SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(4), 351–367 (APRIL 1993)

The Design and Implementation of Dynamic Hashing for Sets and Tables in Icon

WILLIAM G. GRISWOLD
Department of Computer Science & Engineering, 0114, University of California, San Diego, La Jolla, CA 92093-0114, U.S.A.

AND

GREGG M. TOWNSEND
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, U.S.A.

SUMMARY

Two key features in the Icon programming language are tables and sets. An Icon program may use one large set or table, or thousands of small ones. To improve space and time performance for these diverse uses, their hashed data structures were reimplemented to dynamically resize during execution, reducing the minimum space requirement and achieving constant-time access to any element for virtually any size set or table. The implementation is adapted from Per-Åke Larson's dynamic hashing technique by using well-known base-2 arithmetic techniques to decrease the space required for small tables without degrading the performance of large tables. Also presented are techniques to prevent dynamic hashing from interfering with other Icon language features. Performance measurements are included to support the results.

KEY WORDS: Hashing Programming languages Run-time systems

INTRODUCTION

Two powerful features in the Icon programming language[1] are tables—otherwise known as associative arrays—and sets. These are basic data types in the language and are implemented using hash tables.

The original implementation of tables and sets used a traditional hashing scheme with a fixed number of slots, so that all elements hashing to the same slot resided on a linked list at the slot.[2] Fixing the number of slots meant that a large table had long collision chains, slowing insertions and deletions to $O(n)$ for a table of n elements.* In fact, a few data-intensive applications ran slowly due to poor table performance.

Improving table performance by reconfiguring Icon with a larger slot array is not a reasonable solution. For one thing, applications using many small tables or sets are not uncommon in Icon, and these would waste considerable space if the array

* In the following, a *table* typically means a hash table representation, as opposed to the Icon table abstraction. When the distinction is important, the kind is stated explicitly.

0038–0644/93/040361–17$13.50

Received 6 March 1992
Revised 16 October 1992

were enlarged. In fact, owing to memory constraints, on platforms like PCs the slot array is already smaller than on other platforms. No static allocation method can address a wide enough variety of usage patterns.

Providing a size hint as an additional parameter to the table and set creation functions is not satisfactory, either. Hints to improve performance violate the spirit of Icon, and many Icon programmers are researchers in the humanities—not experienced programmers—who wish to handle large data sets without worrying about performance issues. Also, effective use of a size parameter requires advance knowledge of the ultimate table size, which may not be available in advance—for example, if the table is filled from an input stream.

An alternative is a table representation that adapts to the input to improve performance. Dynamic hashing is a technique that increases or decreases the number of slots in a hash table in proportion to the number of elements in the table. By ensuring that the number of slots and the number of elements are related by a constant factor—and assuming an even element distribution—insertions and look-ups can be performed in constant expected-time, as long as the cost of expansion is kept low. Although the high cost of address calculations traditionally limited the use of dynamic hashing to accessing external files, such as in database applications, Per-Åke Larson's adaptation of linear dynamic hashing[3] makes it practical for general-purpose applications.[4]

Larson's approach, unmodified, is expensive for applications using many small tables, because the expansion technique requires a large initial table size to efficiently support large tables. Adapting Larsons's technique by exploiting well-known arithmetic properties of powers of 2 and base-2 logarithms yields a relatively simple solution to the space problem. As a consequence, however, constant expected-time expansion of hash tables was sacrificed in favor of amortized constant-time expansion. This is acceptable for the typical Icon application.

In addition to improving performance, a design was needed that fit well with Icon's other powerful language features. In particular, self-reorganizing tables have to coexist with code that can be part-way through an iteration of a table's elements at the time of an insertion or deletion. The adaptation of Larson's technique was further modified to handle this interaction.

Common Lisp has a hash table data type (Reference 5, pp. 435–441) with features quite similar to Icon's table type, and faces similar design problems. The techniques presented below can apply to a Common Lisp implementation as well.

HASHED STRUCTURES IN ICON

Sets and tables are two built-in datatypes in Icon. They are implemented similarly, both using hash tables and sharing much code.

A set is an unordered collection of values of any type. Set operations include insertion, deletion, membership testing, union, intersection, and difference.

A table is an associative array, and in Icon any data type can be used for either the key or the value of any entry. Insertion is accomplished by assigning a value to `T[key]`; reference to an unassigned key returns a default value, but does not create a table entry. Elements may also be deleted.

Icon provides *generators* that produce sequences of values. If `S` is a set, `!S` produces the elements of the set one at a time. If `T` is a table, `!T` produces the

values of the table entries, and the generator function key(T) produces the keys. Several generators can be simultaneously active, and, when using a generalized form of coroutines called coexpressions, the generators need not start and finish in nested fashion.

As an example of the utility of tables in Icon, consider the following function countwords, which counts the number occurrences of each word in a file:

```
# Count the number of occurrences of the words retrieved by nextword
procedure countwords()
local word, count                    # local variables

   count := table(0)                 # allocate a table, default 0
   every word := nextword() do       # for each word...
      count[word] +:= 1                 # add 1 to the word's entry

   write("Word Counts:")
   every word := key(count) do       # for each word in the table...
      write(word, ": ", count[word])    # write it out with its count
end
```

The first statement allocates a table count in which each new entry has a default initial value of 0. The first iterator every, which in its simplest form operates like a Pascal for or while loop, retrieves successive words with nextword(), defined elsewhere in the program. The body of the every looks up the entry for the value of word in count, which contains the number of occurrences of the value of word, and adds 1 to it. After nextword() produces all of its values, the second loop is entered, which iterates through the keys of the table count using key(). The loop body retrieves the word's number of occurrences from count and prints it out with the word.

STATIC HASHING

As shown in the above example, a table stores a value that can be retrieved by a key that is not necessarily an integer. This key, logically, is the 'location' of the value. Associative look-up like this can be implemented with a static hashing algorithm (Reference 6, pp. 506–549). For an open hash table, the data structure for holding values is a linear array T of size n, indexed by integers from 0 to n–1. Each element of the array, referred to as a *slot*, is a linked list of pairs (*key*, *value*). To look up a value associated with a key in T, the key is converted into a *hash number* with a function h, which in turn is mapped into a slot number, that is $slot = s(key) = h(key) \bmod n$. Then the linked list at the slot, T [*slot*], is retrieved, the linked list is traversed until an element matching the key is found, and the value stored with the key is returned.

Figure 1 shows a hash table of seven slots containing three elements. The top half of each element contains the key; the bottom contains the associated value. The slot computation function is $s(k) = k \bmod 7$. For example, the second element on the first chain as key 21 with string value "b" . Its slot is computed as $s(21) = 21 \bmod 7 = 0$.

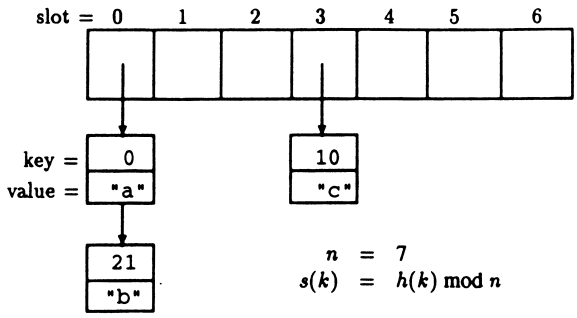

Figure 1. A small static hash table

If two conditions hold, a hash table supports constant expected-time element access: the elements must be relatively evenly distributed among the slots, and the number of slots must vary in direct proportion to the number of elements. Together these ensure that the average length of a slot's linked list remains constant, even as the number of elements grows. Guaranteeing short hash chains without wasting space is the focus of this paper.

An even element distribution over the n slots is achieved by applying an inexpensive hashing function h to key values. Constructing a good h is not trivial, but guidelines together with experimentation can lead to good results. A simple but adequate function is $h(k) = k \bmod M$, where M is prime (Reference 6, pp. 508–509), so that the resulting slot computation is $s(k) = (k \bmod M) \bmod n$. To avoid performing a modulo for both even distribution and slot computation, the number of slots in the table, n, can be chosen as a prime, so that the slot computation can subsume the scrambling property of $h(k)$. The parameter n need not be especially large; in Icon's original implementation of hash tables[2] n was 37 for platforms with large amounts of memory and 19 for those without. For hashing integers, $h(k) = k$ had proven sufficient. For hashing strings, $h(s) = \Sigma s_i$ where s_i denotes the integer value of character i, is easy to implement, portable and provides a passable distribution. However, because it is too slow for long strings, the hashing function in Icon sampled only the first ten characters of a string, and then added in the length of the string to distinguish strings with identical prefixes.

Icon's implementation of hash tables included two further optimizations. The hash value $h(k)$ was stored with the key, and elements were inserted in hash-value order. The first allowed the search of a hash chain to compare the hash values before comparing the keys, because comparing keys can be expensive in certain cases, such as long strings. Of course, when two hashed keys matched, the actual keys still had to be compared to test for an exact match. Keeping the chains ordered by hash value allowed the search to quit after finding a hashed key larger than the look-up hash key, with the knowledge that the rest of the chain contained keys greater than the look-up key. This ordering cut the number of keys examined in half, on average.

LARSON'S ALGORITHM

Given a good hashing function, preserving constant-time access to elements requires keeping hash chains short. Keeping chains short without wasting space requires adjusting the number of slots as the number of elements changes.

Hash table expansion can be supported with dynamic arrays. A dynamic array[2] can be implemented as a list of pointers, called the *segment list*, to pieces of the array, called *segments*.[4] A dynamic array is created with one segment that is the length of the initial array, *m*, pointed to by the first pointer in the segment list. If the segment list is L, then element *k* in the list is accessed as L [*k* div *m*] [*k* mod *m*], where the constant *m* is the size of each segment. When the array needs to be grown, a new segment of size *m* is allocated and the next segment pointer in the segment list is updated to point to it, as shown in Figure 2.

The price of this technique's flexibility is a more complex address calculation. The fixed-length segment table makes list addressing less expensive than a linked-list approach, but has the disadvantage that the number of elements in the list is bounded by the size of the segment table. Bounded array size, however, is not a serious problem for an open hash table, because it expands only to improve performance.

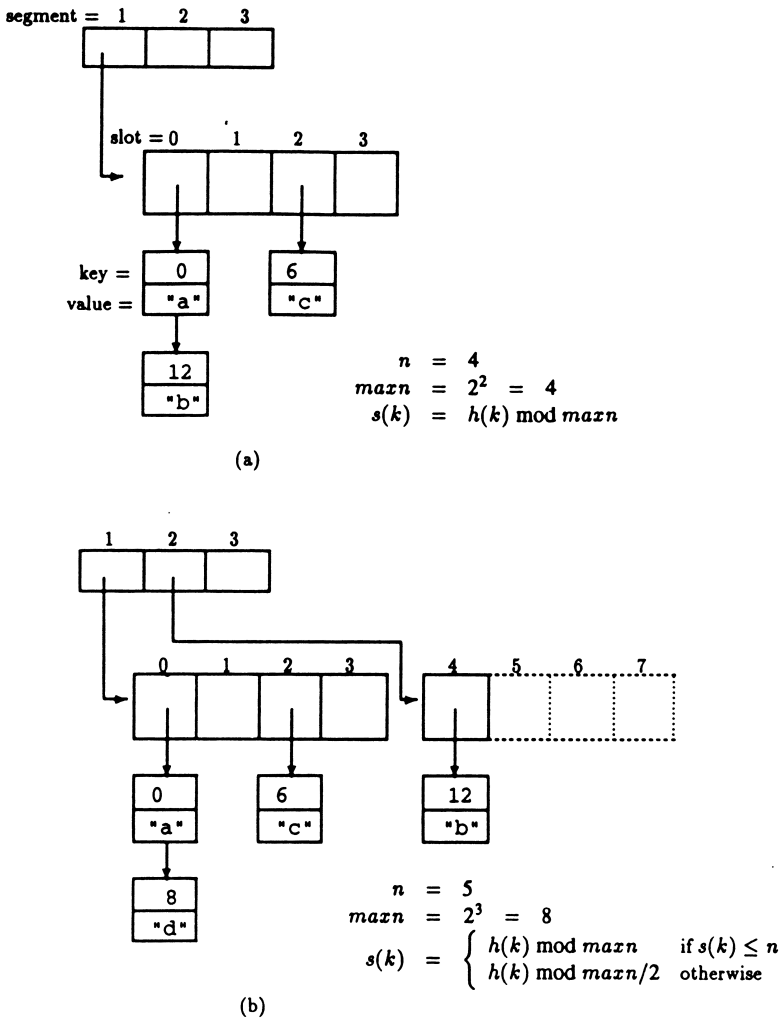

Figure 2. A dynamic hash table before (a) and after (b) expansion

When the average hash chain length, called the *load factor*, exceeds a threshold—a typical choice is 5—a slot must be added to the end of the hash table and some elements moved into it. Using dynamic arrays, a new slot is found in the first unused slot in the last allocated segment. If all the slots are in use, a new segment must be added. When the slot is added, some elements in other slots must be moved into this new slot. Naïvely, every element could be rehashed by $h(k) \bmod (n+1)$, but this could change the slot of every element in the table, hurting performance.

Larson's approach requires moving elements from only one slot—the *buddy slot*—to the new slot, but then some look-ups require computing *two* slot values instead of one. This minimization of slot reassignment is accomplished by treating the table as though it is being *doubled* in size to $2n$, not just increased by one slot.

First, for simplicity, suppose that a table's size is always kept as a power of 2, $n = 2^i$. Doubling a hash table, then, changes the slot computation function from $h(k) \bmod 2^i$ to $h(k) \bmod 2^{i+1}$. How do the elements move in such a doubling? For a power of 2, the modulo operation can be implemented as a bit-mask operation: $h(k)$ & (2^i-1). The number 2^i-1 is called the *hash-mask*. When doubling to 2^{i+1}, then, an additional bit of $h(k)$, bit i, is included in the slot number. However, before beginning a doubling expansion phase, all $h(k)$ with the first $i-1$ bits identical are already in a single slot, $s(k) = h(k) \bmod 2^i$. When slot $s(k)+2^i$ is added, all those elements in $s(k)$ that have bit i set in $h(k)$ are moved to the newly added slot.

Now consider how look-ups are performed when a table is expanded only one slot at a time. To support incremental expansion, a hash table must store *two* size numbers, the number of currently active slots, n, and the maximum number of slots available in the currently active segments, $maxn = 2^i$. In recognition of the fact that the expansion of a single slot is part of a doubling phase, the slot number function $s(k) = h(k) \bmod maxn$ is used. However, it is necessary to examine $s(k)$'s result on each look-up, and determine if slot $s(k)$ exists yet, that is, if $s(k) \leq n$. If so, it is accessed. If not, its buddy slot $s(k)-maxn/2$ must not have been split yet, so the look-up proceeds there. For example, in Figure 2(b), a look-up with the key 6 first produces $s(6) = 6$. However, this is greater than $n = 5$, so the alternative formula is applied, producing $s(6) = 2$, where the key is located.

Larson's technique supports contraction as well as expansion, so it does not waste space if elements are deleted from a table. When the average chain length drops under a threshold, say 3, then the last used slot of the last segment can be removed, placing its linked list of elements in its buddy slot. When all the slots in the uppermost segment have been removed, the segment can be removed from the segment list.

Choosing a hash table's size to be 2^i violates the assumption that $maxn$ is prime-like, and so slot computation is not likely to produce an even hash number distribution. To compensate, $h(k)$ must be more sophisticated. Larson uses the division-based function $h = ck \bmod M$, where c is a constant and M a large prime. Larson used 314,159 for c and 1,048,583 for M.

The advantages of Larson's dynamic hashing approach are its relative simplicity and good performance.[4] The method is a simple extension of the static hashing technique, depending on a straightforward implementation of dynamic arrays. Exploiting the special properties of addressing a table growing in powers of 2 allows inexpensive movement of elements and readdressing. Because only a single slot is

created when the load factor goes over the threshold, and only one slot has to be split, an expansion's cost is constant expected-time, and so the asymptotic cost of the insertion that triggered the expansion is still constant.* Constant expected-time insertion can be useful, for example, in highly interactive and animated applications. The extra cost of address calculation is minimal, especially in comparison to the savings from keeping collision chains short. Any performance loss, then, is noticeable only on tables with few elements, and is not significant. A drawback of this technique is that the segment table must be linear in size with respect to the *maximum* efficient table size, because segments are a fixed size. To be robust, then, a table abstraction implemented with fixed-size segments must have a comparatively large segment table.

Further discussion, theoretical analysis, and code for implementing the algorithm are presented in the original paper.[4]

MODIFICATIONS TO LARSON'S ALGORITHM

The basic ideas of dynamic hashing are simple and efficient, but making them fit the diverse needs of Icon programmers required several changes to the algorithm.

Segment expansion

In Larson's algorithm, each segment of slots is the same size. With such an approach a table of 256 segments of 256 slots can manage about 300,000 elements without performance degradation, assuming 5 as the load factor threshold. A newly created table requires a little over 512 words—2048 bytes—of space to accommodate the segment table and initial segment.

In the new algorithm, to overcome these large space requirements, each new segment *doubles* the number of slots in the table, as shown in Figure 3. This doubling behavior avoids a large segment list without sacrificing expandability. Suppose Icon's segment list contains 13 elements, and segments start at a length of 2^4. Then the last segment holds 2^{15} slots, adding up to 2^{16} slots in total. Consequently, the minimal Icon table for these parameters requires only a little over 116 bytes of storage to accommodate the segment table and initial segment, but can efficiently manage the same number of elements as Larson's configuration. The disparity grows for larger configuration parameters.

As described earlier, an element k in a dynamic array L is accessed by L [k div m] [k mod m], where each segment is of size m. This formula must be modified

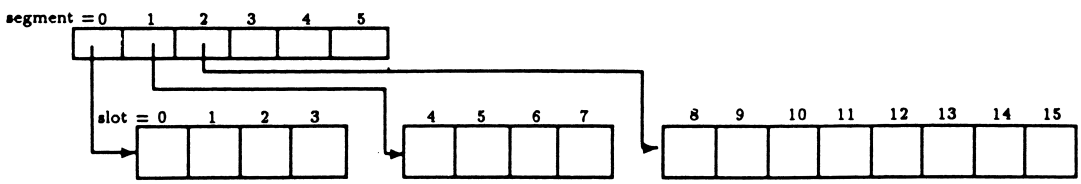

Figure 3. Each segment allocation doubles the number of slots

* Absolute constant-time expansion requires a hash function that guarantees evenly distributing elements among the slots. In theory, even distribution is not feasible unless the distribution of the keys is known beforehand (Reference 6, pp. 506–507).

for use with slot-doubling segments, since m is not fixed. Intuitively, with segments growing in powers of 2, the base-2 logarithm of k yields the segment number, and the index into the segment is found by subtracting the number of slots below the chosen segment from k. However, the initial segment size is not 2^0, but a larger power of 2, complicating the addressing slightly. Suppose segments are of size $m_0 = 2^3,\ m_1 = 2^3,\ m_2 = 2^4,\ldots,\ m_r = 2^{r+2}$. This progression is called three-rooted, since it starts with 2^3. To make the progression appear zero-rooted, each term is divided by m_0, yielding $m_0 = 1,\ m_1 = 1,\ m_2 = 2,\ldots,m_r = 2^{r-1}$. In the implementation, then, the progression is normalized by dividing k by m_0 before taking its logarithm. Note also in these progressions that the number of slots below segment *seg* is the number of slots in that segment, m_{seg}, except for segment 0, which has 0 preceding slots.

The slot access function, then, is $\llcorner [seg] [k–m'_{seg}]$, where

$$seg = \llcorner 1+\log_2(k \text{ div } m_0)\lrcorner, \text{ treating } \log_2 0 \text{ as } -1 \quad (1)$$

$$m'_{seg} = m_{seg} \text{ for } seg > 0, \text{ and } 0 \text{ for } seg = 0 \quad (2)$$

Since there are relatively few segments, the value m'_{seg} can be represented as a small fixed array indexed by *seg*. The segment calculation also uses an array look-up, indexed by k. Together the two arrays avoid the potentially expensive power and logarithm calculations, and avoid treating $k = 0$ as a special case. Also, since m_0 is a fixed power of 2, the operation k div m_0 is cheaply implemented as a right shift of k by $\log_2 m_0$.*

When the slots are doubled, all the corresponding elements are moved to the new hash chains at that time, in contrast to Larson's incremental approach. Eliminating incremental expansion simplifies the slot number calculation as well as other aspects of the implementation, described in the next section. Although the performance benefit is small, the benefit of maintaining simpler code is expected to be considerable in the long term.

Performing the entire doubling and moving all elements necessary at once means that the time required for a slot expansion is linear in the size of the table, rather than constant expected-time. Of course, the *amortized* cost remains the same. Practically speaking, foregoing incremental expansion can cause perceivable pauses in execution of an interactive program. However, Icon has a stop-and-copy garbage collector[2] that pauses execution longer than any table expansion because it copies a large portion of the heap, not just the chain pointers of one table.

Segment contraction

To simplify the implementation, particularly in the area of element generation, sparse hash tables never contract. The typical usage of sets and tables in Icon would make contraction a rare occurrence.† In particular, the delete operation is not frequently used, and set operations such as intersection create a new set rather than modifying an argument.

---

* With a little mathematics, it can be shown that $\log_2 (k \text{ div } m_0) \equiv \log_2 k – \log_2 m_0$. Because the divide is implemented as a shift, there is no performance benefit from this latter form. Also, the look-up table for $\llcorner \log_2 k \lrcorner$ is a factor of 2^{m_0} larger than the table for $\llcorner \log_2 (k \text{ div } m_0) \lrcorner$.

† The sieve program discussed in the performance measurements provides a counterexample.

Hashing

To avoid an expensive modulo operation, integer hashing uses the multiplicative method of hashing, taking a multiple of the golden mean as the multiplier (Reference 6, pp. 509–511): $h(k) = \lfloor 13{,}255k \text{ div } 1024 \rfloor$, where the integer division is implemented by a right shift. String hashing, to improve distribution, uses a simple extension of Larson's division-based approach, multiplying each partial sum in Icon's original string sum by a constant. Hash functions for the other Icon types are similar to the functions used for integers and strings.

CHANGES TO ICON'S IMPLEMENTATION

Changes to set and table operations

Because dynamic hashing adds a level of indirection for accessing slots, the methods used to chain through sets and tables changed. In operations requiring access to all the elements in a hash table, such as `copy`, a level of looping was added to index through the dynamically added segments to access all the slots. In the set operations union, difference and intersection, basic assumptions changed. For example, to perform set intersection with static hashing, individual corresponding collision chains were intersected. Slot-wise intersection worked because the number of slots never changed, so an element appearing in two sets necessarily resided on the same collision chain in each. With the new algorithm, the slots of two sets do not necessarily have a one-to-one correspondence, requiring that the set operations compute an element's slot address for each set to find overlapping values. This generalization was expected to incur a performance penalty, but in fact the operations are not perceptibly slower. It is possible that the extra cost of recomputing the slot address is mitigated by the shorter hash-chains realized by the new algorithm, but this hypothesis is untested.

Reorganization and element generation

The biggest changes to support dynamic hashing are in the generator operations `!S`, `!T`, and `key(T)`. These generators are implemented by traversing, in order, each slot of each segment and passing back each element in turn. Between elements, when the caller's code is in control, insertions can cause the table to be expanded one or more times.

A naïve traversal algorithm could generate a set or table elements more than once due to its movement during an expansion, which is not desirable. With the solution described below, any element that exists throughout the generator's lifetime is produced exactly once regardless of any expansions; elements inserted during generation may or may not be produced depending on when they are inserted and in what slot they happen to land. In the following, only elements that exist throughout the generator's lifetime are considered.

The boundary between the generated and not-yet-generated elements after an expansion is called a *split-point*, and the slot containing the last generated element a *split-slot*. Figure 4(a) shows a table in the process of being generated. It has one segment, which is depicted as a rectangular box; its length represents the segment

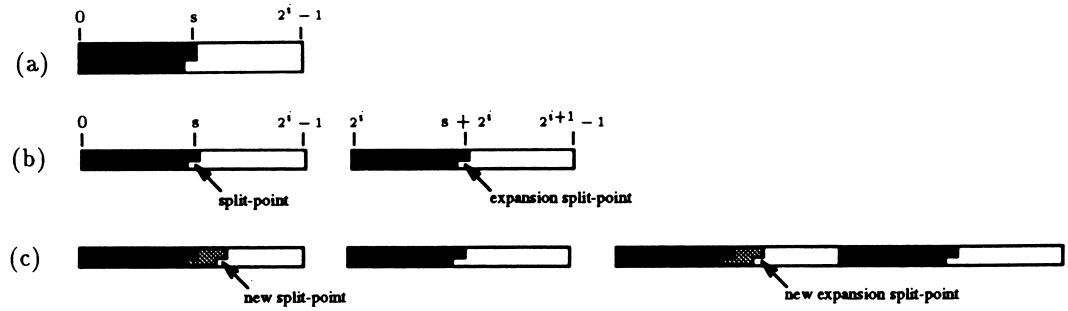

Figure 4. Repeated splitting creates many regions of visited and unvisited elements

length, and its height represents the current load factor. The visited elements in the left-to-right scan of the slots are indicated by the gray region, and unvisited elements by the white region. If the table were to be split in this state, the split-point would be indicated by the horizontal line delimiting the gray and white regions, and the split-slot, *s*, would be the slot containing the split-point.

After an expansion, a table has some new slots, and they are contained in one or more new segments. First, consider the situation in which exactly one expansion occurs between the generation of two elements, and no other expansion has occurred while this generator was active. Given that a new slot r in a table of size 2^{i+1} contains elements only from its buddy slot, $r–2^i$, there are the following cases:

1. A new slot whose buddy is before the split-slot contains only elements that were generated before the expansion.
2. A new slot whose buddy slot is the split-slot can contain some elements that have already been generated and some that have not. This new slot is called the *expansion split-slot* for the new segment. The visited elements must all reside at the front of the element-chain, because the elements are kept in the same relative order as before the split. The boundary between the visited and unvisited elements is the *expansion split-point*. When the distinction between the type of the split-slot or split-point is clear, the modifier is omitted.
3. A new slot whose buddy slot is beyond the split-slot contains only elements that have not been visited.

After the split, then, the visited and unvisited slots are broken into four contiguous groups, the visited and unvisited portions of the old and new segments, as shown in Figure 4(b). Thus, for the case of a single expansion, all new slots before the split-slot need not be visited, and generation of the new segment can begin at its split-slot.

More generally, several expansions may occur during the lifetime of a generator, and there may be several split-slots. In the same way that a single expansion distributes visited and unvisited elements in two pairs, a second expansion produces elements in four pairs, doubling the two in the lower half of the table. Where before there was a single expansion split-point for the first split-point, there are now three: at $s+2^i$, $s+2\times2^i$, and $s+3\times2^i$. Each of these marks the previous s slots as fully generated, where s is the slot-number of the original split-slot. These regions are called the *shadows* of the split-slot. The second expansion creates its own new split-

point and expansion split-point as well. Figure 4(c) shows the elements visited between the two expansions in light gray. The new split-point is guaranteed to be at or beyond the previous split-point since the generation proceeds from left-to-right. The newest segment, though, now contains discontiguous sections of unvisited elements. Each slot, then, needs to be checked for the presence of a split-point or shadowing by one.

Skipping visited elements is aided by saving some state information at the time of an expansion. A generator's state variables include pointers to the current element and the underlying table; the segment number and slot number of the element; and the table's hash-mask, which implicitly gives the total number of slots. Also included, to handle expansions, is a small array of records corresponding to the array of segments. These records save the hash-value of the element before the split-point and the hash-mask of the time of an expansion. The generator also caches the current hash-mask when generating each element.

Before producing a new element, the generator compares the table's current hash-mask against its previous value; a difference indicates that an expansion has occurred. If so, the previous hash-mask and hash-value are saved in the expansion records corresponding to the newly added segment or segments; generation then continues from the split-point. The previous hash-value, *phv*, denotes the last elements that were generated when the expansion occurred: $phv = h(k)$. Used with the previous hash-mask, *phm*, it also denotes the split-slot for that expansion, by *split-shot* = *phv* & *phm*, as well as all the expansion split-slots.

When the generator advances to a new slot, it checks the values saved in the expansion array to determine if it is shadowed by any split-slot. If it is not, there are probably elements in the slot that must be generated. If it is a split-slot, it starts generating from the greatest split-point for that slot. Because split-points are monotonically increasing, this is the newest split-point that occupies that slot. Otherwise, all of its elements must be generated. This bookkeeping is practical because segments are added only occasionally, and are never removed.

There is one potential problem in using a hash-value to record the state of a generator: because two keys can produce the same hash-value, a hash-value does not specify a unique location in a slot. But because elements with identical hash-values remain adjacent across all expansions, the hash value can represent the *sequence* of all such values. Specifically, the generator ignores an expansion—by not checking for a changed hash-mask—while it is in the midst of a sequence of identically-hashed elements. Following the sequence, the expansion is recognized and handled as described above. Thus the generator may actually generate a couple of elements in a newly-split slot before finishing its buddy, but when it returns later to the expansion split-slot, it starts after the elements it had generated earlier. It skips these because it starts with the elements that have a hash-value greater than the hash-value saved in the expansion array, which in this case is the hash-value of these last-generated elements.

PERFORMANCE MEASUREMENTS

The goal was to improve the speed of large hash tables in Icon, while also reducing the space used by small tables. Larson's technique provides the speed improvement, but uses excessive space for small tables and is not compatible with Icon's element-

generation mechanism. The new technique addresses both of these problems, but experimental analysis across a range of table and set usage patterns is required to ensure that significant overhead has not been added in the process.

The speed of Larson's implementation cannot be compared directly with the new technique because Larson's algorithm is too dynamic to handle element generation. The results, however, suggest that the new technique supports essentially constant-time table operations, so Larson's technique could vary by only a small constant. Space comparisons with Larson's technique were performed by taking each application's hash table size characteristics and computing a byte count with a formula combining Larson's resizing formula and Icon's implementation parameters.

All measurements were made on a Sun Sparcstation 2 running SunOS 4.0.1. To make comparisons between the static and new dynamic algorithms more equitable, the static hashing implementation uses the modified hashing functions described in the previous section. These improve the static hashing algorithm's performance slightly. The static hash tables were configured with 37 slots. For dynamic tables, the number of possible segments was set to 10, the initial segment to 8 slots, and the load factor threshold to 5. These parameters were chosen to maximize performance for a wide range of Icon applications. Although the parameters put a limit of about 2^{16} elements before performance degrades, the Icon run-time system can be reconfigured for a small cost in the space required for small tables and some additional space for the auxiliary slot and segment index tables. The space numbers for Larson's method was computed with the number of segments set to 64, the number of slots in a segment to 128, the expansion load factor to 5, and the contraction load factor to 3. These parameters allow Larson's technique the same expandability as the new technique.

Comparing the new method against Larson's, an empty table is 7·5 times smaller and accesses are still constant-time. However, recall that constant-time table expansion was replaced with amortized constant-time expansion and contraction was eliminated. Comparing the new method against the old Icon implementation, an empty table is 1·6 times smaller, and accesses are constant-time as opposed to linear-time. Medium-sized tables occupy about 4 per cent more space, decreasing to about 2 per cent for larger tables. The lack of contraction in the new technique means that space wastage can occur relative to the other techniques when a table has many elements deleted.

Figure 5 shows how look-up time is affected by the new algorithm. The graph compares the look-up performance of two runs of the program `words`, which inserts unique words in a table while periodically reporting the time required to look up every word in the input file. The input is the roots file `/usr/dict/words` of the Unix spelling checker `spell`. One run is with Icon's original static hashing algorithm, the other with the new dynamic hashing algorithm. The graph records user CPU time as measured from within the application. Sufficient heap memory was provided to the run-time systems so that garbage collection did not occur. The *x*-axis is graphed on a logarithmic scale. The graph shows that static hashing has linear-time look-up of strings, whereas the new dynamic hashing implementation achieves the expected constant-time look-up.

Figure 6 is a graph of the space usage, in bytes, of the two runs of `words` and of Larson's original algorithm. For consistency with Figure 5, the *x*-axis is graphed on a logarithmic scale. The graph shows that dynamic hashing uses much less space than Larson's algorithm for smaller table sizes, and performs a little better than the

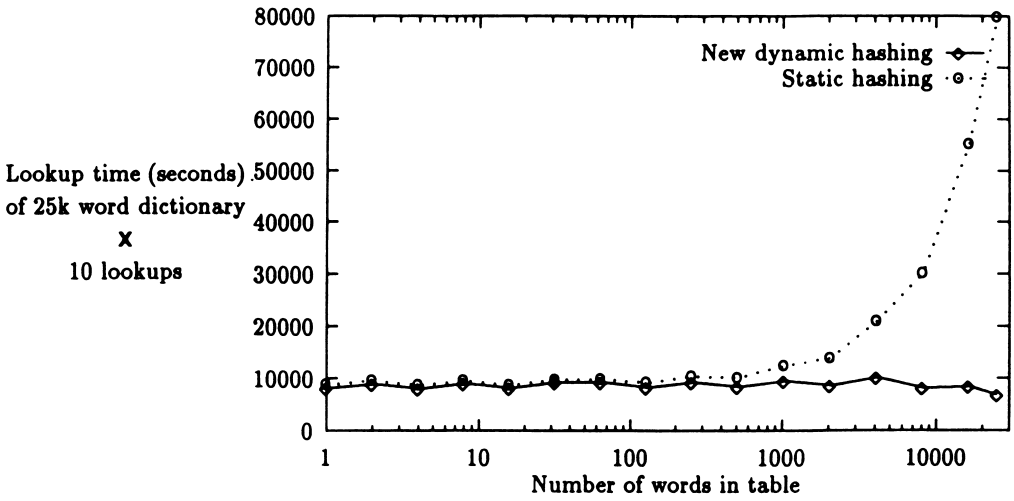

Figure 5. Look-up performance of static versus dynamic hashing

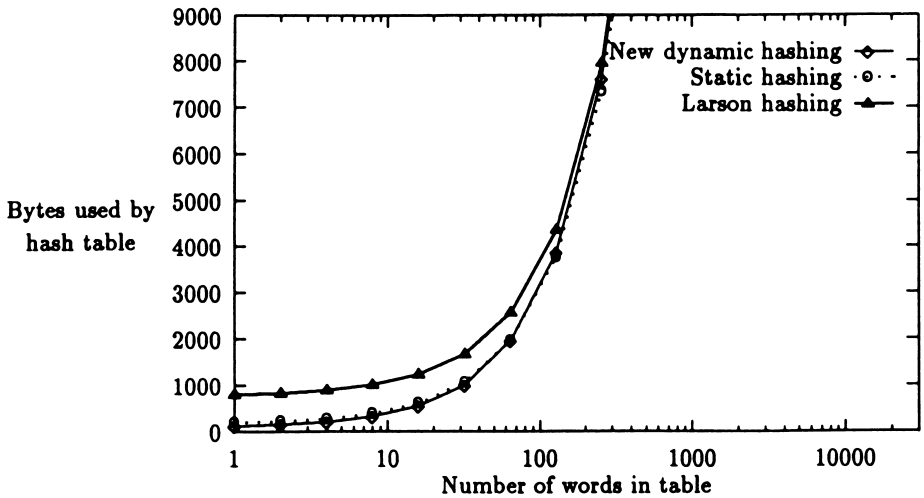

Figure 6. Space performance of static, dynamic, and Larson dynamic hashing

static algorithm. These differences are not significant for words, but an application using many tables, such as koncord, described below, can use an inordinate amount of space. The differences in space usage quickly diminish above 100 elements, because the space required for linking up the elements begins to dominate.

To show the performance impact of the new dynamic hashing technique across a spectrum of table and set usage patterns, seven additional Icon programs were measured. Table I presents the space and time characteristics of these applications. The time improvement yielded by dynamic hashing is reported as the ratio of the static implementation's run-time to the dynamic implementation's run-time, shown in the column marked 'Time, *s*/*d*'. Space results are reported similarly, but also include space data for Larson's method. As with the measurement of words, the

Table I. Performance tests on Icon programs (koncord has many small tables *and* one larger table)

Program	Table usage		Time			Space				
	Number	Size	Static	Dynamic	*s/d*	Static	Dynamic	Larson	*s/d*	*L/d*
words	1	25,144	328·6	144·4	2·28	704,200	720,848	723,952	0·98	1·00
newtwo	1	24,698	40·8	12·6	3·24	494,120	510,768	514,200	0·97	1·01
common	1	16,436	119·0	89·3	1·33	460,376	477,024	473,996	0·97	0·99
letter	1	15,516	16·4	10·9	1·50	434,616	451,264	447,528	0·96	0·99
scrabble	2	10,577	355·1	361·2	0·98	423,400	456,696	441,032	0·93	0·97
sieve*	1	10,000	4·0	1·6	2·50	24,760	33,176	26,920	0·75	0·81
koncord	1680	avg 4	5·8	5·8	1·00	463,484	358,228	1,518,720	1·29	4·24
	1	1680				47,208	49,400	48,856	0·96	0·99
rsg	1	22	14·5	14·8	0·98	784	720	1404	1·09	1·96

*The space numbers are measured at the end of the run, not at the high-water mark.

programs were run with enough initial heap to prevent garbage collection from biasing the results, with one exception explained below. In these measurements the Unix C-shell time command was used, and the value reported is the median of total run-time—user time plus system time—over seven runs.

The program words is the application measured for the results reported in the graph above. The program letter partitions the words of the dictionary into sets of words that have the same letters in them; for example, {*archaic, chair*} is one such set. It finds the partition for a word by converting it into a value of type character set, which is then used as a key in a table storing the partitions. Character sets are implemented as bit sets. The program newtwo finds all words in the dictionary that are composed of two other words in the dictionary: for example, *endear* is *end* followed by *ear*. The program takes every word in the dictionary at least four letters long and looks up all its pairwise partitions in the dictionary. The program common, as configured for this test, finds the 100 most common words in a 2 megabyte mail file. It uses a single large table of word counts, similar to the one for function countwords described at the beginning of this paper. The program scrabble is a Scrabble-playing program, and uses a dictionary to generate legal words for placement on the Scrabble board. The program played against itself for the measurements. Scrabble uses two sets for dictionary accesses, and, incidentally, a small table to represent the values of the different tiles. The program sieve computes primes based on the classic sieve of Eratosthenes (Reference 7, p. 394). It begins with a set of the integers 1 to 10,000, the maximum number to be considered for primality. Then each number in the set from 2 to 10,000 is selected in order, and each multiple of it is removed from the set as a non-prime. The program koncord derives the concordance of a file: for each unique word, it lists the line numbers on which the word appears, and how many times the word appears on each such line. The program uses a table indexed by the unique words for accessing the line information. The line information for each word is also stored in a table, keyed by the line numbers, for retrieving the number of instances of the word on that line. For each word, then, there is a look-up in the word table, followed by a look-up in that word's line-number table. The input for this program was a file of 60 thousand characters. The program rsg is a random sentence generator based on input of a grammar. It uses

just one table, for holding the grammar productions. The input grammar for this test has 22 productions, and produces 1000 poem triplets.

Most of the programs that use a large set or table show improved speed, supporting the results shown in Figure 5. The performance gain in newtwo is particularly large because the dominant computations are the dictionary look-ups of the powersets of substrings for each word. The program scrabble, in spite of the large initial heap it was allocated, still performed many garbage collections. The negligible performance difference is probably due to the large cost of these garbage collections and because dictionary look-ups are just a part of the program's computational tasks.

The programs with small sets, koncord and rsg, show insignificant changes in speed. The small differences agree with Figure 5, which shows a threshold for dynamic hashing in comparison to static hashing at about 500 words. Only programs that regularly access large tables can be expected to run significantly faster with dynamic hashing.

The results on space reveal that neither dynamic hashing technique has significant space overhead for moderate and large tables. For applications such as koncord, however, which use many small tables, Larson's technique performs considerably worse than the new technique. Static hashing uses moderately more space than the new technique on koncord. The downside of the new technique is revealed by sieve, which deletes the non-primes from a set of integers. Because the new technique cannot contract a hash table as elements are deleted, low utilization of slots can result, wasting space. Consequently, when sieve completes with only 1230 elements remaining in the set, the new version of dynamic hashing is wasting about 20 per cent of its table space relative to Larson's technique, and 25 per cent relative to static hashing.

POTENTIAL IMPROVEMENTS

The performance measurements suggest that the running time of the hashing algorithm is quite good, so speed improvement is unlikely. The space requirements of tables have also decreased relative to the previous hash table implementation in Icon. In contrast to Larson's constant-sized segments, which require a segment table *linear* in size with respect to the maximum efficient table size, the slot-doubling approach reduces the segment table's space requirements to logarithmic, without hurting amortized access time. An inherent limitation of both segment table approaches is that only a fixed number of segments can be added.

It is possible, however, to do away with the segment table altogether, not only saving space in programs using many small tables, but also allowing fully extendible dynamic arrays. That is, instead of addressing segments through a segment table, the segments can be linked sequentially—like a linked list—with each segment header indicating the range of indices addressed by the segment. With this representation, a look-up for slot *i* walks the segments until reaching the segment containing slot *i*. Because each segment doubles the number of slots of its predecessors, the maximum number of steps required to reach the correct segment is logarithmic in the number of elements in the table.* Better yet, linking the segments in *reverse* order—largest to smallest—requires traversing only one segment link on average, because half of

* Icon's lists are currently implemented in this fashion.[8]

the slots are in the first segment. Assuming an even look-up distribution, then, reversing the chains preserves amortized constant-time look-up.

Space also can be saved without abandoning the segment table approach. In particular, the first segment of a newly created table is smaller than the portion of the segment table reserved for pointers to later segments. The first segment can be overlaid on the end of the segment table until the table expands enough to need the overlaid segment pointers, at which time the segment can be copied out so that the segment table can be used fully.* The cost of the copy is probably small, because the first segment is small—currently eight words. Interactions with Icon's garbage collector add some minor complications, but we see no serious obstacles.

A simple alternative to the dynamic hashing expansion scheme is to reallocate the entire slot-array sequentially and move all the elements into the new array. This method eliminates the segment table and a level of addressing. However, the performance measurement in Figure 5 suggests that dynamic hashing's addressing costs are not significant with respect to the simpler static single-indirection addressing scheme. One drawback of reallocation is the additional cost of copying all the old slot values into the new segment. A more serious problem is that the old slot array cannot be deallocated until after the new one is created and initialized, causing a temporary and potentially large surge in the demand for memory. With a memory management scheme like Icon's,[2] reallocation increases the frequency of garbage collections by discarding large unused segments. With a non-compacting memory scheme, greater fragmentation is created because no segment discarded by a table expansion is large enough to accommodate any new segment for that table. Overall, then, this initially attractive modification turns out to be less desirable than the others.

CONCLUSION

Larson's design of dynamic hashing overcomes many of the shortcomings of static hash tables and provides constant expected-time table operations, but Icon's unique features required re-examining Larson's solution. By changing segment expansion from allocating constant-sized segments to allocating table-doubling segments, the size required for sets and tables containing few elements was reduced without hurting performance for large structures. Moreover, we were able to change Icon's run-time to allow element generation to coexist with table reorganization. These benefits were achieved at the expense of table contraction and constant expected-time slot-expansion, although the amortized cost of segment expansion remains the same. The improvement in both speed and space accommodates a wide array of symbolic applications without any special attention from the programmer.

As Johnson discovered when transferring theory to practice in compiler design,[9] transferring Larson's general dynamic hashing result to use in a symbolic programming language such as Icon required considerable effort, but it was worth it. Icon, like any complex system, must meet unique demands that entail unique design criteria—criteria that cannot, and need not, be anticipated by an algorithm designer. As a result, existing 'best' algorithms and data structures had to be reexamined and adapted for the specific application domain. Taking Larson's ideas and adapting them using well-known properties of powers of 2 and base-2 logarithms improved

* This idea is due to David S. Cargo of Cray Research.

both the space and time characteristics of tables and sets as they are commonly used in Icon. Although many of Larson's basic assumptions changed in deriving the technique presented here, his technique provided the essential structure of the solution.

ACKNOWLEDGEMENTS

We thank Ralph E. Griswold for his advice on this paper and on an early implementation of the technique. We thank the referees for their help in improving the performance section. This work was supported in part by an IBM Graduate Fellowship, NSF Grant DCR-8901573, and NSF Grant CCR-9211002.

REFERENCES

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, second edition, 1990.
2. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
3. W. Litwin, 'Linear hashing: a new tool for file and table addressing', *Proceedings of the 6th Conference on Very Large Databases*, 1980, pp. 212–223.
4. P.-Å. Larson, 'Dynamic hash tables', *Communications of the ACM*, **31**, (4), (1988).
5. G. L. Steele, *COMMON LISP, The Language*, Digital Press, Burlington, MA, 1991.
6. D. E. Knuth, *The Art of Computer Programming, Volume 3, Searching and Sorting*, Addison-Wesley, Reading, Mass., 1973.
7. D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., 1981.
8. R. E. Griswold, 'Supplementary information for the implementation of version 8 of Icon', *Icon Project Document 112*, Department of Computer Science, The University of Arizona, March 1991.
9. S. C. Johnson, 'A portable compiler: theory and practice', *Proc. 5th Symposium on Principles of Programming Languages*, January 1978, pp. 97–104.