.5.3.3 Non-power-of-two ratio

As claimed previously, we can support a non-power-of-two number of cells with a power-of-two number of home addresses, and here is how that works. For a given number of cells, we use a number of home addresses that is the next power of two larger than the number of cells. This yields a home-address-to-cell ratio in the sweet spot of 1.0 to 2.0, and has the advantage that the number of home addresses is a power of two, so the address portion of each element is a whole number of bits.

An obscure ratio, however, complicates efficient implementation. Laying out 1.0 or 2.0 MAPPED bits per cell is simple, but something like $r = 1.38$ is more difficult. I see three options:

1. Intersperse the MAPPED bits among the cells, and use complicated logic to address each entry and metadata bit. (A special case of this is used in the "accurate" variant of adaptive storage in Chapter 11.)

2. Round the number of MAPPED bits per cell up to the next whole number, so that everything is easily addressable, but the extra MAPPED bits are wasted unless they are also used as home addresses. (The method of implementing non-power-of-two cells in Section 9.5.2 can use such extra MAPPED bits.)

3. Store the MAPPED bits in their own bit vector. (This option is easiest, but reduces memory locality and reduces dynamic adaptability in that it is not compatible with the algorithms in Chapter 10.)

The last main piece needed is a mapping from home addresses (MAPPED bits) to preferred locations (cells with CHANGE bits). To minimize access times, the preferred locations should be spread as evenly as possible over

all the cells, so that the cell occupancy is consistent throughout. A poor mapping, for example, would be the home address MOD the number of cells. This would create a contiguous region where each cell is the preferred location for two home addresses and another region where each cell is the preferred location of just one home address. This would create an imbalance in which lookups could degrade to linear search at occupancies well below 100%. There could easily be no unoccupied entries in one contiguous half of the structure at just 80% occupancy, for example.

A good mapping is to multiply the home address by r^{-1} and round down to get the preferred location. (**Implementation note:** Floating point arithmetic is fine. Store r^{-1} to avoid repeated division, which might be much slower than multiplication.) Like before, some cells are the preferred location for two home addresses each and the rest for just one each. In this case, however, the twos are spread evenly throughout.

It might seem that there is no avoiding the local imbalance in the "good" mapping, which surely has a small effect on access times, but I suspect it is correctable by considering the entire element or hash value—not just the home address—in computing the preferred location. It might seem that every element with the same home address must have the same preferred location, but I believe that can be relaxed to allow elements with the same home address to map to one of two adjacent preferred locations—using the same algorithms. In this dissertation, I do not investigate this possible enhancement any further, largely because it would complicate the adaptation algorithms in Chapter 10.

9.5.4 Edge extension or edge wrapping (marginal benefit)

One of the concerns sometimes raised about the "standard" Cleary table I have described is that entries might "bunch up" at the edges of the structure, where runs can expand only in one direction. The edges do cause

imbalances in the structure, but perhaps not in the way one might expect. We can enhance the Cleary table to correct these imbalances and improve access times, but the imbalances and the improvements seem to disappear asymptotically.

Cleary briefly discusses alternative strategies for handling edges [14, Section II, last paragraph], proposing one could either make the array "circular," which I shall call *edge wrapping*, or make extra cells available beyond those that serve as preferred locations, which I shall call *edge padding*. The latter was proposed by Amble and Knuth [2].

My testing consistently shows that in a standard Cleary table, cells close to the extremes are *less* likely to be occupied. See Figure 9.7 for experimental results. This might be the opposite of what one expects, but it nevertheless leads to a small imbalance in expected occupancy. See Figure 9.6 for a rough explanation of this behavior.

It is also clear from Figure 9.7 that edge wrapping is balanced. This is easy to believe, because with edge wrapping, all cells are created equal, with no prior advantages or disadvantages to any particular cells.

In that same figure, edge padding has imbalances, but has consistently lower occupancy, which is likely to translate into lower access times. This is somewhat misleading, though, because edge padding requires more space for the extra cells. If we compared them with each using the same amount of space, padding would be much less attractive for its imbalances, but such a direct comparison might be misleading, because the amount of padding to preallocate is subjective (more below).

Comparing Figure 9.8 to Figure 9.7 indicates that the imbalances due to edge handling are really only significant for small Cleary tables. For large Cleary tables, the "balance" of an edge handling scheme should have little significance in choosing, because the imbalances are insignificant. Some other factors in choosing an edge handling scheme follow.

Although edge wrapping is very "balanced," it is tricky to implement. An

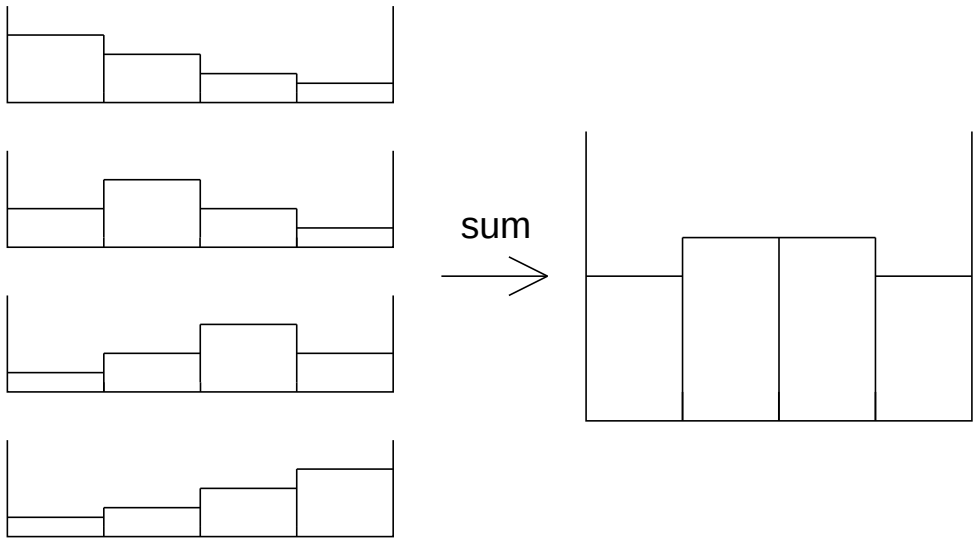

Figure 9.6: *A rough explanation for small imbalances in the expected cell occupancy of a standard Cleary table. Consider a Cleary table with four cells. For each home address, consider the possibility of there being a run associated with it and where the occupied cell or cells for that run would be located. Roughly speaking, add these up for all home addresses, and that suggests the probability of each cell being occupied. The basic idea is that cells on the edge are less likely to be occupied because "pressure" can only come from one direction.*

example of a tricky aspect of implementation is that the empty cell search needs to report not only the location of the nearest empty cell, but also the direction of the nearest empty cell. Without wrapping, the direction is easy to determine from the location, using greater-than and less-than comparisons. With wrapping, the location might be "to the right" even though it was found by searching "to the left" (and vice-versa). This and the other issues are solvable, but it complicates the invariants of an already complicated algorithm.

Also note that edge wrapping requires at least one cell to remain unoccupied, because the edges of the table are not synchronization points if edge wrapping is used.

I suspect that edge padding was not entirely motivated by relieving "edge pressure." I suspect it was used to eliminate bounds checks, simplifying the

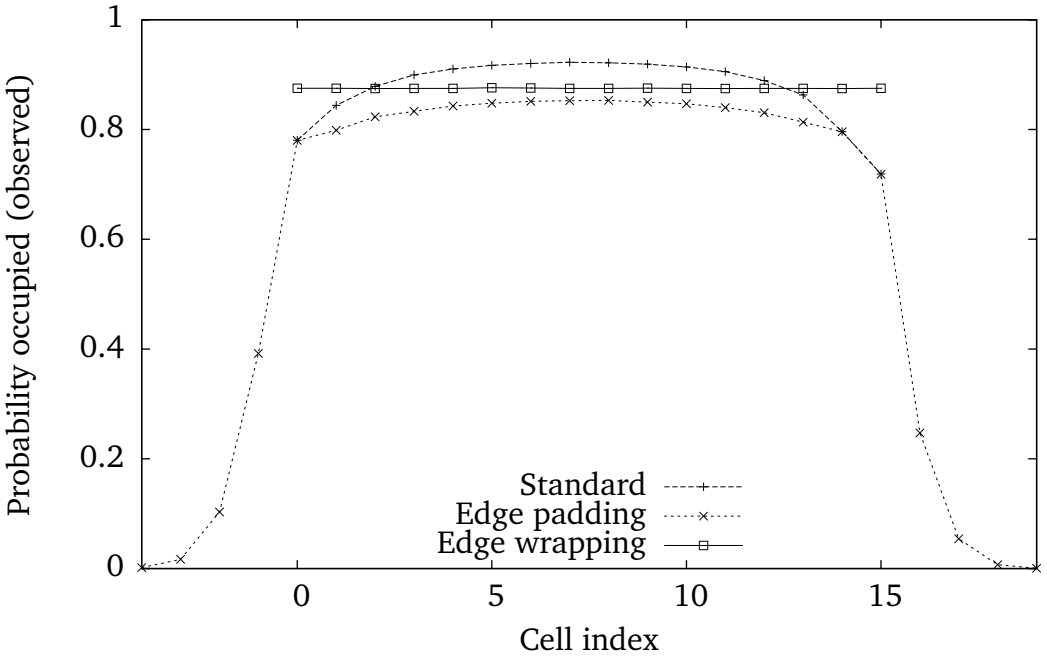

Figure 9.7: *Observed occupancy probabilities for each cell in Cleary tables with different edge behaviors. These results came from repeatedly simulating 14 unique, random additions to Cleary tables with 16 cells, not including padding. The nominal overall occupancy is therefore* $14/16 = 0.875$ *(*87.5%*). For each design, we ran one million iterations, each adding 14 values to a fresh table. The graph shows for each cell position the proportion of iterations in which that cell was occupied after the additions. The "Edge padding" design is the only one that uses cells beyond the standard 16, numbered* 0 *to* 15*.*

standard implementation (rather than complicating it) and eliminating some conditional branches in the CPU. The thinking is that if the padding is large enough that overrunning it is highly unlikely, then the simpler, faster implementation with no bounds checking works—with high likelihood. However, such an implementation could threaten both stability and security, and contemporary computing practices favor eliminating such threats.

To be sure edge handling has no effect for large structures, I evaluated the speed of a structure using edge padding in Figure 9.4. There was no observable difference between that and the standard Cleary table. Because this edge-padding structure used extra memory the standard did not, we

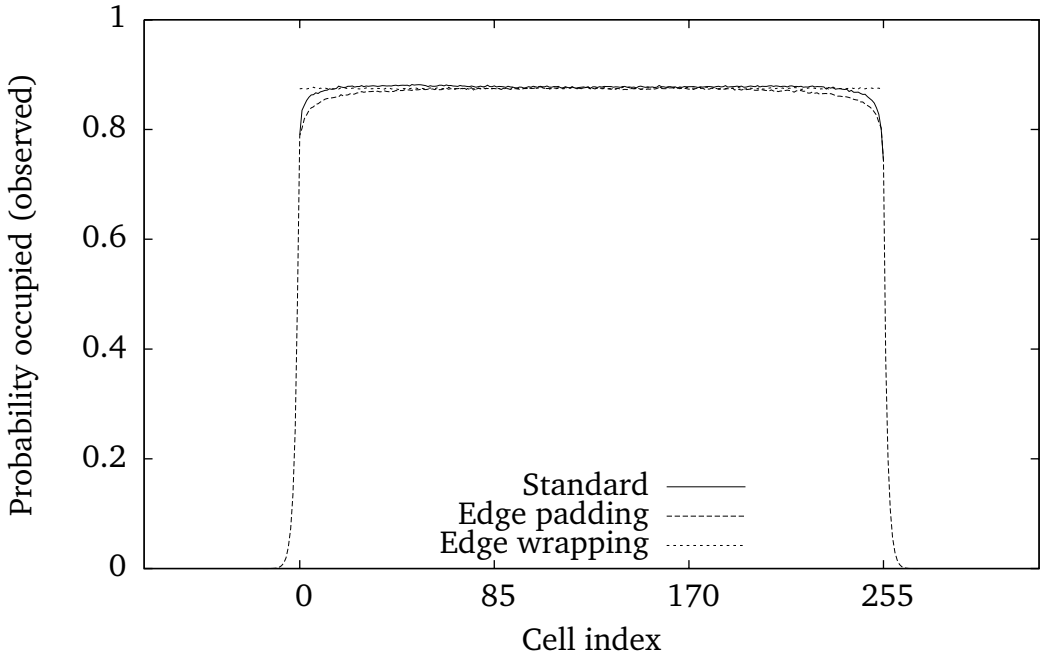

Figure 9.8: *Observed occupancy probabilities for each cell in larger Cleary tables with different edge behaviors. These results are essentially a re-run of the experiment from Figure 9.7 with sizes increased by a factor of 16. In each of* 100 000 *iterations for each design, 224 values were added to 256 cells, not including padding. Comparing these results to Figure 9.7 suggests that occupancy imbalances due to edge handling are asymptotically insignificant.*

would expect edge wrapping to be between the two. (It has not been worth the effort to make a full, optimized implementation of edge-wrapping.)

9.5.5 Correcting directional favor (marginal benefit)

A careful examination of Figure 9.7 reveals that the non-wrapping results are lopsided. The reason is that in a standard implementation, the bidirectional search for an empty cell favors the same direction every time. In this case, it favors left, evident by the "0" cell being more likely to be occupied than its mirror cell "15." In the empty search, after checking the preferred location, there is no avoiding the choice of whether to check the location to the left first or check to the right first. Making the same choice every time, however,

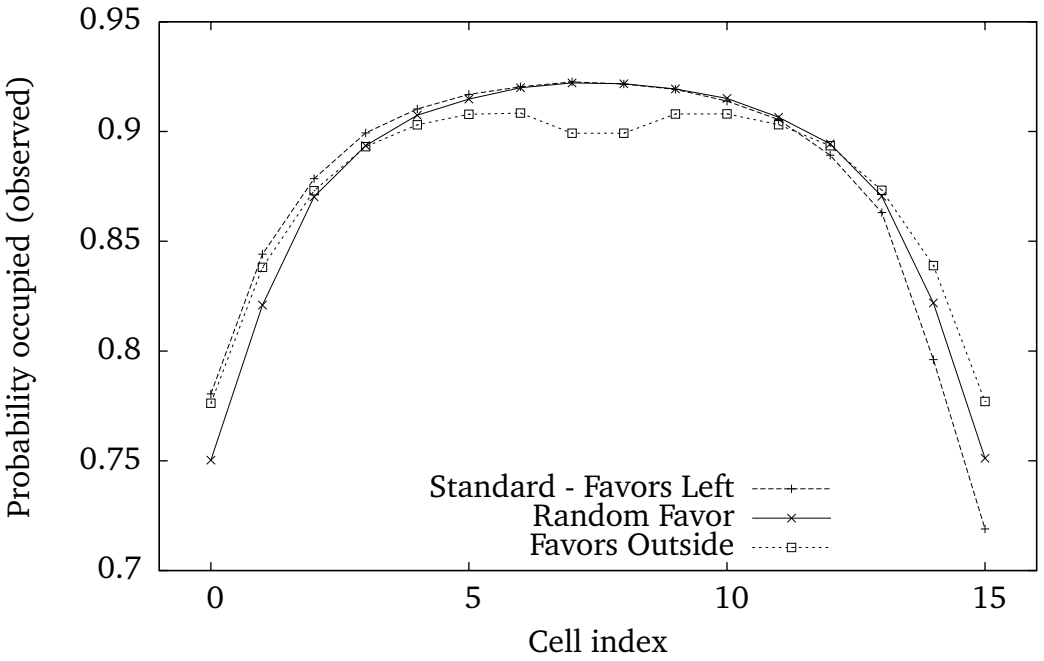

Figure 9.9: *Observed occupancy probabilities for each cell in Cleary tables with different directional favors. These results are based on the same setup as Figure 9.7, except that the "Random Favor" and "Favors Outside" configurations are new. All configurations here use standard edge handling. The "Favors Left" results are the same as the "Standard" from Figure 9.7.*

adds to imbalances due to edge handling. (Note that edge wrapping seems to suffer no ill effect.)

Besides using edge wrapping, there are at least two basic remedies. First, we can effectively randomize which direction to favor, either by acquiring a pseudorandom bit or by using the least significant data entry bit.Second, we can attempt to use the favor to counteract the imbalances due to edge handling, and always favor the "outside," the direction of the nearest edge.

Experimental results in Figure 9.9 indicate that randomizing corrects the lopsidedness and that favoring the outside goes further in reducing the imbalance of standard edge handling (and of edge padding; not pictured). The difference should not matter for large structures, however.

9.5.6 Unidirectional (not recommended)

Unidirectional linear probing can be used in place of bidirectional linear probing in a Cleary table, but comes with a significant increase in access times. It also requires either edge wrapping or edge padding to prevent premature overflow. Basically, we modify the unoccupied search to search only in the forward direction. The rest of the algorithm needs no modification; however, the logic handling the case of the empty cell coming before the preferred location can be removed.

Figure 9.10 is a visual explanation of why bidirectional probing is cheaper overall for a Cleary table. The difference in work involved in adding or querying a value is not in the search for an empty cell, which is similar on average. The difference is in what comes after the empty search. Because the bidirectional empty search finds the empty cell *nearest* to the preferred location, the work in finding or making room for the current value is, in expectation, about half of what it is with unidirectional probing.

Figure 9.11 shows how the different edge handling strategies interact with unidirectional probing with respect to cell occupancies. Edge wrapping continues its robust showing of balance in cell occupancies, while edge padding is particularly imbalanced when used with unidirectional probing.

Figure 9.4 shows the actual speed of using unidirectional probing has a significant impact on verification times. It seems to nearly double the impact of occupancy. This makes sense, since there's a mixture of operations that match an existing entry and those that insert a new entry.

Finally, there is an interesting relationship between "standard" edge handling and unidirectional probing. In a standard Cleary table, the closer to an edge the preferred location for an access is, the more that access behaves like a unidirectional access. If the preferred location is at the edge, the empty search proceeds in just one direction, and we expect to re-search after the empty search about as many cells as we covered in the empty search, rather

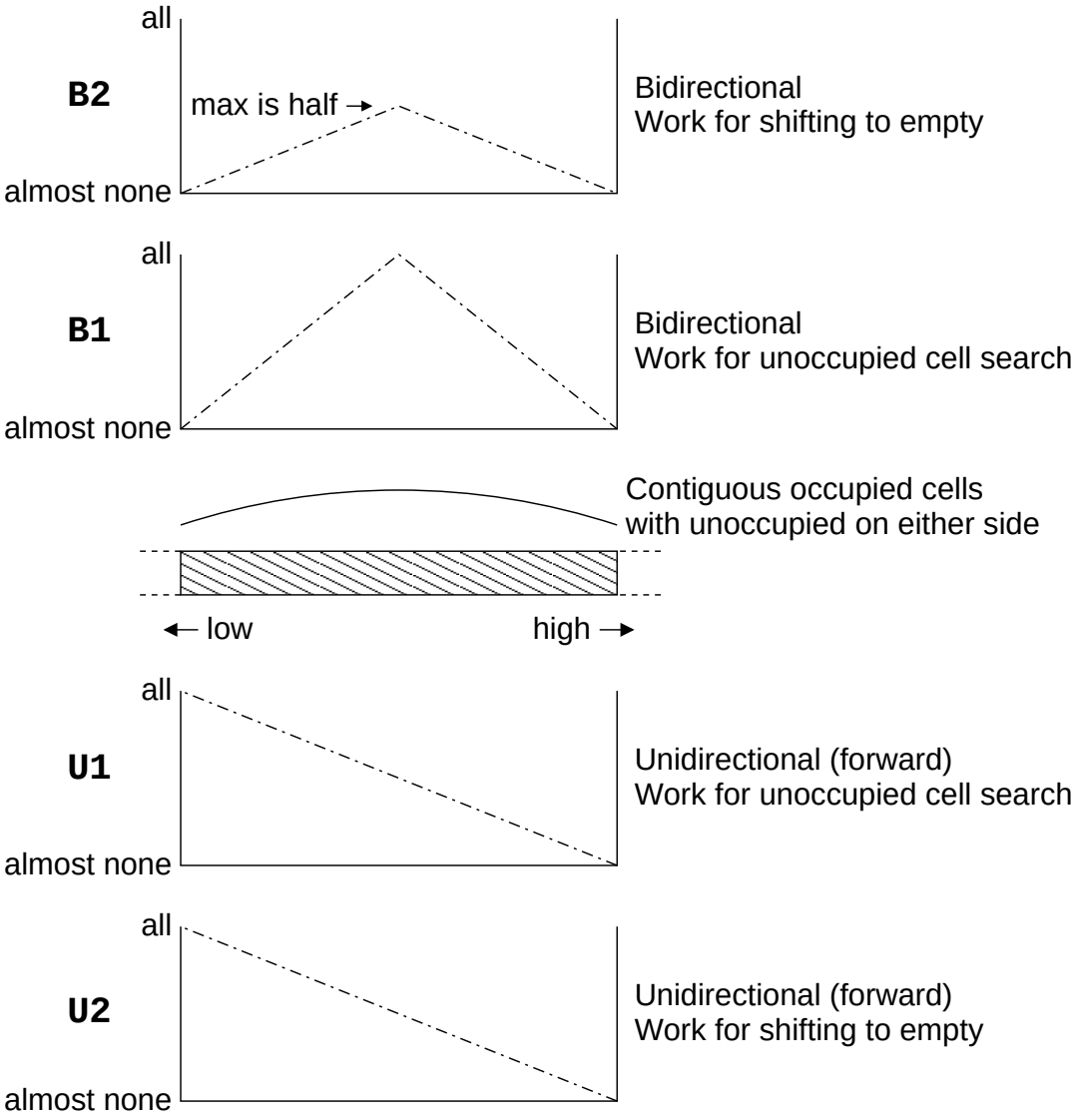

Figure 9.10: *The idea behind why bidirectional linear probing is more efficient than unidirectional in a Cleary table. Consider a contiguous sequence of occupied cells in a Cleary table with unoccupied cells on each side. The* `B1` *and* `U1` *graphs show what portion of the sequence is searched before an unoccupied cell is found, for each starting position in that sequence. For this part of the operation, the aggregate costs (area under each curve) are the same. The* `B2` *and* `U2` *graphs show what portion of the sequence must be shifted for inserting a new element, for each insertion position in that sequence. Here, the aggregate cost (area under the curve) for bidirectional probing is much lower, because shifting is always done toward the nearest unoccupied cell.*

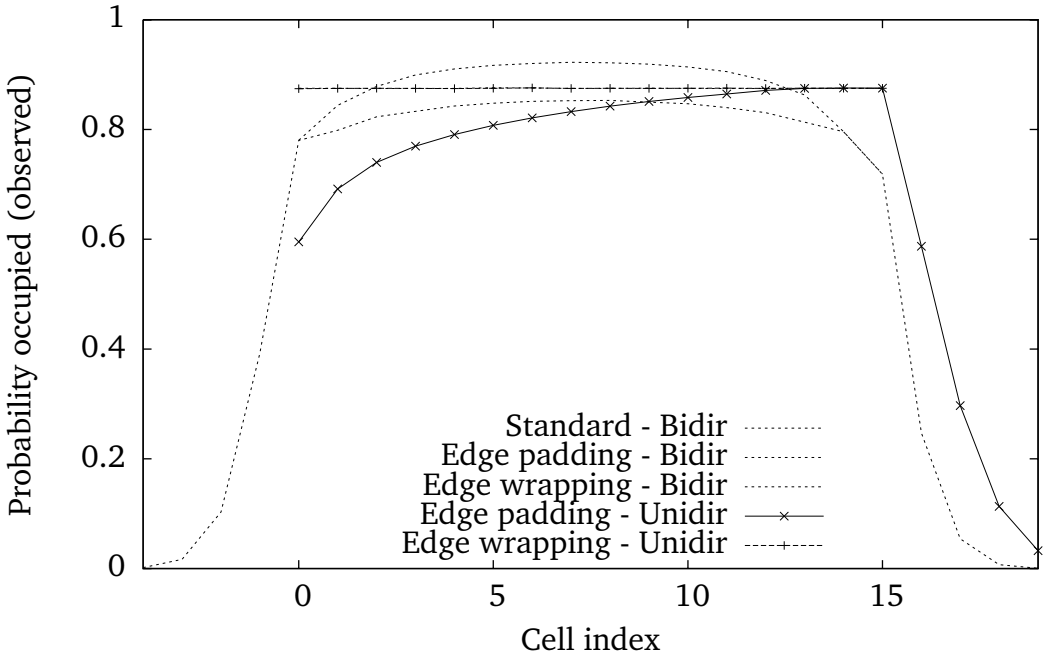

Figure 9.11: *Observed occupancy probabilities for each cell in Cleary tables using unidirectional linear probing. These results are based on the same setup as Figure 9.7, except that the "Unidir" configurations are new. The results from Figure 9.7 are labelled here as "Bidir" because they use bidirectional linear probing. The two configurations using edge wrapping are indistinguishable from their cell occupancies.*

than about half as many, which is the usual case for a table using bidirectional probing.

9.5.7 Summary of Variations

A structure has limited usefulness when it only supports sizes that are powers of two. I have described two ways of supporting non-power-of-two sizes, each with its own strengths for certain environments. Understanding how to best to decouple cells and home addresses, involved in one of the solutions, is not just about a solution to a technical problem; it also gives us a deeper understanding of the standard Cleary table.

Although the standard Cleary table has imbalances that favor some cells

over others, this is not a problem for large tables, and is correctable with edge wrapping nevertheless. I chose not to consider the edge wrapping Cleary table the "standard" version because it is more difficult to explain and more difficult to implement. Cleary himself included edge padding in his standard table [14, Section II, last paragraph], but I do not see any good reasons to use it and I prefer to avoid its issues and complications.

I cannot think of any reason to use unidirectional probing in a Cleary table, unless perhaps one were accessing a Cleary table on disk, or one really needed to optimize for small code size. Bidirectional is faster.

Getting faster QUERY times by using mini-pointers is likely to pay off for large cell sizes, but such an space-vs.-time trade-off is at a disadvantage in the visited set paradigm, which relies heavily on adding more elements. Particularly when using adaptation (Chapter 10), the added code complexity should also be considered.

9.6 Summary

As an exact representation of a set, the Cleary table is remarkably compact, with the help of my optimization of its metadata. It is also among few such structures that scale perfectly, with an adequate hash function. As an over-approximation of a set, the Cleary table has accuracy similar to a Bloom filter, except when memory per added element is very low. The structure is fast unless it is allowed to exceed about 90% occupancy, because it uses linear probing. However, one can use the maximum occupancy to trade speed for compactness/accuracy.

The next chapter utilizes what I consider the true strength of the Cleary design: the potential to adapt the structure dynamically to accommodate more elements.

CHAPTER 10

Dynamic adaptation of Cleary tables

A key insight of this dissertation is that the Cleary table representation allows for fast, on-the-fly adaptation to accommodate more elements in the same space with less precision. **Contribution:** Here I describe algorithms for such adaptation and the "closer-first" traversal that underlies them. The traversal is based on properties of Cleary tables that I prove.

We are able to adapt between various Cleary table configurations and also to a $k = 1$ or hash-reusing $k = 2$ Bloom filter. The simplest Cleary table to Cleary table adaptation is doubling the number of cells by cutting the size of each in half. More complicated is splitting each two cells into three and, as needed, those three into four. Other adaptations are possible, but I focus on those that work well in the adaptive storage scheme of Chapter 11.

Except for the 2-to-3 and 3-to-4 adaptations below, these algorithms supported the adaptive scheme we introduced in a SPIN Workshop paper [26]. The scheme is elaborated in Chapter 11.

10.1 Understanding fast adaptation

The basic algorithm depends on a traversal I call "closer-first," in which all table entries between a given entry and its home address are processed before that entry is. As described below, this enables the adaptation to happen in place, with $O(1)$ auxiliary storage, and moving each entry at most once.

Before describing the algorithm, I will motivate the need for a "closer-first" traversal by considering the drawbacks of other conceivable solutions. For this example, I focus on the case of adapting a Cleary table to have twice as many cells of half the size.

Consider an example in which each cell is 32 bits ($b = 30$ entry bits, the "payload") before adaptation and 16 bits ($b' = 14$ entry bits) after adaptation. If there are 2^{20} cells before adaptation ($a = 20$ address bits), then there are twice as many, 2^{21} cells, after adaptation ($a' = 21$ address bits). Since each hash value is composed of the part in the cell entry and the part implicit in the home address, the size of each represented hash value is the sum of the two sizes: $b + a = 50$. Or after adaptation: $b' + a' = 35$. The represented hash values after adaptation are fifteen bits smaller. Note that although sixteen bits were lost per cell entry, one was gained in each address.

More generally, for any number of address bits, a, and even number of entry bits b, we can derive the new number of address bits, a', and entry bits b' after adapting to twice as many cells of half the size. With twice as many cells after adaptation, $a' = a + 1$. With each cell having two bits of metadata (see Chapter 9) and cells being half the size after adaptation, $\frac{1}{2}(b+2) = b'+2$, or $b' = b/2 - 1$.

The goal of adapting a Cleary table of hash values is for the new Cleary table to represent truncated versions of those same hash values. In our example, we have to choose 15 bits to throw away and 35 to keep, and we have to divide those 35 into 21 address bits and 14 entry bits. The following rule seems to be important to fast, in-place adaptation of Cleary tables and also dictates how the hash value bits should be migrated:

Rule 10.1. *The order of hash values in a Cleary table should be the same before and after adaptation.*

The goal of this rule is to minimize the space, time, and conceptual complexity of the adaptation procedure. See Figure 10.1 for a "before and after"

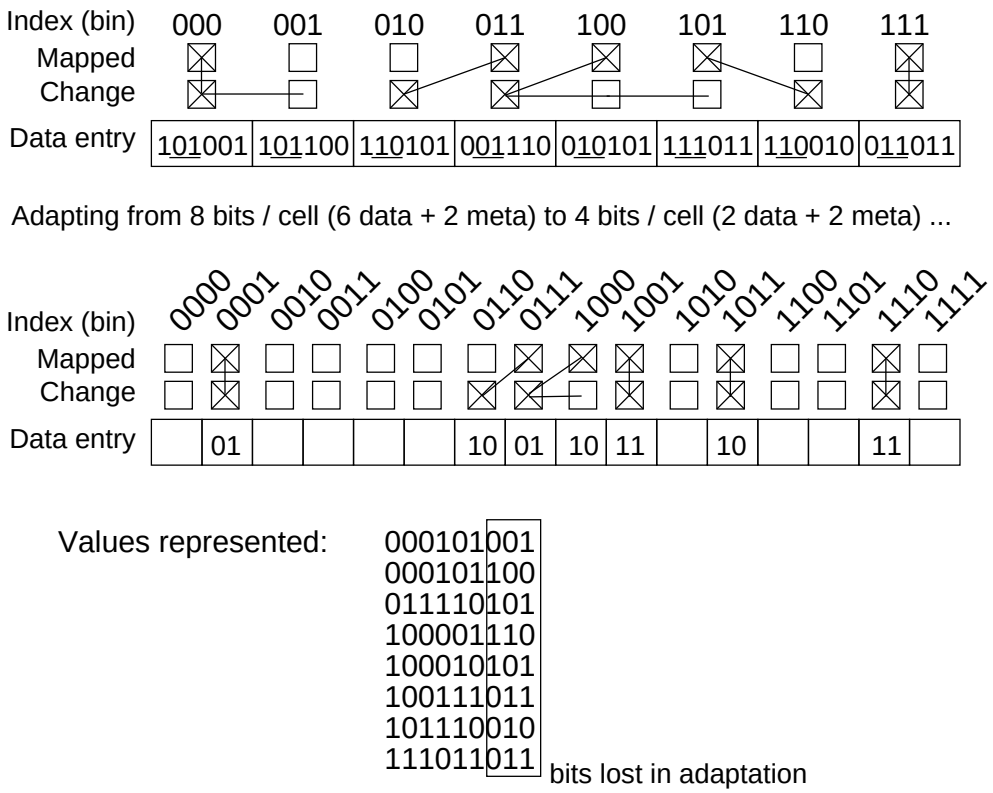

Figure 10.1: *A simple "before and after" example of 1-to-2 Cleary table adaptation. Notice that the total memory bits, 64, is the same before and after adaptation, and that the order of the represented values is unchanged. The underlined portion of the data entry before adaptation is the data entry after. Besides the underlined, one entry bit becomes an address bit and three entry bits are lost. In this example, the first two values are merged because they are only distinguishable by the bits lost in adaptation. An equally valid result of adaptation, representing the same values, would have some of the middle entries shifted to the right by one cell; the result shown matches the result of my algorithm in Section 10.3. Note that the four-bit-per-cell Cleary table is used only for demonstration purposes, as it is inferior to a simple bit table (Section 9.3).*

example of adaptation, going from eight bits per cell to four and keeping the order of represented values the same.

Ideally, the adaptation would be possible with a linear scan over the entries, adapting each in place. This is actually possible if we were to give up Invariant 9.2, but to have random access, we must guarantee no unoccupied cells between an entry and its preferred location. Because of all the extra free space opened up by adaptation, we have to move entries closer to their preferred locations to satisfy the invariant. The example in Figure 10.1 is on point. If adaptation were to put each entry into one of the two new cells made from the old cell containing it, there would be unoccupied cells between some entries and their preferred location, violating Invariant 9.2.

If we attempt to move entries to their new locations one-by-one using a simple linear scan, we fail because the new location in memory might be occupied by an entry we have not yet moved. In simple terms, some entries must be moved to the "left" and some to the "right," as in the entries at addresses 110 and 010 in Figure 10.1, respectively. Thus, neither a left-to-right nor a right-to-left pass is free of interference.

If one used more than $O(1)$ auxiliary space, one could go entirely left-to-right or right-to-left, by queuing up states to be written later. For our problem, however, we want almost all of core memory dedicated to the Cleary table. In some cases, it might be reasonable to use out-of-core storage (disk) for adaptation, but my solution should prove that not necessary.

How efficiently can we visit the entries in an order that ensures the destination location is always available for writing? My solution is close to a simple linear scan, in both practical and theoretical terms, as described in the next section.

10.2 Closer-first traversal

If we are adapting a Cleary table to a configuration with smaller entries, and the memory for the original preferred location of each entry encompasses the memory for the new preferred location, a "closer-first" ordering allows entry-by-entry adaptation without overwriting entries not yet adapted. In other words, if we follow this certain ordering, we can read each entry from its location in the the original structure, throw away part of it, and store the new entry in its new location—one entry at a time.

10.2.1 Description

The ordering constraint is this:

Definition 10.2. *A **closer-first** traversal of Cleary table entries is one that guarantees that the entries between any given entry and its preferred location (inclusive) are visited before that entry.*

To understand why this ordering constraint is feasible and prevents overwriting entries, we need a lemma on the structure of Cleary tables:

Lemma 10.3. *Consider each maximal sequence of adjacent, occupied cells in a Cleary table, in low-to-high ("left" to "right") order. Each can be divided uniquely into subsequences each with the following structure:*

- ***Zero or more** "right-leaning" entries each with its preferred location higher than its actual location, followed by*
- ***One** "pivot" entry at its preferred location, followed by*
- ***Zero or more** "left-leaning" entries each with its preferred location lower than its actual location.*

See Figure 10.2 for an example of categorizations and subsequences.

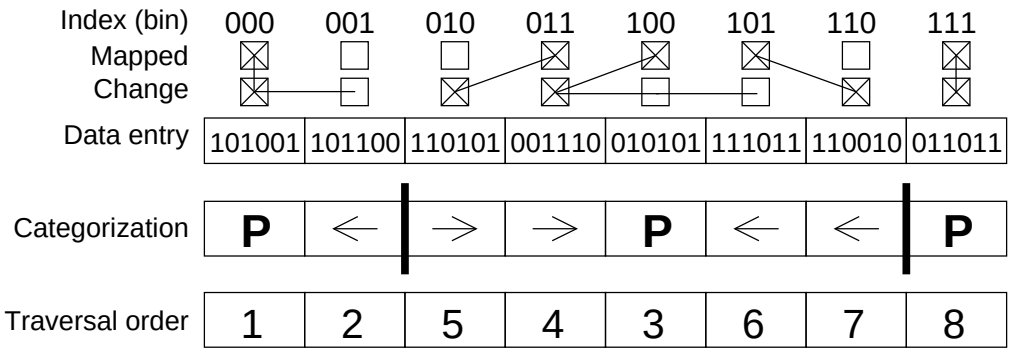

Figure 10.2: *Categorizations and traversal order for Cleary table adaptation. This is the same example as Figure 10.1. The "Categorization" line shows the categorizations and subsequences for Lemma 10.3. The "Traversal order" line shows the order of visitation for the standard closer-first traversal (Definition 10.5).*

Proof First, each cell in a Cleary table can be categorized as either (1) unoccupied, (2) right-leaning, (3) pivot, or (4) left-leaning. Invariant 9.1 ensures that every occupied cell stores an entry associated with a preferred location, so each occupied cell must be either right-leaning, a pivot, or left-leaning, depending on whether its preferred location is higher than, equal to, or lower than its actual location, respectively.

With categories on each cell, we prove the lemma using finite automata. Consider the possible sequences of categorizations of all cells in a well-formed Cleary table. We reduce the lemma to the proof that the automaton of Figure 10.3 (upper) allows all such sequences. But first we prove the reduction, which requires that all sequences allowed by the automaton to satisfy the lemma. First, observe that each time we see a cell in a particular category, we move to the state with that label. This allows us to easily couple those with concepts in the lemma, by considering each transition in the automaton:

- From "(start)" to "right-leaning" is allowed because the first cell in the Cleary table could be right-leaning, as the start of a subsequence.
- From "(start)" to "pivot" is allowed because the first cell in the Cleary

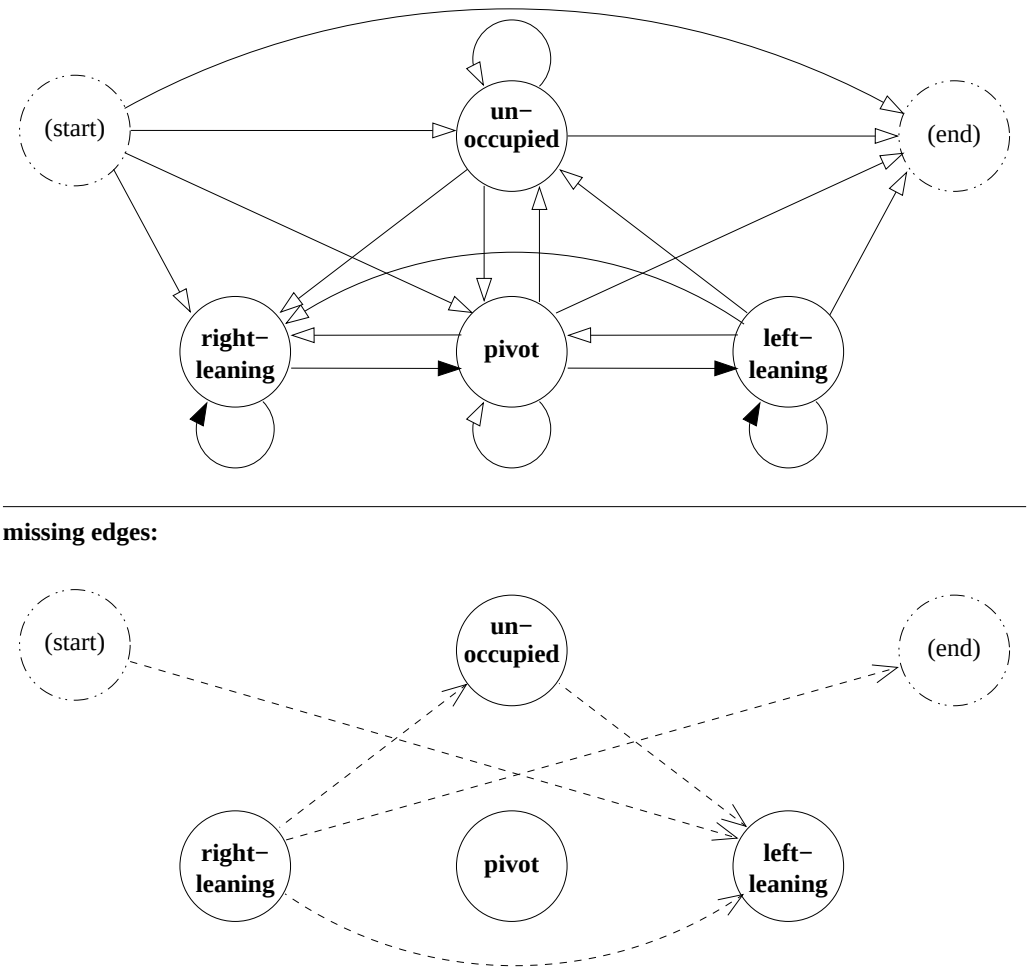

Figure 10.3: *Allowed and disallowed categorization combinations of adjacent cells in a Cleary table. The upper diagram is a finite automaton constraining the possible sequences of cell categorizations in a Cleary table. The automaton matches the categorization of all cells in sequence, starting before the lowest location, "(start)," and ending after the highest, "(end)." To prove that the automaton allows all categorization sequences in a well-formed Cleary table, we prove that each missing edge, depicted in the lower diagram, is infeasible. We use this result in the proof of Lemma 10.3. (In the lower diagram, trivially infeasible edges leading to "(start)" or coming from "(end)" are elided.) (Unlike standard DFAs, which have labels on edges, this automaton has the special property that each incoming transition of a state is based on seeing the same thing, and I have labelled that on the state.)*

table could be a pivot, as the start of a subsequence with zero right-leaning entries.

- From "(start)" to "unoccupied" is allowed because the first cell in the Cleary table could be unoccupied.
- From "(start)" to "(end)" is allowed because the lemma does not technically forbid a zero-cell Cleary table.
- The four edges with filled arrows are within a subsequence, and correspond directly to the description of subsequences in the lemma.
- A "pivot" can directly follow a "pivot," because the first may have no left-leaning cells in its subsequence and the second may have no right-leaning cells in its subsequence.
- . . . (reader can verify the remaining edges similarly)

Since every behavior of the automaton is allowed and classified by the lemma, all we have to verify is that all legal sequences of cell categorizations in a Cleary table are allowed by the automaton. We prove this by showing that all transitions disallowed by the automaton are not legal in a Cleary table. All the disallowed transitions correspond to edges missing from the automaton; they are depicted in the lower part of Figure 10.3. Trivially infeasible edges leading to "(start)" or coming from "(end)" have been elided. We consider those remaining:

- A Cleary table can't start with a left-leaning cell because, based on Invariant 9.1, the preferred location for the entry in that cell must be a valid location, which cannot be to the left.
- Similarly, a Cleary table can't end with a right-leaning cell.
- A left-leaning cell cannot immediately follow an unoccupied cell, because that would violate Invariant 9.2, which disallows unoccupied cells between an entry and its preferred location.

- Similarly, a right-leaning cell cannot immediately precede an unoccupied cell.

- A left-leaning cell cannot immediately follow a right-leaning cell because this would violate how the metadata match entries with home addresses. Invariant 9.1 and interpretation of the metadata guarantee that if entry y is right of entry x, then the preferred location of y cannot be to the left of entry x.

Thus, there are no legal configuration sequences disallowed by the automaton in Figure 10.3. □

Let us review the hierarchy of interesting cell sequences in a Cleary table:

- A *run* is all the cells whose entries map to a particular home address. By the interpretation of metadata and Invariant 9.2, such entries must be adjacent to one another.

- A *Lemma 10.3 subsequence* is a maximal sequence of right-leaning followed by a pivot followed by left-leaning entries, as described in the lemma. Such a subsequence could be composed of any whole number of adjacent runs. As we will see, a subsequence is also "self-contained" for adaptation purposes, meaning the adaptation of one does not interfere with the adaptation of another.

- An *occupied sequence* is a maximal sequence of adjacent, occupied cells. Such a sequence could be composed of any whole number of Lemma 10.3 subsequences. It is also "self-contained" for random access purposes, because each random access stays within the immediate frontier of an occupied sequence.

- A *Cleary table* has zero or more occupied sequences, with unoccupied cells in between.

Lemma 10.3 helps us to realize a closer-first traversal:

Theorem 10.4. *Using the Lemma 10.3 categorizations of cell entries, the following is a realizable partial-order plan for a closer-first traversal of Cleary table cell entries:*

- *Pivot entries have no dependences.*
- *The dependence predecessor of a "right-leaning" entry is the entry to its right.*
- *The dependence predecessor of a "left-leaning" entry is the entry to its left.*

Proof By induction, I show that the dependences are sufficient to guarantee a closer-first ordering. And to show realizability, I use Lemma 10.3 to show that the plan is well-formed and acyclic.

The base cases of the induction are the pivots. By definition, they are entries at their preferred location; thus, it is trivially true that all entries between a pivot and its preferred location have been visited.

For an X-leaning entry (for $X \in \{\text{left}, \text{right}\}$), the dependence on the adjacent entry in the X direction allows us to assume that it has been visited, along with its dependencies. By the definition of X-leaning, the preferred location for the current entry must be in the X direction. Suppose, however, that the preferred location for the current entry is farther in the X direction than the preferred location for the adjacent entry in the X direction; this is not possible due to the interpretation of metadata in a Cleary table. Thus, the induction hypothesis guarantees that the dependences of the current entry are met.

Now I show that the plan is well-formed and acyclic. Lemma 10.3 guarantees that an X-leaning entry has an adjacent entry (occupied cell) in the X direction, because otherwise, one would be able to violate the subsequence structure. This means that the dependences claimed in the partial-order plan actually exist. The only way to get a cycle would be for a left-leaning entry to be to the immediate right of a right-leaning entry. Such a config-

uration would also violate the subsequence structure of Lemma 10.3. The dependences are, therefore, acyclic. □

To simplify terminology, I will call the "dependence predecessors" of Theorem 10.4 **closer-first predecessors**. Thus, a pivot has no closer-first predecessor and the predecessor of an X-leaning entry it the entry immediately in the X direction.

The simplest closer-first traversal is useful and efficient for implementation on a single processor core:

Definition 10.5. *The **standard closer-first traversal** of Cleary table entries is a "left"-to-"right" (low to high index) traversal modified to satisfy closer-first dependences lazily.*

Here is a more specific definition of the standard closer-first traversal: Lemma 10.3 subsequences are processed one at a time in left-to-right order. Within each subsequence, the pivot is visited first, followed by any right-leaning entries in right-to-left order, followed by any left-leaning entries in left-to-right order. For example, see the traversal order in Figure 10.2.

10.2.2 Algorithm

From an implementation standpoint for a single thread, it would be inefficient to compute the boundary of a subsequence before processing ("visiting") its entries. At the start of a subsequence, we search for the next pivot, "visit" it, and visit any right-leaning entries passed over in right-to-left order. Then we visit entries after the pivot in left-to-right order, until we reach one that is not left-leaning.

My highly-tuned Cleary table implementation in 3SPIN does not include a generic closer-first traversal. Instead, the traversal is integrated with each adaptation algorithm that—in concept—uses it. Here I present a generic version of the traversal, so that it can be understood separately from adaptation algorithms. However, their demands on the traversal are more complicated

than what I have already described. For example, as each entry is "visited," its home address needs to be passed along, in addition to its actual location. Also, it is helpful to have separate call-backs for visiting the pivot, left-leaning, and right-leaning entries, and to have these additional call-backs: (1) before "visiting" an entry that is the first with its home address, and (2) after "visiting" the left-most right-leaning entry in a subsequence. These are easy to hook in to the traversal algorithm.

The `closerFirstTraversal` algorithm is specialized by passing it an object that implements the `CloserFirstVisitor` interface, and thus providing implementations for the abstract methods used in the algorithm.

```
// Additional declarations for ClearyTable class, from Section 9.2:
public static interface CloserFirstVisitor {
    void preVisitHome(int home);
    void visitRightLeaning(int loc, int home);
    void visitPivot(int addr);
    void visitLeftLeaning(int loc, int home);
    void postVisitLeftMost(int loc, int home);
}

protected int findPivot(int startLoc) {
    int loc = startLoc;
    int home = startLoc;
    for (;;) { // loop until return
        while (!getMapped(home)) {
            home = home + 1;
        }
        do {
            if (loc == home) return loc;
            loc = loc + 1;
        } while (!getChange(loc));
        home = home + 1;
    }
}
```

```
protected void traverseRtoL(CloserFirstVisitor visitor,
        int pivotLoc, int leftLoc, boolean pivotChange) {
    int loc = pivotLoc;
    int home = pivotLoc;
    boolean prevChange = pivotChange;
    while (loc > leftLoc) {
        if (prevChange) {
            do {
                home = home - 1;
            } while (!getMapped(home));
            visitor.preVisitHome(home);
        }
        loc = loc - 1;
        prevChange = getChange(loc);
        visitor.visitRightLeaning(loc, home);
    }
    visitor.postVisitLeftMost(leftLoc, home);
}

protected int traverseLtoR(CloserFirstVisitor visitor,
        int pivotLoc) {
    int loc = pivotLoc + 1;
    int home = pivotLoc;
    while (loc < num_cells && isOccupied(loc)) {
        if (getChange(loc)) {
            do {
                home = home + 1;
                if (home == loc) return loc;
            } while (!getMapped(home));
            visitor.preVisitHome(home);
        }
        visitor.visitLeftLeaning(loc, home);
        loc = loc + 1;
    }
    return loc;
}
```

```
public void closerFirstTraversal(CloserFirstVisitor visitor) {
    int topLoc = 0;
    while (topLoc < num_cells) {
        if (!isOccupied(topLoc)) {
            topLoc = topLoc + 1;
        } else {
            int pivLoc = findPivot(topLoc);
            boolean pivChange = getChange(pivLoc);
            visitor.preVisitHome(pivLoc);
            visitor.visitPivot(pivLoc);
            traverseRtoL(visitor, pivLoc, topLoc, pivChange);
            topLoc = traverseLtoR(visitor, pivLoc);
        }
    }
}
// End of additional declarations for ClearyTable class
```

The following theorem captures important bounds on the performance of this algorithm.

Theorem 10.6. *Assuming the provided* `CloserFirstVisitor` *object uses* $O(1)$ *space and each method call to the provided* `CloserFirstVisitor` *uses* $O(1)$ *time and* $O(1)$ *additional space, the* `closerFirstTraversal` *algorithm requires* $O(m)$ *time and* $O(1)$ *additional space, where* m *is the total number of bits of memory used by the Cleary table.*

Proof Idea The time bound is easy to see, because we iterate over each unit of memory only a constant number of times. In a sense, we iterate over right-leaning entries twice (once in `findPivot` and once in `traverseRtoL`), pivot entries twice (once in `findPivot` and once in `visitPivot`), and left-leaning entries once (in `traverseRtoL`). In more detail, we actually iterate of the MAPPED bits and the non-MAPPED parts separately in the `traverseRtoL` and `traverseRtoL` methods, but this only affects constant factors.

Note that a time bound of $O(c)$ would not hold in theory unless we assume (as my implementations assume) that each cell is a constant number of machine words. (I am assuming the RAM model for complexity analysis [17, Section 2.2].) Note also that $O(m)$ is equivalent to O-of the number

of machine words of memory used.

The space bound is also easy to see, because the only additional state is a constant number of indices into the table and boolean flags. Note that the algorithm does not use recursion. □

10.3 1-to-2 adaptation

I now use the closer-first traversal to implement our first Cleary table adaptation procedure: doubling the number of cells, cutting their size in half, in place. I call this **1-to-2 adaptation**. Some bits at the end of each represented value will be thrown away in the adaptation, so this procedure is really only applicable when using the Cleary table as an inexact set representation—or going from exact to inexact. Figure 10.1 shows an example input and output of this adaptation operation.

The output occupies exactly the same memory as the input, so during the adaptation, it will be accessed both as a structure with the input configuration and as a structure with the output configuration. In fact, this algorithm depends on the input configuration and output configuration overlapping in the right way: bits for input cell i, including its two metadata bits, should correspond to the bits for output cells $2i$ and $2i+1$, including their metadata bits. In fact, MAPPED bits in the input configuration must also be MAPPED bits in the output configuration. Figure 10.4 shows a layout of the bits to satisfy these constraints. Observe that with doubling the number of cells, the quantity of metadata will double, but only gradually, as each cell is adapted.

Suppose instead we used the layout in which the metadata bits live in their own bit vectors. It is not clear how best to make room for the new metadata in order to split cells. One might need a pass over the entries to compact them slightly to make room for all the new metadata before we start splitting. The elegance of my approach, with the metadata beside the entries in the cells, is that the space for new metadata becomes available

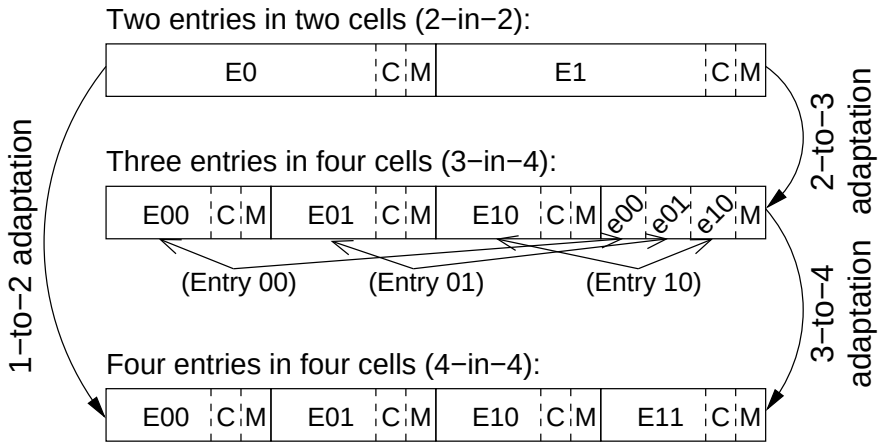

Figure 10.4: *Example of bit layouts for Cleary tables that are compatible with my adaptation algorithms. Each "C" is a* CHANGE *bit, each "M" a* MAPPED *bit, and each "Exx" is an entry. In the 3-in-4 configuration, each entry is actually the "Exx" part followed by the "exx" part. Note that "four entries in four cells" (4-in-4) is treated as twice as many cases of "two entries in two cells" (2-in-2) for further adaptation.*

right as it is needed.

The closer-first traversal takes care of breaking the adaptation procedure down into the adaptation of each entry; thus, the `Adapt1to2Visitor` listed below implements `CloserFirstVisitor`. However, we need to pass information about the adaptation of the closer-first predecessor of each entry to adaptation of the successor. In particular, three pieces of information are needed: (1) the new entry location of the predecessor is needed to determine where space is available for the successor; (2) the new home address of the predecessor is needed to determine where an old run is being split into two new runs; and (3) the new entry value of the predecessor is needed to determine when two entries are no longer distinct and must be coalesced. The new entry address of the predecessor is also used to set the CHANGE bit on the predecessor when visiting right-to-left and splitting an old run in two. These pieces of information could be obtained by re-examining the partially-adapted structure, but that would raise the overall running time and code complexity.

At a given time, there might be two active copies of the information that needs to be passed from predecessor to successor, because we have to remember information from the pivot for passing to a left-leaning successor while visiting the right-leaning entries. This justifies the six fields of `Adapt1to2Visitor` that are auxiliary state for the adaptation.

The story of the MAPPED bits is complicated, because their traversal sometimes lags behind the traversal of the rest of the cells. In particular, when we are visiting "leaning" entries, we cannot zero out the MAPPED bit in the same cell when we zero out the CHANGE and entry bits, because the closer-first traversal will only read that MAPPED bit once it reaches entries with that as their home address. Only then can we clear the MAPPED bit. This happens in the `preVisitHome`. Even though each old MAPPED bit will also be a MAPPED bit in the new configuration, the same bit might not be set. If, for example, the old MAPPED bit corresponding to address 100101 is set, that same bit will only be set if there are values that start with 1001010. If the old run only consists of values starting with 1001011, the bit will not be set. If there are both, `visitRightLeaning` or `visitLeftLeaning` takes care of splitting up the old run, where `newHome` is compared with the previous "new home."

The remaining details of the algorithm should be understandable from comments and trying examples, though a valuable exercise is outlining a proof that the output has the same number of set MAPPED and CHANGE bits (part of Invariant 9.1). **Proof Idea** All the old metadata bits are cleared, the MAPPED bits in `preVisitHome` and the CHANGE bits in `visit{RightLeaning, Pivot,LeftLeaning}`. The setting of a CHANGE bit in `postVisitLeftMost` and a MAPPED bit in `visitPivot` account for one another because they each happen once per Lemma 10.3 subsequence. Otherwise, when we set a MAPPED bit in `visitRightLeaning` or `visitLeftLeaning`, we immediately also set a CHANGE bit. □

```
// Additional declarations for ClearyTable class, from Section 9.2:
static class Adapt1to2Visitor implements CloserFirstVisitor {
    ClearyTable t1; // source
    ClearyTable t2; // destination, with same "BitVect table"

    // these six fields track state between calls
    int rightNewLoc;
    int rightNewHome;
    long rightNewEntry;
    int leftNewLoc;
    int leftNewHome;
    long leftNewEntry;

    /** Constructor, for adapting from t1 to t2,
     *  which should share a memory table. */
    Adapt1to2Visitor(ClearyTable t1_0, ClearyTable t2_0) {
        t1 = t1_0;
        t2 = t2_0;
        assert t1.table == t2.table;
    }

    public void preVisitHome(int home) {
        t1.setMapped(home, false);
    }

    public void postVisitLeftMost(int loc, int home) {
        t2.setChange(rightNewLoc, true);
    }

    protected int getNewHome(int oldHome, long oldEntry) {
        // from oldHome, insert highest bit of oldEntry as lowest
        // order bit to get new home
        int highBit = (int)(oldEntry >>> (t1.entry_bits - 1));
        return (oldHome * 2) + highBit;
    }

    protected long getNewEntry(long oldEntry) {
        // shift away low bits to be thrown away and mask away
        // highest bit (which becomes an address bit)
        int throwAwayCount = t1.entry_bits - t2.entry_bits - 1;
        long newMask = (1L << t2.entry_bits) - 1;
        return (oldEntry >>> throwAwayCount) & newMask;
    }
```

```
public void visitPivot(int addr) {
    // save needed information
    int newHome = getNewHome(addr, t1.getEntry(addr));
    long newEntry = getNewEntry(t1.getEntry(addr));

    // clear away the old
    t1.setEntry(addr, 0);
    t1.setChange(addr, false);

    // compute new location (trivial for pivot)
    int newLoc = newHome;

    // write new info
    t2.setEntry(newLoc, newEntry);
    t2.setMapped(newHome, true);

    rightNewLoc = leftNewLoc = newLoc;
    rightNewHome = leftNewHome = newHome;
    rightNewEntry = leftNewEntry = newEntry;
}
```

```
public void visitRightLeaning(int loc, int home) {
    // save needed information
    int newHome = getNewHome(home, t1.getEntry(loc));
    long newEntry = getNewEntry(t1.getEntry(loc));

    // clear away the old
    t1.clearEntryAndChange(loc);

    // check for coalesce with predecessor
    if (newHome==rightNewHome && newEntry==rightNewEntry) {
        return;
    }

    // compute new location
    int newLoc = Math.min(rightNewLoc - 1, newHome);

    // write new info
    t2.setEntry(newLoc, newEntry);
    if (newHome != rightNewHome) {
        t2.setMapped(newHome, true);
        t2.setChange(rightNewLoc, true);
    }

    rightNewLoc = newLoc;
    rightNewHome = newHome;
    rightNewEntry = newEntry;
}
```

```
    public void visitLeftLeaning(int loc, int home) {
        // save needed information
        int newHome = getNewHome(home, t1.getEntry(loc));
        long newEntry = getNewEntry(t1.getEntry(loc));

        // clear away the old
        t1.clearEntryAndChange(loc);

        // check for coalesce with predecessor
        if (newHome==leftNewHome && newEntry==leftNewEntry) {
            return;
        }

        // compute new location
        int newLoc = Math.max(leftNewLoc + 1, newHome);

        // write new info
        t2.setEntry(newLoc, newEntry);
        if (newHome != leftNewHome) {
            t2.setMapped(newHome, true);
            t2.setChange(newLoc, true);
        }

        leftNewLoc = newLoc;
        leftNewHome = newHome;
        leftNewEntry = newEntry;
    }
} // end of class Adapt1to2Visitor

public ClearyTable adapt1to2() {
    assert (cell_bits & 1) == 0; // cell size must be even
    ClearyTable t2 = // new ClearyTable uses same memory table
        new ClearyTable(addr_bits+1, cell_bits / 2 - 2, table);
    // construct visitor and execute traversal
    closerFirstTraversal(new Adapt1to2Visitor(this, t2));
    return t2;
}
// End of additional declarations for ClearyTable class
```

The following theorem captures important bounds on the performance of this algorithm.

Corollary 10.7. *Assuming each cell in the Cleary table is a constant number of memory words, the* `adapt1to2` *algorithm requires* $O(m)$ *time and* $O(1)$ *addi-*

tional space, where m is the total number of bits of memory used by the Cleary table.

Proof Idea The time bound is an obvious corollary of Theorem 10.6, because the methods in `Adapt1to2Visitor` do not have any loops or recursion.

For the space bound, we need an assumption that was not needed for Theorem 10.6: that the cell size is $O(1)$ space. Any given cell size can be considered $O(1)$ space, but for exact storage, the size usually depends on the model complexity, and should not be considered $O(1)$. For a given accuracy, as an expected proportion of states hash-omitted, the cell size is $O(1)$.

With that assumption, the space bound is a corollary of Theorem 10.6, because the `Adapt1to2Visitor` object and its methods use only a constant number of table indices and cell-sized data variables. The `ClearyTable` `t1` is just a reference to the input table. The `ClearyTable` `t2` wraps the same `BitVector` with a different configuration, which uses a constant number of index-size fields. □

10.4 2-to-3 and 3-to-4 adaptation

Cutting the size of each cell in half has a sudden, drastic impact on the false positive rate of inexact storage with a Cleary table. The expected hash omissions also jump nearly as suddenly. Going from 32 bits per cell to 16, for example, increases the false positive rate by a factor of $2^{15} = 32\,768$. Thus, we only have to visit 1% more states after that adaptation for the expected hash omissions to increase by a factor well over 300.

To ease such drastic rises in inaccuracy, I have designed an intermediate step between doubling the number of cells. Updates to the ADD and adaptation algorithms to support this intermediate step are not scientifically interesting enough to warrant detailed listing. I describe the changes needed in enough detail to convince the casual reader that they have the same essence and computational complexity as those already detailed, or to guide an in-

terested reader in creating his/her own implementation.

10.4.1 3-in-4 design

The intermediate structure is tricky because we either have to give up on having one home address for each preferred location (Section 9.5.3) or we have to move only part of one bit of each entry to be part of the address. I deal with values that are not a whole number of bits in order to accommodate non-power-of-two amounts of memory (Section 9.5.2), but I do not like dealing with it otherwise. Thus, I prefer the former approach.

The middle of Figure 10.4 shows my design for this intermediate Cleary table, which I call a **3-in-4 Cleary table**. Essentially, we split adjacent pairs of cells into a group of four "cells" that has four MAPPED bits but only stores up to three entries. The last "cell" in each group stores parts of each of the previous three entries. I will explain some of the motivators for this design.

First of all, by either doubling the number of MAPPED bits or keeping it the same, we either add one address bit or keep the same number. This makes computing updated addresses for adapted values no more complicated than for 1-to-2 adaptation.

Second, we should double the number of MAPPED bits for the 2-to-3 adaptation rather than the 3-to-4 adaptation, because the best home-address-to-cell ratios are between 1.0 and 2.0 (Section 9.5.3). Four home addresses for every three entries uses less memory than two home addresses for every three entries would.

Third, by making sure the output MAPPED bits encompass at least the input MAPPED bits, dealing with MAPPED bits is no more complicated than for the 1-to-2 adaptation algorithm. (Recall that the 1-to-2 required the same constraint.)

Unfortunately, this combination means that we cannot store each entry contiguously in our "three entries in four cells" configuration. There is sim-

ply not enough space between each MAPPED bit. Thus, every fourth cell contains parts of the entries in the previous three. (See Figure 10.4.)

10.4.2 Algorithm changes for 3-in-4 ADD

Updating the Cleary table ADD algorithm to work with this 3-in-4 scheme presents a challenge in terms of addressing entries. Consider adding the first element to the table. From its home address, h, the position of the MAPPED bit is easy to compute. We also need a preferred location, which needs to be roughly the $i = 3h/4$th entry, but computing the bit position of the ith entry is tricky because we essentially need to undo what we just did, because we need to relate every three entries with the group of four "cells" that contain it.

I find the following a simpler approach to the entry addressing problem: entry locations are the same as "cell" and home addresses, but some addresses are invalid as entry locations. In particular, every fourth address, ending in bits "11," is invalid as an entry location. To make this work, we just have to hook in to (1) computing the preferred location from a home address, (2) each time we compute the "next" entry, and (3) each time we compute the "previous" entry. For (1), home addresses that are also valid entry locations map to themselves; those ending in "11" map to the respective one ending in "10." For (2) and (3), we just have to skip the "11" cases. We can actually do each of these with simple ALU operations, as these macro definitions from my 3SPIN implementation show:

```
#define 3IN4_ADDR_PREF_LOC(addr) \
      ((addr) - (((addr) & 1) & (((addr) & 2) >> 1)))
#define NEXT_3IN4_LOC(loc) \
      ((loc) + 1 + (((loc) >> 1) & 1))
#define PREV_3IN4_LOC(loc) \
      ((loc) - 1 - ((((loc) - 1) >> 1) & 1))
```

10.4.3 Algorithm changes for 2-to-3 and 3-to-4 adaptation

Adapting from a standard Cleary table to a 3-in-4 Cleary table (2-to-3 adaptation) has the same complications as its ADD algorithm and a little more. If we ignore for the moment the extra entry pieces in every fourth cell and suppose we just have to zero out those bits, the adaptation is very much like the 1-to-2. The memory for each cell in the input configuration encompasses the memory for at least one usable cell in the output configuration, and a home address or preferred location in that input cell maps to an output home address or preferred location in the same piece of memory. Thus, ignoring the extra entry pieces, the only thing different from the 1-to-2 adaptation is the preferred location computation and location increment and decrement.

The extra pieces of entry in every fourth cell cannot be written directly by following a closer-first order, but we only have to remember up to two uncommitted pseudo-cells of partial entries during the traversal. Processing of the last subsequence might have ended with the first of an input pair (middle of an output four-"cell" group), with a partial entry that cannot yet be written, and the pivot of the current subsequence might also be the first of an input pair. In this case, we have to delay writing those "fourth cell" partial entries until the corresponding input cell has been processed ("visited"). Since this extra storage is only a constant amount, the computational complexity of the adaptation algorithm is not affected.

Adapting from the 3-in-4 Cleary table to a standard Cleary table (3-to-4 adaptation) would seem simple, but it is actually somewhat tricky. It seems simple because if you zero out the "extra" entry information, contained in every fourth "cell", we have an *almost* valid Cleary table in the output configuration [1] What makes it tricky is that a quarter of the input entries will have a new preferred location in the output configuration; specifically, those whose

[1] It would not satisfy Invariant 9.2, and could also have repeated entries that should be coalesced.

preferred location was not equal to their home address (those whose home address ends in binary "11") have a new preferred location, equal to their home address. This means we could have a pivot entry (see Lemma 10.3) whose home address ends in "11" but is at its preferred location ending in "10" and needs to be moved during adaptation. Adaptation is easiest if the new memory area for each pivot is contained in the memory area it currently occupies, as in 1-to-2 adaptation. 3-to-4 adaptation has the next best thing, however: upon processing each pivot, the new memory area for it is allowed to be overwritten. This is because we can throw away the "extra" entry information contained in every fourth cell as soon as we know we won't be confused by a location appearing unoccupied, in case the only non-zero data was in the extra information. That is not a problem visiting right-to-left from the pivot because we know all those cells are occupied; it is not a problem visiting left-to-right because we would only need to overwrite a cell with extra entries after visiting the previous three.

As in every previous adaptation routine, we are throwing away entry information and might need to coalesce entries that are no longer distinct, not just for compactness, but also to ensure my hack to eliminate OCCUPIED bits does not lead to violation of Invariant 9.2, by throwing away the only non-zero information contained in an occupied cell (and not taking corrective action, coalescing).

10.5 Post-adaptation access times

Based on how my adaptation algorithm arranges runs and empty cells in the resulting Cleary table, I formulated a hypothesis that access times for the resulting table should be better than if we had added all the elements to a table of that configuration to begin with. Tests indicate my hypothesis is correct, probably because adaptation lays out runs in a way that seems to leave empty cells interspersed more regularly than we would otherwise

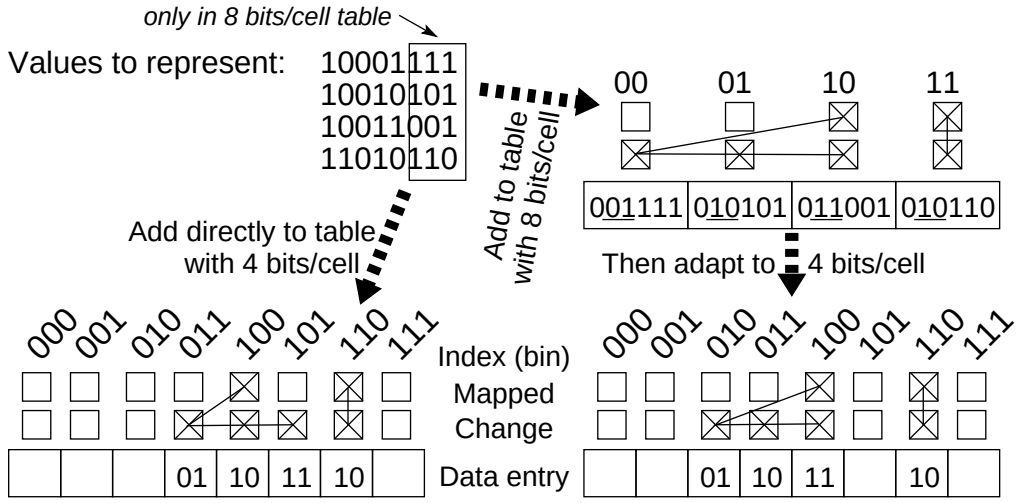

Figure 10.5: *Comparison of Cleary table layout after adaptation and layout with no adaptation. This shows how using different algorithms I have presented can result in different layouts for the runs in a Cleary table. In particular, adapting from a Cleary table with fewer entry locations seems to result in shorter spans of occupied cells.*

expect.

Recall that even if we satisfy all three structural invariants of a Cleary table, there are usually many ways to lay out the runs in the cells and still satisfy the invariants. Consider the two Cleary tables at the bottom of Figure 10.5. They represent the same set of values in different ways, despite using the same Cleary table configuration. The table on the right seems to be better for access times, because the average distance from each occupied cell to the nearest unoccupied cell is only 1.25 rather than 1.5. As the figure indicates, the right table was the result of adaptation while the left table was the result of simply adding the same set of values. Examples like this were the basis of my hypothesis that the post-adaptation structure has better-than-usual access times.

To test this hypothesis, I instrumented my Cleary table implementation in 3SPIN to time how long it takes to get from one specified number of visited states to another. For the first data point, I configured it to start with 64 bits

per cell, adapt to 32-bit 3-in-4 at 90% occupancy, and then adapt to 32-bit standard at 90% occupancy. Right after that, at 160 million states (67.8% occupancy), it recorded the starting time, and then at 200 million states (84.8% occupancy), it recorded the stopping time. The elapsed time was 10.131 seconds. For the second data point, I configured it to start with the 32-bit-per-cell standard Cleary table, and sample times at the same numbers of states. The elapsed time was 10.445 seconds, about 3% slower. Repeated trials gave virtually the same result. The model I used was specifically designed to control for differences due to live caches for recently added states; specifically, no states reached in the search match a previously added state. I got similar results with a realistic model.

I performed a similar test comparing the result right after 1-to-2 adaptation to no adaptation, and the difference was a much smaller, only 0.5%. This makes sense considering the occupancy must be below 50% after 1-to-2 adaptation.

Do not misinterpret this result; the overall running time was significantly greater when the model checker had to adapt down to 32 bits rather than starting at 32 bits. I examine that difference in overall running time in Chapter 11. Here I am evaluating whether and how access times can vary among structures with the same current configuration and representing the same set of values, which might be a surprising result.

To help verify my hypothesized explanation, I added code to compute the average length of each span of occupied cells, just before recording the starting time at 67.8% occupancy. If it had just come from 3-to-4 adaptation, the average length was 4.020 cells. With no adaptation, the average length was 4.278 cells.

There are two things to take away from this examination of access times after adaptation. First, the access times one can expect from a post-adaptation Cleary table are slightly better than those from the corresponding table built without adaptation. Second, it seems possible to alter a Cleary table

constructed using only the standard algorithms (Chapter 9, no adaptation) to improve its access times. This second point might be useful in applications that do not follow the "visited set" usage paradigm, so it is not worth exploring further in this dissertation.

10.6 Adaptation to Bloom filter

Adapting a Cleary table to a Bloom filter is reasonable and possible when enough states are reached for the memory per visited state (m/v) to be small. As described before, the Cleary table is not great when the memory per visited state is quite small. For instance, there is no known reason for using a 4-bit-per-cell Cleary table instead of a bit table, since the bit table can represent the same values in the same space, with $O(1)$ access times and no susceptibility to overflow (see Section 9.3). Using a bit table for inexact storage, based on a reduction from Chapter 8, is congruent to a $k = 1$ Bloom filter, described in Chapter 6.

10.6.1 $k = 1$

Adapting to a $k = 1$ Bloom filter (bit table) is remarkably easy when the cell size of a standard Cleary table is a power of two. The closer-first traversal makes this easy, because the bit to set in the new table for an entry is in the cell at the entry's home address. Thus, only cells that had their MAPPED bit set will have bits set in the resulting Bloom filter. If each cell is 2^i bits, then i bits of entry become address bits for the bit table/Bloom filter. See Figure 10.6 for an example, in which $i = 3$.

One way to think about this adaptation is that we are adapting to a Cleary table with zero-bit entries, and because all entries in a "run" are indistinguishable, there is no need for CHANGE bits. The structure is only MAPPED bits. This is equivalent to a $k = 1$ Bloom filter.

In more detail, the algorithm erases relevant MAPPED bits in the imple-

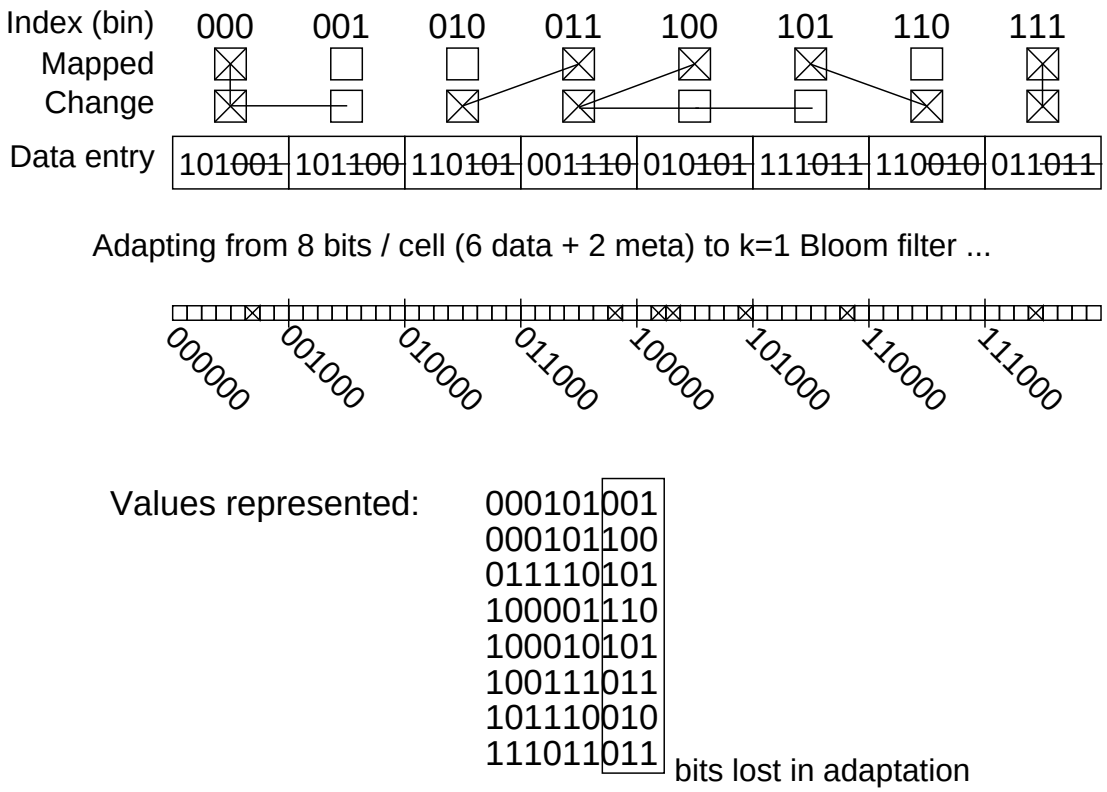

Figure 10.6: *A simple "before and after" example of adapting a Cleary table to a $k = 1$ Bloom filter. Notice that the total memory bits, 64, is the same before and after adaptation. The input table is the same as in Figure 10.1, and the outputs in the two figures represent the same set of values.*

mentation of `preVisitHome`. In `visit{RightLeaning,Pivot,LeftLeaning}`, it erases CHANGE and entry bits in the current cell and then writes the Bloom filter bit to the proper location, which is safe to write to. (Code for this algorithm is not shown.)

10.6.2 Hash-reusing $k = 2$

Adapting to a special $k = 2$ Bloom filter is also possible, and is usually more attractive than adapting to a $k = 1$ Bloom filter. The $k = 1$ Bloom filter is rarely the ideal Bloom filter configuration for a model checker even when the state space is intractably large, because the search usually starves for new states before reaching the v/m ratio for which $k = 1$ is best. The design

I use here is a natural generalization of what I described in [26].

Unless we are adapting from a Cleary table with very large cells, we do not have enough hash information from the original elements to construct a standard $k = 2$ Bloom filter. Most typically, we would be adapting from an 8-bit-per-cell Cleary table, which represents hash values that are three bits larger than an address in the resulting Bloom filter (six entry bits per cell, minus three new address bits per cell, because addressing bits instead of bytes results in $2^3 = 8$ times as many addresses). Another way to see this is in the three bits "thrown away" when converting to a $k = 1$ Bloom filter in Figure 10.6. This is the only additional hash information available to the newly-formed Bloom filter. We cannot recover any more information about the input elements unless we dramatically change the conditions of adaptation, such as recreating the structure from disk, which would be slow.

There is another problem with a standard $k = 2$ Bloom filter. Bloom filters are most accurate when there is no locality in the indices for an element. Ideally, the location of the second index should be random with respect to the location of the first index. Independent positioning of the second index is a problem for adaptation, however, because it breaks the locality needed for the adaptation to be confined to a closer-first traversal using $O(1)$ auxiliary space.

These two problems, the shortage of hash information and the need for locality in indexes, are actually serendipitous. My hash-reusing Bloom filter (Section 6.4.4) is the best known way to deal with limited hash information, and it makes indices local to each other. This was one of the strange conclusions of how best to deal with limited hash information in a Bloom filter. Thus, we use the hash-reusing Bloom filter for adapting from a Cleary table, when we want $k = 2$. All the accuracy analysis of that design (Section 6.4.4) is applicable here.

In more detail, when adapting from a 2^j-bit-per-cell Cleary table to our special $k = 2$ Bloom filter, the first index is computed as for the $k = 1$ adap-

tation, and the second is computed by replacing the last j bits of that index with the next j bits of available hash information and adding 2^j. By adding 2^j (or, equivalently, adding 1 to the part not replaced), we are essentially placing the second index randomly within the next cell in the input table. For an 8-bit-per-cell Cleary table, for example, the second index is in the next byte. Note that at the end of the table (highest index, "right-most"), the second index should either wrap around to the first cell or occupy an extra cell beyond the end of the table. I use the latter in 3SPIN, by keeping one extra allocated cell (byte) in reserve for use when adapting to a $k = 2$ Bloom filter.

This design keeps the adaptation procedure from getting too complicated. Its basic difference from the $k = 1$ adaptation procedure is that the $k = 2$ adaptation keeps track of two input cells worth of uncommitted Bloom filter data, which is Bloom filter data that will by copied into the memory occupied by unvisited cells. (Recall that the 2-to-3 adaptation required tracking uncommitted data.) One contains the uncommitted data from processing the last Lemma 10.3 subsequence, and the other contains the uncommitted data from processing the pivot. Note that we only need one of those after all right-leaning entries in the current subsequence have been visited, because at that point we can commit the data left over from the previous subsequence. In my implementation in 3SPIN, I found it more convenient to have three variables for uncommitted Bloom filter data. In either case, the tracked data is $O(1)$ size in the RAM model [17].

10.7 Summary

In this chapter I have shown how the representation of the Cleary table allows for fast, in-place adaptation to accommodate more elements. I have proven properties of Cleary tables that enable a "closer-first" traversal to perform this adaptation using $O(1)$ auxiliary space.

I validate the performance and practicality of these algorithms in Chapter 11, in which I put them together into an adaptive storage solution with unmatched dynamic flexibility.

CHAPTER 11

Adaptive storage scheme

Contribution: Here I describe a scheme for state storage with an unprecedented combination of dynamic flexibility, speed, and accuracy. In particular, it is the first approach that does not assume any prior information about the state space size in order to be near the best possible accuracy and speed in all practical cases. The scheme is based on Cleary tables (Chapter 9) and Bloom filters (Chapter 6) and my algorithms for quickly adapting from configuration to configuration.

We introduced a basic version of the scheme in a SPIN Workshop paper [26], which focused on practical aspects. Here I refer to that basic version as the "fast" variant, because it is among the fastest known approaches to over-approximate state storage. Here I also describe the "accurate" variant, which uses the 3-in-4 Cleary table to substantially improve worst-case competitive accuracy, at the cost of higher code complexity and a little time.

The theoretical design of the scheme allows it to be near optimal accuracy in all practical cases. I show that the "accurate" variant is at least as accurate as is the information-theoretic optimal for half as much memory, or 40% as much memory for the "fast" variant. My state storage algorithm could be considered an online algorithm, and my analysis is related to competitive analysis of online algorithms [6, 1]. The detailed analysis is in Section 11.1.

In practice, to be observably near optimal accuracy and speed, any implementation only needs to use a finite subset of the configurations allowed by the design, based on the word size. This ensures fast access and adaptation times by keeping cell sizes and hashing requirements to normal levels: a small, constant number of machine words. This is discussed in detail in Section 11.2.

A great advantage of my scheme is that it enables the user of an explicit-state model checker to forget about configuring state storage to get the best speed and accuracy. Using my scheme—at least by default—ensures that speed and accuracy are close to the best possible [1].

11.1 Near optimal accuracy by design

Theorem 11.1. *Suppose we use m bits of memory to exactly or over-approximately represent visited states in exploration of a state graph, in which each state is only known to be among u representable possibilities. Suppose also that m is not close to u ($m \leq u/8$) and that m is not trivial in size ($m \geq 2^{16}$; at least 8KB). For any number of unique states reached by the search, v, up to the number of bits of memory available (assume $v \leq m$), the expected hash omissions from a search using the "accurate" storage scheme from Figure 11.1 is no greater than the expected hash omissions from a search using an information-theoretic optimal representation in half as much memory ($m/2$). The analogue is true for the "fast" variant of the scheme and 40% as much memory ($m/2.5$).*

Throughout this section, I outline a proof of this theorem, with the assistance of numeric computation. But first, I motivate that this theorem is useful in practice, and that its key assumptions are reasonable. I also describe in more detail the storage scheme outlined in Figure 11.1.

[1] Significantly better performance is likely possible in one or both dimensions by using heuristic methods, state caching, or out-of-core storage. As discussed in Chapter 1, these methods are outside the scope of this dissertation.

Entry bits	How
exact	2^i bits per cell, possibly 3-in-4, enough for exact storage
...	... (follow pattern from below)
62	= 64 bits per cell - 2 metadata bits per cell
40	= 32 bits per cell - 2 metadata + $\lfloor$(32 - 1)/3$\rfloor$ extra (3-in-4)
30	= 32 bits per cell - 2 metadata bits per cell
19	= 16 bits per cell - 2 metadata + $\lfloor$(16 - 1)/3$\rfloor$ extra (3-in-4)
14	= 16 bits per cell - 2 metadata bits per cell
8	= 8 bits per cell - 2 metadata + $\lfloor$(8 - 1)/3$\rfloor$ extra (3-in-4)
6	= 8 bits per cell - 2 metadata bits per cell
n/a	$k = 2$ Bloom filter (Hash-reusing)

Figure 11.1: *Life cycle of Cleary table and Bloom filter configurations in my adaptive storage design. The structure starts with the configuration on the top line, where i is the smallest integer to enable exact storage. After reaching 85% occupancy, we adapt the Cleary table to the next smaller entry size that matches the pattern established by the last several Cleary table configurations, which are listed explicitly. These Cleary tables should implement inexact storage using even partitioning (Section 8.3) based on a randomized state descriptor. The process continues by adapting as needed to the next configuration upon attempting to add to a table that has reached 85% occupancy. If needed, we finally adapt to a $k = 2$ hash-reusing Bloom filter. This is the final configuration, as it is listed last. The configurations surrounded with boxes are only present in the "accurate" variant of the storage scheme. Note that in some rare cases, the starting exact configuration could use 8 bits per cell, $i = 3$, the minimum allowed in our scheme; the $k = 2$ Bloom filter is inherently inexact. Or in the rare case that $m \geq 0.9u$, we recommend a bit table in place of this scheme entirely.*

11.1.1 Utility of the theorem

Theorem 11.1 is useful primarily because it puts a modest price on taking the guesswork out of configuring non-heuristic data structures for storing visited states. That modest price is doubling the amount of memory available for storing visited states. If you pay that price and use our storage scheme, your searches will always be at least as accurate as they would be if you had continued trying to pick the most accurate configuration in each case. And if you are interested in both fast falsification and high-assurance verification, trying to pick the best configuration in advance is a lot like playing the lottery, because you will eventually lose big in either fast falsification or high assurance verification.

The scheme is not practical and the theorem is not useful if the hashing requirements and cell sizes bog down the performance of any implementation, however. I address these issues in Section 11.2.

Here, I argue that the key assumptions for the inaccuracy bounds of Theorem 11.1 are reasonable. The first assumption is that "each state is only known to be among u representable possibilities," which implies my characterization of an "information-theoretic optimal representation" does not take advantage of any regularity or predictability in state descriptors. As explained in Chapter 1, heuristic storage is outside the scope of this dissertation. Such techniques are often effective at reducing memory requirements for exact storage, but usually at a price in execution speed. Perhaps more importantly, having a concept of optimality is muddled if we cannot assume each state represents a certain amount of information. Also note that my design does not preclude bit-packing raw states before storing them; in fact, the more we can constrain the representable universe with static analysis of the model, the more reasonable this assumption of Theorem 11.1 is.

The second assumption is that the number of memory bits is not close to the universe size. Specifically, the theorem is restricted to $m \leq u/8$, or

equivalently, $\lg u - \lg m \geq 3$. That means the full state descriptors cannot be smaller than the values represented by a standard 8-bit-per-cell Cleary table, but this should not be a practical limitation, because $\lg u$ is typically many times larger than $\lg m$. Observe that the adaptive storage scheme is not needed if $m \geq u$, because in that statically-determinable case we can use a bit table (Section 5.2) to represent any subset of U exactly, with ideal access times. The case of $0 < \lg u - \lg m < 3$ should be rare, but is difficult to work with. For a Bloom filter, this would mean each value is less than three bits larger than an index. Our scheme is not terrible in that case, because hash-reusing is probably the "least bad" option for inexact storage, but the competitive inaccuracy of the storage scheme can reach about 2.5. Therefore, this assumption does not change the essence of the theorem.

The third assumption is that the number of memory bits is not too small. I specifically assume $m \geq 2^{16}$ ($\geq$ 8KB memory), but this is the arbitrary "large enough" point at which I compute bounds numerically and generalize to everything larger analytically. I expect the bounds to hold for much smaller m as well, but I do not care about those cases. Since Theorem 11.1 is based on an accuracy metric that is tied to the visited set usage paradigm, and the point of the scheme is to make best use of constrained memory, I expect users to be dedicating more than 8KB of memory to a problem that calls for our technique.

Finally, I assume the number of states seen during the search does not exceed the number of memory bits available ($v \leq m$). For this assumption to be violated, a search would have to visit enough states for a $k = 1$ Bloom filter to have been the best known approach, which means $v/m \geq 0.88$ ($\approx 1.13459^{-1}$; see Figure 6.3). Recall that Holzmann's extensive testing has shown that even when memory is short, $k = 3$ is usually more accurate than $k = 2$, which is almost always more accurate than $k = 1$ (see Section 6.3.3). The reason is that as the false positive rate of the structure gets high, the search starves for new states in the stack or queue and much of the state

graph is transitively omitted. The worst that happens in the rare case that this assumption is not met is that the storage scheme is further from optimal than the bounds in Theorem 11.1, but this is not exactly a fair critique, because the best known practical approach for this case, the $k = 1$ Bloom filter, also diverges from the information-theoretic optimal as v surpasses m. If we compare the hash-reusing $k = 2$ Bloom filter of my adaptive storage scheme to the best known approach in these cases, $k = 1$, the accuracy is well within a memory factor of two.

11.1.2 Design

Figure 11.1 shows the data structure configurations used in the full design of my storage scheme. We shall assume it starts with a Cleary table capable of representing states exactly and adapts from there to less accurate Cleary tables, as needed, until it adapts to the hash-reusing $k = 2$ Bloom filter. This is probably not the best way to implement the scheme, mostly because of high hashing requirements, but it should be a fair way of analyzing the scheme without tying ourselves to a particular problem scale solvable by today's machines. Discussion of practical implementation of this scheme is in Section 11.2.

Let us consider an example. Suppose $\lg u = 800$; thus, it takes 100 bytes to describe a state. Suppose we have $m = 2^{33}$ bits, or 1 GB, of memory for state storage. To find the most compact starting configuration that represents states exactly but is also compatible with our scheme, we can start with a cell size that is the next power of two higher, and work our way down. Consider, therefore, cells of $2^{10} = 1024$ bits. The represented values would be $(1024-2)+(33-10) = 1045$ bits, which is clearly sufficient for exact storage. The next configuration would be a 3-in-4 Cleary table using $2^9 = 512$ bits per cell. The represented values would be $(512-2)+(33-9)+\lfloor(512-1)/3\rfloor = 704$. In general, this is not sufficient for exact storage. Thus the starting configu-

ration uses a standard Cleary table with 1024 bits per cell.

The 512-bit-per-cell 3-in-4 Cleary table would be the next configuration in the example, in the case of the first structure reaching 85% occupancy. Only 704 bits of the original 800 (padded to 1045) would remain after the first adaptation. Ignoring for now the practical implications, we assume the state vector has been randomized by an ideal randomization function (one-to-one hash function), such that any prefix of the bits has as much information as possible about the original state. If in this example there are 750 bits of information (entropy) in the original 800-bit state, we assume the 704-bit prefix represents 704 bits of that. If the original only contained 100, we assume the 704-bit prefix contains all 100, as an ideal randomization would guarantee. (See Section 11.2 for practical considerations.)

If we reach the 8-bit-per-cell standard Cleary table and reach 85% occupancy on it, the $(8-2)+(33-3)=36$ bit values represented are used to convert to a $k=2$ hash-reusing Bloom filter. This final adaptation is unique because we are converting to an inherently inexact structure. Information is lost in the conversion, but not in an explicit way; we need all 36 bits from the values in the previous structure to build the Bloom filter, but we cannot reverse-engineer the precise set of 36-bit values that went into creating the Bloom filter. This indicates information loss, and why the Bloom filter is less accurate than the 8-bit-per-cell Cleary table. The Bloom filter's lower accuracy, however, comes with the ability to accommodate more elements. In fact, it cannot overflow in the same sense that a Cleary table can, because any Bloom filter can represent the entire universe of elements, by having all its bits set to "1".

11.1.3 Exact storage case

I first demonstrate the portion of Theorem 11.1 relating to exact storage. This requires showing that for any allowed combination of m, u, and v, my

adaptive storage scheme uses exact storage at least when the competitive inaccuracy, $m/\breve{m}_{v,u,0}$, is greater than or equal to 2 (or 2.5 for the fast variant). (See Definition 3.4 and Corollary 4.4.) This rules out any case of my scheme using inexact storage when the bound requires exact storage.

For a given m and u, the lowest competitive inaccuracy (highest competitive accuracy) for exact storage will always occur right before we adapt from exact storage to inexact storage. Assuming this lowest $m/\breve{m}_{v,u,0}$ is less than or equal to 2 (or 2.5), we satisfy the exact storage portion of Theorem 11.1.

Figure 11.2 shows numeric computation of the lowest $m/\breve{m}_{v,u,0}$ for various m and u. Observe that the data points for the accurate variant stay below 2.0 and the fast variant below 2.5.

To compute lower bounds for $\breve{m}_{v,u,0}$, I use some code I wrote to compute $\breve{m}_{v,u,f}$ based on Theorem 4.2:

$$\lg \frac{\binom{u}{v}}{\binom{w}{v}} = \lg \frac{(u)_v}{(w)_v} = \frac{\ln(u)_v - \ln(w)_v}{\ln 2}$$

The notation $(u)_v$ is the falling factorial or Pochhammer symbol, and I have written a routine to compute the natural logs of these, with iteration for small numbers and with asymptotic approximations for large numbers. The approximations are based on well-known approximations of the natural logarithm of the gamma function. Care must be taken to avoid the limitations of floating point arithmetic, especially in representing real numbers close to 1.0. See Figure 6.1 for further validation of these computations.

Focusing on Figure 11.2 again, I use $\lg u - \lg m$ for the X axis rather than u so that the graph has little dependency on the magnitude of m. Observe that there is very little difference between the results for $\lg m = 16$ (8 KB) and $\lg m = 33$ (1 GB), except when $u \gg m$ gets large. There is an interesting explanation for the jagged lines and generally lower memory factors associated with the $\lg m = 16$ case. Toward the right side of the graph, $\lg u$ is becoming a sizeable proportion of $m = 2^{16}$, which means the number of cells

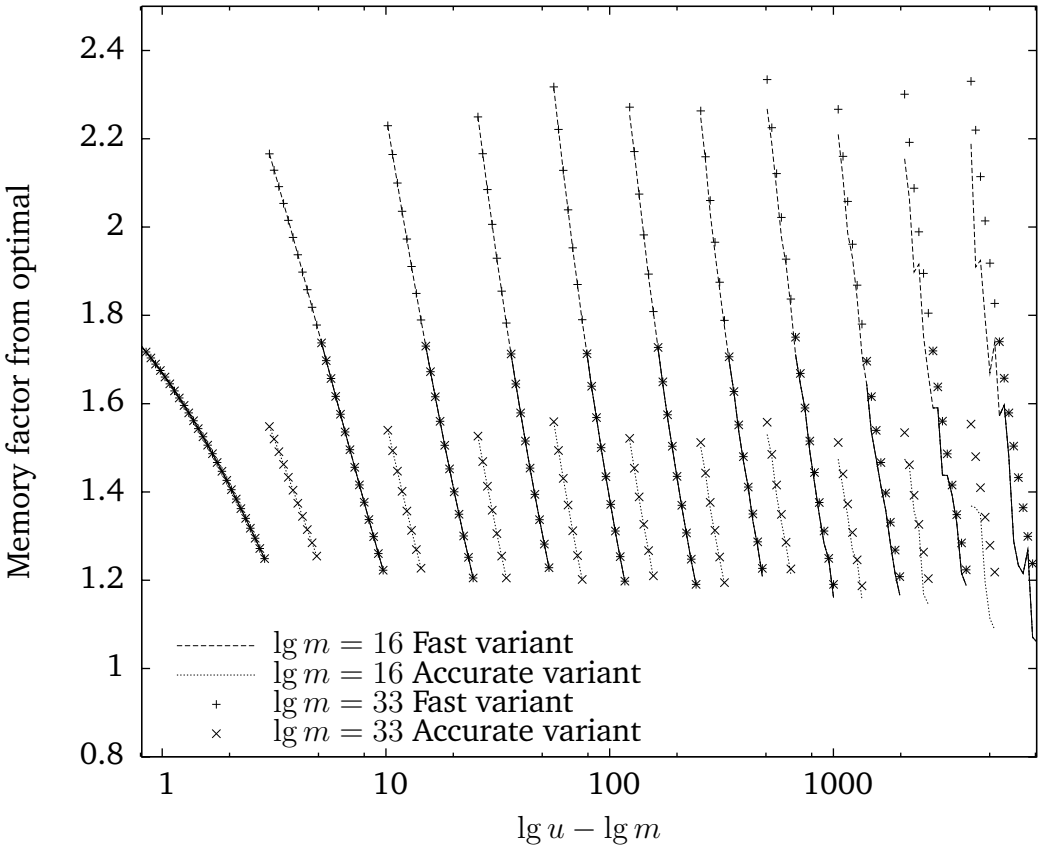

Figure 11.2: *Competitive inaccuracy of exact storage in variants of adaptive storage scheme. Right before converting from exact storage to inexact, the adaptive storage scheme is using some factor more memory than is theoretically possible for the v and u. These factors are graphed, based on a vast range for $\lg u - \lg m$, which is used instead of $\lg u$ so that the graph will be largely independent of the scale of m. (Notice there is not much difference between $m = 16$, the lines, and $m = 33$, the points.) The optimal is based on the maximum v before adaptation for each case, since this is most relevant to Theorem 11.1. The factor is below 2.0 for $\lg u - \lg m$ much smaller than Theorem 11.1 assumes, which is $\lg u - \lg m \geq 3$. See referencing text for more discussion on the interpretation of this graph.*

in the table is getting to be small. In that case, the occupancy is actually well above 85% the first time it reaches at least 85%; thus it is making better use of memory upon adaptation, and we can trust these bounds to hold even if the number of cells is very small. The spikes in the $\lg m = 16$ indicate where the maximum v for before adaptation goes down by one, thus creating an instant loss of memory utilization and competitive accuracy.

The results in Figure 11.2 need not satisfy the appropriate bounds for $\lg u - \lg m$ smaller than what is show, because the minimum relevant value for $\lg u - \lg m$ is approximately 0.152, based on the $m \leq 0.9u$ assumption in Theorem 11.1.

The results in Figure 11.2 continue to satisfy the appropriate bounds for $\lg u - \lg m$ beyond what is shown (about 8000, or about a 1 KB state descriptor). In this case, an asymptotic argument is easy to formulate, because as $\lg u - \lg m$ approaches ∞, the competitive accuracy ($\breve{m}_{v,u,0}/m$) approaches the proportion of the Cleary table memory that is entry bits from actual element descriptors (call it j/m). (j does not include bits for metadata, unoccupied cells, or wasted/unused space in occupied cells.) Both $\breve{m}_{v,u,0}$ and j are $v(\lg u - \lg v + O(1))$ (see Equation 4.6), where the $O(1)$ is insignificant as $\lg u - \lg m \to \infty$ because an assumption of Theorem 11.1 ensures $\lg u - \lg v > \lg u - \lg m$.

Now, we just have to verify that the adaptive storage scheme ensures that $j/m \geq 0.5$ (or 0.4 for the fast variant) at the point when adapting from exact to inexact. In the worst case, the entry size required for exact storage will be slightly larger than a size allowed by the design. (See the pattern established in Figure 11.1.) The two cases are (for some i) $2^i + \epsilon$ and $\frac{4}{3}2^i + \epsilon$ bits per cell, where ϵ is a negligibly small value. In each case, we are forced to go with the configuration with the next larger entry size, $\frac{4}{3}2^i$ in place of $2^i + \epsilon$ and $2^{i+1} + \epsilon$ in place of $\frac{4}{3}2^i + \epsilon$. This means we are using as little as $3/4$ths or $2/3$rds (respectively) of the entry bits for actual data from the element. (The rest is wasted due to the constraints of my scheme.)

On top of that, as little as 85% of cells are occupied. Thus, asymptotically, $j/m \geq 0.85 \cdot 2/3 = 0.5\overline{6}$, for the accurate variant. For the fast variant, entries only double in size, so $j/m \geq 0.85 \cdot 1/2 = 0.425$. That justifies that the appropriate bounds extend indefinitely to the right in Figure 11.2. (Look for the accurate variant staying below $0.567^{-1} = 1.76$ and the fast variant below $0.425^{-1} = 2.35$ in Figure 11.2.)

The results in Figure 11.2, shown for $m = 2^{16}$ and $m = 2^{33}$, also generalize to arbitrarily large m. If we double m, we should also double u, so as not to infringe upon our assumptions and to keep $\lg u - \lg m$ the same. Note that the cell size stays the same because the additional state descriptor bit is covered by an additional address bit in the Cleary table. Doubling m and u almost exactly doubles the memory lower bound (using Corollary 4.4):

$$2 \lg \tbinom{u}{v} = 2 \sum_{i=0}^{v-1} \lg \frac{u-i}{v-i} \approx \sum_{i=0}^{2v-1} \lg \frac{u-i/2}{v-i/2} = \lg \tbinom{2u}{2v}$$

Thus, the competitive (in)accuracy is virtually unaffected by doubling m and u.

11.1.4 Inexact storage case

The remaining part of Theorem 11.1 is the case of inexact (over-approximate) storage. This requires showing that for any legal m, u, and v for which my adaptive storage scheme uses inexact storage, the expected number of hash omissions is less than the information-theoretic optimal for half as much memory (or 40% as much memory, for the fast variant).

I start with numerical computation that shows the bounds are satisfied in many important cases, as shown in Figure 11.3. I use m of adequate size for Theorem 11.1, $m = 2^{16}$. u is sufficiently large that inexact storage is used well before adapting to the configuration using 64 bits per cell, such as $u = 2^{150}$. v ranges to consider every configuration after 64 bits per cell, up to the maximum v considered by Theorem 11.1: $\frac{1}{80} \leq \frac{v}{m} \leq 1$.

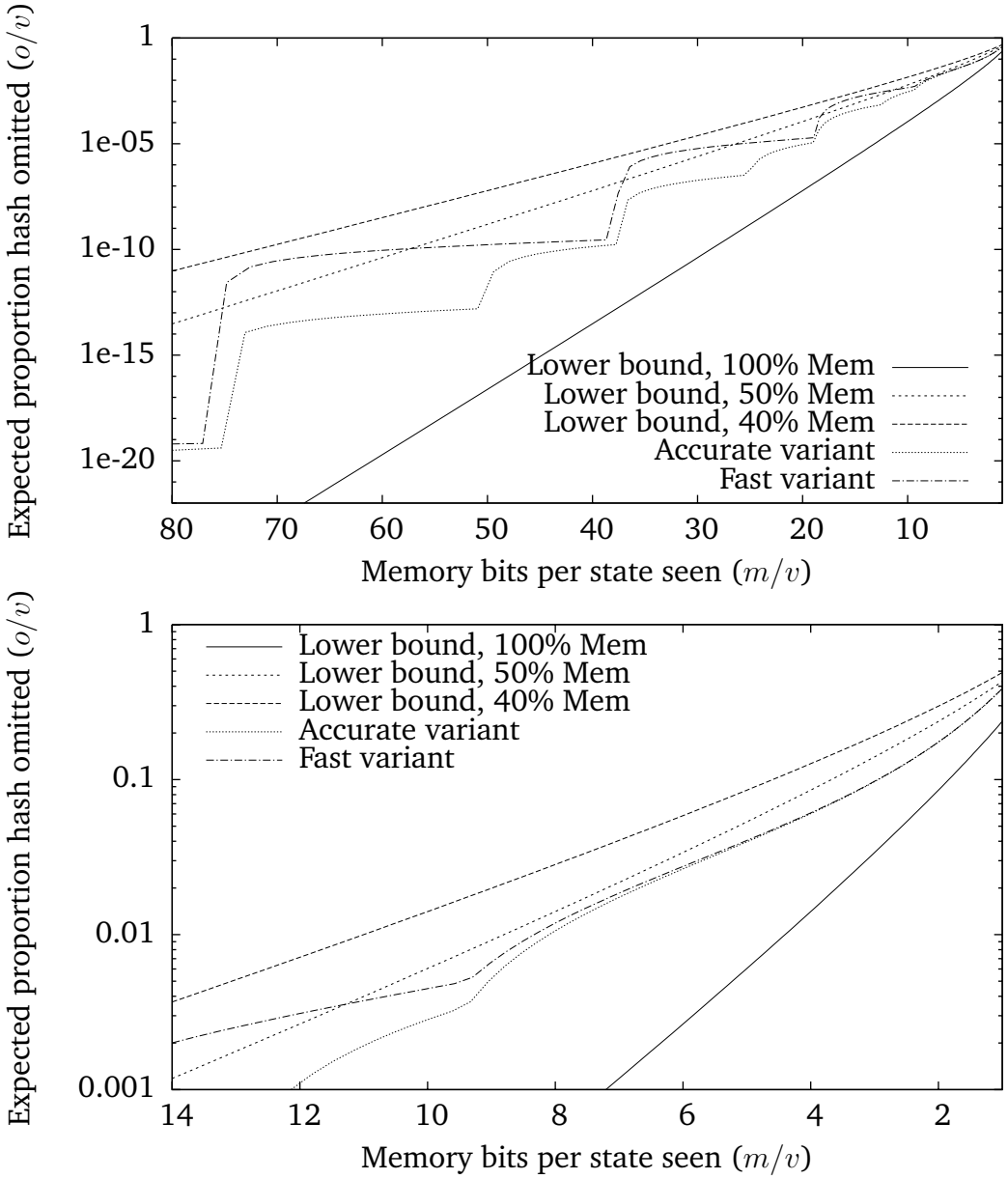

Figure 11.3: *Comparison of predicted inaccuracy of adaptive storage variants with information-theoretic lower bounds. Lower is better. The lower graph is a zoom of the top-right corner of the upper graph. These show that within this range, the accurate variant stays below the information-theoretic bound for 50% as much memory and the fast variant stays below the bound for 40% as much memory. These expected inaccuracies are computed from formulas as described in referencing text, using $m = 2^{16}$ and $u = 2^{150}$, though the results should not change substantively for larger cases.*

Recall that there are two forms of inexact storage used by my storage scheme: Cleary tables, using either reduction from Chapter 8, and the hash-reusing $k = 2$ Bloom filter (Section 6.4.4). To predict hash omissions, we use Equation 3.9, which expects the expected false positive rates for each number of visited states up to the maximum. For the Cleary table, I use the false positive rate formula for "even" partitioning, in Equation 8.11, using the expected value of n based on the history of false positive rates. For the Bloom filter, I use the formula from my analysis of hash-reusing, Equation 6.15. For the information-theoretic lower bound, I invert the memory lower bound from Theorem 4.2 in the naive fashion, by using a binary search in each case to find the largest w that the bound precludes as representable; using $f = \frac{w-v}{u-v}$, this gives a lower bound on the false positive rates. These analyses are the basis for the numerical results in Figure 11.3.

It is interesting to note that the bounds do not quite hold for the false positive rates that underlie the expected hash omission numbers. See Figure 11.4. Increasing the maximum occupancy a little should satisfy the bounds, however.

The first analytical generalization of the computed expected omissions in Figure 11.3 is that they are "scale independent," in the sense that if we scale up u, m, and v by some constant, such as two, they do not change substantively. The accuracies in Figure 11.3 are stated in terms of the expected proportion of states hash-omitted ($\hat{o}/v$), because it is scale-independent if the underlying solution is "asymptotically compact" (see Section 4.4), unlike the expected hash omissions ($\hat{o}$). The notion of "asymptotically compact" was derived from the information-theoretic lower bound, so we can trust its results to be scale-independent. By the approximation of Equation 6.5, Bloom filters are asymptotically compact, and it is easy to confirm that is also true for hash-reusing Bloom filters. The Cleary tables for inexact storage are also asymptotically compact, using an argument related to that of the Bloom filter. Doubling u, m, and v results in twice as many cells of the same size;

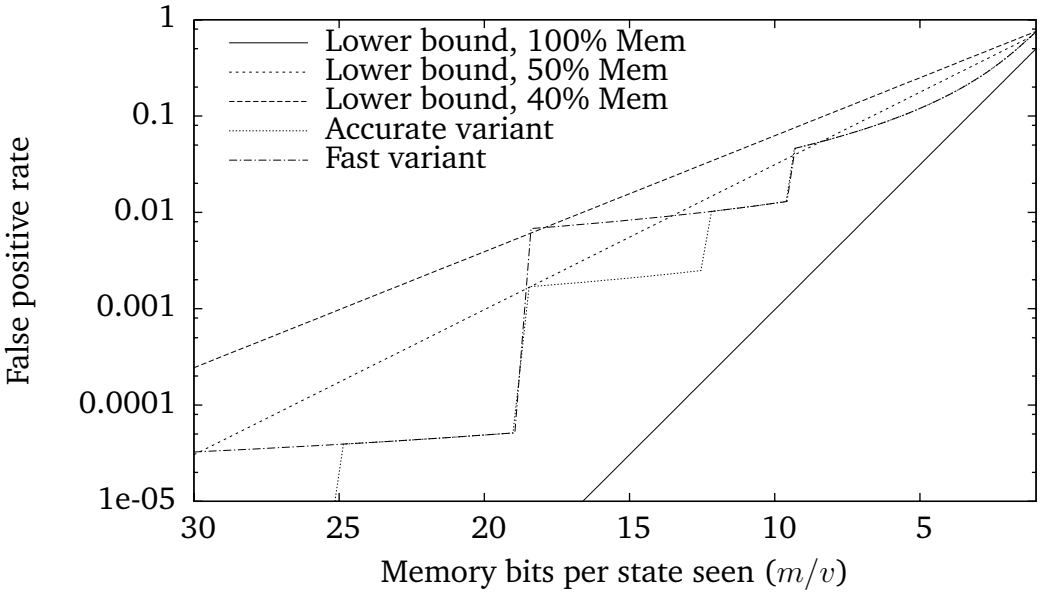

Figure 11.4: *Comparison of predicted false positive rates of adaptive storage variants with information-theoretic lower bounds. This is like Figure 11.3, except intended to show that the same bounds do not quite hold for the individual false positive rates that contribute to the expected hash omissions.*

because of the extra address bit, the number of partitions p doubles. By Equation 8.3, which is like $k = 1$ Bloom filter analysis, the false positive rate is essentially unaffected by scaling these variables together.

I have also confirmed the generalization of scaling u, m, and v simultaneously by generating the graph in Figure 11.3 for various scaled values, with visually identical results.

Next I describe how the bounds continue to be satisfied indefinitely to the left in Figure 11.3, meaning for $m/v > 80$, or equivalently, $v/m < \frac{1}{80}$, but for now I will assume that u is practically infinite. Now, from a m/v that calls for Cleary table storage, suppose we double m but keep v the same. This calls for Cleary table storage with the same number of cells but twice the size, which is always allowed by the pattern of the scheme, as established in Figure 11.1. The expected occupancy of the Cleary table, α, should be approximately the same in both cases. Thus, by using $2m$ memory instead

of m, we have added $\alpha m/v$ bits to each entry in the Cleary table, cutting the false positive rate by a factor of $2^{\alpha m/v}$. Based on our assumption that u is practically infinite, we can use the simpler lower bound, $\breve{m}_{v,\infty,f} \geq \lg f^{-v}$, from Corollary 4.3. Equivalently,

$$2^{\breve{m}_{v,\infty,f}/v} \geq f^{-1}$$

Thus, the false positive rate for the optimal is cut by a factor of $2^{m/v}$ by doubling memory, or in the case of the optimal for α as much memory, $2^{\alpha m/v}$. Thus, generalizing the results to $m/v > 80$ is a matter of ensuring that the occupancy α is greater than the competitive accuracy bounds claimed by Theorem 11.1, and I verified a nearly identical property in demonstrating the $j/m \geq 0.5$ (or 0.4) properties for exact storage.

Finally, I generalize the bound for any allowed m/v to any allowed u. Combined with scaling all three together, this completes the generalization of inexact storage bounds. For this generalization, I appeal to Corollary 4.7, which says that unless exact storage is possible, the difference between the real lower bound and the one assuming u is practically infinite comes down to just a constant number of bits per added element. Such a difference is less significant when $\lg u - \lg m$ is large, making inexact storage extend to large m/v. Thus, any problems with various u should appear when $\lg u - \lg m$ is near the minimum permitted by Theorem 11.1, $\lg u - \lg m = 3$.

I have tested the bounds for various u with $\lg m \approx 16$, and they clearly hold if using "even" partitioning (as specified in Figure 11.1). Interestingly, if we were to use "balls and bins" partitioning (Section 8.2), the bounds would not quite hold, unless we increase the maximum occupancy in the scheme from 85% to about 87%. See the graphs in Figure 11.5, which show the bounds for 100% memory and 50% memory as solid lines, the accurate variant of the adaptive storage scheme as a fine-dotted line, and the accurate variant modified to use "balls and bins" partitioning as the dash-dotted line.

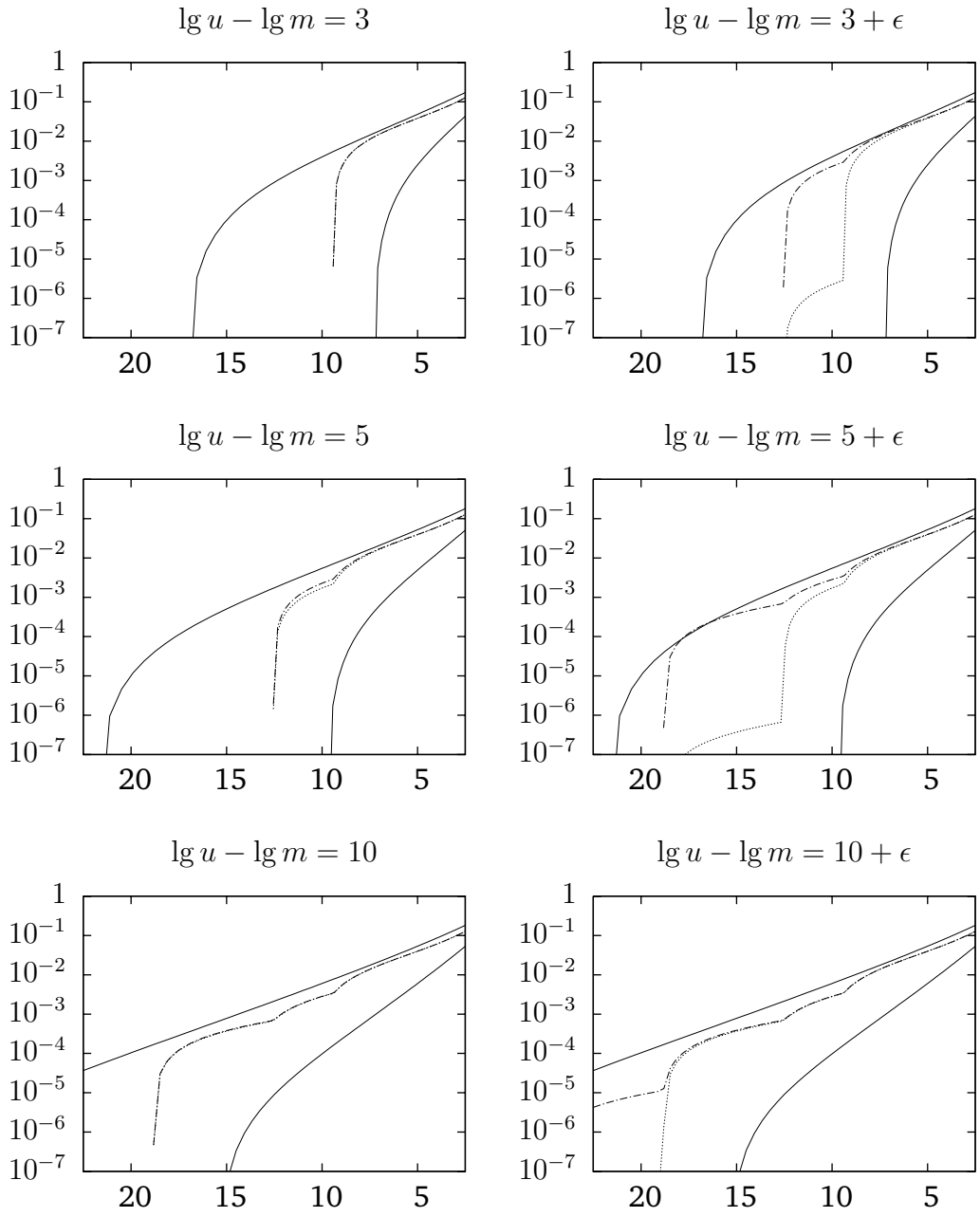

Figure 11.5: *Demonstration of inaccuracy bounds on adaptive storage for various universe sizes. These have the same axes as Figure 11.3; the X axis is memory bits per state seen (m/v) and the Y axis is the expected proportion of states hash-omitted (o/v). The solid lines are the bounds for 100% and 50% as much memory. The fine-dotted line is the accurate variant of the adaptive storage scheme, using "even" partitioning for inexact storage. The dash-dotted line is the same except with "balls & bins" partitioning, which is worse in some rare cases like these.*

The first interesting case, shown in Figure 11.5, is exactly $\lg u - \lg m = 3$. In that case, no inexact Cleary table configurations are used, so the difference in partitioning methods is not even exercised. Next is $\lg u - \lg m = 3 + \epsilon$, which is the most accuracy-demanding configuration to use the standard 8-bit-per-cell Cleary table for inexact storage. Even partitioning shows its advantage because the descriptors are only barely too large for the standard 8-bit-per-cell Cleary table to represent them exactly. $\lg u - \lg m = 5$ is just the right size for the 3-in-4 Cleary table with 8 bits per "cell" to represent exactly, so when two bits of each descriptor are thrown away for the standard 8-bit-per-cell, there is not a huge difference between the partitioning methods.

$\lg u - \lg m = 5 + \epsilon$ is the case in which using "balls and bins" partitioning would violate the competitive inaccuracy bound of 2.0; the dash-dotted line in the graph for this case in Figure 11.5 pokes above the bound. Using "even" partitioning is not even close to the bound in this case. The rest of the graphs are not so interesting, as they fill out to look like Figure 11.3, staying below the claimed bound for any acceptable u.

11.1.5 Final notes on the theoretical bound

Though well short of a full proof, my defense of Theorem 11.1 should be compelling and informative. Basically, I have shown the ways in which the storage scheme is fundamentally scalable with the generalizations of the numerical results. In the next section, I address certain speed concerns, that are not scalable if the full design is used.

The accuracy bound in Theorem 11.1 is essentially a practical conception of dynamic flexibility, which in Definition 3.5 is not limited to "all practical cases." The adaptive storage scheme as described in Figure 11.1 is optimized for such practical concerns. If we wanted to have the best known dynamic flexibility according to Definition 3.5, we would end with a $k = 1$ Bloom

filter instead of the hash-reusing $k = 2$. For example, the "accurate" variant of adaptive state storage is more dynamically flexible than most other data structure configurations considered in this dissertation, but its dynamic flexibility is technically incomparable with the standard $k = 2$ and $k = 1$ Bloom filters. Though typically worse in practical cases, ending with a $k = 1$ Bloom filter would have better dynamic flexibility than all other data structure configurations considered, under some reasonable assumptions. Details of these arguments did not make it into this dissertation. Instead, Theorem 11.1 captures a notion of dynamic flexibility that is more practical for problems of visited set storage.

11.2 Near optimal speed and accuracy in practice

The full adaptive storage scheme in Figure 11.1 can be too slow to be practical, but for a given machine, we can implement a fast subset of the design with accuracy practically indistinguishable from the full design. Basically, if the largest (starting) cell size is the size of just one or two machine words, the probability of that causing more actual hash omissions than the full design is one in millions or billions.

Exact storage with Cleary tables also has low hashing requirements, so it would be possible also to include that case in a practical implementation, but work on that case was not completed in time for writing.

11.2.1 Practical problems with full design

There are a few practical problems with the full design of my adaptive storage scheme. The first has to do with hashing. The scheme assumes that the entire state descriptor is hashed with a truly randomizing bijective hash function; if the state descriptor is large, this is a ridiculous assumption. One

might think we could use a standard block cipher to randomize a long descriptor, but it only randomizes a block at a time, with limited "memory" of the past. Thus, only a limited summary of the information about the first half of the descriptor would be "mixed in" with the second half.

A more practical assumption is that we can quickly compute a hash of the descriptor equal to three machine words in length. This is reasonable because Jenkins hash functions, known for a great combination of speed and quality [52], are based on mixing pieces of input into a three-word state. (I have before validated using the full state of a Jenkins hash function as the hash value [24, Section 4].) And if the static size of the descriptor is small enough (three words or less), it should be reasonable to compute a one-to-one hash. Jenkins functions are based on bijective transformations of the internal state, and it should be easy to restrict that to a given size.

The second problem with the full design is that access and adaptation require shifting around and copying descriptors that are potentially rather large. In that case, some adaptation operations would not satisfy the precondition for $O(1)$ auxiliary storage in Corollary 10.7, though requiring a few descriptors worth of working space is not in itself a big practical concern. More concerning is the cost of shifting around large cell entries when adding elements to a heavily occupied table. Restricting Cleary table cells to a small number of machine words keeps required auxiliary storage and access times low.

The final problem with the full design is that it does not handle the case of infinite universe in a reasonable way. Because of the limitations in analyzing exact storage requirements (see Section 4.5), my bounds assume exact storage from an infinite universe is not possible in finite space, so the scheme would require us to start out with one cell almost the size of the whole structure and split it from there as we add more elements. Clearly, starting with some reasonable cell size is a good choice in this case.

11.2.2 Practical implementation

The natural way to deal with these shortcomings is to start with a configuration whose accuracy is practically indistinguishable from that of the full design, but whose cells are of reasonable size. Luckily, the absolute accuracy of an "asymptotically compact" structure improves exponentially with additional memory per added element. More specifically, the following theorem guides implementation of the scheme such that its accuracy is not practically distinguishable from the full design:

Theorem 11.2. *Let $\dot{m}$ be the maximum number of bits of memory addressable by an implementation of inexact Cleary table storage. If the cell size is at least $q + 2 + \lg \dot{m}$ bits and the number of visited states is not too many to overflow the structure, the probability of any omissions is no more than 2^{-q}.*

Proof Note that in the formula for the prior probability of no omissions (Equation 3.11), we makes the assumption at each step that the affecting additions n equals the unique additions v. Thus, the false positive rate using "even" partitioning (Equation 8.11) is no higher than that using "balls and bins" partitioning, which by Equation 8.4 is n/p, where p is the number of partitions of the universe. To get the probability of any omissions, we subtract from one the probability of no omissions, from Equation 3.11, and use the $n = v$ assumption:

$$P(o > 0) = 1 - \prod_{i=0}^{n-1} 1 - \frac{i}{p}$$

Using some properties of arithmetic,

$$P(o > 0) \leq \sum_{i=0}^{n-1} \frac{i}{p} = \frac{(n-1)^2 + (n-1)}{2p} \leq \frac{n^2}{p}$$

From here, we know $n \leq 2^{\dot{m}}$ from the assumption that the number of visited states is not too many to overflow the structure, and we know $p \geq n2^{\dot{m}+q}$

based on there being at least n home addresses and at least $\dot{m} + q$ entry bits per cell. Thus,

$$P(o > 0) \leq \frac{n2^{\dot{m}}}{n2^{\dot{m}+q}} = 2^{-q}.$$

□

Using an implementation on a machine with byte-addressing ($2^3 = 8$ bits per byte), this corollary is more directly useful:

Corollary 11.3. *Using an implementation of inexact Cleary table storage for an architecture with word size b bits, able to address up to 2^b bytes of memory, the probability of any omissions is no more than 2^{-q} if the cell size is at least $q + 5 + b$ bits and the number of visited states is not too many to overflow the structure.*

For example, suppose we are using a 32-bit machine, and we can accept a one in a million chance of any omissions (2^{-20}). We only need cells of at least $20 + 5 + 32 = 57$ bits. Thus, if our starting configuration for the adaptive storage scheme is 64 bits per cell, our chances of being able to notice a difference in accuracy, by noticing at least one state was omitted, is one in millions. In fact, using Theorem 11.2, 64 bits per cell is "one in a million" good up to 2^{42} bits of memory, or 512 GB. Thus, 64 bits per cell is a fine upper limit for cell sizes at the time of writing.

In general, for word sizes of at least 32 bits, one or two words is enough to guarantee that it is practically impossible to observe a loss in accuracy compared to the full adaptive design, and it allows for fast hashing, access, and adaptation. A hash value of three words, such as that from a Jenkins-style function, is sufficient, because one word is enough for the home address and two words are enough for entry bits in the cell. And two words is not an unwieldy size for access or adaptation.

11.2.3 Active state matching

Cell-based structures such as Cleary tables are particularly well-suited for integration with partial-order reduction (P.O.R.), which has drastically increased the effectiveness of explicit-state verification methods [47, 33]. Adaptation complicates that integration, as does the $k = 2$ hash-reusing Bloom filter, but the problems are surmountable. The resulting integration is faster than having separate structures to support P.O.R., and can save a lot of memory also.

I introduced and validated this integration in the SPIN Workshop paper that introduced the "fast" variant of my adaptive storage scheme [26]. Here I describe the same integration.

The typical implementation of P.O.R. requires runtime support in the form of a "cycle proviso," which needs to know whether a state is on the DFS stack [47] or BFS queue [8] (depending on whether depth-first or breadth-first search is being used). We will refer to this as checking whether the state is **active**, or **active state matching**.

If the visited set is based on cells, one bit with each cell can be used to indicate whether the state stored in that cell is active. I call this **integrated matching** of active states. In the adaptive storage scheme, the one bit per cell for the ACTIVE flag comes from using one less entry bit for each configuration, so that cell sizes are still powers of two.

Integrated matching is more compact than using a separate structure if the proportion of visited states that are active reaches a significant level, and adding random access to the stack or queue of active states would be expensive. If, for example, less than 1% of visited states are active at a time, then a separate structure using 64 bits per active state would be more compact than dedicating a bit for each visited state. However, predicting the relative proportion of visited states and active states is similarly as difficult as predicting the number of visited states, and integrated matching is the

more robust approach, because it cannot overflow. For details, see [26].

Adding random access ("active state matching" for P.O.R.) to the stack or queue of active states could be expensive or impractical for two reasons. First, a stack or queue of states can reasonably use out-of-core storage (disk) because stack and queue operations are adjacent to either the last "add" or "remove" operation, meaning the access is "linear" or "streaming," which disk is optimized for. Thus, if most of the stack is on disk, as with SPIN's `-DSC` option, we cannot add efficient random access to the search stack of states as it is stored. In fact, not allowing the search stack/queue to spill to disk impairs robustness of the search, because there has to be a cap on the number of active states to prevent thrashing.

Second, adding random access to the stack/queue might also be impractical because of lossless compression. It is common for adjacent states to share most of the content of their descriptors. Because access to the stack/queue is always in a prescribed order, only the differences between states need to be stored. This can save a lot of memory and/or disk usage and bandwidth, but inhibits random matching of states on the stack/queue.

Clearly, without knowledge that can only be acquired through luck or trial and error, the search needs integrated stack/queue matching and a search stack or queue that mostly lives on disk in order to be robust.

The details of how active state matching is integrated into my adaptive storage scheme, including the hash-reusing Bloom filter, are in [26].

11.2.4 Practical speed

The speed of the storage scheme depends on many factors, but typically it is a little faster than the standard $k = 3$ bitstate/Bloom filter approach before the first Cleary table reaches high occupancy, slower for the intermediate Cleary tables, and faster for the hash-reusing $k = 2$ Bloom filter. The difference in speed with standard approaches is often hidden by other overhead

in industrial examples.

Figures 11.6 and 11.7 give us a lot of information about the speed of the variants of my adaptive storage scheme. The speed of the $k = 1$ Bloom filter should be regarded as "practically optimal," and the speed of the $k = 3$ Bloom filter is generally recognized as good.

The first thing to notice in Figure 11.6 is that the time for each adaptation operation is relatively small. These are the places in the graph at which the number of explored transitions is momentarily flat, and these are marked with points on the line. Because adaptation is done with essentially streaming access to memory and no hash computation, it is very fast. In these tests, each adaptation takes between 1% and 2.5% of the running time so far. That could be higher for models with less overhead or much lower for industrial models with a lot of overhead, such as in hashing large descriptors. Note that because the time between adaptations increases exponentially through the verification process, the total time spent in adaptation never really exceeds a small factor more than the last adaptation (about 3x for "accurate" variant and about 2x for "fast" variant).

Next, for Cleary tables, the speed of each lookup depends on the occupancy, as was also shown in Figure 9.4. When sparsely occupied, the speed of each lookup is noticeably faster than the $k = 3$ bitstate, which uses three random memory accesses. As expected with a structure using linear probing, the access times rise quite significantly as the structure fills to 85% occupancy, and abruptly return to faster times after adapting, which lowers occupancy. The visited set usage paradigm tolerates such rises because they are are only temporary. Using a smaller maximum such as 65% predictably mitigates this slowing at a moderate cost in accuracy. Note that the maximum occupancy before adaptation could be chosen dynamically, so for example, it could be chosen based on what percentage of time the model checker (or other program) is spending on Cleary table accesses, allowing for dynamic balancing of speed versus accuracy.

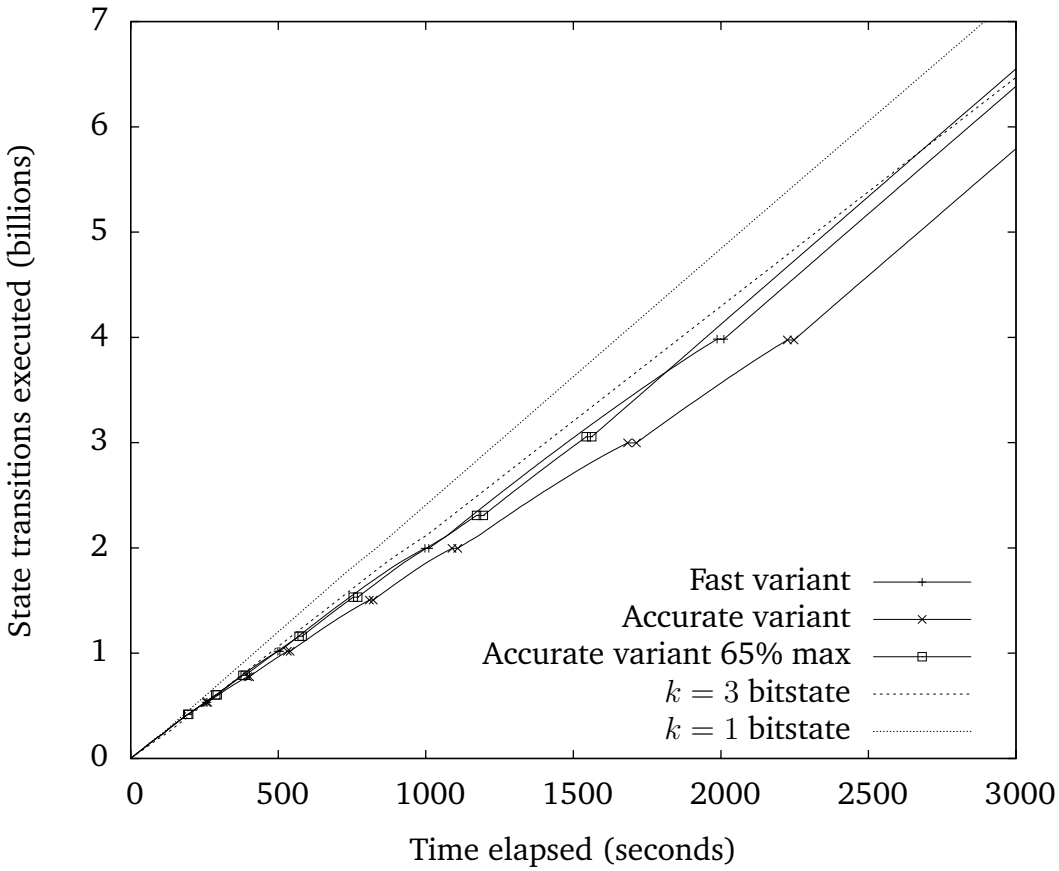

Figure 11.6: *The progress over time in exploring a state graph with different storage schemes. Progress is measured by the number of state transitions executed (graph edges explored). The points on the lines for adaptive storage delineate the beginning and end of each adaptation, during which no transitions are executed. All the adaptive schemes started with a 64-bit-per-cell table and ended with the hash-reusing $k = 2$ Bloom filter. The model is one from the* SPIN *distribution, PFTP, scaled up to have an enormous number of states and 296-byte state descriptors. Each of these runs executed about 13.5 billion transitions, distinguishing about 6 billion unique states in about 6200 seconds, except the $k = 1$ bitstate (Bloom filter) configuration only explored 12.5 billion transitions and 5.5 billion states, in 5200 seconds. Hash computation, using a Jenkins hash function, was the same for each configuration. Beyond the bounds of this graph, the lines continue with the same slope. I used 3SPIN with 2 GB for visited state storage and limited depth to 20 million. Partial-order reductions were used, using a chaining hash table to match states on the stack. Tests were compiled with gcc 4.4.3 (-O3) and run alone on a 64-bit Linux system with Intel Xeon X5677 CPUs (3.47GHz).*

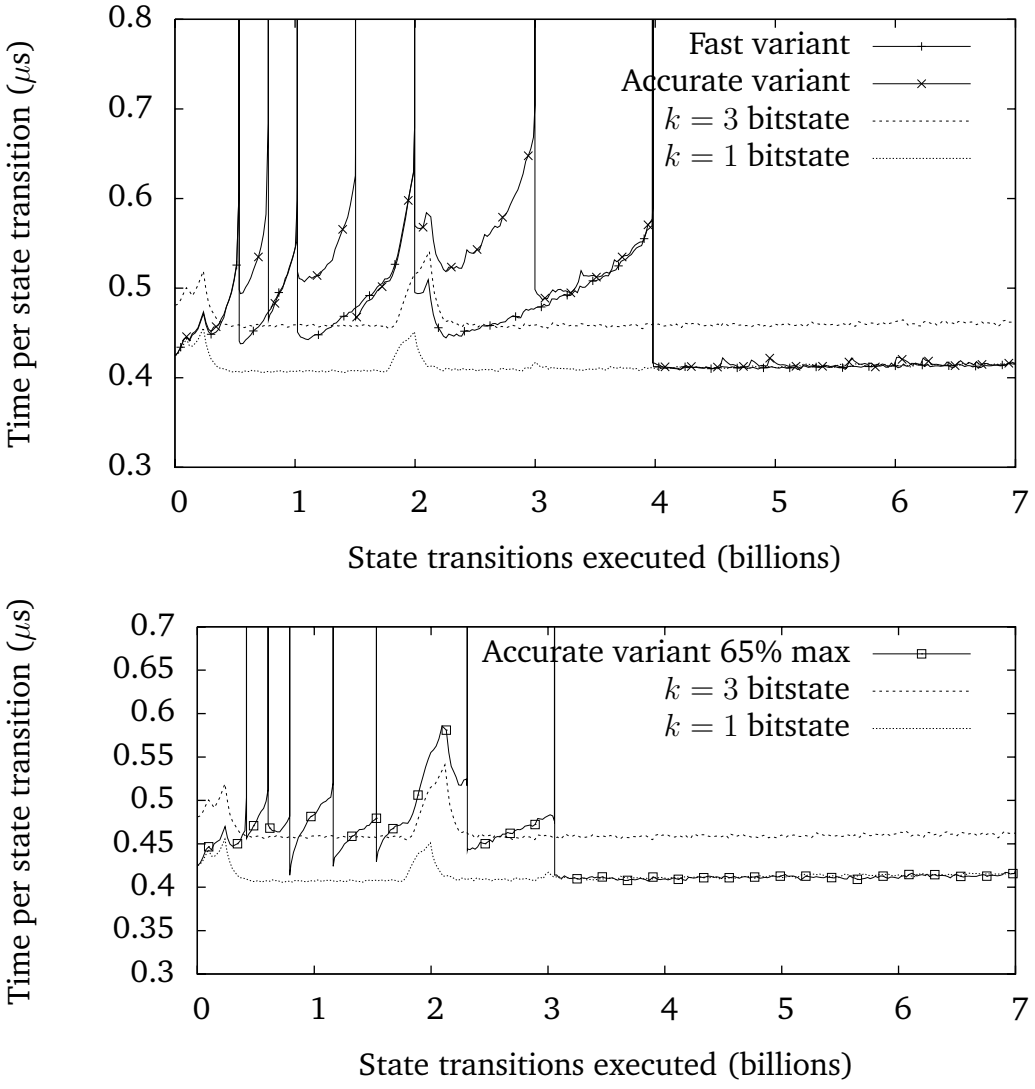

Figure 11.7: *The time per state transition over the duration of exploring a state graph with different storage schemes. These graphs are essentially the inverse of the derivatives of the curves in Figure 11.6, because it shows the time per transition rather than total transitions. To avoid clutter, the test using a different maximum occupancy is in the separate lower graph. The adaptations occur where the time is briefly enormous; the points are simply at regular intervals to help distinguish curves. To have adaptations line up between the "fast" and "accurate" variants, the X axis is total transitions rather than elapsed time. The dips in the "optimal" time, given by the $k = 1$ Bloom filter, coincide with drastic changes in search stack depth. Based on profiling and other tests, I estimate about 0.3 microseconds of overhead per transition: about 0.13 microseconds on hashing and about 0.17 on other things for transition execution, invariant checking, and partial-order reduction.*

The hash-reusing $k = 2$ Bloom filter appears to be essentially as fast as the $k = 1$ Bloom filter, which must be because each access is confined to two adjacent bytes of memory. Thus, only one random access to memory is required for each access. It is well-known that Bloom filter access times can be improved by increasing locality of the indices (e.g. [72]), but it comes with an accuracy cost. In fact, by preceding the hash-reusing Bloom filter with more accurate Cleary table configurations, we are only using the Bloom filter when the accuracy impact of index locality is smallest ($m/v < 10$). Though in many cases the $k = 3$ Bloom filter is more accurate than the adaptive storage scheme, its higher peak competitive accuracy comes at a cost in dynamic flexibility.

In many cases, the adaptive storage scheme is using the hash-reusing $k = 2$ Bloom filter for most of the duration of verification. This is because adaptation to the Bloom filter happens around $v/m = 0.1$ and can easily extend to $v/m = 0.4$ or higher. In that case, having gone through the slower Cleary tables has little relative impact on the overall running time. Figure 11.6 shows less than half of the overall running time for the test, and all the adaptive storage configurations end up with an average speed faster than the standard $k = 3$ Bloom filter configuration.

The adaptive storage scheme is also near "optimal" speed until the first Cleary table configuration gets to be about half full. This is visible in Figure 11.7, up to about $1/4$ billion transitions. If an error exists in the model, it is often manifest in multiple places in the state space, meaning it is expected to be discovered before exploring the majority of the state space. Thus, in many cases, my adaptive storage scheme will find an error before the first adaptation, and find it slightly more quickly than the standard $k = 3$ Bloom filter configuration.

When the storage scheme is using a Cleary table that is mostly full, then the speed is not as clearly or consistently "near optimal," but it has favorable scaling properties. First of all, it should be pointed out that others have

shown that only $O(1)$ search is needed from the home address if we enforce a maximum occupancy less than 100% and have an adequate hash function [66]. However, an increasingly dominant cost in accessing large data structures is the cost of random accesses to main memory [53], and because the Cleary table uses linear probing, only one random access is needed per operation. Integrating matching of active states for partial-order reduction accomplishes even more with that one random access.

The low need for random memory accesses led me to hypothesize that on a system in which multiple CPU cores are active and routinely accessing main memory, my adaptive storage scheme would have less degraded performance than a $k = 3$ Bloom filter, which uses three random memory accesses. I confirmed that hypothesis in some detail in [26], and here I briefly show more support. I re-ran the tests from Figure 11.6 by running eight configurations simultaneously on the 8-core machine, as in a "swarm" configuration [46] or a grid or cloud computing environment. The $k = 3$ Bloom filter took 10.7% longer. My adaptive scheme took about 8% longer (7.8% longer for "accurate" variant and 8.2% longer for "fast" variant), which indicates less degradation in a high-load, parallel environment. (See Section 11.3 for discussion of parallelization.)

Another important consequence of the scheme's low reliance on random memory accesses is that its performance relative to a Bloom filter is worse when random accesses are cheap, such as when all or most of the data structure fits in CPU cache. In such cases, the searching and shifting with linear probing are, relatively speaking, much more expensive. Therefore, do not be deceived by the speed of the scheme in artificially small experiments. And small experiments are not realistic, because the user of an explicit-state model checker should always allow use of as much memory as is readily available.

On large industrial problems, time spent on other things such as hash computation should dominate the extra time associated with the adaptive

storage scheme during part of its life cycle. My tests have been performed using a modified version of SPIN, which is known to be the fastest explicit-state model checking tool. A well-known related tool, Murφ (or "Murphi") [76, 78], is usually one to two orders of magnitude slower, in terms of rate of states or transitions explored. Murφ's symmetry reductions, for example, reduce state space sizes dramatically, but incur a relatively large per-state time cost [15]. Even in SPIN, industrial examples tend to take longer per state than my examples, partially because they tend to have larger state descriptors, which take more time to hash. See Tables V and VI in [43].

Nevertheless, the inherent time associated with my storage scheme is not much different from the practical "optimal," the $k = 1$ Bloom filter. Each requires only one random access to main memory. By using the "fast variant" and/or lowering the maximum occupancy, the actual speed can be made very close to that optimal. If a little extra time in state storage is not going to slow the process down much, optimizing the accuracy with the "accurate" variant and higher maximum occupancies is the natural choice.

11.2.5 Practical accuracy

Using realistic examples, the observed accuracies of the adaptive storage scheme are consistent with expectation from the formulas. To get close matches with expectation over a wide range of configurations, we have to control for transitive omissions, however. When we test a model constructed to have almost no transitive omissions, the accuracies very closely match expectation.

As discussed at length in Section 3.6.2, it is hard to predict transitive omissions from hash omissions; thus, we cannot empirically validate the precision of our hash omission analysis using a model susceptible to transitive omissions. Fortunately, it is not hard to construct a model that has almost no transitive omissions. In particular, if the in-degree of each state

(node) in the state graph is large (such as 10), then for state, each of the 10 predecessors would have to be omitted for it to be transitively omitted. We can simply make the state of each model an integer from 1 to the desired number of states, and from each, non-deterministically transition to the states you get by adding each of the first 10 prime numbers, assuming the result is in the desired range.

Figure 11.8 uses that highly-connected model to validate predicted accuracies against observation, for a range of m/v values. The results are remarkably close in all the cases visible. Getting statistically significant results for more accurate cases would require an enormous number of runs. In fact, in those cases, we are essentially testing the probability of any omissions, which does not require a contrived model.

In Figure 11.9, I validate that computed probabilities of no omission for the adaptive storage scheme are accurate for various models of asynchronous systems. These models are more representative of industrial examples than the previous synthetic model, because they exhibit some transitive omissions, such as one of the 750 verification runs on Peterson's algorithm, which had 36 total omissions instead of the usual 0, 1, or 2. (The probability of that many hash omissions is one in billions.) However, if you believe in the exact storage of the Cleary table, the exact-to-inexact reduction, and the effectiveness of the 64-bit Jenkins hash function used, these results are predictable.

11.3 Parallel model checking, etc.

Parallel algorithms usually have the advantage of making use of more computing resources to solve a problem quickly. This has been studied heavily in the context of explicit-state model checking[2], with mixed results. Typically, the best results are shown when non-storage overhead in the checker is high

[2]At the time of writing, the Stern/Dill paper [79] has more than one hundred citations.

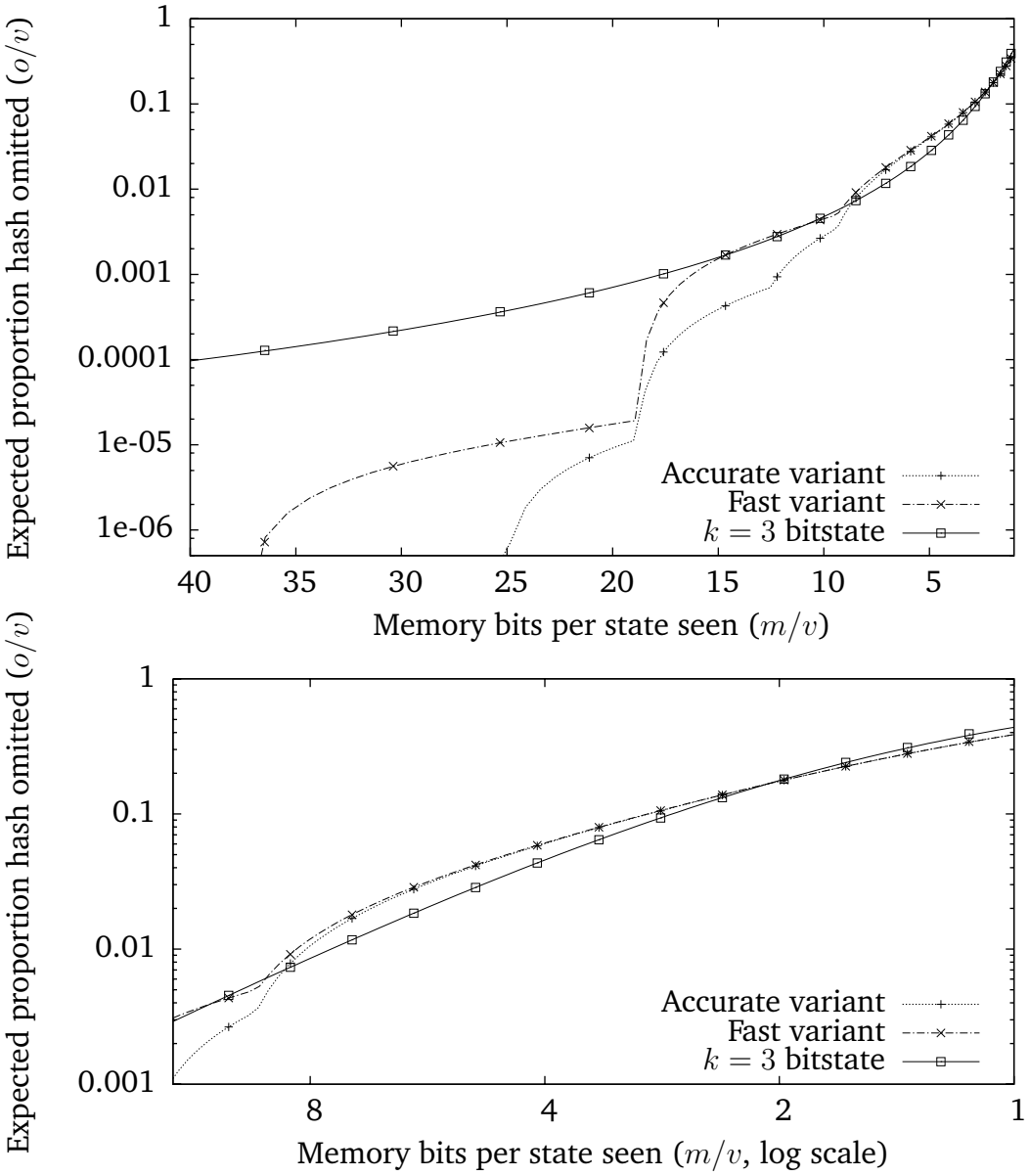

Figure 11.8: *Empirical validation of predicted hash omissions for the adaptive storage scheme. The lower graph is essentially a zoom of the top-right of the upper graph. This is essentially a repeat of Figure 11.3 with empirical results, represented by the points plotted. The lines are the prediction based on iterating the formulas to get expected hash omissions. The empirical results use 3SPIN on a model with a controllable number of states and virtually no transitive omissions. 20 to 1000 iterations with distinct hash function seeds were averaged for each configuration, depending on what was needed to keep sampling error low.* $m = 2^{23}$ *was used for empirical testing, though* $m = 2^{16}$ *was used to compute expectations. (The axes are scale-independent for these solutions.)*

Model	State size	Reachable states	Memory for visited set
Leader(8)	340 bytes	4 926 645	18 MB
Peterson(4)	48 bytes	16 819 903	71 MB
Sliding Window(5)	68 bytes	11 876 485	50 MB
Philosophers(9)	156 bytes	1 640 881	5 MB
Phone Switch	92 bytes	32 898 808	145 MB

Model	$P(o = 0)$, Predicted	$P(o = 0)$, **Observed**
Leader(8)	0.4370	**0.450** (**450** / 1000)
Peterson(4)	0.4323	**0.412** (**309** / 750)
Sliding Window(5)	0.5392	**0.527** (**790** / 1500)
Philosophers(9)	0.5897	**0.588** (**1175** / 2000)
Phone Switch	0.6139	**0.635** (**254** / 400)

Figure 11.9: *Empirical validation of predicted probabilities of no omissions for the adaptive storage scheme. The empirical results are **bold**. These models are standard test models distributed with* SPIN *[43], tested here using 3SPIN with adaptive storage ("accurate variant") on a 64-bit machine. The top table has information about each model and the configuration used in testing it. The memory size was chosen to have a probability of full coverage near 50%, to maximize the entropy in the boolean result of each trial, which is either full coverage ($o = 0$) or not. The number of trials was chosen by running trials for each model for a couple of hours. Partial order reduction was not used, for the state space sizes to be larger.*

and/or partial order reductions are disabled. Here I review some of these approaches and discuss how applicable my adaptive storage scheme is.

11.3.1 Message-passing parallel

The pioneering technique introduced by Stern and Dill uses a hash function to divide the state space among the computational nodes [79, 80]. As each node computes a successor state, it is sent to the owner node for that state, based on the hash. The original method uses a random hash of the full descriptor, to divide the state space evenly and non-heuristically, but in that case, successor states are expected to require transmission to another node.

Lerda and Sisto showed how to reduce communication overhead, using a partitioning of the state space that heuristically favors keeping transitions local to a node [60].

My adaptive storage scheme is well-suited to this form of parallel model checking, because each node has its own private memory space for state storage. However, this approach is generally considered to incur too much communication overhead to offer an advantage.

11.3.2 Shared memory parallel

Holzmann and Bosnacki designed an extension of SPIN designed to take advantage of multicore processors, using a shared-memory algorithm that supports partial-order reduction [43]. The processing threads need concurrent access to the visited set, so it should either support atomic access or use mutual exclusion at sufficient granularity. However, a small level of revisitation of states might be acceptable [43, Section VI.A], and that makes some data structures such as Bloom filters concurrently accessible without locking. Laarman, van de Pol, and Weber describe a lock-free hash table designed for this application [59], but their experimental results are suspect for not using partial-order reduction; nor do they consider approximate state storage.

There does not seem to be a way to access Cleary tables concurrently without locking; the invariants are complicated and span ranges that are highly dynamic. One strategy for locking in my adaptive state storage scheme would be to partition the state space using bits from the computed hash and allocate independent structures of equal size for each partition. This is essentially the same as allocating one big structure with extra "walls" in the Cleary table that cannot be crossed in a single access. It is not clear how much overhead the locking would entail compared to other schemes, but this would be worth investigating.

This shared memory approach is not seen as consistently scalable, including by Holzmann [46, Section 1, last par.]. My limited experience has indicated that multicore verification can easily "upset" partial order reduction such that orders of magnitude more states are visited.

11.3.3 Independent parallel

An approach more recently promoted by Holzmann is "swarm verification" [46] in which many single-threaded verifiers are run simultaneously, with different seeds for search randomization but no communication at run time. This ensures that communication does not slow down the process or interfere with partial-order reduction. A depth-first search with randomized successor ordering works well because it seems to give the independent processes the best chance of exploring distinct parts of the state space as soon as possible without communication. Bitstate storage (Bloom filter, $k = 2$ or 3) with different hash seeds works well.

My adaptive storage scheme is well-suited to this form of parallel model checking, but the advantages of adaptive storage do not quite match the use-case that swarm verification was intended for. Swarm verification is designed for bug hunting in models that are presumed so large that high-assurance verification is not practical. It is possible that the model might be unexpectedly tractable to the high-assurance verification possible with adaptive storage, but even in that case, the fact that multiple runs have likely covered the vast majority of the state space makes it less important that each run be highly accurate. The transitive omission problem is simply not as compelling a justification for an individual search being highly accurate if many searches are being run in parallel.

Nevertheless, in order to find bugs quickly, the best choice would likely be the fastest technique that also has decent accuracy. Recall that when the Cleary table is less than 50% occupied, it is faster than a $k = 3$ Bloom filter. A

hash-reusing Bloom filter (with index locality) or a $k = 1$ Bloom filter would be faster still, though with a likely moderate cost in accuracy. Therefore, if one were going to run swarm for a pre-determined, short period of time on machine(s) with a lot of memory, the standard $k = 3$ Bloom filter is not the most attractive option, especially since my adaptive storage scheme permits changing one's mind about aborting the search and instead allowing it to continue until starvation. (Without adaptation, Cleary table storage would often overflow before search starvation.)

Also recall that storage schemes that minimize random accesses to main memory scale better with respect to running many instances in parallel on a single machine. This is likely to become a more important factor in speed as the number of cores in a CPU increases to tens or hundreds. So even if we are not doing parallel model checking, this could be important to performance in a "grid" or "cloud" computing environment.

11.3.4 Summary

Although not likely to be the fastest data structure for concurrent, shared-memory access, my adaptive storage scheme is likely to find application in parallel computing environments, because of its dynamic flexibility and favorable access pattern to main memory.

CHAPTER 12

Other Related Work

Besides the related work cited throughout, there are other data structures that might be used to solve Problem 3.1. Some structures, such as Pagh et al.'s "optimal Bloom filter replacement" [65], look good in theory but do not have good practical performance (see Section 5 of [65]). Another structure with nice asymptotics but little indication of being practical is by Brodnik and Munro [11].

Here I survey other promising structures that have been subject to practical evaluation.

12.1 Golomb-compressed sequence

Putze, Sanders, and Singler describe a space-efficient Bloom filter alternative that adds random access to a sorted list encoded with an efficient, variable-length encoding [72, Section 4 (gcs)]. This involves keeping a table of bit-accurate pointers to where particular ranges of values start. Insertion and deletion is supported by adding to another table for recently added and deleted elements and periodically rebuilding the main table to reflect the changes.

This approach, and the paper generally, is geared toward the use of these data structures as summaries, not as visited sets. Their basis for evaluation is similar to that of the "summary cache" paper [28] or the "compressed

Bloom filter" paper [62]. Consequently, the authors do not account for the total memory footprint of the structure as it is being built, only the size required for transmission over a network. If we were to account for the metadata needed for random access, it is clear the structure is not asymptotically compact. To have $O(1)$ access times, we would need $\Theta(n)$ pointers into the data pool, which, like the compacted chaining hash table (Section 5.3), makes the space requirements diverge from the lower bound asymptotically. Even if we relax the access times somewhat, the situation does not improve much.

Perhaps more importantly, the design of this structure is not suited for the visited list paradigm. The structure is much simpler and faster if you commit to not adding any more elements. Unless a 100% false positive rate is acceptable, adding more elements is an essential part of the visited list usage paradigm (Definition 3.2). In fact, in many verification runs with partial-order reductions, there are more ADD operations than positive QUERY operations; thus, efficient adding is crucial. It is possible the design of the Golomb-compressed sequence could be tweaked to have better random access and perhaps better asymptotic memory requirements, but I suspect the Cleary table discipline of elements in fixed-size cells is the better solution for the visited set paradigm.

It is worth noting that one of the structures considered in the Putze, Sanders, and Singler paper is the Cleary table, though with a variation that improves access times by using a small relative pointer that, with high likelihood, enables jumping straight to the run for a home address. This is another design choice that is useful when the structure is used as a summary, but not as much when used as a visited set. (A Cleary table variant uses a related indexing structure. See Section 9.5.1.)

12.2 Cuckoo hashing

Pagh and Rodler's cuckoo hashing is an effective method of building a hash table with worst-case $O(1)$ query time [69, 70, 29]. Basically, hash functions determine a constant number of locations, originally just two, at which a value can be stored in the table. Querying is always $O(1)$ time because it is just a matter of checking each of the constant number of allowed locations. To add, an element is put in one of its allowed locations and, if necessary "bumps" an element already there to another of its allowed locations, recursively until the operation succeeds or an infinite loop is suspected. In a well-known generalization by Dietzfelbinger and Weidling, each location has a bucket that can hold up to some fixed number of elements [21].

It is known, but not often mentioned, that the structure can be made asymptotically compact [68]. This is done by using the location in the structure to encode part of the stored element, which Pagh calls *quotienting* [67], though the basic idea can be traced to Morris [64] and possibly others like Cleary [14].

Cuckoo hashing is another design that seems to be optimized for usage paradigms *unlike* the visited set usage paradigm. When we add to a cuckoo hashing table, we often spend a little time re-arranging elements until we get them in a configuration that satisfies the nice $O(1)$ worst-case query time. Because each negative QUERY for a visited set entails an ADD, there is limited potential to recoup that expense in ADD time with savings in QUERY time.

Also, the constant factors in the query time are not very good for the visited set usage paradigm. In particular, the time for a negative query remains constant through the life of the structure, and entails checking each allowed location of the value. (Using buckets can improve the situation slightly.) Contrast this to the Cleary table, in which most negative queries require the checking of just one bit, until the structure gets more than half full. Except in extreme or unusual cases, there will be roughly as many queries to

a visited set when the structure is 20%, 40%, or 60% full as there are when it is 80% or 90% full. Thus, in the visited set paradigm, the query times when the structure is nearly overflowing are not as important as the average query time over all occupancies. In fact, in a model checker, having faster operations at lower occupancies facilitates finding errors more quickly.

These conjectures about the relative performance of cuckoo hashing in the visited set usage paradigm are not interesting to me, however, unless we can make cuckoo hashing adaptive in the way I have done for Cleary tables. The use of multiple hash functions seems to be a large obstacle to throwing away specific information about each state while maintaining locality in the adaptation procedure. Consider the original cuckoo hashing structure, with two hash functions and a dedicated table for each hash function. To eliminate redundant storage of element home addresses (via quotienting), the hash functions need to be one-to-one (randomization functions). The inverse of each hash function is used to recover the original descriptor when "bumping" an entry. It seems quite simple to double the number of cells by cutting the size in half, but by storing only part of each hashed descriptor, we can no longer recover the original descriptor for bumping to another table with a different hash function.

Another possible strategy is to use one randomization function, but to use different parts of the randomized descriptor as the index in different tables. This strategy starts to cause trouble when there is not enough descriptor left to split into fully independent indices. As in Bloom filters, there is some kind of performance degradation associated with dependence in the hash function results. For cuckoo hashing, the degradation will be in insertion times and the occupancy at which the table is not able to add the next element. Observe that if there is no independence in the computed indices, it is equivalent to the corresponding indices being part of the same bucket, but without the locality. It would be worth investigating the effectiveness of this strategy, but unless a lot of re-arranging is allowed in adaptation, the de-

pendence among hash functions required for the cuckoo configuration with the smallest entries would need to be included from the start. On the face, the Cleary table's use of a single hash function seems better-suited for fast adaptation.

One attractive aspect of cuckoo hashing compared to the Cleary table is that it does not require any locking to be accessed in parallel, assuming each entry can fit in an atomically-accessible memory word.

12.3 Multilevel hashing

A related scheme that predates cuckoo hashing is Broder and Karlin's multilevel hashing [10][1]. It is different from cuckoo hashing in that there are typically $\Theta(\log \log n)$ tables, each with its own hash function, that are probed in a prescribed order, with earlier ones much larger than later ones. With each entry stored in the earliest table with the corresponding location available, negative queries can return upon encountering an empty cell. The original inception of the structure did not do any re-arranging of elements already added, but Mitzenmacher and Kirsch have demonstrated the value of such a cuckoo-like enhancement [57].

Like the Cleary table, multilevel hashing starts out with very fast queries, but can be accessed in parallel as a cuckoo hashing table can. Multilevel has the same complications for adaptation as cuckoo hashing, due to the need for multiple hash functions, plus another complication. Unlike cuckoo, multilevel hashing has the invariant that the locations for an entry on earlier

[1] While listening to Antti Valmari's invited talk at TACAS 2004, I independently invented a structure I came to call the "hash pyramid" which used $\Theta(\log n)$ levels of exponentially diminishing size and quotienting to store values very compactly. In my room at the conference, I hacked together a prototype implementation and found that it beat the compacted chaining structure Valmari described in his talk (left out of proceedings; journal version published [82]). Discussions about the structure with Valmari and Jaco Geldenhuys led them to draft a paper around my idea, but it was never accepted. We later discovered threads of related work, such as Broder and Karlin's, that made the hash pyramid not-so-novel. As with most things, it had been done before. But with a little luck, the seemingly obvious can be novel [24].

levels must be occupied. The invariant is crucial for good query times in software. Making room for more entries by shrinking the size of each cell, seems to require breaking the invariant temporarily. The first pass would be trivial, streaming adaptation, and the second pass would traverse all the elements in levels beyond the first and attempt to re-place them in the earliest available level. This would be intensive in random accesses and hash computation, so it would be much slower than Cleary table adaptation, even with the cheap hash-reusing among the levels (to support in-place adaptation, as described for cuckoo hashing).

In fact, it might be worth paying a small price in space to do multilevel adaptation by building a new structure from the old instead of adapting in place. This could be done in one pass and would not require early compromises on hash function independence. The required auxiliary space could be kept reasonably small by splitting the state space 2^i ways and using quotienting to pick a multilevel structure to add to. The memory available would be split $2^i + 1$ ways, so that a free space is available for adapting to when a partition gets too full. The space for the old structure for that partition then becomes the free space.

12.4 Summary

Other structures do not seem to have the properties that make Cleary tables well-suited for fast adaptive storage, at in the case of being accessed by a single thread.

Bibliography

[1] Susanne Albers. Competitive online algorithms. BRICS Lecture Series LS-96-2, University of Aarhus, BRICS, Department of Computer Science, 1996.

[2] O. Amble and Donald E. Knuth. Ordered hash tables. The Computer Journal, 17(2):135–142, 1974.

[3] Tonglaga Bao and Michael Jones. Time-efficient model checking with magnetic disk. In TACAS, volume 3440 of LNCS, pages 526–540. Springer, 2005.

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), pages 193–207. Springer-Verlag, 1999.

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

[6] Allan Borodin. Online Computation and Competitive Analysis. Cambridge University Press, Cambridge, 1998.

[7] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang. On the false-positive rate of Bloom filters. Submitted for publication, May 2004. http://cg.scs.carleton.ca/~morin/publications/ds/bloom-submitted.pdf.

[8] Dragan Bosnacki and Gerard J. Holzmann. Improving spin's partial-order reduction for breadth-first search. In 12th SPIN Workshop on Model Checking Software, volume 3639 of LNCS, pages 91–105. Springer, 2005.

[9] Andrei Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. In Proc. of the 40th Annual Allerton Conference on Communication, Control, and Computing, pages 636–646, 2002.

[10] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, SODA '90, pages 43–53, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[11] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. SIAM Journal of Computing, 28:1627–1640, May 1999.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. Information and Computation, 98(2):142–170, June 1992.

[13] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing, pages 59–65. ACM, 1978.

[14] John G. Cleary. Compact hash tables using bidirectional linear probing. IEEE Trans. Computers, 33(9):828–834, 1984.

[15] C.N. Ip and D.L. Dill. Better verification through symmetry. In Computer Hardware Description Languages and their Applications,

pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[16] Saar Cohen and Yossi Matias. Spectral bloom filters. In Proceedings of the 2003 ACM SIGMOD international conference on on Management of data, pages 241–252. ACM Press, 2003.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. McGraw-Hill Higher Education, 2001.

[18] Bernard Cousin and Jean-Michel Hélary. Performance improvement of state space exploration by regular and differential hashing functions. In 6th Internation Conference on Computer-Aided Verification, pages 364–376, 1994.

[19] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership. Technical Report arXiv:0803.3693v1 [cs.DS], arXiv.org, 2008.

[20] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In ICALP (1), volume 5125 of Lecture Notes in Computer Science, pages 385–396. Springer, 2008.

[21] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. Theoretical Compututer Science, 380:47–68, July 2007.

[22] Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Hacking and extending ACL2. In Ruben Gamboa, Jun Sawada, and John Cowles, editors, Seventh International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2007), 2007.

[23] Peter C. Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In Formal Methods in Computer-Aided Design (FMCAD) 2004, volume 3312 of LNCS. Springer, 2004.

[24] Peter C. Dillinger and Panagiotis Manolios. Fast *and* accurate bitstate verification for SPIN. In 11th SPIN Workshop on Model Checking Software, volume 2989 of LNCS. Springer, April 2004.

[25] Peter C. Dillinger and Panagiotis Manolios. Enhanced probabilistic verification with 3Spin and 3Murphi. In 12th SPIN Workshop on Model Checking Software, volume 3639 of LNCS. Springer, August 2005.

[26] Peter C. Dillinger and Panagiotis Manolios. Fast, all-purpose state storage. In 16th SPIN Workshop on Model Checking Software, volume 5578 of LNCS. Springer-Verlag, June 2009.

[27] Paul Erdős and Joel Spencer. Probabilistic Methods in Combinatorics. Academic Press, New York, 1974.

[28] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. IEEE/ACM Transactions on Networking, 8(3):281–293, 2000.

[29] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. Theory of Computing Systems, 38(2):229–248, 2005.

[30] Jaco Geldenhuys. State caching reconsidered. In 11th SPIN Workshop on Model Checking Software, volume 2989 of LNCS, pages 23–38. Springer, 2004.

[31] Jaco Geldenhuys and Antti Valmari. A nearly memory-optimal data structure for sets and mappings. In 11th SPIN Workshop on Model Checking Software, volume 2648 of LNCS, pages 136–150. Springer, 2003.

[32] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. Formal Methods in System Design, 7(3):227–241, 1995.

[33] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In Logic in Computer Science, pages 406–415, 1991.

[34] Gaston H. Gonnet. Handbook of Algorithms and Data Structures. Addison-Wesley, 1984.

[35] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification, Third Edition. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[36] Gerard J. Holzmann. On limits and possibilities of automated protocol analysis. In Protocol Specification, Testing and Verification VII, pages 339–344, 1987.

[37] Gerard J. Holzmann. An improved protocol reachability analysis technique. Software–Practice & Experience, 18(2):137–161, 1988.

[38] Gerard J. Holzmann. Algorithms for automated protocol validation. Technical Report 69:32-44, AT&T Technical Journal, 1990.

[39] Gerard J. Holzmann. Design and Validation of Computer Protocols. Prentice Hall, 1991.

[40] Gerard J. Holzmann. An analysis of bitstate hashing. In Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP, pages 301–314, Warsaw, Poland, 1995. Chapman & Hall.

[41] Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In Proceedings of the Third International SPIN Workshop, 1997.

[42] Gerard J. Holzmann. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Boston, Massachusetts, 2003.

[43] Gerard J. Holzmann and Dragan Bosnacki. The design of a multi-core extension of the spin model checker. IEEE Trans. Softw. Eng., 33(10):659–674, 2007.

[44] Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In 11th SPIN Workshop on Model Checking Software, volume 2989 of LNCS, pages 76–91. Springer, 2004.

[45] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Model driven code checking. Automated Software Engineering, 15(3-4):283–297, 2008.

[46] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification. In 23rd IEEE/ACM International Conference on Automated Software Engineering, pages 1–6, 2008.

[47] Gerard J. Holzmann and Doron Peled. Partial order reduction of the state space. In First SPIN Workshop, Montrèal, Quebec, 1995.

[48] Gerard J. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. International Journal on Software Tools for Technology Transfer (STTT), 2(3):270–278, 1999.

[49] Gerard J. Holzmann and Margaret H. Smith. Automating software feature verification. Bell Labs Technical Journal, 5(2):72–87, 2000.

[50] Gerard J. Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. IEEE Transactions on Software Engineering, 28(4):364–377, 2002.

[51] Bob Jenkins. Algorithm alley: Hash functions. Dr. Dobb's Journal, September 1997.

[52] Bob Jenkins. http://burtleburtle.net/bob/hash/index.html, 2007.

[53] Sndor Juhsz and kos Duds. Optimising large hash tables for lookup performance. In IADIS International Conference Informatics 2008, pages 107–114, 2008.

[54] Pim Kars. The application of promela and spin in the bos project. In Second SPIN Workshop, 1996.

[55] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In 14th European Symposium on Algorithms, volume 4168 of LNCS, pages 456–467. Springer, 2006.

[56] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. Random Structures and Algorithms, 33(2):187–218, 2008.

[57] Adam Kirsch and Michael Mitzenmacher. The power of one move: Hashing schemes for hardware. In INFOCOM, pages 106–110, 2008.

[58] Donald Ervin Knuth. The Art of Computer Programming, volume 3: Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., 2nd edition, 1997.

[59] Alfons Laarman, Jaco van de Pol, and Michael Weber. Boosting multi-core reachability performance with shared hash tables. In Formal Methods in Computer-Aided Design, 2010.

[60] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with spin. In Theoretical and Practical Aspects of SPIN Model Checking, volume 1680 of LNCS, pages 22–39. Springer, 1999.

[61] Wenbin Luo and Gregory L. Heileman. Improved exponential hashing. IEICE Electronics Express, 1(7):150–155, 2004.

[62] Michael Mitzenmacher. Compressed Bloom filters. In Proc. of the 20th Annual ACM Symposium on Principles of Distributed Computing, IEEE/ACM Trans. on Net., pages 144–150, 2001.

[63] Michael Mitzenmacher and Eli Upfal. Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York, NY, USA, 2005.

[64] Robert Morris. Scatter storage techniques. Commun. ACM, 11(1):38–44, 1968.

[65] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, pages 823–829. SIAM, 2005.

[66] Anna Pagh, Rasmus Pagh, and Milan Ruzic. Linear probing with constant independence. In STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, pages 318–327, New York, NY, USA, 2007. ACM.

[67] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. SIAM Journal of Computing, 31(2):353–363, 2001.

[68] Rasmus Pagh. On the cell probe complexity of membership and perfect hashing. In Symposium on Theory of Computing (STOC), pages 425–432, 2001.

[69] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In European Symposium on Algorithms (ESA), volume 2161 of Lecture Notes in Computer Science, pages 121–133. Springer, 2001.

[70] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. Journal of Algorithms, 51(2):122–144, 2004.

[71] Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci, and Marisa Venturini Zilli. Exploiting transition locality in automatic verification of finite-state concurrent systems. International Journal on Software Tools for Technology Transfer (STTT), 6(4):320–341, 2004.

[72] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash- and space-efficient bloom filters. In 6th International Workshop on Experimental Algorithms (WEA), volume 4525 of LNCS, pages 108–121. Springer, 2007.

[73] M. V. Ramakrishna. Practical performance of bloom filters and parallel free-text searching. Communications of the ACM, 32(10):1237–1239, 1989.

[74] Bradley J. Smith, Gregory L. Heileman, and Chaouki Abdallah. The exponential hash function. Journal of Experimental Algorithmics (JEA), 2, January 1997.

[75] Ulrich Stern. Algorithmic Techniques in Verification by Explicit State Enumeration. PhD thesis, Technical University of Munich, 1997.

[76] Ulrich Stern and David L. Dill. Automatic verification of the sci cache coherence protocol. In CHARME, volume 987 of LNCS, pages 21–34. Springer-Verlag, 1995.

[77] Ulrich Stern and David L. Dill. Improved probabilistic verification by hash compaction. In P.E. Camurati and H. Eveking, editors, Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95, volume 987 of LNCS, pages 206–224. Springer-Verlag, 1995.

[78] Ulrich Stern and David L. Dill. A new scheme for memory-efficient probabilistic verification. In IFIP TC6/WG6.1 Joint Int'l Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, pages 333–348, 1996.

[79] Ulrich Stern and David L. Dill. Parallelizing the murphi verifier. In Computer Aided Verification (CAV), volume 1254 of LNCS, pages 256–278. Springer, 1997.

[80] Ulrich Stern and David L. Dill. Parallelizing the murphi verifier. Formal Methods in System Design, 18(2):117–129, 2001.

[81] Antti Valmari. The state explosion problem. In Lectures on Petri Nets I: Basic Models, pages 429–528. Springer-Verlag, 1998.

[82] Antti Valmari. What the small Rubik's cube taught me about data structures, information theory and randomisation. International Journal on Software Tools for Technology Transfer (STTT), 8(3):180–194, 2006.

[83] Willem Visser. Memory efficient state storage in SPIN. In Proceedings of the 2nd SPIN Workshop, pages 21–35, 1996.

[84] P. Wolper and D. Leroy. Reliable hashing without collision detection. In 5th International Conference on Computer Aided Verification, pages 59–70, 1993.