Threshold Data Structures and Coding Theory

Eric Bach
Marcos Kiwi

Technical Report #1380

July 1998

Threshold Data Structures and Coding Theory*

¡ Feliz Cumpleaños a Don Manuel !

Eric Bach† Marcos Kiwi‡

July 21, 1998

Abstract

Data structures for combinatorial objects are traditionally designed to handle objects up to a certain size. We introduce the idea of threshold data structures: representations that allow a richer collection of operations on small objects than large ones. As illustrations of the general concept we discuss threshold data structures for sets and multisets, and show how the former can be applied to cache memory design.

Consider threshold representations for subsets of a universe of size n, supporting insertions and deletions at any level, and enumeration of sets whose size does not exceed the threshold t. We derive lower bounds on the space used by any such representation. When t is fixed and $n \to \infty$ (the case of interest in memory design), any such representation must use, asymptotically, at least $(t+1)\log_2 n$ bits of memory. Applying the theory of error-correcting codes, we design a structure, efficiently supporting the required operations, whose space consumption matches the lower bound. Similar results are proved for multisets.

1 Introduction

In this work we study a data structure design problem that is at the heart of directory–based cache coherence protocols. Data structures have a long and honorable history within computer science. In this paper, we introduce the idea of *threshold data structures* and show how algebraic coding theory can play a role in its realization. Both the specific technique and the application of coding theory to cache coherence seem to be new.

This work was motivated by a problem that arises in the implementation of directory–based cache coherence protocols. Its larger context is a parallel programming environment in which processors keep local copies of shared memory. To read a block, a processor requests a copy and stores it in local memory. In other words it *caches* the block. The copy is explicitly returned when it is no longer needed. To handle a write request, the memory manager must invalidate all copies of the block. Failing to do so would cause other processors to maintain outdated copies of a block, giving rise to a data coherence problem. We henceforth refer to the latter problem as the *cache coherence problem.*

In order to address the cache coherence problem the memory manager must determine where invalid copies of the block are. Traditionally, this is not done, and the addresses of invalid blocks are simply broadcast to all processors. Broadcasting involves the following difficulties, however. First, a broadcast will increase network traffic and slow down processors that have no need of the information. Messages are also issued serially, so the time for a broadcast will be proportional to the number of processors, whereas we

*Presented at the International Conference on Theoretical Computer Science in Honour of Manuel Blum's 60th Birthday, City University of Hong Kong, April 24, 1998.

†Computer Sciences Dept., University of Wisconsin, Madison, WI 53706, bach@cs.wisc.edu. Gratefully acknowledges the support of NSF, via grant CCR-9510244, and the hospitality of U. de Chile.

‡Dept. de Ingeniería Matemática, Facultad de Ciencias Físicas y Matemáticas, U. de Chile, mkiwi@dim.uchile.cl. Partially supported by FONDECYT No. 1981182 and FONDAP in Applied Mathematics 1998.

would prefer a scalable solution using time bounded by a small power in the logarithm of the number of processors. Furthermore, latency (the time needed for the memory manager to determine the addresses of invalid blocks) is a crucial issue. Thus, one is interested in very fast solutions to the cache coherence problem that have simple and efficient hardware implementations.

Studies of sharing patterns indicate that there are essentially two cases: either a few processors share a block, or all of them do. (See, for example, [22].) For this reason, it is worthwhile to handle the first case quickly, if that can be done, before resorting to a broadcast. In addition, since we must ensure that extant copies of every block are identical, we wish to do this with a small amount of memory. To see why, note that the storage space needed to solve the cache coherence problem for b blocks will be a factor of b larger than the one needed to solve the same problem for a single block.

In this work we describe an elegant and efficient solution to the cache coherence problem that is simple to implement. We will do this, but in fact do more. Before summarizing our contributions let us begin by formally stating the problem motivating this work and how it relates to data structures.

1.1 Data Structures and the Cache Coherence Problem

Consider a piece of shared memory, i.e. a block, copies of which can be held by n processors. We focus on a single block. Copies are managed by a device, the memory manager, that ensures that at any given time all copies of the block held by the processors are identical. At various times, not under the memory manager's control, it is desired to know how many copies of the block are extant, and if that number does not exceed a threshold t, quickly identify which processors have copies. The set of labels of the processors will be denoted Ω. The goal is to provide a data structure that maintains a dynamic representation of the set of labels, denoted S, of processors that hold copies of the block. In addition, the data structure should be able to support the following three operations:

- Insert: If $\alpha \in \Omega \setminus S$, $\texttt{Insert}_\Omega(\alpha)$ updates the representation of S to $S \cup \{\alpha\}$.
- Delete: If $\alpha \in S$, $\texttt{Delete}_\Omega(\alpha)$ updates the representation of S to $S \setminus \{\alpha\}$.
- Enumerate: If $|S| \leq t$, $\texttt{Enumerate}_{\Omega,t}$ lists the elements of S, and if $|S| > t$, it returns NIL.

However, $\texttt{Insert}_\Omega(\alpha)$and $\texttt{Delete}_\Omega(\alpha)$ should be supported provided they are *feasible operations*, i.e. $\alpha \in \Omega \setminus S$ in the case of insertion and $\alpha \in S$ in the case of deletion. (Note that whether or not an operation is feasible depends solely on the history of operations performed and not on the state of the data structure.) We will refer to a data structure with the above described properties as a *threshold data structure for subsets of* Ω. To be considered a satisfactory solution to the cache coherence problem, any such data structure should have an efficient implementation with low memory overhead.

There is a naive solution to the above stated problem that is correct even when $t = n$. Associate to the block an n bit vector representing S whose coordinates are indexed by the labels assigned to processors. (In the literature, these bits are called *presence flags* [7].) Set the α-th bit of the vector only if $\alpha \in S$. Deletions and insertions can be easily implemented by complementing the α-th coordinate of the vector. To implement $\texttt{Enumerate}_{\Omega,n}$ simply list all the indices α for which the α-th bit of the vector is set. Clearly, the above data structure is very "simple". Moreover, insertions, deletions, and enumerations can be implemented efficiently. But, our problem arises in the design of memory systems. In this context, memory is limited. Empirical studies of directory protocols consider 32-byte blocks [23, Sect. 3.5]. On a system with 1024 processors, the naive solution incurs a 400% memory overhead! Hence, it is unsatisfactory.

For $t = 1$, there is an elegant solution. Indeed, let $\lg n$ denote the length of n in binary and assign to each processor an identifier that is a nonzero bit vector of length $\lg n$. The system keeps count of how many copies are checked out, plus a register containing the exclusive–or of all identifiers of processor holding the block. When there is only one such process, the register indicates the unique processor holding a copy. Thus, the overall memory usage is $2 \lg n$. This solution is due to David Wood, improving on an earlier method of Chandra and Palacharla. We will extend the latter ideas to values of t larger than 1.

1.2 Our Contributions

Our main technical contributions are a lower bound on the space required to approximately represent dynamic subsets of a space Ω (in a sense we will make precise) and a data structure that efficiently supports insertions, deletions, and enumeration of up to t elements, whose space consumption asymptotically matches the lower bound. Moreover, our data structure can be implemented efficiently and, when the threshold parameter t is a small constant, it has an efficient hardware level implementation. Indeed, the most expensive operation supported by our data structure is essentially equivalent to single decoding step in a Bose–Chaudhuri–Hocquenghem (BCH) t error correcting code [14, Ch. 3 & Ch. 9]. For such codes, specially designed efficient hardware level decoding procedures exist [14, Ch. 9,§6]. (For further properties of these codes we refer the reader to [14] and [21].) In summary, we provide a satisfactory solution to a "threshold" version of the cache coherence problem which is optimal in terms of memory usage.

In order to discuss the memory usage of our data structure as well as the cost of the operations it performs we henceforth denote by $\lg n$ the length in binary of a positive integer n. Note that $\lg n = \lfloor \log_2 n \rfloor + 1$ and $\lg n = \lceil \log_2(n+1) \rceil$ as well. When the size of Ω is n our data structure uses $(t+1)\lg n$ bits of memory. This implies, for $t = 4$ and $n = 1024$, that our data structure solves the cache coherence problem with a 20% memory overhead. This compares favorably to protocols considered in the literature, some of which are in some sense inferior[1] to our solution. For large values of t and n, the cost of insertions and deletions is $O(t(\log n)^{1+o(1)})$ bit operations and enumeration of up to t elements requires $O((t \log n)^{2+o(1)})$ bit operations. For small values of t and n, the first two operations cost $O(t(\log n)^2)$ bit operations while the latter requires $O((t \log n)^3)$ bit operations.

We also show that our threshold data structure for subsets of Ω has an interesting and potentially useful property. Namely, the set Ω need not be assumed to be fixed (provided some conditions are satisfied and that sufficient additional memory can be allocated to the data structure in the case that the size of Ω grows). The above stated property might be used to improve performance of our data structure. Moreover, it could prove useful in the design of a solution to the cache coherence problem that takes advantage of processors that get freed by jobs being executed in a parallel environment.

The design of our data structure heavily relies on algebraic coding theory. The use of coding theoretic techniques to address the cache coherence problem is novel and, to the best of our knowledge, the application of coding theory problems in cache coherence is new.

This work, in addition to its technical contributions, contains a conceptual contribution. Indeed, our solution to the problem that motivated this work rests in a realization of a new concept. Traditionally, a data structure for a combinatorial object must support a family of operations no matter what the size of the object. Our innovation is to introduce a *threshold* t and only support certain operations when the object's size does not exceed the threshold. For certain applications, this may be sufficient, and even desirable if economies of space and/or time usage thereby result. We call this new type of data structure a *threshold data structure*. Below we describe one such data structure; a *dynamic–set threshold data structure*.

Recall that many algorithms manipulate sets that can grow, shrink, and change over time. We will call these *dynamic sets*. Algorithms that manipulate dynamic–sets may require several types of operations to be performed on them. For example, insertions and/or deletions of elements, membership tests, etc. In a typical implementation of a dynamic–set it is assumed that the objects representing elements of the set have an identifying field called a *key*. Objects usually contain *satellite data*, i.e. information that is carried around in other object fields but are otherwise unused by the dynamic–set implementation. Typical operations on a dynamic–set S, where S is contained in some fixed set Ω, are:

- Membership test: given a key α returns an element of S whose key is α, or `NIL` if no such element belongs to S.
- Insert: given a pointer p augments the set S with an element pointed to by p.
- Delete: given a pointer p to an element in the set S, removes that element from S.
- Enumerate: a query that lists all the elements in S.

[1] E.g. some protocols avoid broadcasts by preventing more than t processors from sharing a block, others use software when more than t processors hold a copy of a block.

- Minimum (respectively Maximum): a query on a totally ordered set S that returns the element of S with the smallest (respectively largest) key.
- Predecessor (respectively Successor): a query that, given a pointer p to an element from a totally ordered set S, returns the next smaller (respectively larger) element of S, or `NIL` if no such element exists.

For a threshold t such that $t < |\Omega|$ we say that a dynamic–set operation on a set S is a *dynamic–set threshold operation* if it is an operation, different from insertion and deletion, that returns what is supposed to if $|S|$ is at most t, and returns `NIL` otherwise.

Definition 1 *A dynamic–set threshold data structure is a dynamic–set data structure that supports the insertion and deletion operations, and one or more dynamic–set threshold operation.*

In designing dynamic–set threshold data structures, one can think of the goal as that of designing a data structure for a set S that supports insertion and deletion operations and at least one other standard operation, but only doing these other operations at times when the set's size drops to t or less. If the size of S is greater than t, we might not even be able to support other standard operations besides insertion and deletion.

We expect that dynamic–set threshold data structures will find applications in domains where sets are too large to be represented in main or auxiliary memory. In this work we exhibit only one such application. But, there are several realizations of the concept of threshold data structure. In order to substantiate the latter claim we provide additional examples of threshold data structures. In particular, we build a data structure which maintains a representation of a dynamic multi–set and supports insertions, deletions, and enumeration of up to t elements. Its construction is based on some facts that could prove useful in the design of other threshold data structures.

The idea of threshold data structures is intriguing, and quite possibly our ideas will find other uses. In some sense, then, our technique is a solution looking for more problems.

1.3 Related Work

Chien and Frazer [11] applied BCH codes to solve an information retrieval problem. One is given a large set of documents, indexed by the presence of one or more features. Given a set S of features, one wishes to list all the documents whose feature sets T contain S. To solve this problem, Chien and Frazer define a *decipherable linear superimposed code* to be a collection of bit vectors representing feature subsets of cardinality $\leq t$, with the property that no two distinct linear combinations of size $\leq t$ are equal, and observe that the columns of a parity check matrix for a t-error correcting BCH code form such a code. When S and T have cardinality $\leq t$, deciding if $S \subset T$ is reduced to BCH decoding.

Such codes were further studied by Ericson and Levenshtein [12], who imposed the additional constraint that they should be resistant to transmission errors, and proved various bounds on their size. The goal of this work was to economically represent the set of active users in a token ring network.

In contrast to the works cited above, we want our set representations to still be useful when the set size exceeds t. Put another way, we do not restrict the set size, but partition the allowed operations into those that can be always applied and those that only work on "small" sets. For this reason, the previous work is not directly applicable. Our lower bounds also apply to *any* representation of a threshold set data structure, not just those based on linear superposition. Finally, no one seems to have considered multisets at all.

1.4 Organization

In Sect. 2 we describe a t-threshold data structure for subsets of Ω that uses $(t+1)\lg|\Omega|$ bits of memory. Furthermore, we discuss the cost of the operations it supports and additional properties of the data structure. In Sect. 3, we focus on small values of the threshold parameter t and discuss several issues that would be useful for achieving a good performance in an actual implementation of our data structure, and thence in a solution to the cache coherence problem. In Sect. 4, we show that our data structure is optimal in terms of memory usage. In Sect. 5, we illustrate the concept of threshold data structures with additional examples.

2 A Threshold Data Structure for Dynamic–sets

In this section we describe a threshold data structure relying on algorithms from the theory of error-correcting codes, in particular the algebraic decoding algorithm for binary BCH codes invented by Peterson [17] and refined by Berlekamp [3] and Massey [15].

We will now describe a t-threshold data structure for subsets of Ω. We henceforth identify Ω with a subset of $GF(2^m)^*$, the nonzero elements of the finite field of order 2^m. Each element of Ω is thus represented by a nonzero bit vector of length m. All sets under consideration will be subsets of the universe Ω. If there are n distinct elements we wish to keep track of, we must have $|\Omega| = 2^m - 1 \geq n$, so $m \geq \log_2(n+1)$.

We now describe how our data structure represents a dynamic–set S. Let $S_i = \sum_{\alpha \in S} \alpha^i$ be the i-th power sum of elements in S and C be a non–negative integer equal to $|S|$. The set $S \subseteq \Omega$ is represented by the vector

$$(S_1, S_3, \ldots, S_{2t-1}, C)\,.$$

Initially the vector is $(0, 0, \ldots, 0, 0)$ which represents the empty set. For $\alpha \in \Omega$, the operations are as follows:

1. $\texttt{Insert}_\Omega(\alpha)$: Increment C. Add the vector $(\alpha, \alpha^3, \ldots, \alpha^{2t-1})$ to $(S_1, \ldots, S_{2t-1})$, doing these operations in $GF(2^m)$.

2. $\texttt{Delete}_\Omega(\alpha)$: Decrement C. Add the vector $(\alpha, \alpha^3, \ldots, \alpha^{2t-1})$ to $(S_1, \ldots, S_{2t-1})$, doing these operations in $GF(2^m)$.

3. $\texttt{Enumerate}_{\Omega,t}$: If $C > t$ return NIL. If $0 < C \leq t$, for $j = 1, \ldots, C-1$, determine S_{2j} as S_j^2. Recover the elements of S from the power sums $S_1, S_2, \ldots, S_{2C-1}$.

Note that insertions and deletions are implementable using addition in $GF(2^m)$, which is just bitwise addition mod 2. The enumeration operation requires more discussion. First, when $x, y \in GF(2^m)$, we have $(x+y)^2 = x^2 + y^2$. Therefore, $S_{2j} = S_j^2$. Suppose that $|S| = d$, a value that can be read from the counter. Determining the α's from the power sums is exactly the problem that is solved when decoding binary BCH codes. Let us briefly outline how this is done. Using the Berlekamp–Massey algorithm [3, 15], we take the power sums $S_1, S_2, \ldots, S_{2d-1}$ and produce the coefficients of the polynomial

$$f(X) \;=\; \prod_{\alpha \in S} (1 - \alpha X)\,.$$

In coding theory, this polynomial is called the *error locator*. Suppose the elements of S are $\alpha_1, \ldots, \alpha_d$. We note that

$$f(X) \;=\; 1 - \sigma_1 X + \sigma_2 X^2 + \cdots + (-1)^d \sigma_d X^d\,,$$

where $\sigma_1, \ldots, \sigma_d$ are the elementary symmetric functions of $\alpha_1, \ldots, \alpha_d$. That is, we now have the coefficients of

$$g(X) \;=\; \prod_{\alpha \in S} (X - \alpha) \;=\; X^d - \sigma_1 X^{d-1} + \sigma_2 X^{d-2} + \cdots + (-1)^d \sigma_d\,.$$

By finding the roots of this polynomial, we can recover the α's.

We now discuss the space required by this method. Since $2^m - 1 \geq n$, the smallest value of m that can be used is $\lg n$. We also have $0 \leq C \leq n$, so that $\lg n$ bits suffice for storing the counter. These estimates give our first result.

Theorem 1 *Let $\Omega = GF(2^m)^*$. There is a t-threshold data structure for subsets of Ω that supports any feasible sequence of the operations* $\texttt{Insert}_\Omega$, $\texttt{Delete}_\Omega$, *and* $\texttt{Enumerate}_{\Omega,t}$ *using $(t+1)\lg|\Omega|$ bits of storage.*

Proof: Consider the data structure described above. Recall that since $GF(2^m)$ is of characteristic two, $x - y = x + y$ for all $x, y \in GF(2^m)$. From this it follows that the data structure correctly maintains a representation of $S \subseteq \Omega$ for any sequence of feasible operations (meaning that they could be performed on an exact representation of the set). The space consumption follows from the fact that to store each of the

t power sums requires $\lg |\Omega|$ bits of storage and to store the counter $\lg |\Omega|$ additional bits of memory are needed. ∎

Note that there are $2^{|\Omega|}$ possible subsets of Ω and that the data structure of the previous theorem has $|\Omega|^{t+1}$ distinct states. Thus, it is impossible, information theoretically, for the latter data structure to support $\texttt{Enumerate}_{\Omega,n}$. This is a feature of our threshold data structure that most standard dynamic-set data structures do not share.

We now discuss the asymptotic time complexity of our procedures. We assume that $GF(2^m)$ is implemented using polynomials over $GF(2)$ modulo an irreducible degree m polynomial. In this case, addition and subtraction require $O(m)$ bit operations, and multiplication and division require $O(m^{1+o(1)})$ bit operations.

Theorem 2 *Let $\Omega = GF(2^m)^*$ and $n = |\Omega|$. For $d, n \to \infty$, $\texttt{Insert}_\Omega$ and $\texttt{Delete}_\Omega$ can be implemented using $O(t(\log n)^{1+o(1)})$ bit operations. If $|S| = d \leq t$, $\texttt{Enumerate}_{\Omega,t}$ can be implemented using $O((d \log n)^{2+o(1)})$ bit operations.*

Proof: Consider insertions first. If $t > 1$, the required power sums can be found with t multiplications in $GF(2^m)$. The remaining steps are t additions in $GF(2^m)$ and the increment of a counter less than n. Since $m = O(\log n)$, the first bound follows. Clearly, deletions have the same cost.

Now consider enumerations. Note that $d = |S|$ is available from the counter C. We can obtain the required S_{2i} and run the Berlekamp–Massey algorithm using $O(d^2 m^2)$ bit operations [3, 15]. Finally, Shoup [19, p. 14] has proved that a degree d polynomial with all its zeroes in $GF(2^m)$ can be completely factored using $O((d \log n)^{2+o(1)})$ bit operations. Since $m = O(\log n)$, the result follows. ∎

Apropos of this result, some remarks are appropriate.

First, we are implicitly assuming that communication has been minimized in transmitting instructions such as $\texttt{Insert}(\alpha)$ to the data structure. If we are willing to transmit all the power sums, then insertions and deletions can be done using $O(t \log n)$ bit operations.

Second, the running time for enumerations can be improved if randomized algorithms are permitted. Let us sketch a proof of this. Blahut [5, p. 340] shows that the Berlekamp–Massey algorithm can be implemented with $O(t(\log t)^2 \log\log t)$ multiplications in $GF(2^m)$. Kaltofen and Shoup [13, Theorem 3] prove that a degree d polynomial f in $GF(2^m)[X]$ can be factored into its irreducible factors by a randomized (Las Vegas) algorithm using $O(md(d^{1+o(1)} + m^{0.67+o(1)}))$ bit operations. Therefore we find the expected number of bit operations for enumerations to be

$$O(d(\log n)(d^{1+o(1)} + (\log n)^{0.67+o(1)}))\,.$$

We emphasize that the randomization used here only affects the running time; when the algorithm finishes, the members of S are accurately determined.

Finally, the assumption that d and n are large may be unrealistic for applications. For this reason it is worthwhile to see what can be proved using standard arithmetic algorithms, which use $O(m^2)$ bit operations to multiply and divide elements of $GF(2^m)$ [2, Chap. 6]. With this assumptions, insertions and deletions cost $O(t(\log n)^2)$ bit operations. To implement enumerations we must use the Berlekamp–Massey algorithm, which costs $O(m^2 d^2)$ as before, and find the d roots of the degree d polynomial $g(X)$. This is done as follows. Let R denote the ring of polynomials modulo g. By the Chinese Remainder Theorem we have

$$R \;\cong\; GF(2^m) \oplus \cdots \oplus GF(2^m)\,.$$

We first find a matrix F for the $GF(2)$-linear map $a \mapsto a^2$ on R. This will use $O((md)^3)$ bit operations [2, Ex. 7.15]. If $Fa = 0$, we have

$$a \;=\; (a_1, \ldots, a_d)$$

with a_i either 0 or 1, and if $a \neq 0, 1$, then we can split g by computing $\gcd(a, g)$. We do this systematically as follows. First, use Gaussian elimination to find a basis $B = \{b_1, \ldots, b_r\}$ for the kernel of F. (Cost: $O((md)^3)$ bit operations.) Next, let $S = \{g\}$. Then, for $i = 1, \ldots, d$, we execute the following refinement step: For each $s \in S$, compute $\gcd(s, b_i)$, and replace s by its factors thus found, should they be nontrivial. We can see this will completely factor g by considering α and α', two distinct zeroes of g. Suppose at the end of the

procedure we have some $s \in S$ with $s(\alpha) = s(\alpha') = 0$. Then, each b_i is simultaneously zero or not zero at α, α', and B is not a basis. At all times, S contains a factorization of g. Therefore, the i-th refinement step uses

$$O(m^2) \sum_{s \in S} (\deg s)(\deg b_i) \;=\; O(m^2 d) \sum_{s \in S} (\deg s) \;=\; O(m^2 d^2)$$

bit operations. There are d refinements, so the total cost of enumerations is $O((d \log n)^3)$ bit operations.

We have established the following

Theorem 3 *Let $\Omega = GF(2^m)^*$ and $n = |\Omega|$. With standard arithmetic, the instructions* $\texttt{Insert}_\Omega$ *and* $\texttt{Delete}_\Omega$ *can be implemented using $O(t(\log n)^2)$ bit operations. If $|S| = d \leq t$,* $\texttt{Enumerate}_{\Omega,t}$ *can be implemented using $O((d \log n)^3)$ bit operations.*

We now discuss an interesting property of the data structure described at the beginning of this section. Namely, it is able to support dynamic expansions and contractions of the set of labels Ω. Below we will state formally what we mean by the latter claim. But, let us first describe our motivation for addressing such an issue. Parallel machines, by their very nature, may execute several different jobs at the same time. A fixed number of processors might be allocated to each job. Hence, while running a particular task other jobs might finish executing thus freeing processors. In this case, it is useful to be able to re–allocate the freed processors to jobs that are still executing. But, our proposed solution to the cache coherence problem seems to require that the set of labels of available processors be known from the beginning. We now show that this is not the case. Indeed, our data structure allows for a smooth transition from an underlying set of processors' labels Ω to a new set of labels Ω', provided $\Omega \subseteq \Omega'$ and that Ω' is sufficiently large. In other words our data structure supports the operation of expansion of the underlying set of labels. We stress that the latter can be achieved without significantly disrupting execution of the tasks being performed.

Formally, we have the following:

Lemma 1 *Let $\Omega = GF(2^m)^*$, $m' = km$ for some positive integer $k > 1$, and $\Omega' = GF(2^{m'})^*$. There is a threshold data structure for subsets of Ω that supports any feasible sequence of the operations* $\texttt{Insert}_\Omega$, $\texttt{Delete}_\Omega$, *and* $\texttt{Enumerate}_{\Omega,t}$. *In addition, the data structure supports the operation*

> $\texttt{Expand}_\Omega(\Omega')$: *Updates the data the structure to a threshold data structure for subsets of Ω' supporting any feasible sequence of the operations* $\texttt{Insert}_{\Omega'}$, $\texttt{Delete}_{\Omega'}$, *and* $\texttt{Enumerate}_{\Omega',t}$.

The cost of the latter operation is equivalent to the allocation of $(t+1)(\lg |\Omega'| - \lg |\Omega|)$ bits of memory.

Proof: The initial data structure is the one of Theorem 1. Since $m' = km$, then $GF(2^{m'})$ is a field extension of $GF(2^m)$. Hence, addition and multiplication in $GF(2^m)$ of elements in $GF(2^m)$ is consistent with addition and multiplication in $GF(2^{m'})$. It follows that the $\texttt{Insert}_\Omega(\alpha)$ and $\texttt{Insert}_{\Omega'}(\alpha)$ (respectively $\texttt{Delete}_\Omega(\alpha)$ and $\texttt{Delete}_{\Omega'}(\alpha)$) update the data structure in exactly the same way when $\alpha \in \Omega$.

The initial state of the new data structure after the $\texttt{Expand}_\Omega(\Omega')$ will be the last state taken by the initial data structure. Moreover, after an $\texttt{Expand}_\Omega(\Omega')$ operation, the arithmetic involved in $\texttt{Insert}_{\Omega'}$, $\texttt{Delete}_{\Omega'}$, and $\texttt{Enumerate}_{\Omega',t}$ should be done in $GF(2^{m'})$ and the counter should be allowed to reach values of up to $|\Omega'|$. This requires allocating $\lg |\Omega'| - \lg |\Omega|$ additional bits of memory for the counter and each of the power sums. ∎

Note that an implementation of an expansion operation does not cause an increase of communication overhead and is independent of the size of the set represented by the data structure. The reverse of the expansion operation is a contraction, denoted $\texttt{Contract}_{\Omega'}(\Omega)$ (where Ω and Ω' are as in Lemma 1). Provided this operation is feasible, i.e. the set represented by the data structure is fully contained in Ω, it can be supported by our data structure at the cost of freeing $\lg |\Omega'| - \lg |\Omega|$ bits of memory.

3 Solution to the Cache Coherence Problem

In this section we return to memory design and discuss some ideas that would be useful for this application.

We first observe that enumeration is only required after a write request. Since the complexity of enumeration grows with the size of the set enumerated, it is worth considering writes in detail. These will be of two types. If the processor α writes to a block it already holds, the others holding the block can be found by doing `Delete`(α), then `Enumerate`. A further `Insert`(α) restores the set to its original state. Writing to a "new" block, can be handled with `Enumerate` followed by `Insert`(α). Using this trick, t can be effectively increased by 1 at little additional cost.

Second, we note that we will only be interested in small values of t. For example, Weber and Gupta [22] suggest that in most sharing patterns, the number of readers is less than 4. For such cases, a feasible alternative to the general enumeration procedure is to directly compute the σ_i's and then find the roots of a small degree polynomial. This is important in BCH decoding, and many methods for this have been published. No one source, however, contains all the useful ideas, and so a brief review of the topic seems appropriate.

In the remainder of this section, we will assume that a power basis is used for $GF(2^m)$. That is, elements of this field are represented as the coefficients c_i of $\sum_{i=0}^{m-1} c_i\xi^i$, where ξ has degree m over $GF(2)$. On $GF(2^m)$, squaring is an invertible $GF(2)$-linear map, given in the power basis by a matrix we will denote by F.

Berlekamp, Rumsey, and Solomon [4] observed that square roots in $GF(2^m)$ can be found by applying F^{-1}. Their method for solving the quadratic equation

$$y^2 + y \;=\; \gamma\,,$$

whose left hand side is a $GF(2)$-linear function of y, goes as follows. If $y = \sum_{i=0}^{m-1} y_i\xi^i$ and $\gamma = \sum_{i=0}^{m-1} \gamma_i\xi^i$, the equation takes the form

$$\begin{pmatrix} U & 0 \\ * & 0 \end{pmatrix} \begin{pmatrix} y_{m-1} \\ \vdots \\ y_0 \end{pmatrix} \;=\; \begin{pmatrix} \gamma_{m-1} \\ \vdots \\ \gamma_0 \end{pmatrix},$$

where U is an $(m-1)\times(m-1)$ matrix of full rank. Then

$$\begin{pmatrix} y_{m-1} \\ \vdots \\ y_1 \end{pmatrix} \;=\; U^{-1} \begin{pmatrix} \gamma_{m-1} \\ \vdots \\ \gamma_1 \end{pmatrix}.$$

Taking $y_0 = 0, 1$ gives two values for y. (The equation has two solutions or none. We will only be concerned with the first case. For a solvability criterion, see [4, p. 555].)

Equations of the form

$$z^4 + az^2 + bz \;=\; c$$

are equivalent to

$$Mz \stackrel{\text{def}}{=} (F^2 + aF + bI)z \;=\; c$$

and solvable by Gaussian elimination [4, p. 562]. If matrices for F^2 and powers of ξ times F and I are precomputed, these can be added together to form M. This scheme uses about $2m^3$ bits of memory, but we can reduce this by spending more time. First, as discussed in [10, pp. 331], if $a \neq 0$, the substitution $z = \sqrt{a}w$ leads to the standard form

$$M'w = (F^2 + F + b'I)w \;=\; c'\,.$$

(If $a = 0$, the form is the same without the F.) This cuts the memory needed in half. We can reduce it further to $O(m^2)$ if we evaluate $b'I$ by nested multiplication; for example if K represents multiplication by ξ we have

$$(c_0 + c_1\xi + c_2\xi^2 + c_2\xi^3)I \;=\; c_0I + K(c_1I + K(c_2I + c_3K))\,.$$

The matrix K is sparse, so the work is roughly that of m matrix–vector multiplications.

With these preliminaries taken care of, we can now give our "formulary" for enumeration. In the theorems below, $\alpha_1, \alpha_2, \ldots, \alpha_d$ denote elements of $GF(2^m)$ and d the size of the set S which we wish to enumerate. We let $\sigma_1 = \sum_i \alpha_i$, $\sigma_2 = \sum_{i<j} \alpha_i \alpha_j$, ... denote their elementary symmetric functions, and $S_1 = \sum_i \alpha_i$, $S_2 = \sum_i \alpha_i^2$, ... their power sums.

Theorem 4 *(Enumeration for $d = 2$.) Let $\alpha_1 \neq \alpha_2$. If y is a solution to*

$$y^2 + y = S_3/S_1^3 + 1,$$

then (in some order)

$$\alpha_1 = y/S_1; \qquad \alpha_2 = \alpha_1 + S_1.$$

Proof: Follows from the discussion in [4]. We note that $S_1 \neq 0$. ∎

Our approach has been to reduce a cubic equation to an easily solved quartic. Alternatively, one can extract a square root and a cube root and then solve an auxiliary quadratic equation [8]. Cube root computation can be reduced to exponentiation; see [2, Thm. 7.3.2].

Theorem 5 *(Enumeration for $d = 3$.) Let $\alpha_1, \alpha_2, \alpha_3$ be distinct. Then $D = S_1^3 + S_3 \neq 0$. If $C = (S_1^5 + S_5)/D$, the equation*

$$y^4 + Cy^2 + Dy = 0$$

has distinct nonzero solutions y_1, y_2, y_3. We may take (in some order) $\alpha_i = y_i + S_1$.

Proof: Blokh [6, pp. 27] shows that $D \neq 0$ and $y^3 + Cy + D = 0$, from which the result follows. ∎

Theorem 6 *(Polynomial coefficients for $d = 4$.) Let $\alpha_1, \ldots, \alpha_4$ be distinct. Let*

$$\begin{aligned} \beta_3 &= S_3 + S_1^3, \\ \beta_5 &= S_5 + S_1^5, \\ \beta_7 &= S_7 + S_3^2 S_1, \\ D &= S_3\beta_3 + S_1\beta_5. \end{aligned}$$

Then $D \neq 0$, and

$$\begin{aligned} \sigma_1 &= S_1, \\ \sigma_2 &= \frac{1}{D}(S_3(\beta_5 + S_1^2\beta_3) + S_1(\beta_7 + S_1^4\beta_3)), \\ \sigma_3 &= \beta_3 + S_1\sigma_2, \\ \sigma_4 &= \frac{1}{D}(\beta_5(\beta_5 + S_1^2\beta_3) + \beta_3(\beta_7 + S_1^4\beta_3)). \end{aligned}$$

Proof: From Newton's identities we have

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ S_2 & S_1 & 1 & 0 \\ S_4 & S_3 & S_2 & S_1 \\ S_6 & S_5 & S_4 & S_3 \end{pmatrix} \begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \end{pmatrix} = \begin{pmatrix} S_1 \\ S_3 \\ S_5 \\ S_7 \end{pmatrix}.$$

The matrix has determinant $D = \prod_{i<j}(\alpha_i - \alpha_j)$ [17, p. 462], which cannot vanish. Solving for the σ_i by Gaussian elimination gives the result. ∎

d	$v * w$	$v \div w$	Mv	$M\backslash v$	Memory
2	2	2	1	0	$\sim m^2$
3	3	1	0	1	$\sim 2m^3$
4	17	5	1	1	$\sim 2m^3$

Table 1: Cost of enumerations for small d. Here, v and w are m-dimensional bit vectors (i.e. elements of $GF(2^m)$, M is a $m \times m$ matrix, and we have borrowed the notation $M\backslash v$ from MATLAB® to stand for solution of the linear system $Mx = v$.

Theorem 7 *(Root finding for $d = 4$.) Let the zeroes of*

$$\prod_{i=1}^{4}(x - \alpha_i) = x^4 + \sigma_1 x^3 + \sigma_2 x^2 + \sigma_3 x + \sigma_4$$

be distinct. If $\sigma_1 \neq 0$, Let D be as in Theorem 6 and define σ_i' for $i = 2, 3, 4$ by

$$\begin{aligned} \delta &= \sqrt{\sigma_3/\sigma_1}, \\ \sigma_4' &= S_1^2/D, \\ \sigma_3' &= S_1^3/D, \\ \sigma_2' &= (\sigma_2 + \sigma_1\delta)\sigma_4'. \end{aligned}$$

If $\sigma_1 = 0$, let $\sigma_i' = \sigma_i$. The equation

$$z^4 + \sigma_2' z^2 + \sigma_3' z = \sigma_4',$$

has distinct solutions $z_1, \ldots, z_4$, and we may take (in some order)

$$\alpha_i = \begin{cases} z_i^{-1} + \delta, & \text{if } \sigma_i \neq 0; \\ z_i, & \text{otherwise;} \end{cases}$$

for $i = 1, 2, 3$ and $\alpha_4 = S_1 + \alpha_1 + \alpha_2 + \alpha_3$.

Proof: Substituting $x = y + \delta$ in the original equation brings it to the form

$$y^4 + \sigma_1 y^3 + (\sigma_2 + \sigma_1)y^2 + (\delta^4 + \sigma_2\delta^2 + \sigma_4) = 0.$$

Its constant term must be nonzero, otherwise it would have a double root. The further substitution $z = 1/y$ and some algebra gives the result. ∎

The main idea of this method is already in [4], but the simple formulas for σ_i' do not seem to be in the literature. It is also possible to solve a quartic equation by extracting two square roots and solving one auxiliary cubic and three auxiliary quartic equations [8].

Assuming that matrices for F^{-1}, F^2, and powers of ξ times F and I have been precomputed, Table 1 summarizes the complexity of the above described methods.

There are specific methods for larger d, but we limit ourselves to a few remarks. In general, the polynomial coefficients σ_i are rational functions of the power sums S_i. These functions are given explicitly for $d \leq 6$ in [16, pp. 144–147].

For degree 5 equations, Trager and Winograd [20] have given an elegant method, relying on inversion of an $m \times m$ matrix, for finding their roots.

Finally, we consider the hardware performance of our methods. A detailed engineering analysis of this matter is beyond the scope of this paper, but one can get an idea of what is possible by considering the recent literature on decoders for error–correcting codes. In recent years, code engineers have concentrated on Reed–Solomon codes rather than the binary BCH codes favored in the 1960's. Fortunately, our enumeration

procedure can make use of the components provided in modern Reed–Solomon decoders. Roughly speaking, an enumeration operation can be done in about half the time used to decode one Reed–Solomon codeword.

Let us briefly recall Reed–Solomon codes. A codeword consists of $n = 2^m - 1$ symbols from $GF(2^m)$, thought of as a polynomial

$$C(X) = c_0 + c_1 X + \cdots + c_{n-1} X^{n-1} .$$

Codewords satisfy $C(\alpha^i) = 0$, for $i = 1, \ldots, 2t$, where α generates the cyclic group $GF(2^m)^*$. The decoder gets the polynomial $R(X) = C(X) + E(X)$, where

$$E(X) = e_0 + e_1 X + \cdots + e_{n-1} X^{n-1}$$

is the error contributed by the channel. Using R, the decoder finds the syndromes

$$S_k = R(\alpha^k) = E(\alpha^k) = e_0 + e_1 \alpha^k + \cdots + e_{n-1} \alpha^{k(n-1)}$$

for $k = 1, \ldots, 2t$. From these, error locations are determined (places where $e_i \neq 0$) and error magnitudes (the values of nonzero e_i), allowing up to t transmission errors to be corrected.

The connection to our enumeration procedure is as follows. In our case, e_i is either 0 or 1. Therefore, we already have half of the syndromes and can find the others by squaring, using t multiplications in $GF(2^m)$. It is evident that we need only to determine the error locations. To do this, decoders use the Berlekamp–Massey algorithm (or a variant thereof), followed by a Chien search for the roots of a polynomial. This search method uses special hardware to reduce the time per root to one clock cycle.

Chi [9, pp. 951] analyzes the performance of several popular Reed–Solomon decoders. He assumes that the decoder has $2t$ parallel units, each capable of one multiplication or division, plus one addition, in $GF(2^m)$ per clock cycle. Taking into account that we can obtain the syndromes with one clock cycle and that we don't need the error values, we find that enumeration can be done within $6t + n + 1$ cycles. (The time for full decoding is about double this.) More decoder designs can be found in [18] and the references therein.

With specialized hardware, it should be possible to perform an enumerate operation in a few microseconds when $n = 10^2 - 10^3$. Indeed, commercial Reed-Solomon decoders, which are programmed for differing code sizes and hence do not have the best possible performance, achieve latencies around 20 μsec when $n = 255$ [1].

4 A Lower Bound on Memory Usage

The data structure for subsets of Ω discussed in Sect. 2 requires about $(t+1)\log_2 |\Omega|$ bits of memory. In this section, we will show this is optimal up to low order terms, when t is fixed and $|\Omega| \to \infty$.

Let Ω be a universe of size n. We would like to consider a finite state machine M that models a t-threshold data structure for subsets of Ω. Intuitively, each state of M should stand for one or more subsets, and the input alphabet of M should consist of instructions to insert or delete various elements of Ω. The action of M should be compatible with the intended interpretation of machine states as subsets.

Unfortunately, the formalism of finite automata is cumbersome to apply here, and we are better off working directly with a mapping from 2^Ω to the state space Q of M. Our goal is to prove a lower bound on the size of Q. In order to achieve this, recall that if S is the set whose representation is maintained by the data structure, then $\texttt{Insert}_\Omega(\alpha)$ (respectively $\texttt{Delete}_\Omega(\alpha)$) is a feasible operation provided that $\alpha \in \Omega \setminus S$ (respectively $\alpha \in S$). Thus, whether or not an operation is feasible depends solely on the history of operations performed and not on the state of the data structure. Consider the mapping q from the set of feasible operation sequences into Q. Note that different feasible operations sequences might give rise to the same subset S of Ω, but leave the finite state machine M in different states. We abuse notation, and view q as a mapping from 2^Ω to Q, where $q(S)$ corresponds to a state of Q reached by M when its input is a feasible operations sequence that gives rise to S (many such sequences exist, so in order to define $q(S)$ pick one arbitrarily). We first require q to satisfy two conditions that say, in effect, that the state transition function of M is well defined.

(1) Let $\alpha \notin S \cup T$. If $q(S) = q(T)$, then $q(S \cup \{\alpha\}) = q(T \cup \{\alpha\})$.

(2) Let $\alpha \in S \cap T$. If $q(S) = q(T)$, then $q(S \setminus \{\alpha\}) = q(T \setminus \{\alpha\})$.

Lemma 2 *Let q satisfy conditions (1) and (2) above and let S and T be subsets of Ω. If $q(S) \neq q(T)$, and σ is a feasible operations sequence for both sets, then $q(S_\sigma) \neq q(T_\sigma)$, where S_σ (respectively T_σ) is the set that the sequence of operations σ gives rise too when performed starting from S (respectively T).*

Proof: Use induction on the length of σ. ∎

So far, our requirements could be satisfied by a trivial mapping (one with $|Q| = 1$). To get a lower bound, we add a third requirement. This says, intuitively, that each set of size at most t is represented within M by a distinct state, which is also distinct from any state representing a larger set, i.e.,

(3) If $|S| \leq t$ and $T \neq S$, then $q(S) \neq q(T)$.

Lemma 3 *Let $q : 2^\Omega \to Q$ satisfy (1)–(3) above. If S and T are distinct subsets of Ω such that $|S \cap T| \geq \min\{|S|, |T|\} - t$, then $q(S) \neq q(T)$.*

Proof: The conclusion is true by assumption if $|S|, |T| \leq t$, so assume that $|S| \leq |T|$ and $|S \cap T| \geq |S| - t$. If $q(S) = q(T)$, we can delete all elements in the (non-empty) intersection and obtain sets S' and T', respectively. By Lemma 2, we must have $q(S') = q(T')$. But

$$|S'| \;=\; |S| - |S \cap T| \leq t\,,$$

so requirement (3) forces $q(S') \neq q(T')$. This is a contradiction. ∎

The following result gives a lower bound on the number of states used by any t-threshold data structure for subsets of Ω.

Theorem 8 *Let $|\Omega| = n$. If $q : 2^\Omega \to Q$ satisfies (1)–(3) above, then*

$$|Q| \;\geq\; \sum_{i=0}^{t+1} \binom{n}{i}\,.$$

Proof: Let us denote $\{1, \ldots, n\}$ by $[n]$. Identify Ω with $[n]$. We define a family C of subsets of $[n]$, any pair of which satisfies the hypothesis of Lemma 3. Let

$$C_0 \;=\; \{S \subseteq [n] : |S| \leq t\}\,,$$

and for $1 \leq i \leq n - t$ define

$$C_i \;=\; \{S \cup [i] : S \subseteq [i]^c, |S| = t\}\,.$$

Then $C = \bigcup_{i=0}^{n-t} C_i$. The size of C is

$$\begin{aligned}
\sum_{i=0}^{t} \binom{n}{i} + \sum_{i=1}^{n-t} \binom{n-i}{t} \;&=\; \sum_{i=0}^{t} \binom{n}{i} + \sum_{i=t}^{n-1} \binom{i}{t} \\
&=\; \sum_{i=0}^{t} \binom{n}{i} + \binom{n}{t+1} \\
&=\; \sum_{i=0}^{t+1} \binom{n}{i}\,.
\end{aligned}$$

∎

When t is fixed and $n \to \infty$, the bound of Theorem 8 is asymptotic to $n^{t+1}/(t+1)!$. Hence, the number of memory bits used by the method of Sect. 2, namely $(t+1)\log_2 n$, is asymptotically optimal. When $t = n/2$, Theorem 8 tells us that we cannot expect much gain over explicitly listing all states, since $\log_2 \binom{n}{n/2} \sim n$.

Finally, when $t = 0$, the bound becomes $n + 1$, telling us that the simple method of detecting empty sets by means of a counter is optimal.

We note that the bound of Theorem 8 is not best possible in all cases. Take, for example, $n = 3$ and $t = 1$. We start with

$$C_0 = \{\emptyset, \{1\}, \{2\}, \{3\}\} .$$

We get C_1 by adding 1 to each singleton not containing 1, so

$$C_1 = \{\{1,2\}, \{1,3\}\} .$$

In a similar fashion,

$$C_2 = \{\{1,2,3\}\} .$$

Thus, C consists of all subsets of $\{1,2,3\}$ save $\{2,3\}$, and the lower bound is 7. However, $\{2,3\}$ must be distinguished from every set of size at most 1 (by requirement 1), from any set of size 2 (since there is always a common intersection), and from any set of size 3 (since we can delete 2 and 3). Hence in this case, 8 states are required.

5 Extensions

As mentioned in the introduction to this work, we believe that threshold data structures could prove useful not only in the design of memory systems, but also when there is too much data to be stored in main or auxiliary memory. Thus, in our opinion, identifying new threshold data structures and providing additional techniques for constructing them is a worthwhile endeavor. In this section we undertake it. Moreover, the results we describe below establish that the concept of threshold data structure has several realizations besides the one described in Sect. 2.

We first observe that a threshold data structure for subsets of Ω can support additional dynamic–set operations, like membership test, by combining it with standard data structures. Indeed, consider a universe Ω' much larger than Ω and that we wish to maintain a representation of a dynamic subset S of Ω' whose size is small compared to Ω'. In addition, we wish to efficiently perform insertions, deletions, enumerations of up to t elements, and membership tests. Note that the latter can be achieved by hashing the elements of Ω' into a hash table indexed by the elements of Ω where collision resolution is resolved through chaining[2] and using our data structure for subsets of Ω to keep track of the hash table entries storing elements of S. Such data structure can support the operations of deletion, insertion, and membership test in the standard way. Moreover, enumeration of sets of size up to t can be performed efficiently by first listing the entries of the hash table to which the elements of S hash and then enumerating the elements of S associated to those entries.

We will now construct a threshold data structure that maintains a dynamic representation of a multi–set, i.e. of a "set with repeated elements." Formally, consider a universe Ω and a threshold parameter t. As usual we will identify the elements of Ω with the elements of $GF(2^m)^*$ for some sufficiently large m. We will tackle the problem of designing a data structure similar, in the sense that it supports the same type of operations, to a threshold data structure for subsets of Ω, but which maintains a dynamic representation of a multi–subset of Ω instead of a subset of Ω. We will call such data structure a *threshold data structure for multi–subsets of* Ω. The formal definition of the dynamic–multi-set operations of insertion, deletion, and enumeration of up to t elements are the natural generalizations of those provided in Sect. 1.1 for the analogous operations over sets. We abuse notation and also refer to the multi–set operations by $\texttt{Insert}_\Omega$, $\texttt{Delete}_\Omega$, and $\texttt{Enumerate}_{\Omega,t}$.

The construction of our new data structure arises from the following observation. In the dynamic–set threshold data structure described in Sect. 2, it is possible to avoid the Berlekamp–Massey algorithm by

[2] In other words, elements that hash to the same value are placed on a linked list.

spending a little more work in the $\texttt{Insert}_\Omega$ and $\texttt{Delete}_\Omega$ procedures. Indeed, we could work with the polynomials directly and store

$$f(X) \;=\; \prod_{\alpha\in S}(X-\alpha) \bmod X^t,$$

together with a counter C. Then, $\texttt{Insert}_\Omega(\alpha)$ could be implemented by taking the remainder mod X^t of the product of $f(X)$ by $X-\alpha$, and incrementing the counter C. To support $\texttt{Delete}_\Omega(\alpha)$, we could take the remainder mod X^t of the product of $f(X)$ and

$$\alpha^{-t}X^{t-1}+\alpha^{-t+1}X^{t-2}+\cdots+\alpha^{-1}$$

(which is $(X-\alpha)^{-1} \bmod X^t$), and then decrement the counter C. To carry out $\texttt{Enumerate}_{\Omega,t}$ we rely on the following identity:

$$\prod_{\alpha\in S}(X-\alpha) \;=\; \begin{cases} X^t+f(X), & \text{if } |S|=t; \\ f(X), & \text{if } |S|<t. \end{cases}$$

Hence, from $f(X)$ we can obtain $\prod_{\alpha\in S}(X-\alpha)$. Once this polynomial is determined, we find its roots as we did before, i.e. by polynomial factorization. This yields the elements of S. We now come to a crucial observation. The above procedures allow for multiple insertions and deletions of elements in Ω. Thus, they maintain a dynamic representation of a multi–subset of Ω. Hence, the advantage of these procedures is that they extend to multi–sets, albeit with a more complex deletion and enumeration procedure. Indeed, deletion will require a polynomial multiplication and enumeration a more involved factorization procedure. More precisely, we get the following

Theorem 9 *Let N be a positive integer, $\Omega = GF(2^m)^*$, and $n = |\Omega|$. There is a t-threshold data structure for multi–subsets of Ω of size up to N that supports any feasible sequence of the operations $\texttt{Insert}_\Omega$, $\texttt{Delete}_\Omega$, and $\texttt{Enumerate}_{\Omega,t}$ using $\lg N + t \lg n$ bits of storage. The instructions $\texttt{Insert}_\Omega$ and $\texttt{Delete}_\Omega$ can be implemented using $O(\log N + t(\log n)^2)$ and $O(\log N + (t\log n)^2)$ bit operations, respectively. If $|S| = d \le t$, $\texttt{Enumerate}_{\Omega,t}$ can be implemented using $O((d\log n)^3)$ bit operations.*

Proof: Consider the data structure described above. The memory requirement follows by observing that a counter of size up to N consumes $\lg N$ bits of storage and that to store the coefficients of a degree $t-1$ polynomial in $GF(2^m)[X]$ consumes $t \lg n$ bits of storage.

Insertion can be done with t multiplications and t additions in $GF(2^m)$, and by incrementing a $\lg N$ bit counter. Hence, it costs $O(\log N + t(\log n)^2)$ bit operations. Deletion can be done with t^2 multiplications and $t(t-1)$ additions in $GF(2^m)$, and by incrementing a $\lg N$ bit counter. Hence, it costs $O(\log N + (t\log n)^2)$ bit operations.

Now, consider enumeration. Note that $|S| = d \le t$ is available from the counter. Thus, $\texttt{Enumerate}_{\Omega,t}$ can be done by finding the roots of the degree d polynomial $g(X) = \prod_{\alpha\in S}(X-\alpha)$. In order to achieve this, we first determine the monic squarefree polynomials $g_1,\ldots,g_r$ and the positive integers $e_1,\ldots,e_r$ such that $g = g_1^{e_1}\cdots g_r^{e_r}$. (Cost: $O(d^3(\log n)^2)$ bit operations [2, Theorem 7.5.2].) Then, for $i \in \{1,\ldots,r\}$, we find the d_i roots of g_i as described in Sect. 2. (Cost: $O((d_i \log n)^3)$ bit operations for $i \in \{1,\ldots,r\}$.) Recalling that $\sum_{i=1}^r d_i = d$ we get that the total cost of enumeration is $O((d\log n)^3)$ bit operations. ∎

The costs stated in Theorem 9 may be reduced when $d, n \to \infty$ and if randomization is used. We leave the details to the interested reader.

If the cost of insertions and deletions is crucial and efficient memory usage is not so important, there is an alternative to the above described threshold data structure for multi–subsets of Ω. Indeed, assume $\Omega = GF(p^m)^*$ where p is a prime greater than t. As usual, let $n = |\Omega|$. Consider the same data structure described in Sect. 2, but where: (1) $\lg N$ bits are allocated for the counter; (2) the i-th power sums, for $i \in \{j : 1 \le j \le 2t-1, \gcd(p,j) = 1\}$ are stored; (3) arithmetic is performed over $GF(p^m)$ instead of $GF(2^m)$. In such data structure, provided precomputation is permitted, insertions (respectively deletions) can be performed by incrementing (respectively decrementing) the counter and with $\lceil (2t-1)(1-1/p) \rceil$

additions in $GF(p^m)$, i.e. $O((2t-1)(1-1/p)\log n)$ bit operations. Enumeration is essentially equivalent to a single decoding step in a t–error correcting narrow–sense BCH code over $GF(p)$. We note, however, that memory usage is not so efficient since the new data structure requires $\lg N$ bits to store the counter and $\lceil (2t-1)(1-1/p) \rceil \lg n$ bits to store the power sums.

The first threshold data structure for multi–subsets of Ω discussed above requires about $\log_2 N + t \log_2 |\Omega|$ bits of memory. We will show, by extending the argument of Sect. 4, this is optimal up to low order terms, when t is fixed and $|\Omega| \to \infty$. More precisely, let Ω be a universe of size n, and let M' be a finite state machine that models a t-threshold data structure for multi–subsets of Ω of size up to N. In the same way as the mapping q was associated to M in Sect. 4, we can now associate to M' a mapping q' which takes multi–subsets of Ω to elements of the state space Q' of M'. Conditions (1)–(3) of Sect. 4 should still hold for q' instead of q if we view the operations $\cap$ and $\cup$ as their natural extension to multi–sets. We henceforth refer to such view of conditions (1)–(3) as conditions (1')–(3'). Similar arguments to those given in Sect. 4 yield the following:

Lemma 4 *Let q' be a mapping from multi–subsets of Ω into Q' satisfying conditions (1')–(3'). If S and T are distinct multi–subsets of Ω such that $|S \cap T| \geq \min\{|S|, |T|\} - t$, then $q'(S) \neq q'(T)$, where $\cap$ corresponds to the multi–set intersection operator.*

We can now prove a lower bound on the required size of Q'.

Theorem 10 *Let $|\Omega| = n$. If q' is a mapping from multi–subsets of Ω into Q' satisfying conditions (1')–(3'), then*

$$|Q'| \;\geq\; \binom{n+t}{t} + \binom{n+t-1}{t}(N-t)\,.$$

Proof: Identify Ω with $\{1,\ldots,n\}$. We now define a family C of multi–subsets of $\{1,\ldots,n\}$ any pair of which satisfies the hypothesis of Lemma 4. For $0 \leq i \leq t$, let C_i be the collection of multi–sets of size i chosen from the universe $\{1,\ldots,n\}$. Note that the cardinality of C_i is

$$\binom{n+i-1}{n-1} \;=\; \binom{n+i-1}{i}\,.$$

We denote multi–sets by double brackets and let

$$\begin{aligned} C_{t+1} &= \{\{\{S,1\}\} : S \in C_t\}\,, \\ C_{t+2} &= \{\{\{S,1\}\} : S \in C_{t+1}\}\,, \\ &\vdots \\ C_N &= \{\{\{S,1\}\} : S \in C_{N-1}\}\,. \end{aligned}$$

Then, $C = \bigcup_{i=0}^{N} C_i$. The size of C is

$$\begin{aligned} \sum_{i=0}^{t} \binom{n+i-1}{n-1} + \binom{n+t-1}{t}(N-t) &= \sum_{i=n-1}^{n+t-1} \binom{i}{n-1} + \binom{n+t-1}{t}(N-t) \\ &= \binom{n+t}{t} + \binom{n+t-1}{t}(N-t)\,. \end{aligned}$$

Since all multi–sets in C must be represented by distinct states, we get the desired bound on $|Q'|$. ∎

When t is fixed and $n \to \infty$ the bound of the above stated theorem is asymptotic to $(N-t)n^t/t!$. Hence, the number of memory bits used by any t-threshold data structure for multi–subsets of Ω of size up to N, is asymptotic to $\log_2 N + t \log_2 n$ when $N, n \to \infty$. It follows that the number of memory bits used by the first method described in this section, namely $\lg N + t \lg n$, is asymptotically optimal.

Acknowledgments

We thank Satish Chandra and Subbarao Palacharla for bringing this problem to our attention. We also thank Stephen Pope for useful discussions on decoder performance and Anne Condon for comments on an earlier draft.

References

[1] Anonymous, Product Specification: AHA 4011 10 MBytes/sec Reed-Solomon Error Correction Device, Advanced Hardware Architectures, Inc., Pullman, Washington, no date. [Web site: `http://www.aha.com`.]

[2] E. Bach and J. Shallit, Algorithmic Number Theory, vol. 1: Efficient Algorithms, MIT Press, 1996.

[3] E. R. Berlekamp, Algebraic Coding Theory. McGraw-Hill, 1968.

[4] E. R. Berlekamp, H. Rumsey, and G. Solomon, On the solution of algebraic equations over finite fields, Inform. Control, vol. 10, pages 553–564, 1967.

[5] R. E. Blahut, Theory and Practice of Error Control Codes, Addison–Wesley, 1983.

[6] E. L. Blokh, Method of decoding Bose-Chaudhuri triple error correcting codes, Engineering Cybernetics 3, 1964, pages 22–32.

[7] L. M. Censier and P. Feautrier, A new solution to coherence problems in multicache systems, IEEE Trans. Comput. C-27, 1978, pages 1112-1118.

[8] C.-L. Chen, Formulas for the solutions of quadratic equations over $GF(2^m)$, IEEE Trans. Inform. Theory IT-28, 1982, pages 792–794.

[9] D. T. Chi, A new fast Reed-Solomon decoding algorithm without Chien search, in Proc. 1993 IEEE Military Communications Conference, IEEE Press, 1993, pages 948–952.

[10] R. T. Chien, B. D. Cunningham, and I. B. Oldham, Hybrid methods for finding roots of a polynomial – with application to BCH decoding, IEEE Trans. Inform. Theory IT-15, 1969, pages 328–335.

[11] R. T. Chien and W. W. Frazer, An application of coding theory to document retrieval, IEEE Trans. Inform. Theory IT-12, 1966, pages 92–96.

[12] T. Ericson and V. I. Levenshtein, Superimposed codes in the Hamming space, IEEE Trans. Inform. Theory 40, 1994, pages 1882–1893.

[13] E. Kaltofen and V. Shoup, Fast polynomial factorization over high algebraic extensions of finite fields, in Proc. 1997 International Symposium on Symbolic and Algebraic Computation, ed. W. Küchlin, ACM Press, 1997, pages 184–188.

[14] F. J. MacWilliams and N. J. A. Sloane, The Theory of Error-Correcting Codes, North-Holland, 1997.

[15] J. L. Massey, Shift register synthesis and BCH decoding. IEEE Trans. Inform. Theory, vol. IT-15, 1969, pages 122–127.

[16] A. M. Michelson and A. H. Levesque, Error–control Techniques for Digital Communication, Wiley, 1985.

[17] W. W. Peterson, Encoding and error-correction procedures for the Bose-Chaudhuri codes, IRE Trans. Inform. Theory IT-6, 1960, pages 459–470.

[18] Y. R. Shayan and T. Le-Ngoc, A cellular structure for a versatile Reed-Solomon decoder, IEEE Trans. Computers 48, 1997, pages 80–85.

[19] V. Shoup, A fast deterministic algorithm for factoring polynomials over finite fields of small characteristic, in Proc. 1991 International Symposium on Symbolic and Algebraic Computation, ed. S. Watt, ACM Press, 1991, pages 14-21.

[20] B. Trager and S. Winograd, to appear.

[21] J. H. van Lint, Introduction to Coding Theory, 2nd edition, Springer–Verlag, 1992.

[22] W.-D. Weber and A. Gupta, Analysis of cache invalidation patterns in multiprocessors. In Proc. 3rd Int. Conf. Arch. Supp. Prog. Langs. Op. Sys. (ASPLOS III), pages 243–256, 1989.

[23] D. Wood, et al., Mechanisms for cooperative shared memory, CMG Transactions, Issue 84, pages 51–62, 1994. [Also in Proc. 20th Ann. Intern. Symp. Computer Architecture, pages 156–168. IEEE Press, 1993.]