Report Number: WUCSE-2003-15

2003-03-18

Hash Tables for Embedded and Real-time systems

Scott Friedman, Anand Krishnan, and Nicholas Leidenfrost

Common collection objects such as hash tables are included in modern runtime libraries because of their widespread use and efficient implementation. While operating systems and programming languages continue to improve their real-time features, common implementations of hash tables and other collection objects are not necessarily 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... **Read complete abstract on page 2.**

Hash Tables for Embedded and Real-time systems

Scott Friedman, Anand Krishnan, and Nicholas Leidenfrost

Complete Abstract:

Common collection objects such as hash tables are included in modern runtime libraries because of their widespread use and efficient implementation. While operating systems and programming languages continue to improve their real-time features, common implementations of hash tables and other collection objects are not necessarily 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.

2003-15

Hash Tables for Embedded and Real-time systems

Authors: Friedman, S; Krishnan, A; Leidenfrost, N; Brodie, B. C; Cytron, R. K; and Niehaus, D

March 18, 2003

Abstract: Common collection objects such as hash tables are included in modern runtime libraries because of their widespread use and efficient implementation. While operating systems and programming languages continue to improve their real-time features, common implementations of hash tables and other collection objects are not necessarily 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.

Hash Tables for Embedded and Real-Time Systems *

Scott Friedman, Anand Krishnan, Nicholas Leidenfrost,

Benjamin C. Brodie, Ron K. Cytron, and Douglas Niehaus†

Washington University Box 1045

Department of Computer Science

St. Louis, MO 63130 USA

March 18, 2003

Abstract

Common collection objects such as hash tables are included in modern runtime libraries because of their widespread use and efficient implementation. While operating systems and programming languages continue to improve their real-time features, common implementations of hash tables and other collection objects are not necessarily 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

†This work was done while this author was on sabbatical from the EECS Department, University of Kansas

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 resource usage can have dramatic and widespread impact on the suitability of their offerings for RTES. The research community has been involved in the specification and development of real-time aware functionalities associated with programming languages [2] and middleware [10].

In this paper, we consider the adaptation to RTES of a data structure that is common to most runtime libraries and middleware systems [11]. The *hash table* [4] is among the most popular of data structures, occuring 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, reasonable bounds are needed for every method invoked on a hash table. 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 resemble 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 $\nexists\, value \mid (key, value) \in HT$.

PUT($key, value$) causes HT to become the set

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

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

Textbook treatments [4] of hash tables sometimes expose internal data structures for maintaining $(key, value)$ pairs. The resulting API can avoid successive lookups of the same key, because the *container* for the $(key, value)$ pair is exported for subsequent manipulation.

The above API closely matches Java's early `Hashtable` and later `HashMap` classes—differences are discussed in Section 4 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 Hash Functions

In this section we review the basics of *hash functions* and describe the role such functions play in common implementations of hash tables. We also discuss the relevance of those implementations for RTES.

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

$$h : Keys \mapsto S$$

where $S \subset Z$ is a finite sequence of integers. Following standard practice, we say that a key k *hashes* to index i if $h(k) = i$.

1.2.1 Collision Resolution

Open Addressing The domain for the key-space of a hash table is typically large as compared with the size of the hash table. It is therefore inevitable that two different keys hash to the same index, or *collide*. Collisions can be resolved using *open addressing*, which stores all entries in the hash table itself. Collisions are resolved by checking for the next available slot in the table following the index provided by the hash function. The indices at which a given key can be found in this manner is termed a *probe sequence*.

With open addressing, hash tables are expanded by allocating a table with more slots and then hashing the old table's entries into the larger table.

Chaining Another common approach for conflict resolution is *chaining*. Here, the entries that hash to the same index are placed in a linked list (commonly called a *bucket*). Hash tables managed in this fashion can grow without expanding the index space, because each new entry is added to the linked list at its hash index. However, when the number of entries in a hash table becomes too large, the average time to locate a key suffers, and the table itself is typically expanded as in the open-addressing case.

Comparison Open addressing does not require linked lists for each table index, and thus provides a smaller footprint. However, in searching for a given key, the probe sequence can include entries that do not collide with the key at hand, but happen to have hashed near the key's table index. The time to find (or

not find) a given key cannot be bounded by the number of keys that hash to a particular index. For example, to discover that a key is *not* in a nearly full table requires searching the entire table. With chaining, the search is limited to the number of entries that hash to the key's index. Moreover, removal of an entry is complicated with open addressing, because an empty slot usually ends the search for a given key.

With RTES in mind, we therefore do not further consider open addressing but instead adopt chaining for resolving conflicts.

1.2.2 Universal Hash Functions

Since any fixed hash function is vulnerable to worst-case behavior, an effective way to improve the average-case performance was suggested by Carter and Wegman [3]. The choice of a hash function is made in a random fashion, independent of the keys, from a pool of hash functions designed at the beginning of execution. If there are H hash functions to choose from, this scheme guarantees that the probability of collision by using a randomly chosen hash function to map two distinct keys is at most $1/m$ where m is the total number of slots in the hash table.

1.2.3 Perfect Hash Functions

If the set of keys inserted into a hash table is known *a priori*, then *perfect hashing* [4] allows lookups to be peformed in worst-case constant-time. A function is developed that immediately maps each known key to an index of the table. If the index is occupied, it contains the entry for that key; otherwise no entry is currently present in the table for that key.

This is the best solution for real-time applications if the keys are known *a priori*.

1.3 Hash Table Organization

For reasons described above, we use chaining to resolve hash function conflicts. A hash table is thus constructed to have $t = |S|$ buckets, denoted $B(1), B(2), \ldots, B(t)$. Given a key, the hash function h indicates that bucket $B(i)$ should be consulted to find the key and its value, if $h(key) = i$. We follow standard practice [4] of assuming the *simple uniform hash* property: h maps its inputs *uniformly* across the range of buckets, so that the probability of hash collisions follows a binomial distribution [7].

Each bucket contains a set of $(key, value)$ pairs, with each key occuring at most once in all the buckets. 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

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

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 $B(i) = \{\, (key, value) \in HT \mid h(key) = i \,\}$. 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 Implementing the API

We implement each of the API's methods using the Command pattern [8], so that each method can make incremental progress when a rehash operation is in progress. We have a generic method call interface, LOCATE() that 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.

1.5 RTES Concerns

Operating systems such as Linux/RT[1] 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 [9], 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 a task that records the progress of a nuclear reaction in a controlled reactor at intervals of various frequencies. This would merit the use of a hashtable for storing data pertaining to the sampling rate of various kinds of data which could be accessed by other tasks in the system in the process of controlling the nuclear reaction. The task provisioning that should occur in this case comprises in itself

[1] See URL `http://www.timesys.com/products/linux2.html`

the provisioning for performing 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 are wasteful and 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 4 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 the following actions.

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 *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. We analyze the requisite number of calls in Section 3.

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$(key, command)$—this is unacceptable given our requirements. We therefore restrict our design and implementation to 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 the following two ways:

operation-driven 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.

methodical cleaning: 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 at each 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.

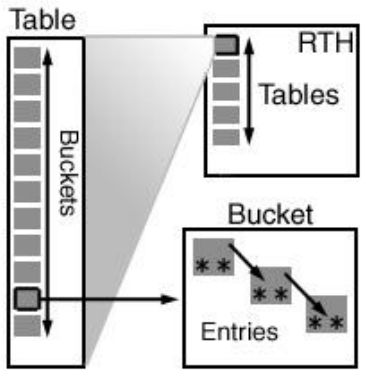

Figure 1: Two-level hashing scheme.

2.2 Space Utilization

Because of the constraints of RTES systems, particularly the embedded-systems concerns, our hash table does not allocate and populate a new hash table and 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.

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 Rehash Trigger Design

Our redesign of the hash table does not assume that all clients necessarily require real-time behavior. To allow control of the implementation's real-time behavior, while providing the intended functionality, we seperate 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 Point Of Design (POD) object, 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. For example, an application that cares more about average performance might want to trigger a rehash based on the current load factor of the hash table, defined as the ratio of number of elements to the number of buckets in the hash table.

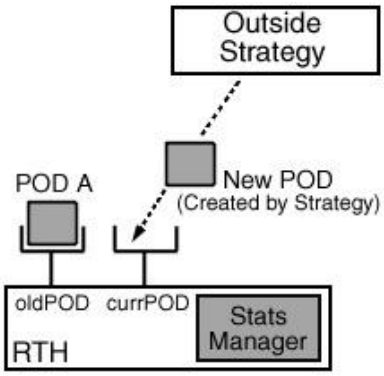

Figure 2: Observer can request a rehash.

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 to 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 real-time compliant properties. We are thus able to substitute our implementation for Java's, and the experiments of Section 4 are based on this approach.

3 Analysis

We use the following notation. We are interested in the state of the hash structure *prior* to rehashing, during rehashing, and just after rehashing.

b is the total number of buckets, across all tables, just prior to rehashing.

b' is the total number of buckets during and after rehashing.

Rehash Triggering Length (RTL) is the bucket-chain length that triggers a rehash operation, going from b to b' total buckets.

n is the number of $(key, value)$ pairs in the hash structure just prior to rehash.

n' is the number of $(key, value)$ pairs in the structure just after rehashing.

Definition 1 *Let $\alpha = \frac{n}{b}$ be the* load factor *of the hash structure just prior to rehashing. Similarly, let $\alpha' = \frac{n'}{b'}$ be the load factor of the hash structure just after rehashing.*

We provide analysis that identifies a lower bound on hash structure expansion to provide a bounded expected maximum-chain length.

Lemma 1

$$(b' > 0) \wedge \left(b' \geq \frac{b \times RTL + 1}{RTL - 1}\right) \rightarrow \alpha' < RTL$$

Proof: Consider the hash structure just prior to rehashing. One bucket has exactly RTL entries, but all other buckets have fewer than RTL entries. Thus

$$n \leq (b-1)(RTL-1) + RTL$$

Simplifying, we obtain

$$n \leq b(RTL-1) + 1 \tag{1}$$

Once rehashing begins, it is guaranteed to complete within b hash structure operations. The number of entries after rehashing, n', is therefore bounded as follows:

$$n' \leq n + b$$

Rearranging, we obtain

$$n' - b \leq n \tag{2}$$

By transitivity (1) and (2)

$$\begin{aligned} n' - b \leq \quad & n \\ n \quad & \leq b(RTL - 1) + 1 \end{aligned}$$

we obtain

$$n' - b \leq b(RTL - 1) + 1$$

Rearranging, we obtain

$$n' \leq b(RTL - 1) + 1 + b$$

which reduces to

$$n' \leq b \times RTL + 1$$

By assuming the lemma's antecedent $b' > 0$ we can divide both sides of the above inequality to obtain

$$\frac{n'}{b'} \leq \frac{b \times RTL + 1}{b'} \tag{3}$$

By assuming the lemma's antecedent $b' \geq \frac{b \times RTL+1}{RTL-1}$ we can rearrange to obtain

$$\frac{1}{b'} \leq \frac{RTL-1}{b \times RTL+1}$$

The above can be reduced to

$$\frac{b \times RTL+1}{b'} \leq RTL-1 \tag{4}$$

By transitivity of (3) and (4)

$$\begin{aligned} \frac{n'}{b'} \leq & \quad \frac{b \times RTL+1}{b'} \\ & \quad \frac{b \times RTL+1}{b'} \quad \leq RTL-1 \end{aligned}$$

we obtain

$$\frac{n'}{b'} \leq RTL-1 \tag{5}$$

Since $\alpha' = \frac{n'}{b'}$ by definition, we obtain

$$\alpha' < RTL$$

□

Theorem 1 *Just after rehashing,* $\forall\, i\ E(|Bucket(i)|) < RTL$.

Proof: By Lemma 1, we have $\alpha' < RTL$. The rehashing algorithm runs every element in the hash structure through the new hash function; the new hash function is a simple uniform hash function. We can therefore expect that elements are distributed uniformly among the b' buckets. □

Corollary 1 *During rehashing,* $\forall\, i\ E(|Bucket(i)|) < 2 \times RTL$

Proof: From (5), we expect the number of elements per bucket to be bounded as shown. Because rehashing is triggered when some bucket reaches RTL in size, any bucket during rehashing can only have a maximum of $RTL + (RTL - 1)$ elements on the average. Hence no bucket is expected to contain more than $2 \times RTL$ entries. □

Corollary 2 *Between the end of a rehash and the beginning of the next,* $\forall\, i\ E(|Bucket(i)|) < RTL$.

Proof: From the above proof of Theorem 1, we assume that the elements are distributed uniformly among the b' buckets so that $\alpha' < RTL$. In this phase, once an element is inserted in such a manner that its bucket exceeds a capacity of RTL, we begin rehashing. □

4 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.

4.1 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 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.

4.2 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().
```

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

	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.

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.

5 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 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

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

[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, Mass., 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] W. Feller. *An introduction to probability theory and its applications*. John Wiley & Sons, 1970.

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

[9] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, New Jersey, 2000.

[10] Realtime Platform SIG. Realtime CORBA. White Paper, Object Management Group, December 1996. Editor: Judy McGoogan, Lucent Technologies.

[11] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, Boston, 2002.