Hashtables for Embedded and Real-Time Systems *

Extended Abstract

Scott Friedman, Nicholas Leidenfrost, Benjamin C. Brodie, and Ron K. Cytron
Washington University Box 1045
Department of Computer Science
St. Louis, MO 63130 USA
August 12, 2001

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.

*Sponsored by DARPA under contract F33615–00–C–1697; contact author cytron@cs.wustl.edu
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 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.

1.1 The Hash Table Interface

Essentially, a hash table is an implementation by memoization 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 \(HT\):

\[\text{GET}(key)\] returns the \(value\) currently associated with \(key\), if \((key, value) \in HT\); otherwise, a predeter-
mined value (null in Java) is returned to indicate that $\not\exists value \mid (key, value) \in HT$.

$\text{PUT}(key, value)$ causes $HT$ to become the set

$$(HT - \{(k, v) \mid k = key\}) \cup \{(key, value)\}$$

$\text{REMOVE}(key)$ causes $HT$ to become $(HT - \{(k, v) \mid k = key\})$

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 The API Expressed using the Command Pattern

For expository and software-construction reasons, we find it convenient to implement the hash table’s API in terms of the Command pattern [7], so that each of the methods described above actually invokes the LOCATE() operation described as follows:

$\text{LOCATE}(key, \text{command}(args))$ 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

$$h : \text{Keys} \mapsto S$$

where $S \subset \mathbb{Z}$ is a finite sequence of integers.

Based on the above, a hash table is constructed to have $t = |S|$ buckets, denoted $B(1), B(2), \ldots, B(t)$. Thus, $h(\text{key})$ is suitable for selecting a bucket that might contain the key and its associated value. We follow practice [4] and assume that $h(\text{key})$ 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 $(\text{key}, \text{value})$ pairs, with each $\text{key}$ occurring at most once in all the buckets.\footnote{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(\text{key}) (more generally, LOCATE($\text{key}, \text{command}$)) by searching the bucket $B(h(\text{key}))$ for an entry with the specified $\text{key}$. The hash table $HT$ maintains that

$$\text{GET}(\text{key}) = \text{value} \iff (h(\text{key}) = i \rightarrow (\text{key}, \text{value}) \in B(i))$$

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

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 $B(i) = \{ (\text{key}, \text{value}) \in HT \mid h(\text{key}) = i \}$. 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 practice to consider redistributing that table’s contents into a larger table with the goal of 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 RTES 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’ specifications 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.

- Underprovisioning for cost is dangerous: a scheduler may determine feasibility but then a task may overrun its stated cost. As a result, deadlines can be missed.

- Overprovisioning is safe, but a scheduler may reject a set of tasks whose feasibility could be guaranteed if only the costs were provided more precisely.

Based on the above, the suitability of a hash table implementation can be judged by the amount of overprovisioning 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

\(^2\)See URL http://www.timesys.com/products/linux2.html
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.

- The program exhibits a storage \textit{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.

Our approach avoids the temporary increase in storage footprint.

\section{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 \texttt{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 Rehashing by Cleaning

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 the linked list. We refer to this transfer as cleaning, a process we perform in two ways: operation-driven cleaning and methodical cleaning. Whenever the user performs an operation on the hash table (\texttt{Get()}, \texttt{Put()}, \texttt{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.

The cleaning takes place during the Command-pattern implementation of the API. Specifically, if the hash table is moving from hash function $h_{old}$ to $h_{new}$, then our implementation does the following when \texttt{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

![Two-level hashing scheme](image)

**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, the new space is allocated and the rehashing (cleaning) maps from and to the expanded space which includes the old space as well.
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 Point Of Design, 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.](image)

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 involved small footprint, we decided not to put this functionality within the Hashtable class itself. Rather, 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. The details of our implementation will be presented in the full paper.
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 (\texttt{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 \texttt{gethrtime()} function and Java Native Interface (JNI).

Figure 3 shows that the average time 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 \textit{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.
<table>
<thead>
<tr>
<th></th>
<th>Avg Time</th>
<th>Max Time</th>
<th>Max/Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rh</td>
<td>java</td>
<td>rh</td>
</tr>
<tr>
<td>n = 100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put:</td>
<td>181.83</td>
<td>11.11</td>
<td>363</td>
</tr>
<tr>
<td>Get:</td>
<td>56.68</td>
<td>5.81</td>
<td>342</td>
</tr>
<tr>
<td>Remove:</td>
<td>202.41</td>
<td>7.17</td>
<td>490</td>
</tr>
<tr>
<td>n = 1,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put:</td>
<td>185.83</td>
<td>9.76</td>
<td>446</td>
</tr>
<tr>
<td>Get:</td>
<td>103.43</td>
<td>5.78</td>
<td>521</td>
</tr>
<tr>
<td>Remove:</td>
<td>208.72</td>
<td>7.11</td>
<td>435</td>
</tr>
<tr>
<td>n = 10,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put:</td>
<td>184.64</td>
<td>11.32</td>
<td>556</td>
</tr>
<tr>
<td>Get:</td>
<td>63.46</td>
<td>5.59</td>
<td>557</td>
</tr>
<tr>
<td>Remove:</td>
<td>209.89</td>
<td>7.07</td>
<td>535</td>
</tr>
<tr>
<td>n = 100,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put:</td>
<td>186.43</td>
<td>11.02</td>
<td>569</td>
</tr>
<tr>
<td>Get:</td>
<td>150.34</td>
<td>6.15</td>
<td>620</td>
</tr>
<tr>
<td>Remove:</td>
<td>209.70</td>
<td>7.23</td>
<td>800</td>
</tr>
<tr>
<td>n = 150,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put:</td>
<td>188.22</td>
<td>12.77</td>
<td>1059</td>
</tr>
<tr>
<td>Get:</td>
<td>74.96</td>
<td>6.16</td>
<td>698</td>
</tr>
<tr>
<td>Remove:</td>
<td>215.33</td>
<td>7.26</td>
<td>692</td>
</tr>
</tbody>
</table>

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

As discussed in Section 2, we created an adapter class 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().
```

Figure 4 shows that our implementation provides more predictable performance for the SPEC benchmarks that 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 uniformness of Java’s `hashCode()` method on the objects used as keys in those runs. In the full paper, we provide detailed data concerning the uniformity of the hash functions in the SPEC benchmarks.

4 Conclusions

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

- 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
<table>
<thead>
<tr>
<th>Size</th>
<th>Avg Time</th>
<th>Max Time</th>
<th>Max/Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rh</td>
<td>java</td>
<td>rh</td>
</tr>
<tr>
<td>size = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jess:</td>
<td>194.87</td>
<td>11.78</td>
<td>1023</td>
</tr>
<tr>
<td>raytrace:</td>
<td>279.14</td>
<td>35.80</td>
<td>672</td>
</tr>
<tr>
<td>db:</td>
<td>268.99</td>
<td>35.22</td>
<td>655</td>
</tr>
<tr>
<td>mpegaudio:</td>
<td>284.46</td>
<td>36.39</td>
<td>702</td>
</tr>
<tr>
<td>mtrt:</td>
<td>280.18</td>
<td>35.92</td>
<td>690</td>
</tr>
<tr>
<td>jack:</td>
<td>26.28</td>
<td>5.55</td>
<td>834</td>
</tr>
<tr>
<td>size = 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jess:</td>
<td>59.25</td>
<td>6.08</td>
<td>1018</td>
</tr>
<tr>
<td>raytrace:</td>
<td>281.62</td>
<td>34.57</td>
<td>699</td>
</tr>
<tr>
<td>db:</td>
<td>272.91</td>
<td>34.95</td>
<td>737</td>
</tr>
<tr>
<td>mpegaudio:</td>
<td>286.97</td>
<td>37.09</td>
<td>880</td>
</tr>
<tr>
<td>mtrt:</td>
<td>280.19</td>
<td>36.23</td>
<td>577</td>
</tr>
<tr>
<td>jack:</td>
<td>25.00</td>
<td>5.29</td>
<td>834</td>
</tr>
<tr>
<td>size = 100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jess:</td>
<td>46.19</td>
<td>5.30</td>
<td>1177</td>
</tr>
<tr>
<td>raytrace:</td>
<td>289.97</td>
<td>35.46</td>
<td>848</td>
</tr>
<tr>
<td>db:</td>
<td>265.51</td>
<td>36.32</td>
<td>550</td>
</tr>
<tr>
<td>mpegaudio:</td>
<td>286.27</td>
<td>36.24</td>
<td>704</td>
</tr>
<tr>
<td>mtrt:</td>
<td>280.88</td>
<td>36.28</td>
<td>625</td>
</tr>
<tr>
<td>jack:</td>
<td>5.25</td>
<td>1838</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Results on SPEC benchmarks.
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 unboundable 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.

Acknowledgements

We thank Irfan Pyarali for suggesting this problem.

References


