Cache-Oblivious Dictionaries and Multimaps with Negligible Failure Probability

Michael T. Goodrich[1], Daniel S. Hirschberg[1], Michael Mitzenmacher[2], and Justin Thaler[2]

[1] Dept. of Computer Science, University of California, Irvine
[2] School of Engineering and Applied Sciences, Harvard University

Abstract. A *dictionary* (or *map*) is a key-value store that requires all keys be unique, and a *multimap* is a key-value store that allows for multiple values to be associated with the same key. We design hashing-based indexing schemes for dictionaries and multimaps that achieve worst-case optimal performance for lookups and updates, with minimal space overhead and *sub-polynomial* probability that the data structure will require a rehash operation. Our dictionary structure is designed for the Random Access Machine (RAM) model, while our multimap implementation is designed for the cache-oblivious external memory (I/O) model. The failure probabilities for our structures are sub-polynomial, which can be useful in cryptographic or data-intensive applications.

1 Introduction

A *dictionary* (or *map*) is a key-value store that requires all keys be unique, and a *multimap* [3] is a key-value store that allows for multiple values to be associated with the same key. Key-value associations are used in many applications, and hash-based dictionary schemes are well-studied in the literature (e.g., see [12]). Multimaps [3] are less studied, although a multimap can be viewed as a dynamic *inverted file* or *inverted index* (e.g., see Knuth [21]). Given a collection, Γ, of documents, an inverted file is an indexing strategy that allows one to list, for any word w, all the documents in Γ where w appears. Multimaps also provide a natural representation framework for adjacency lists of graphs, with nodes being keys and adjacent edges being values associated with a key. For other applications, please see Angelino *et al.* [3].

Such structures are ubiquitous in the "inner-loop" computations involved in various algorithmic applications. Thus, we are interested in implementations of these abstract data types (ADTs) that are based on hashing and use a near-optimal amount of storage – ideally $(1+\epsilon)n$ words of storage, where n is the number of items in the dictionary or multimap and $\epsilon > 0$ is some small constant. In addition, because such solutions are used in real-time applications, we are interested in implementations that are *de-amortized*, meaning that they have asymptotically optimal worst-case lookup and update complexities, but may have small probabilities of overflowing their memory spaces.

Crucially, we additionally focus on two further design goals. The first is that we aim for our data structures to succeed with *overwhelming probability*, i.e. probability $1 - 1/n^{\omega(1)}$, rather than merely with high probability, i.e. probability $1 - 1/\operatorname{poly}(n)$, achieved by most previous constructions. While our aim of achieving structures that provide worst-case constant time operations with overwhelming probability instead of with high probability may seem like a subtle improvement, there are many applications where it is essential. In particular, it is common in cryptographic applications to aim for negligible failure probabilities. For example, cuckoo structures with negligible failure probabilities have recently found applications in oblivious RAM simulations [17]. Moreover, a significant motivation for de-amortized cuckoo hashing is to prevent timing attacks and clocked adversaries from compromising a system [4]. An inverse polynomial failure probability may render a structure unsuitable for these applications.

In addition, guarantees that hold with overwhelming probability allow us to handle super-polynomially long sequences of updates, as long as the total number of items resident in the dictionary is bounded by n at all times. This is useful in long-running or data-intensive applications. It is also crucial for applications in which it is not possible to anticipate certain parameters, such as the length of the sequence of operations to be handled, at the time the data structure is deployed.

Our final design goal is relevant for our solutions that operate in the *external-memory* (I/O) model. Specifically, we would like our external-memory solutions to be *cache-oblivious* [15], meaning that they should achieve their performance bounds without being tuned for the parameters of the memory hierarchy, like the size, B, of disk blocks, or the size, M, of internal memory. The advantage of cache-oblivious solutions is that one such algorithm can comfortably scale across all levels of the memory hierarchy and can also be a better match for modern compilers that perform predictive memory fetches.

Previous Related Work. Since the introduction of the cache-oblivious framework by Frigo *et al.* [15], several cache-oblivious algorithms have subsequently been presented, including cache-oblivious B-trees [6], cache-oblivious binary search trees [8], and cache-oblivious sorting [9]. Pagh *et al.* [27] describe a scheme for cache-oblivious hashing, which is based on linear probing and achieves $O(1)$ expected-time performance for lookups and updates, but it does not achieve constant time bounds for any of these operations in the worst case.

As mentioned above, the multimap abstract data type is related to the inverted file and inverted index structures, which are well-known in text indexing applications (e.g., see Knuth [21]) and are also used in search engines (e.g., see Zobel and Moffat [32]). Cutting and Pedersen [13] describe an inverted file implementation that uses B-trees for the indexing structure and supports insertions, but doesn't support deletions efficiently. More recently, Luk and Lam [24] describe an internal-memory inverted file implementation based on hash tables with chaining, but their method also does not support fast item removals. Lester *et al.* [22,23] and Büttcher *et al.* [11] describe external-memory inverted file implementations that support item insertions only. Büttcher and Clarke [10] consider trade-offs for allowing for both item insertions and removals, and Guo *et al.* [18] give a solution for performing such operations by using a B-tree variant. Finally, Angelino *et al.* [3] describe an efficient external-memory data structure for the multimap abstract data type, but like the above-mentioned work on inverted files, their method is not cache-oblivious.

Much prior work on de-amortized dictionaries use variants of *cuckoo hash tables*, which were presented by Pagh and Rodler [26] and studied by a variety of other researchers. In their basic form, these structures use a freedom to place each key-value pair in one of two hash tables to achieve worst-case constant-time lookups and removals and amortized constant-time insertions with high probability. Kirsch, Mitzenmacher, and Wieder [20] introduced the notion of a stash for cuckoo hashing, which allows the failure probability to be reduced to $O(1/n^{\alpha})$, for any constant $\alpha > 0$, by using a constant-sized adjunct memory to store items that wouldn't otherwise be able to be placed. The failure probability of this solution is still inverse-polynomial, and insertions take $O(1)$ *amortized time* rather than worst-case time. Kirsch and Mitzenmacher [19] and Arbitman *et al.* [4] study a method for de-amortizing cuckoo hashing, which achieves constant-time lookups, insertions, and deletions with high probability, and uses space $(2+\epsilon)n$ for any constant $\epsilon > 0$ (as is standard in cuckoo hashing). In a subsequent paper, Arbitman *et al.* [5] study a hashing method that achieves worst-case constant-time lookups, insertions, and removals with high probability while maintaining loads very close to 1. They accomplish this by using a two-level hashing scheme, where the first level uses a simple bin-based hash table, and the second uses the de-amortized cuckoo hashing scheme of [4]. Our dictionary construction uses their first level, but replaces their second level with a different structure based on the Q-heaps of Fredman and Willard [14].

A lower bound of Andersson, Bro Miltersen, Riis, and Thorup is also relevant [1]. They prove that even for *static* dictionaries, query time $\Theta\left(\sqrt{\log n/\log\log n}\right)$ is necessary and sufficient on AC^0 RAMs, a restriction of the RAM model in which the only operations permitted in unit time are those computable by polynomial-sized constant-depth circuits. This applies even if $n^{\mathrm{polylog}(n)}$ space is permitted for the dictionary. We clarify that the lower bound only applies for substantially super-polynomial sized universes (though stronger lower bounds are known for RAMs whose instruction set is restricted further [30]). Our algorithms do not contradict the Andersson *et al.* result because we work in the standard RAM model rather than the AC^0 RAM model (we remark that, like much prior work on dictionary data structures, the only place we use non-AC^0 operations is in the evaluation of sufficiently random hash functions).

An earlier version of this paper [16] with similar goals presented a more intricate two-level dictionary data structure, where both levels were implemented as de-amortized cuckoo hash tables. The present paper substantially simplifies the earlier dictionary structure, while achieving a smaller failure probability.

Contributions. Our contributions are two-fold. Our first contribution is a dictionary structure achieving worst-case optimal lookup and update complexities with *sub-polynomial* failure probability, while incurring minimal space overhead. Specifically, our structure requires $(1+\epsilon)n$ space for an *arbitrary* constant $\epsilon > 0$. The lookup and update complexities of our structure are given in Table 1.

To the best of our knowledge, ours is the first structure suitable for the Random Access Machine (RAM) model that achieves all of these goals assuming random hash functions that can be evaluated in constant time. We also discuss several solutions that work with hash functions that are realizable (although impractical) in the standard RAM model, while achieving slightly sub-optimal lookup and update complexities. These solutions partially address an open question raised by Arbitman *et al.* [5].

Our second contribution is to develop a multimap implementation suitable for the *external-memory* (I/O) model. Prior work [3] achieved a solution in this model with worst-case optimal update and lookup complexity, but their solution was *cache-aware*, requiring knowledge of the size, B, of disk blocks.

Table 1. Performance bounds for our dictionary and multimap implementations, which all hold in the worst-case with overwhelming probability, assuming truly random hash functions. These bounds are asymptotically optimal. We use B to denote the block size, k to denote an arbitrary key, v to denote an arbitrary value, and n_k to denote the number of items with key equal to k.

Method	Dictionary I/O Performance	Multimap I/O Performance
add(k, v)	$O(1)$	$O(1)$
containsKey(k)	$O(1)$	$O(1)$
containsItem(k, v)	$O(1)$	$O(1)$
remove(k, v)	$O(1)$	$O(1)$
get(k)/getAll(k)	$O(1)$	$O(1 + n_k/B)$
removeAll(k)	–	$O(1)$

2 Dictionary Data Structure

Our dynamic dictionary data structure is designed for the standard RAM model. The instruction set available will be arithmetic, bitwise logical, and comparison operations on $b = \Omega(\log n)$ bit words.

Recall that a collection H of functions $h : U \rightarrow V$ is k-wise independent if for any distinct $x_1, \ldots, x_k \in U$ and for any $y_1, \ldots, y_k \in V$ it holds that $Pr[h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k] = 1/|V|^k$. Throughout this section, we assume the existence of an n^α-wise independent family of hash functions (for some constant $\alpha > 0$) mapping the universe U to the set $\{0, \ldots, n-1\}$, that can be evaluated in constant time using $o(n)$ space. We present results on the validity of this assumption in Section 3.

We mention that, with the exception of hash function evaluation, all of the pieces in our construction can be made to run in the AC^0 RAM model [2].

Our dictionary data structure combines two pieces. First, we modify a dictionary construction due to Willard to achieve a data structure with optimal worst-case update times and failure probability just $1/n^{\text{polylog}(n)}$. However, the resulting data structure uses $O(n)$ space, rather than $(1+\epsilon)n$ space. As our second step, we combine the result of Step 1 with the first level of the two-level hashing scheme of Arbitman *et al.* [5], to bring the space usage down to $(1 + \epsilon)n$ words for any constant $\epsilon > 0$.

2.1 The First Piece

Willard [31] describes a simple dictionary data structure using $O(n \,\text{polylog}(n))$ words of memory that supports worst-case constant time lookups and updates with failure probability $1/n^{\text{polylog}(n)}$. The primary contribution of this subsection is to give a variant of his structure that achieves the same guarantees using $O(n)$ words of space.

Willard's construction is based on a variant of Fredman and Willard's *Q-heap*. Using $O(n)$ space and preprocessing time, the Q-heap supports constant-time insertions, deletions, member, and predecessor queries into sets of size $O(\log^{1/5} n))$. By using a Q*-heap [31], which is essentially a B-tree whose internal nodes are implemented as Q-heaps, one can in fact achieve worst-case constant time insertions, deletions, and lookups for sets of size $O(\log^c n)$ for an arbitrary constant $c > 0$ [31, Lemma 2].

With the Q*-heap functionality in hand, Willard proposes the following simple dictionary structure with failure probability $1/n^{\log^k(n)}$ for any constant $k > 0$. Let h be a hash function chosen at random from hash family H. Consider a hash table with n buckets, each implemented as a Q*-heap with capacity $\log^{k+2}(n)$. As long as no bucket overflows its capacity, this hash table ensures that a bucket can be searched in $O(1)$ time, and that inserts and deletions can be processed in $O(1)$ time. Moreover, as long as the hash family H is $\log^{k+2}(n)$-wise independent, the probability that any particular bucket overflows is at most $1/n^{\log^{k+1}(n)}$. A union bound then implies that *no* bucket overflows with probability at least $1 - 1/n^{\log^k(n)}$.

As mentioned above, a major downside of Willard's construction is that it uses $O(n \operatorname{polylog}(n))$ space. We now show a modification that brings the space usage down to $O(n)$ machine words.

Instead of using an array of n buckets, use an array of $n/\log^k(n)$ buckets for some constant $k > 1$. Implement each bucket with a Q*-heap of capacity $6\log^k(n)$. Assuming truly random hash functions, a suitable Chernoff bound [25, Theorem 4.4, Part 3] implies that the probability any individual bucket overflows is at most $1/2^{6\log^k(n)}$. By a union bound, *no* bucket overflows with probability at least $1 - 1/2^{5\log^k(n)}$.

Notice each "bad event" (namely a bucket overflowing) in the above analysis involves the hash values of a set S of at most $6\log^k(n)$ items, and as long as H is a $6\log^k(n)$-wise independent hash family, the values of h on S are fully independent. Thus, the same analysis applies as long as H is a $6\log^k(n)$-wise independent hash family. Notice our modified construction requires $O(n)$ words of memory.

Lemma 1. *Let $k > 1$ be any constant. Assume there exists a $\log^k(n)$-wise independent hash family H such that each $h \in H$ can be evaluated in constant time using $o(n)$ words of memory. Then there exists a dynamic dictionary scheme $\mathcal{A}$ using $O(n)$ words of memory that supports insertions, deletions, and membership queries in $O(1)$ worst-case time, with failure probability $1/n^{5\log^{k-1}(n)}$.*

2.2 The Second Piece

Arbitman *et al.* [5] present a novel two-level hashing scheme, which they call Backyard Cuckoo Hashing. The first level of their scheme uses $m = (1+\epsilon/2)n/d$ "bins" of size d, where d is a suitably chosen constant that depends on ϵ. In the simplest version of their scheme, lookups, insertions, and deletions into each bin can trivially be implemented in constant time (which depends on ϵ), because each bin has constant size. However, a constant fraction of items inserted into the first level may "overflow", and must be handled separately by the second level of their scheme, which they instantiate as a de-amortized cuckoo hash table.

We briefly remark that Arbitman *et al.* [5] also present a more involved scheme based on de-amortized perfect hashing that works for slightly subconstant values of ϵ; this variant can also be adapted to our setting, but we omit these details for brevity.

Our intention is to use the first level of their scheme to absorb all but a small fraction of the items in our table. We use the dictionary data structure described in Section 2.1 to handle the overflowing items. Details follow.

Inspection of the proof of [5, Lemma 3.2] shows that their scheme achieves the following guarantee.

Lemma 2. *Let α be any constant $0 < \alpha < 1$. Assume there exists an n^{α}-wise independent hash family H such that each $h \in H$ can be evaluated in constant time using $o(n)$ words of memory. Then for any constant $\epsilon > 0$, there exists a data structure using space $(1+\epsilon/2)n$ satisfying the following guarantees. For some $\beta > 0$, with probability $1 - 2^{-n^{\beta}}$, all but $\epsilon n/16$ items are successfully inserted into the data structure. Moreover, insertions (both successful and unsuccessful) and deletions take worst-case $O(1)$ time, and membership queries succeed in $O(1)$ time for any item successfully placed in the data structure.*

Thus, with probability $1 - 2^{-n^{\beta}}$, during any sequence of n insertions, at most $t := \epsilon n/16$ items overflow from the primary structure, and we can place these items in the data structure $\mathcal{A}$ described in Section 2.1.

The remaining issue is that, in order to guarantee that the second level never contains more than $\epsilon n/16$ items, we must move an item from the second level to the first level whenever an item is deleted from the first level. This issue is also encountered by Arbitman *et al.*, who suggest multiple ways to address it. One solution is to associate with each first-level bin a doubly-linked list pointing to all overflowing items from the bin (the doubly-linked lists in total require at most $c'\epsilon n/16$ space for some universal constant c'). This way, whenever an item is deleted from a first-level bin, we can replace it in constant time with one of the items that previously overflowed from the bin.

We thereby ensure that for any sequence of $\text{poly}(n)$ insertions and deletions such that at most n items are actually stored in the data structure at any point in time, with probability $1 - 2^{-n^{\beta}}$ the second level never contains more than $\epsilon n/16$ items. Conditioned on this event, Lemma 1 implies that for any constant $k > 1$, the scheme $\mathcal{A}$ successfully supports worst-case constant time insertions, deletions, and lookups with probability $1 - 1/n^{5\log^{k-1}(n)}$ using space $c\epsilon n/16$ for some universal constant $c > 0$. Thus, our combined data structure supports worst-case constant time operations with failure probability $1/n^{5\log^{k-1}(n)} + 2^{-n^{\beta}} < 1/n^{\log^{k-1}(n)}$.

Setting $\epsilon = 16\epsilon'/(8+c+c')$, the two levels of our data structure use $(1+\epsilon/2)n + (c+c')\epsilon n/16 = (1+\epsilon')n$ words of memory in total. Combined with the above analysis, we obtain the following theorem.

Theorem 1. *Let α be any constant $0 < \alpha < 1$. Assume there exists an n^{α}-wise independent hash family H such that each $h \in H$ can be evaluated in constant time using space $o(n)$. For any constants $\epsilon' > 0$ and $k > 1$, there exists a data structure using $(1+\epsilon')n$ words of memory and supports insertions, deletions, and membership queries in worst-case $O(1)$ time with probability $1 - 1/n^{\log^{k-1}(n)}$.*

3 Hash Families

The results of the previous section require an n^{α}-wise independent hash family H such that each $h \in H$ can be evaluated in constant time using space $o(n)$. We feel this assumption is supported in practice, for instance, by the fact that one of the most widely-used hash functions, SHA-1, can be implemented in $O(1)$ time even in the AC^0 RAM model. The aim of this section is to provide a careful treatment of the theoretical foundations of this assumption.

Siegel's Hash Functions. Although there are no known constructions of n^α-wise independent hash functions achieving the above desiderata, Siegel achieved a close (albeit impractical) approximation in an influential paper [29]. Assume for the moment that the universe size $|U|$ is at most n^r for some constant $r > 0$, and that the desired range of the hash function is V, where $|V|$ is a power of two. Siegel's construction makes use of a bipartite graph G with constant left-degree d. The left vertex set of G corresponds to the universe U; and the right vertex set is $\{0, \ldots, n^\beta\}$ for some $0 < \beta < 1$. Each right vertex y is assigned a random value $M[y] \in V$ at initialization, and the hash value of element $x \in U$ is defined as $h(x) = \bigoplus_{(x,y) \in E(G)} M[y]$, where $\bigoplus$ denotes the bitwise XOR operation (if $|V|$ is not a power of two, we can replace $\bigoplus$ with any commutative group operation). Using a peeling argument, Siegel proves that as long as G has suitable *vertex expansion*, then the resulting hash family is n^α-wise independent for some $0 < \alpha < 1$.

Naively, storing the adjacency information of G would require space $\Omega(|U|) = \Omega(n^r)$, which is unacceptably large. To avoid this, Siegel uses a tensoring operation to turn a small expander graph (which can be stored explicitly) into larger expander. This approach increases the left degree and hence the evaluation time, but it remains constant as long as the universe size $|U|$ is polynomial in n.

From our usage standpoint, there are two potential sources of "failure" in Siegel's construction. The first is that there are currently no known explicit constructions of unbalanced vertex expanders that are sufficient to guarantee n^α-wise independence of Siegel's hash family. As a result, Siegel's hash family must either be *non-uniform*, with a suitable expander hardwired into the hashing algorithm, or the graph G must be generated at random. Siegel shows that for any constant $c > 0$, a suitable random graph will satisfy the requisite expansion properties with probability $1 - 1/n^c$.

The second source of failure comes into play if the universe size is super-polynomial in n. In this case, the universe should first be "mapped down" to a set U' of size n^r by a hash function h' from an almost-universal hash family H', before applying Siegel's construction to U'. For any set $T \subseteq U$ of size n^r, the resulting hash function will be fully independent on T, conditioned on the event that no elements in T collide under h', i.e., conditioned on the event that for all distinct elements $w, x \in T$, $h'(x) \neq h'(w)$. Unfortunately, any two elements in T will collide under h' with probability $1/n^r$.

In our applications, we cannot tolerate inverse-polynomial failure probabilities, and so neither source of failure is acceptable. Addressing these sources of failure was posed as an open question by Arbitman *et al.* [5]. In what follows, we give partial remedies to these sources of failure.

Obtaining Expanders. For polynomial-sized universes, Siegel's construction does yield *non-uniform* families of n^α-wise independent hash functions with $O(1)$ evaluation time and $o(n)$ space usage, by hardwiring in a suitable expander. However, if one requires a uniform algorithm, Siegel chooses the graph at random, generating a hash family H

that is n^α-wise independent only with probability $1 - 1/n^c$ (this is the probability that a randomly generated graph will satisfy the requisite expansion properties).

The most natural approach to eliminate this failure probability is to obtain *explicit constructions* of suitable expanders. Sadly, we do not resolve this question here: it remains an intriguing open question. Instead, we specify partial solutions to the problem.

Our first solution relies on the following observation: the probability a random graph G fails to satisfy the requisite expansion condition is dominated by the probability that *small sets* of vertices fail to satisfy the condition. Thus, one can obtain sub-polynomial failure probability by randomly generating a graph and exhaustively checking the vertex expansion of all sufficiently small sets. If a non-expanding set is found, the graph is rejected and a new graph is generated. This requires quasi-polynomial pre-processing time, but this may be acceptable in certain applications as the expensive phase need not happen online.

Theorem 2. *Assume the universe U has size n^r for some constant $r > 0$. Then for some α, for every pair of constants $k, r' > 0$, there is a set V of size $n^{r'}$ and a uniform algorithm $\mathcal{A}$ outputting a collection H of functions $h : U \to V$ achieving the following guarantees.*

1. *With probability $1 - 1/n^{\log^k(n)}$ over the internal coin tosses of $\mathcal{A}$, H is an n^α-wise independent family of hash functions.*
2. *$\mathcal{A}$ runs in $n^{\Theta\left(\log^{k+1}(n)\right)}$ time.*
3. *Any function $h \in H$ can be represented with $o(n)$ bits and evaluated in $O(1)$ time.*

We omit the proof because of space constraints; it can be found in the full version of the paper on the arXiv.

For polynomial-sized universes, we can instantiate the hash functions required in Section 2 using the algorithm of Theorem 2. This introduces an additional additive $1/n^{\log^k(n)}$ failure probability into our dictionary structure, which does not affect our results. We remark that the algorithm of Theorem 2 is implementable in the AC^0 RAM model, not just in the standard RAM model.

For polynomial-sized universes, a second partial solution is to avoid expensive preprocessing phase at the expense of slightly super-constant evaluation runtime. A first approach is to achieve this by simply increasing the degree of the randomly generated graph to $d(n)$, where d is some very slow-growing function of n. This reduces the probability that the graph fails to satisfy the requisite expansion properties to $\frac{1}{n^{\Omega(d(n))}}$. A problem with this approach is that the tensoring operation used by Siegel to blow up a small expander into a larger one will cause the larger graph to have degree $\omega(d(n))$.

A superior approach is described next.

Arbitrary Universe Sizes. We give a partial solution for achieving sub-polynomial failure probabilities with arbitrary universe sizes.

Theorem 3. *There exists some α, such that for every pair of constants $r, r' > 0$, and any function $k(n) = \omega(1)$, there is a set V of size $n^{r'}$ and a uniform algorithm $\mathcal{A}$ outputting a collection H of functions $h : U \to V$ achieving the following guarantees.*

1. *$\mathcal{A}$ runs in polynomial time.*

2. *Any function $h \in H$ can be represented with $o(n)$ bits and evaluated in time $\Theta(k(n))$.*
3. *For any set $T \subseteq U$ of size n^r, it holds that with probability $1 - 1/n^{k(n)}$ over the internal coin tosses of $\mathcal{A}$, the distribution $(h(x_1), \dots, h(x_{n^\alpha}))$ is uniform over V^{n^α} for any distinct $x_1, \dots, x_{n^\alpha} \in T$.*

Proof. Recall that in this setting Siegel gives a uniform algorithm $\mathcal{A}'$ generating a family of hash functions H such that, with high probability over the internal coin tosses of $\mathcal{A}'$, for any set $T \subseteq U$ of size n^r, H is fully independent on all subsets of T of size at most n^α. The algorithm $\mathcal{A}'$ works by first mapping the universe U down into a smaller universe U' of size n^r by a hash function h' from an universal hash family H', and then applying Siegel's expander-based hash function to U' using a randomly-generated expander G. As long as h' is one-to-one on T and G is an $(n, \epsilon, d, n^\alpha)$-weak expander (see the proof of Theorem 2), then the hash family H is fully independent on all subsets S of T of size at most n^α. However, both the universe-reduction step and the expander generation step introduce inverse-polynomial probabilities that the hash family H will *not* be fully independent on all such subsets S of T.

The idea to reduce the failure probability is to evaluate $k(n)$ independent instances of Siegel's hash function and XOR the results together (if $|V|$ is not a power of two, we replace XOR with any commutative group operation). For any set S at size at most n^α, as long as at least one of the k individual hash functions is fully independent on S, the result will be fully independent on S.

Formally, let $H_1, \dots, H_{k(n)}$ be hash families generated by $k(n)$ independent runs of Siegel's algorithm $\mathcal{A}'$. We define our final hash family H to be $\{h : h(x) = \bigoplus_{i=1}^{k(n)} h_i(x), (h_1, \dots, h_{k(n)}) \in H_1 \times \cdots \times H_{k(n)}\}$. Thus, a random element of h corresponds to randomly picking a hash function h_i from each hash family H_i, and XORing the results together.

Let $T \subseteq U$ be any set of size n^r. An easy calculation shows that with probability $1 - n^{\Omega(k(n))}$, the universe-reduction step is one-to-one on T for at least $k(n)/2$ runs of $\mathcal{A}'$. For each such run of $\mathcal{A}'$, with probability $1 - 1/n^c$, the expander-generation step successfully produces a graph with the requisite expansion properties for H_i to be n^α-wise independent on the "mapped down" universe U'.

Thus, with probability at least $1 - n^{\Omega(k(n))}$, it holds that for at least one run i of $\mathcal{A}'$, the hash family H_i is fully independent on all subsets of T of size at most n^α. That is, for any set $T \subseteq U$ of size n^r, with probability $1 - 1/n^{\Omega(k(n))}$, the distribution $(h_i(x_1), \dots, h_i(x_{n^\alpha}))$ is uniform over V^{n^α} for any distinct $x_1, \dots, x_{n^\alpha} \in T$. This is easily seen to imply that our final hash family H satisfies the same property for any distinct $x_1, \dots, x_{n^\alpha} \in T$. □

Theorem 3 can be used to obtain sub-polynomial failure probabilities for super-polynomially long sequences of data structure updates, as long as at most n items reside in the data structure at a time. Theorem 3 is applied on a step-by-step basis, where the set T at each step corresponds to the n items extant in the structure. For example, setting $k(n) = \Theta(\log \log n)$, we conclude that the data structure operation at any particular step can be performed in $O(\log \log n)$ time with probability $1/n^{\log \log n}$. If there are $n^{(1/2) \log \log n}$ steps, then all steps succeed in $O(1)$ worst-case time with probability $1 - 1/n^{\Omega(\log \log n)}$.

4 Cache-Oblivious Multimaps

In this section, we describe our cache-oblivious implementation of the multimap ADT. To illustrate the issues that arise in the construction, we first give a simple implementation for a RAM, and then give an improved (cache-oblivious) construction for the external memory model. Specifically, we describe an amortized cache-oblivious solution and then we describe how to de-amortize this solution.

In the implementation for the RAM model, we maintain two dictionary data structures, as described in Section 2. The first table enables fast containsItem(k, v) operations; this table stores all the (k, v) pairs using each entire key-value pair as the key, and the value associated with (k, v) is a pointer to v's entry in a linked list $L(k)$ containing all values associated with k in the multimap. The second table ensures fast containsKey(k), getAll(k), and removeAll(k) operations: this table stores all the unique keys k, as well as a pointer to the head of $L(k)$.

Operations in the RAM Implementation

1. containsKey(k): We perform a lookup for k in Table 2.
2. containsItem(k, v): We perform a lookup for (k, v) in Table 1.
3. add(k, v): We add (k, v) to Table 1 using the insertion procedure of Section 2. We perform a lookup for k in Table 2, and if k is not found we add k to Table 2. We then insert v as the head of the linked list corresponding to Table 2.
4. remove(k, v): We remove (k, v) from Table 1, and remove v from the linked list $L(k)$; if v was the head of $L(k)$, we also perform a lookup for k in Table 2 and update the pointer for k to point to the new head of $L(k)$ (if $L(k)$ is now empty, we remove k from Table 2.)
5. getAll(k): We perform a lookup for k in Table 2 and return the pointer to the head of $L(k)$.
6. removeAll(k): We lookup k in Table 2, and follow the pointer to $L(k)$. We walk through the linked list $L(k)$, and for each entry (k, v) of L_k, we remove (k, v) from Table 1. We also remove k from Table 2.

With the exception of the removeAll(k) operation, all operations above are performed in $O(1)$ time in the worst case with overwhelming probability by the results of Section 2. The removeAll(k) operation takes $O(1)$ *amortized* time with overwhelming probability, because each time we remove a pair (k, v) from Table 1, we can charge the operation to the corresponding insertion of the pair (k, v). We will explain how to de-amortize the removeAll(k) operation in Section 4.2

Two major issues arise in the above construction. First, the space-usage remains $O(n)$ only if we assume the existence of a garbage-collector for leaked memory, as well as a memory allocation mechanism, both of which must run in $O(1)$ time in the worst case. Without the memory allocation mechanism, inserting v into $L(k)$ cannot be done in $O(1)$ time, and without the garbage collector for leaked memory, space cannot be reused after remove and removeAll operations. Second, in order to extract the actual values from a getAll(k) operation, one must actually traverse the list $L(k)$. Since $L(k)$ may be spread all over memory, this suffers from poor locality.

We now present our cache-oblivious multimap implementation. Our implementation avoids the need for garbage collection, and circumvents the poor locality of the above getAll operation. We do require a cache-oblivious mechanism to allocate and deallocate power-of-two sized memory blocks with constant-factor space and I/O overhead; this assumption is justified by the results of Brodal *et al.* [7], who design a system for allocating and deallocating memory using constant time in the worst case, and sub-linear (indeed sub-polynomial, i.e. $o(n^\delta)$ for any $\delta > 0$) auxiliary storage.

Amortized Cache-Oblivious Multimaps. As in the RAM implementation, we keep two dictionary data structures. In Table 1, we store all the (k, v) pairs using each entire key-value pair as the key. With each such pair, we store a count, which identifies an ordinal number for this value v associated with this key, k, starting from 0. For example, if the keys were (4, Alice), (4, Bob), and (4, Eve), then (4, Alice) might be pair 0, (4, Bob) pair 1, and (4, Eve) pair 2, all for the key, 4.

In Table 2, we store all the unique keys. For each key, k, we store a pointer to the array, A_k, that stores all the key-value pairs having key k, stored in order by their ordinal values from Table 1. With the record for a key k, we also store n_k, the number of pairs having the key k, i.e., the number of key-value pairs in A_k. We assume that each A_k is maintained as an array that supports amortized $O(1)$-time element access and addition, while maintaining its size to be $O(n_k)$.

Operations

1. containsKey(k): We perform a lookup for k in Table 2.
2. containsItem(k, v): We perform a lookup for (k, v) in Table 1.
3. add(k, v): After ensuring that (k, v) is not already in the multimap by looking it up in Table 1, we look up k in Table 2, and add (k, v) at index n_k of the array A_k, if k is present in this table. If there is no key k in Table 2, then we allocate an array, A_k, of initial constant size. Then we add (k, v) to $A_k[0]$ and add key k to Table 2. In either case, we then add (k, v) to Table 1, giving it ordinal n_k, and increment the value of n_k associated with k in Table 2. This operation may additionally require the growth of A_k by a factor of two, which would then necessitate copying all elements to the new array location and updating the pointer for k in Table 2.
4. remove(k, v): We look up (k, v) in Table 1 and get its ordinal count, i. Then we remove (k, v) from Table 1, and we look up k in Table 2, to learn the value of n_k and get a pointer to A_k. If $n_k > 1$, we swap $(k', v') = A_k[n_k - 1]$ and $(k, v) = A_k[i]$, and then remove the last element of A_k. We update the ordinal value of (k', v') in Table 1 to now be i. We then decrement the value of n_k associated with k in Table 2. If this results in $n_k = 0$, we remove k from Table 2. This operation may additionally require the shrinkage of the array A_k by a factor of 2, so as to maintain the $O(n)$ space bound.
5. getAll(k): We look up k in Table 2, and then list the contents of the n_k elements stored at the array A_k indexed from this record.
6. removeAll(k): For all entries (k, v) of A_k, we remove (k, v) from Table 1. We also remove k from Table 2 and deallocate the space used for A_k.

In terms of I/O performance, containsKey(k) and containsItem(k, v) clearly require $O(1)$ I/Os in the worst case. getAll(k) operations use $O(1 + n_k/B)$ I/Os in the worst case, because scanning an array of size n_k uses $O(\lceil n_k/B \rceil)$ I/Os, even though we don't know the value of B. removeAll(k) utilizes $O(n_k)$ I/Os in the worst-case with overwhelming probability, but these can be charged to the insertions of the n_k values associated with k, for $O(1)$ amortized I/O cost. add(k, v) and remove(k, v) operations also require $O(1)$ amortized I/Os with overwhelming probability; the bound is amortized because there is a chance this operation will require a growth or shrinkage of the array A_k, which may require moving all (k, v) values associated with k and updating the corresponding pointers in Table 1.

In the next sections, we explain how to deamortize add(k, v), remove(k, v), and removeAll(k) operations.

4.1 De-amortizing Add(k, v) and Remove(k, v) Operations

To de-amortize the array operations, we use a rebuilding technique, which is standard in de-amortization methods (e.g., see [28]).

We consider the operations needed for insertions to an array; the methods for deletions are similar. The main idea is that we allocate arrays whose sizes are powers of 2. Whenever an array, A_k, becomes half full, we allocate an array, A'_k, of double the size and start copying elements A_k in A'_k. In particular, we maintain a crossover index, i_{A_k}, which indicates the place in A_k up to which we have copied its contents into A'_k. Each time we wish to access A_k during this build phase, we copy two elements of A_k into A'_k, picking up at position i_{A_k}, and updating the two corresponding pointers in Table 1. Then we perform the access of A_k, as would otherwise, except that if we wish access an index $i < i_{A_k}$, then we actually perform this access in A'_k. When we are done building A'_k, we deallocate the memory used for array A_k. Since we copy two elements of A_k for every access, we are certain to complete the building of A'_k prior to our needing to allocate a new, even larger array, even if all these accesses are insertions. Thus, each access of our array will now complete in worst-case $O(1)$ time with overwhelming probability. It immediately follows that add(k, v) and remove(k, v) operations run in $O(1)$ worst-case time.

4.2 De-amortizing removeAll(k) Operations

We describe two solutions for de-amortizing the removeAll(k) operation. The first is conceptually simpler but induces a constant-factor increase in memory usage; the second avoids using more memory than the de-amortized scheme above.

At a high level, in order to ensure removeAll(k, v) runs in worst-case $O(1)$ time, we simply remove k from Table 2 and deallocate the space used for the array A_k. We do *not* update the corresponding pointers of (k, v) pairs in Table 1 however; this leaves "spurious" pointers in Table 1, which we define to be (k, v) pairs satisfying the property that removeAll(k) has been called after the most recent insertion of (k, v). We need to explain how to modify all the other operations to deal with the presence of these spurious pointers.

First Solution. Our first solution is to maintain a global clock t, which is initialized to zero and is incremented after every operation. Assuming there are poly(n) total operations, the global time t can always be stored using $O(1)$ words of memory. Whenever we insert a key k into Table 2, or a key-value pair (k, v) into Table 1, we store with it the value t at the time of insertion. These timestamps increase the space usage of Tables 1 and 2 by a constant factor.

Whenever an operation invokes a lookup of a key-value (k, v) pair in Table 1 and finds that it is present, we have to check whether (k, v) is spurious. We do this by looking up key k in Table 2. If k is not found, then we know (k, v) is spurious, and we remove (k, v) from Table 1 and proceed as if (k, v) was not found in Table 1. If k is found in Table 2, we compare the timestamp t associated with k in Table 2 to the timestamp t' associated with (k, v) in Table 1. (k, v) is spurious if and only if $t' < t$; in the former case we remove (k, v) from Table 1 and proceed as if (k, v) was not found in Table 1; in the latter case we proceed as if (k, v) was found in Table 1.

The final issue we must deal with is ensuring that the presence of spurious key-value pairs does not cause the dictionary structure used to implement Table 1 to fail or to overflow its $(1 + \epsilon)n$ space bound. Recall that our dictionary structure consists of two levels, where the first level is implemented as an array of constant-sized "bins", and the second level is implemented as an array of Q^*-heaps, where each Q^*-heap can store $\log^{k+2}(n)$ items. Whenever we go to insert a (k, v) pair into this data structure, we first check whether any items in its first-level bin are spurious, and delete any spurious items from the first-level bin – this can be done in $O(1)$ time because the first-level bin has constant size. If there is room in the first-level bin after deleting spurious items, we insert the (k, v) pair into the bin and return. This ensures that spurious items residing in the first layer of our dictionary structure never affect the capacity of the structure. If we fail to insert the item (k, v) into the first layer of our structure, we attempt to place it into the second level of the data structure.

Dealing with spurious items in the second layer of our structure is more complicated: because a Q^*-heap may contain $\log^{k+2}(n)$ many items, we do not have time to iterate through all the items in the Q^*-heap to which (k, v) is assigned and check if any of the items are spurious. Instead, we increase the capacity of each Q^*-heap from $\log^{k+2}(n)$ items to $2\log^{k+2}(n)$ items. We also maintain with each Q^*-heap Q a doubly-linked list $L(Q)$ containing all items in Q; in addition we maintain for each item in Q a pointer to its entry in $L(Q)$. These pointers into $L(Q)$ are used so that, when an item is deleted from Q, we can also remove its entry from $L(Q)$ in constant time. Doubling the capacity of all the Q^*-heaps, as well as maintaining the lists $L(Q)$ and the pointers into $L(Q)$ causes the space usage of the second layer of our dictionary structure to increase by a constant factor, which can be absorbed into the parameter ϵ.

Recall that in the absence of spurious items, with all but sub-polynomial probability, no Q^*-heap should ever contain more than $\log^{k+2}(n)$ items. So when a Q^*-heap Q surpasses $\log^{k+2}(n)$ items, we know (with all but sub-polynomial probability) that this is due to the presence of spurious items. At this point, every time an item is inserted into Q, we take two items from the front of the doubly-linked list $L(Q)$ and check if they are spurious. Each time we find a spurious item, we delete it from Q and from the list $L(Q)$; otherwise we move the item to the end of the list $L(Q)$. This ensures that there are never more than $\log^{k+2}(n)$ spurious items in any Q^*-heap at any one time, so with all but sub-polynomial probability, no Q^*-heap will overflow its $2\log^{k+2}(n)$ capacity.

Second Solution. Our second solution differs from our first only in the manner in which we check whether a (k, v) pair is spurious. Specifically, we can avoid the use of timestamps. As before, whenever an operation invokes a lookup of a key-value (k, v) pair in Table 1 and finds that it is present, we have to check whether (k, v) is spurious. To accomplish this, we first lookup key k in Table 2. If k is not found, then we know (k, v) is spurious, and we remove (k, v) from Table 1 and proceed as if (k, v) was not found in Table 1. If k is found in Table 1, then we need to determine whether or not (k, v) is actually a member of the array A_k.

To determine this, let i be the count associated with pair (k, v) in Table 1. Recall i is supposed to represent (k, v)'s position in the array A_k if (k, v) is not spurious. We check if $i < n_k$ (recall n_k is stored with k in Table 2 and gives the number of pairs having the key k); if not we know (k, v) is spurious. If $i < n_k$, we check whether $A_k[i] = v$. This equality holds if and only if (k, v) is not spurious. Finally, it is straightforward to modify this solution to work in the case where we are in the process of moving items from an old array A_k to a new array A'_k as in the description of the de-amortized add(k, v) and remove(k, v) operations.

All time bounds in Table 1 follow.

5 Conclusion

In this paper, we have studied dictionary and multimap algorithms that support worst-case constant-time operations with sub-polynomial failure probability. Such structures should prove useful in cryptographic applications, as well as in long-running applications or those in which the duration of deployment is not known in advance. Our multimap solution is suitable for the cache-oblivious I/O model, and is to the best of our knowledge the first dynamic multimap achieving asymptotically optimal performance using linear space in this model.

Several interesting questions remain for future work. Are there (mildly) explicit constructions of unbalanced bipartite expanders sufficient to implement Siegel's hash family? Combined with our results, for polynomial sized universes this would yield an algorithm in the AC^0 RAM model for maintaining a dynamic dictionary with sub-polynomial failure probability, $(1 + \epsilon)n$ space, polynomial preprocessing time, and worst-case constant time operations. More ambitiously, we ask whether dictionaries supporting worst-case constant time operations with sub-polynomial failure probabilities can be achieved in the standard RAM model with quasipolynomial sized universes?

Acknowledgments. We are grateful to Jon Ullman and Mikkel Thorup for enlightening discussions. This research was supported in part by the U.S. National Science Foundation, under grants 0713046, 0830403, and 0847968, and by an Office of Naval Research: Multidisciplinary University Research Initiative (MURI) Award, number N00014-08-1-1015. Justin Thaler is supported in part by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program, and an NSF Graduate Research Fellowship. Michael Mitzenmacher was supported in part by the U.S. National Science Foundation, under grants 0964473, 0915922, 0721491, and 1011840.

References

1. Andersson, A., Miltersen, P.B., Riis, S., Thorup, M.: Static Dictionaries on AC^0 RAMs: Query Time $\Theta(\sqrt{\log n/\log\log n})$ is Necessary and Sufficient. In: Proc. of FOCS, pp. 441–450 (1996)
2. Andersson, A., Miltersen, P.B., Thorup, M.: Fusion trees can be implemented with AC^0 instructions only. Theoretical Computer Science 215(1-2), 337–344 (1999)
3. Angelino, E., Goodrich, M.T., Mitzenmacher, M., Thaler, J.: External-Memory Multimaps. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 384–394. Springer, Heidelberg (2011)
4. Arbitman, Y., Naor, M., Segev, G.: De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 107–118. Springer, Heidelberg (2009)
5. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: Proc. of FOCS, pp. 787–796 (2010)
6. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious b-trees. In: Proc. of FOCS, pp. 399–409 (2000)
7. Brodal, G.S., Demaine, E.D., Munro, I.: Fast allocation and deallocation with an improved buddy system. Acta Inf. 41, 273–291 (2005)
8. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache oblivious search trees via binary trees of small height. In: Proc. of SODA, pp. 39–48 (2002)
9. Brodal, G.S., Fagerberg, R., Vinther, K.: Engineering a cache-oblivious sorting algorithm. J. Exp. Algorithmics 12, 2.2:1–2.2:23 (2008)
10. Büttcher, S., Clarke, C.L.A.: Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In: Proc. of CIKM, pp. 317–318 (2005)
11. Büttcher, S., Clarke, C.L.A., Lushman, B.: Hybrid index maintenance for growing text collections. In: Proc. of SIGIR, pp. 356–363 (2006)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
13. Cutting, D., Pedersen, J.: Optimization for dynamic inverted index maintenance. In: Proc. of SIGIR, pp. 405–411 (1990)
14. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. System Sci. 47, 424–436 (1993)
15. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. of FOCS, pp. 285–298 (1999)
16. Goodrich, M.T., Hirschberg, D.S., Mitzenmacher, M., Thaler, J.: Fully de-amortized cuckoo hashing for cache-oblivious dictionaries and multimaps. CoRR, abs/1107.4378 (2011)
17. Goodrich, M.T., Mitzenmacher, M.: Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011)
18. Guo, R., Cheng, X., Xu, H., Wang, B.: Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In: Proc. of CIKM, pp. 751–760 (2007)
19. Kirsch, A., Mitzenmacher, M.: Using a queue to de-amortize cuckoo hashing in hardware. In: Proc. of 45th Allerton Conference, pp. 751–758 (2007)
20. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: cuckoo hashing with a stash. SIAM J. Comput. 39, 1543–1561 (2009)
21. Knuth, D.E.: Sorting and Searching. The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1973)

22. Lester, N., Moffat, A., Zobel, J.: Efficient online index construction for text databases. ACM Trans. Database Syst. 33, 19:1–19:33 (2008)
23. Lester, N., Zobel, J., Williams, H.: Efficient online index maintenance for contiguous inverted lists. Inf. Processing & Management 42(4), 916–933 (2006)
24. Luk, R.W., Lam, W.: Efficient in-memory extensible inverted file. Information Systems 32(5), 733–754 (2007)
25. Mitzenmacher, M., Upfal, E.: Probability and computing - randomized algorithms and probabilistic analysis. Cambridge University Press (2005)
26. Pagh, R., Rodler, F.: Cuckoo hashing. Journal of Algorithms 52, 122–144 (2004)
27. Pagh, R., Wei, Z., Yi, K., Zhang, Q.: Cache-oblivious hashing. In: Proc. of PODS, pp. 297–304 (2010)
28. Rao Kosaraju, S., Pop, M.: De-amortization of Algorithms. In: Hsu, W.-L., Kao, M.-Y. (eds.) COCOON 1998. LNCS, vol. 1449, pp. 4–14. Springer, Heidelberg (1998)
29. Siegel, A.: On universal classes of extremely random constant-time hash functions. SIAM J. Comput. 33(3), 505–543 (2004)
30. Thorup, M.: On AC^0 implementations of fusion trees and atomic heaps. In: Proc. of SODA, pp. 699–707 (2003)
31. Willard, D.E.: Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. SIAM J. Comput. 29, 1030–1049 (1999)
32. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. 38 (July 2006)