Hashtables for Embedded and Real-Time Systems

Scott Friedman, Nicholas Leidenfrost, Benjamin C. Brodie, and Ron K. Cytron

Abstract-- **Data abstractions such as hash tables are included in most runtime libraries because of their widespread use and straightforward implementation. While operating systems and programming languages continue to improve their real-time features, much of what is offered by a runtime library is not yet suitable for real-time or embedded-systems.**

In this paper, we present an algorithm for managing hash tables that is suitable for such systems. The algorithm has been implemented and deployed in place of Java's `Hashtable` class. We present evidence of the algorithm's performance, experimental results documenting our algorithm's suitability for real-time, and lessons learned from migrating this data structure to real-time and embedded platforms.

1 INTRODUCTION

With the advent of operating systems and programming languages that support predictable execution times and reliable scheduling, the Real-Time and Embedded Systems (RTES) communities are beginning to consider the use of higher-level systems and abstractions for software development.

Most programming languages have rich libraries that offer strong implementations of commonly used data structures. Similarly, middleware [1] offers services, patterns, and frameworks that support the development of robust and portable software. For both runtime libraries and middleware, the need for predictable execution can have dramatic and widespread impact on the suitability of their offerings for RTES.

In this paper, we consider the adaptation to the RTES of a data structure that is common to most runtime libraries and middleware systems [9]. The *hash table* [4] is among the most popular of data structures, occurring in systems code, tools, and application code. In Sun's Java Virtual Machine (JVM) system, 7 hash tables are created before an application is even started; `jack` of the SPEC [5] benchmarks instantiates 18,805 additional hash tables. The theory [3, 6] of hash tables predicts nearly constant-time performance of hash tables *on average*, and experience has verified this theoretical efficiency. However, for RTES systems, bounds are needed on every call. Thus, for such systems, worst-case behavior per call is of greater concern than average performance.

Sponsored by DARPA under contract F33615--00--C—1697 and by Rockwell Collins
Contact author: cytron@cs.wustl.edu
Washington University Box 1045
Department of Computer Science
St. Louis, MO 63130 USA

1.1 Hash Table Interface

Essentially, a hash table is an implementation by memorization of a series of partial functions; a hash table can also be regarded as a mutable set of *(key, value)* pairs.

More informally, a hash table provides an implementation of a *dictionary* interface. Hash table implementations vary as to their Application Programming Interface (API), but most offer something resembling the following methods, called on an instance of a hash table we denote as *HT*:

GET(*key*) returns the *value* currently associated with *key*, if *(key, value)* $\in$ *HT*; otherwise, a predetermined value (`null` in Java) is returned to indicate that $\neg\exists$ *value* | (*key*,*value*) $\in$ *HT*.

PUT(*key*, *value*) causes *HT* to become the set (*HT* - { *(k,v)* | *k*=*key* } $\cup$ { *(key*,*value*) }

REMOVE(*key*) causes *HT* to become (*HT* - {*(k,v)* | *k*=*key*)}

Textbook treatments [4] of hash tables sometimes expose internal data structures for maintaining *(key, value)* pairs. The resulting API can accomplish REMOVE() in constant time, because the *container* for the *(key, value)* pair is provided directly to the method, which spares the method the need to find the pair before removing it.

The above API closely matches Java's `Hashtable` class---differences are discussed in Section 3 where they become relevant. Java's API deals only with keys and values and avoids exposing *container* objects that hold a given key and value.

Because of the strength of its design and because our experiments are based on a Java system, we adopt Java's hash table API as articulated above, without significant loss in generality.

1.2 Implementing the API

We implement each of the API's methods using the Command pattern [7], so that each method can make incremental progress when a rehash operation is in progress. The LOCATE() operation is described as follows:

LOCATE(*key*, *command*(*args*)) runs the supplied *command* which requires locating the entry for the supplied *key*, if such an entry exists.

All three of the API's methods involve determining if the hash table contains an entry *(key, value)* for the supplied

key. Our mechanism for enlarging the hash table takes action upon accessing such an entry.

The LOCATE() method allows us to perform additional work on behalf of the *command* at the supplied *key*'s bucket. We elaborate on this in Section 2.

1.3 Hash Functions

A hash table is typically implemented by mapping the space of all possible keys to a relatively small sequence of integers---suitable for indexing a table. A *hash function h* is defined as

$$h: Keys \rightarrow S$$

where $S \subset Z$ is a finite sequence of integers. Based on the above, a hash table is constructed to have $t=|S|$ buckets, denoted B(1), B(2), …, B(t). Thus, h(*key*) is suitable for selecting a bucket that might contain the key and its associated value. We follow practice [4] and assume that h(*key*) will map its inputs uniformly across the range of buckets. This property is commonly called the *simple uniform hash* property.

Each bucket contains a set of *(key,value)* pairs, with each *key* occurring at most once in all the buckets. (This is the *chaining* approach for handling collisions; open-addressing [4] is an alternative but is not well suited to RTES applications.) We can then perform GET(*key*) (more generally, LOCATE(*key*, *command*)) by searching the bucket B(h(*key*)) for an entry with the specified *key*. The hash table *HT* maintains that

GET(*key*)=*value* $\leftrightarrow$

h(*key*)=$i \rightarrow$ (*key*,*value*) $\in$ B(i)

We say that *key* hashes to bucket i using the hash function h. A *key* is *located* in a hash table by searching the bucket B(h(*key*)) for a (*k*,*value*) pair such that k=*key*.

The domain of a hash function h is typically much larger than its range. It is therefore likely that multiple keys of interest will hash to the same bucket. From the above, we can see that a bucket

B(i) = {(*key*,*value*) $\in$ *HT* | h(*key*)=i}.

As the number of entries in *HT* increases, the number of entries per bucket increases correspondingly. If the simple uniform hash assumption holds, then the increases are spread uniformly among the buckets.

The average and worst-case times to access *HT* are determined by the average and worst-case sizes of the buckets in the hash table. When a table's contents reaches some predetermined size, it is common pratice to consider redistributing that table's contents into a larger table with the goal of reducing and balancing the buckets' sizes. Because this typically occurs in response to a PUT call, it is possible that some PUT calls will be *much* more expensive than others. It is this behavior that makes extant hash table implementations unsuitable for RTES applications.

1.4 Real-Time and Embedded Systems Concerns

Operating systems such as Linux/RT[2] and languages such as Real-Time Specification for Java (RTSJ) [2] offer interfaces for declaring real-time concerns, such as a task's cost, periodicity, and deadline. Based on scheduling theory [8], a scheduler can determine whether a given set of tasks is *feasible*, in the sense that the tasks' deadline requirements are guaranteed to be accommodated on a given platform.

Because feasibility testing requires each task to declare its cost, it is important to state such costs as precisely as possible. Consider the provisioning that should occur when an application performs a PUT() on a hash table. As described above, most PUT() operations are performed relatively quickly; however, an occasional PUT() causes the hash table to be resized, with all its entries redistributed according to a new hash function. At a given PUT(), it is difficult to determine whether a hash-table resize would occur. How should the time for a PUT() operation be provisioned?

- Provisioning for the average or typical case is dangerous. The resulting requirements may be deemed feasible by a scheduler, but the PUT() in question may greatly exceed its stated cost. As a result, deadlines can be missed and an application can fail.
- Provisioning for the worst case is safe, but the resulting requirements may be infeasible on a given platform---every PUT() operation is provisioned as if a rehash operation is necessary.

Based on the above, the suitability of a hash-table implementation can be judged by the amount of over provisioning it imposes on a real-time application. This in turn can be quantified by an implementation's ratio of its worst- to average-case performance: as that ratio approaches *1*, so does the implementation's suitability for RTES applications. In Section 3 we measure this ratio for our implementation and for Java's reference implementation.

For embedded systems, storage behavior can be a determining factor. Hash tables adapt to greater load typically by reprovisioning the space in which *(key, value)* pairs are kept. For languages like Java, this can imply

1.) allocation of a new table (sometimes twice as large as the old (extant) table)
2.) rehashing of extant entries into the new table
3.) deallocation of the old table

Such storage behavior momentarily increases the program's footprint as items are copied, and then decreases the footprint as the old table is deallocated. This behavior is not well suited to embedded systems for the following reasons:

[2] See URL http://ww.timesys.com/products/linux2.html

- The program exhibits a storage *blip* during rehashing. The size of this blip is typically 50% of the new table's size. For an embedded system, this can be unacceptable.
- The deallocation of old hash tables can leave holes in the runtime storage heap. Such holes can cause the heap to be fragmented and thus trigger heap compaction.

As explained in Section 2.2, our approach avoids the temporary increase in storage footprint caused by allocating the new table before deallocating the old one.

2 APPROACH

To avoid burdening a single call with the overhead of an entire rehash, we spread the transition between table configurations over multiple operations. During a rehash, we maintain two hash functions, one "old" and one "new". The old function applies to the old table configuration and will be used to locate data that was mapped prior to the rehash and has not yet been relocated according to the new hash function. Rehashing does not occur in the context of a single hash-table method-call, but is instead amortized over as many calls as are necessary to complete the transition to the new hash function.

Although an implementation could feasibly maintain more than two hash functions, and thus perform multiple simultaneous rehashes, an unbounded number of such functions leads to unbounded time for LOCATE(*key*, *command*)---unacceptable given our requirements. This paper and its accompanying implementation allow only one transition in effect at any time: from the *old* to the *new* hash functions.

2.1 Incremental Rehashing

To complete the rehashing process, every element must be removed from its old location and correctly mapped to its new location using the new hash function. Since we resolve collisions by chaining, this involves rehashing each element in a linked list. We refer to the process of rehashing a bucket's contents according to the new hash function as *cleaning* that bucket.

We perform cleaning in two ways: operation-driven cleaning and methodical cleaning. Whenever the user performs an operation on the hash table (GET(), PUT(), REMOVE()), we rehash the elements contained in the bucket located by the old hash function.

To ensure progress toward completion of the rehashing, we cannot depend only on operation-driven cleaning. It is possible that the operations at-hand avoid a particular bucket. Therefore, *methodical* cleaning is also performed per operation; here, the bucket chosen for cleaning is based on the state of an incremental sweep of the hash table. While implementations may vary, it is essential that a bucket record whether it has been cleaned and that a table record whether any of its buckets are still dirty.

Bucket-cleaning is implemented as part of the Command-pattern. Specifically, if the hash table is moving from hash function h_{old} to h_{new}, then our implementation does the following when LOCATE(*key*, *command*) is called:

1.) The bucket $B(h_{old}(key))$ is cleaned.
2.) The bucket $B(h_{new}(key))$ is cleaned.
3.) The next bucket in the methodical list is cleaned.
4.) The *command* is executed.

2.2 Space Utilization

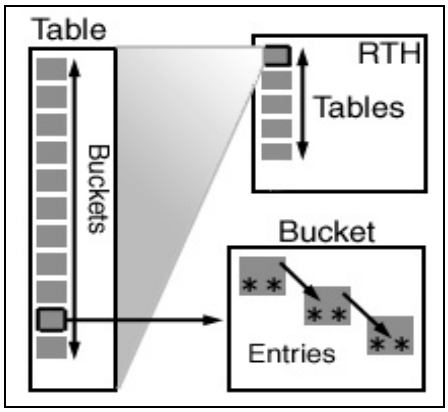

Figure 1: Two-level hashing scheme

Because of the constraints of RTES systems, particularly the embedded-systems concerns, our hash table does not free old storage when moving to the new hash function. Instead, the hash table grows by adding more tables to a two-dimensional hash scheme, as shown in Figure 1. Thus, the old subtables continue to participate as the hash table increases in size by adding additional subtables.

A hash function then must select both a table and a bucket; we assume both outcomes are subject to the simple uniform hash property.

The benefit of our approach is that the hash table does not temporarily blow up in size while the rehash takes place; instead, new space is added to the existing table, and the new hash function maps into the enlarged space.

2.3 Design

Our redesign of the hash table does not assume that all clients necessarily want real-time behavior. In order to give the user control over this aspect of the hash table, while providing the intended functionality, we separate the idea of the hash table mechanism---which implements the API and cleaning functionality---from the strategy that decides *when* the hash table should perform a rehash.

We introduce a POD object, or *P*oint *O*f *D*esign, through which an Observer of the hash table may call for rehashing. Such an Observer could subscribe to certain statistics about

the hash table that are of interest, and based on those statistics decide to ask for a rehash, as shown in Figure 2.

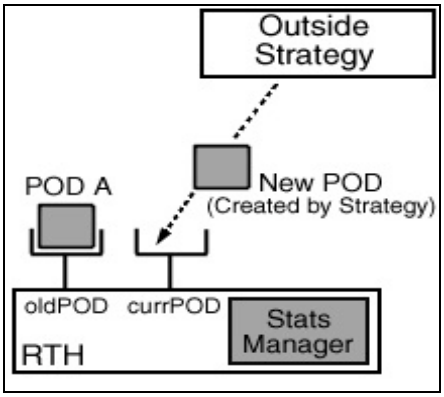

Figure 2: Observer can request a rehash.

2.4 Adapting to java.util.Hashtable

The class `java.util.Hashtable` implements `java.util.Map` and extends `java.util.Dictionary`. Because one of our criteria for the hash table is a small footprint, we decided not burden our primary implementation with all methods found in `java.util.Hashtable`. Instead, we created a wrapper around our hash table that provided the necessary functionality, but maintained realtime-compliant properties. We are thus able to substitute our implementation for Java's, and the experiments of Section 3 are based on this approach.

3 EXPERIMENTS

In this section we report on the results obtained from our implementation. These include careful timings to verify the real-time properties of our approach, as well as experiments conducted by substituting our algorithm for Java's implementation.

Our experiments were conducted on a Sparc Ultra~5 with 128 Mbytes of primary memory. To avoid unwanted interference, pages were locked into primary memory and our processes ran in real-time priority mode. Garbage-collection was disabled during hash table methods---a situation akin to running real-time threads under RTSJ.

3.1 Careful Timings of Contrived Benchmarks

We generated hash table entries using *n* random integers (`java.lang.Integer`) as the keys, for *n* = {*100, 1000, 10000, 100000, 150000*}. Due to randomness, some integers occur multiple times. The set of numbers is sequentially inserted, searched, and deleted from the data structures. The operations' times were gathered using Solaris's `gethrtime()` function and the Java Native Interface (JNI).

	Avg Time		Max Time		Max/Avg	
	rh	java	rh	java	rh	java
n = 100						
Put:	181.83	11.11	363	162	2.01	14.58
Get:	56.68	5.81	342	13	6.03	2.24
Remove:	202.41	7.17	490	15	2.42	2.09
n = 1,000						
Put:	185.83	9.76	446	1474	2.40	151.06
Get:	103.43	5.78	521	81	5.04	14.02
Remove:	208.72	7.11	435	71	2.08	9.99
n = 10,000						
Put:	184.64	11.32	556	19916	3.01	1759.35
Get:	63.46	5.59	557	84	8.78	15.03
Remove:	209.89	7.07	535	287	2.55	40.61
n = 100,000						
Put:	186.43	11.02	569	180138	3.05	16350.70
Get:	150.34	6.15	620	280	4.21	45.56
Remove:	209.70	7.23	800	274	3.81	37.89
n = 150,000						
Put:	188.22	12.77	1059	370297	5.63	28983.27
Get:	74.96	6.16	698	269	9.31	43.70
Remove:	215.33	7.26	692	309	3.21	42.55

Figure 3: Comparison of our algorithm (rh) against Java's (java) for the contrived benchmark. Times are shown in microseconds.

Figure 3 shows that the average time for both implementations is independent of the number of entries---nearly constant as predicted by theory. Also, the average time taken by Java's implementation is less than ours. This is a direct result of the resize-amortization feature of our implementation: it spreads the operations of an entire resize over multiple calls to the table, so the average time per call suffers.

The maximum time over all calls in our implementation appears to climb, but settles at a reasonable value; Java's maximum times are much worse, and are clearly dependent on the number of entries. These times are attributed to the single-call rehashing that occurs (during a *Put*) when its target load factor is exceeded.

These results show that the ratio of maximum to average time for each operation is reasonably bounded in our implementation while seemingly unbounded in Java's.

3.2 SPEC Benchmarks

As discussed in Section 2, we created an adapter class in our implementation so we could substitute it for Java's. This allowed us to test our implementation on the Java SPEC benchmarks jess, raytrace, db, mpegaudio, mtrt, and jack.

The following methods of the Map class were timed and recorded:

```
Object put(Object key, Object value),
Object get(Object key),
Object remove(Object key),
boolean containsKey(Object key),
boolean containsValue(Object value),
Set entrySet(),
Set keySet(),
Collection values().
```

	Avg Time		Max Time		Max/Avg	
	rh	java	rh	java	rh	java
size = 1						
jess:	194.87	11.78	1023	2551	5.25	216.58
raytrace:	279.14	35.80	672	2248	2.4	62.79
db:	268.99	35.22	655	2499	2.44	70.95
mpegaudio:	284.46	36.39	702	1801	2.47	49.49
mtrt:	280.18	35.92	690	1665	2.46	46.36
jack:	26.28	5.55	834	1740	31.26	313.35
size = 10						
jess:	59.25	6.08	1018	1708	17.18	280.85
raytrace:	281.62	34.57	699	1680	2.48	48.60
db:	272.91	34.95	737	2451	2.7	70.12
mpegaudio:	286.97	37.09	880	2463	3.07	66.40
mtrt:	280.19	36.23	577	1687	2.05	46.57
jack:	25.00	5.29	834	1704	33.35	322.42
size = 100						
jess:	46.19	5.30	1177	2034	25.48	383.93
raytrace:	289.97	35.46	848	2241	2.92	63.19
db:	265.51	36.32	550	3423	2.07	94.24
mpegaudio:	286.27	36.24	704	2032	2.46	56.07
mtrt:	280.88	36.28	625	2068	2.23	57.00
jack:		5.25		1838		351.47

Figure 4: Results on SPEC benchmarks.

Figure 4 shows that our implementation provides more predictable performance (as measured by ratios of worst-case to average times) for the SPEC benchmarks than does the standard implementation. We expect nothing less given the simple uniform hash assumption; thus, Figure 4 is in some sense a measure of the uniformity of Java's `hashCode()` method on the objects used as keys in those runs.

4 Conclusion

We have described an adaptation of a common data structure to real-time. From our experience, we offer thefollowing:

- Amortization of expensive operations can play an important role in migrating a data structure to real-time.
- The API of a data structure can make a big difference concerning the feasibility of ever obtaining a real-time implementation of that data structure. For example, some of the methods in Java's API insist on returning an array. Java cannot partially instantiate an array; thus, the full cost of allocating and initializing an array (to zero) must be paid by any call that returns an array. This can lead to unbounded behavior.

Our data supports our claim that the time to use our hash table is reasonably bounded. We are currently proving the real-time properties of our implementation; such a proof must show:

- Bounded behavior prior to resize
- Bounded behavior during resize
- No need to resize while already resizing

The last point can be proven only with respect to a reasonable strategy for picking the next table size. Our early results indicate that it is not necessary to double the table size on rehash, as seems to be the common wisdom, to obtain bounded behavior.

5 Acknowledgements

We thank Irfan Pyarali for suggesting this problem, and Doug Niehaus for his advice about real-time scheduling and his careful reading of this paper. We thank Gary Daugherty and Dave Haverkamp of Rockwell Collins for their support.

6 References

[1] David Bakken. Middleware. In J. Urban and P. Dasgupta, editors, *Encyclopedia of Distributed Computing*. Kluwer, 2001.

[2] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143-154, 1979

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[5] SPEC Corporation. Java spec benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.

[6] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Mey auf der Heid, H. Rohnert, and R. E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM Journal of Computing*, 23(4):738-761, August 1994

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[8] Jane W. S. Liu. *Real-Time Systems.* Prentice Hall, Upper Saddle River, NJ, 2000.

[9] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming: Resolving Complexity Using ACE and Patterns.* Addison-Wesley Longman, Reading, MA, 2001.