ETFA'2009 - 14th IEEE International Conference on Emerging Technologies and Factory Automation

On the Use of Hash Tables in Real-Time Applications

Rômulo Silva de Oliveira Carlos Montez Rodrigo Lange

Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina (DAS-UFSC)
Caixa Postal 476 – 88040-900 – Florianópolis-SC – Brasil

romulo@das.ufsc.br, montez@das.ufsc.br, lange@das.ufsc.br

Abstract. *Software applications use hash tables for many different purposes. Hash tables have an excellent average-case behavior but, in the worst-case, it degrades to something like a chained list. Because of that, the use of hash tables in real-time systems is not usual, since those systems may be required to guarantee deadlines. This paper discusses the use of hash table in real-time systems, considering that when the probability of an undesirable behavior is low enough, it can be ignored. It also compares approaches simple and 2-choice for the table design.*

1. Introduction

Real-time computer systems are defined in the literature as those submitted to timing requirements. In these systems, results should be correct not only logically but also be generated in the correct moment. In most applications, timing requirements have the form of deadlines for the execution of certain tasks [Liu 2000].

Generally, real-time systems are classified according to the criticality of their deadlines. In hard real-time systems the missing of a deadline may result in catastrophic consequences in an economic or environmental sense, or even in risk for human beings. For systems of this type it is necessary an off-line schedulability analysis. This analysis determines whether or not the system is going to meet its timing requirements even in a worst-case scenario, when we have the largest demands for computer resources. When deadlines are not critical (soft real-time) they just describe the desirable behavior.

In general, it is possible to use any conventional programming technique in the development of non-critical real-time systems. However, in hard real-time systems, because of the criticality of its deadlines, one should use only programming techniques that present a worst-case timing behavior that will not jeopardize the meeting of the deadlines.

Tables are one of the most used data structures in any application. One of the most used table implementations is the hash table [Cormen et al 1990]. Hash tables present excellent behavior in the average case. However, in the worst-case, hash tables present a behavior that degrades to something similar to a not ordered chained list. Because of this, the use of hash tables is not common in real-time applications, where it is necessary to guarantee deadlines even in worst-case scenarios. In general, for a hash table, the worst-case behavior is much worse than the average behavior. An enormous processor sub-utilization results from trying to guarantee deadlines in the worst-case scenario. This sub-utilization can be compensated partly by the use of servers that collect the reserved but not used processor time and then use it to execute tasks with soft deadlines. Even so, the need to design the hardware for the worst-case makes many systems economically unfeasible.

Most real-time software designers would prefer to use some type of balanced tree instead of hash table. Several types of trees described in the literature present a worst-case behavior that is close to its average behavior. However, the average case of the hash table is much better than the average case of balanced trees. Although the expense of memory is generally similar, it is easier to implement hash tables. Besides, it is not necessary to spend processor time to maintain the data structure balanced, as in the case of balanced trees.

Considering a hash table with `N` inserted elements, the worst-case will happen when the hash function generates the same value for all `N` keys associated with these `N` elements. In most implementations this scenario makes an insert operation in this table do degrade to a sequential search operation on all `N` elements. On the other hand, the probability of such a total collision to happen is insignificant in most applications. The probability of a hardware fault that compromises the whole system will be probably much larger than a temporary fault caused by the collision of a great amount of elements of the hash table.

Assume `P(e)` the probability of the event `e` to happen during the useful life of a given system. It is possible to define a minimally

relevant probability Pr, such that, for any practical ends, an event e with P(e)<Pr can be considered an impossible event. This way, if it is possible to show that the probability of a given number of collisions in the hash table during the useful life of the system is smaller than Pr, it will be then possible to consider this collision level impossible and to use, in the schedulability analysis, a smaller number of collisions.

This paper discusses the use of hash tables in real-time applications from the assumption that, when the probability of an undesirable event is sufficiently low, it can be ignored. This way, we propose the use of "relevant worst-case" instead of the "possible worst-case" in schedulability tests. In this paper it will be supposed that the number of elements presented in the hash table at every moment will be N in the worst-case, being N a well-known value. It will also be assumed that collisions are handled with by chaining.

This paper is divided in 6 sections. Section 2 surveys about hash table and real-time related works. Section 3 introduces the design criteria proposed in this work to use hash table in real-time applications. Sections 4 and 5 present a numerical solution and a numerical example. Conclusion remarks are presented in section 6.

2. Hash Tables

A hash table is a data structure that associates keys with values. The key is transformed through a hash function in a number that is used as an index to the table in order to locate the position where the associated value can be found [Cormen et al 1990]. Hash tables support the efficient insertion of new records with expected time O(1). The time spent in a search depends on the hash function and on the load of the table (number of records). Both insertion and search will have expected time O(1) with a careful implementation.

In most complexity analyses it is assumed a simple condition that the hash function presents uniform distribution. It is assumed that each key has the same probability of generating each one of the possible values of the function hash. That is to say:

- For any two keys k1 and k2, the chance of h(k1)=h(k2) it is exactly $1/m^2$, where m is the number of entries of the table.
- For two keys k1 and k2, the probability of h(k1)=h(k2) is exactly 1/m.

For the general case, it is not possible to guarantee that one given hash function will exhibit such behavior. An approach to deal with this problem is using a universal hash table function. In this case, the hash function is chosen randomly, independently of the keys, from a finite group H of hash functions. Group H is said universal if, for each pair of different keys k1 and k2, the number of hash functions h in H, such that h(k1)=h(k2), is at most |H|/m; where |H| is the cardinality of group H. Universal hash results in a good performance for the average case that can be demonstrated. However, since the group of keys is dynamic and unknown, there is a non-null probability of the table presenting a terrible behavior.

In the case of a perfect hash, the hash function is built in such a way that two keys never result in the same hash value. Although possible and very efficient for static and well-known groups of keys, efficient methods don't exist to generate perfect hash functions for dynamic groups of keys.

2.1. Collision Resolution

The domain of the keys of a hash table is typically much larger than the number of entries of the table. It is inevitable that two different keys end up being mapped to the same entry of the table by the hash function. In this case it is said that the two keys collided.

A form of working with the problem of collisions is to use open addressing. In this approach, all records are stored in the table itself. Collisions are resolved through the location of the next free space in the table after the entry which address was supplied by the hash function. This search is made in the table in a circular way.

Another possibility to resolve collisions is to chain the entries of the table whose keys resulted in the same address provided by the hash function. This way, each address of the table corresponds to the beginning of a chained list of records.

2.2. Approach 2-Choice

Consider an ordinary hash table that treats collisions with chaining and assume a uniform hash function. In this case, after the insertion of n keys, the length of the longest chain will be O(log n / log log n) with high probability. In this paper, high probability means: with a probability of at least $1-O(1/n^{alfa})$, for some constant alpha [Gonnet 1981].

Suppose now that two uniform hash functions are used. In the moment of inserting an element in the table, both hash functions are calculated and therefore two possible entries of the table are identified (2-choice). The new record will be inserted in the entry with the smaller chained list. If n keys are inserted in the table, the length of the longest chain is O(log log n) with high probability. In the case of a search, again it will be necessary to calculate the

value of the two hash functions and to search the two suitable chained lists, once the sought key could have been inserted in any one of the two entries. In this approach, the search time is related with the scanning of the two chained lists associated with a given key. Anyway, the search will also present a complexity of $O(\log \log n)$ with high probability [Mitzenmacher et al 2000].

Approach 2-choice has the advantage of using only two hash functions, being easy to parallelize and not needing re-hashing of previously inserted records. It also offers robustness in the case of the hash functions not being perfectly uniforms [Karp et al 1996]. The use of balanced allocations (which is a more comprehensive treatment of this same approach) is discussed in [Azar et al 1994].

2.3. Use of Hash Tables in Real-Time Systems

In [Friedman et al 2003], the use of hash tables in embedded real-time systems is considered. An incremental approach is proposed for the reorganization of hash tables that is similar to incremental garbage collection. The proposed approach is applied to hash tables using chaining. That work is motivated by the performance problems that appear when a hash table with chaining is very loaded, resulting in many collisions and linear search through long lists. The conventional solution is to build a new table, larger than the original, all at once, removing records from the original table and inserting them in the new table. This conventional approach makes the table unavailable while it is completely reconstructed. Consequently, this approach is not adequate to be used in real-time applications. The proposal in [Friedman et al 2003] makes insertions in an incremental way, so as to minimize the time the table is unavailable.

In [Parson 2004], an algorithm is discussed to reorganize hash tables whose collision resolution uses the technique called open hash table. The performance of open hash tables degrades after many insertions and deletions, so it is necessary to reorganize the whole table. Typically that is done in a monolithic way, that is, the whole table stays unavailable while the whole table is reorganized all at once. The algorithm proposed in [Parson 2004] alternates among the incremental construction of a new table through the selective copy of the entries, and the incremental cleaning of the original table through the emptying of its entries. This limits the response time in the worst case to an event that requests a search in the table.

Few works can be found on hash tables in the literature of real-time systems. In this paper it is assumed that the maximum number of records at any time in the hash table is known. This way the table is already created with an appropriate length, which never changes. Growing of the table in run-time is not necessary. The subject approached in this paper is the identification of a behavior that, although better than the worst-case (where all keys end up in the same entry of the table), it is so unlikely that it can be considered, for all practical ends, as the effective worst-case possible behavior of the system. It is therefore a different subject from those discussed in the literature referenced.

3. Design Criteria

We propose the following steps to use hash tables:

- Definition of a probability level value, such that, below this value events are considered irrelevant, because they are too rare;
- Computation of the probability of a list of length k to happen during the system operational life;
- Determination of length t of the longest list that still have a relevant probability of happening, this will be the worst-case list length for the sake of computing the worst-case scenario for the system.

For example, assume that the hash table is modified once every second. Also, assume that at any time the hash table has no more than N records, where N is a characteristic of the application. During system operation there are insertions and deletions, but the number of records in the table at any moment is never bigger than N. So, we may have a new configuration for the table with N records at every two seconds. It is possible to compute the number of different configurations that might be generated through 100 million years:

$$10^8 \text{ year} \times 365{,}25 \text{ day/year} \times 24 \text{ hour/day} \times 3600 \text{ s/hour} \times 0{,}5 \text{ config/s} \leq 0{,}16 \times 10^{16} \text{ config.}$$

A possible design criterion is to consider that any event that happens once every 100 million years in average is irrelevant to the design of the computer systems under analysis. An example of that is the collision of a giant meteor with planet Earth that will destroy most life on the planet. This way, a probability of 10^{-16} would be considered the threshold. Events with smaller probabilities than that are still possible, but not relevant.

4. Numerical Solution

With the purpose of showing the concept, we implemented in Java a tool that calculates the PMF (Probability Mass Function) of the random variable $Lh(N)$; that is, the length of the chained

list associated with entry h of the hash table, when there are N records in the table. For the numeric example it will be adopted the probability value of 10^{-16} as threshold, based on the discussion presented in the previous section. We will assume for the hash table the values M=1000 for the number of entries in the table and N=750 for the maximum number of records in the table at any moment during the useful life of the system. Collisions are resolved with chaining.

In order to compute the Probability Mass Function of the length of the chained lists it will be made the addition operation on random variables, that is, it will be made the convolution of their respective Probability Mass Functions. Let X to be the random variable length of the chained list of any entry of the table. Starting from N=0, we have that the PMF of variable X is defined by P(X=0)=1.

For example, assuming that the hash function is perfectly uniform (this restriction will be discussed in Section 5 and removed after in the experiences), we can have Ph as the probability of a new record to be inserted in a given entry h of the table. This value can be calculated by Ph=1/M, since it does not depend on the number of records already inserted in the table. Whenever a new insertion happens, the probability of the list associated with entry h to increase in length is 1/M, and the probability of the list associated with entry h to stay with the same length is 1-1/M. The random variable Ih defines the increase in length of the list associated with entry h of the table, and its PMF is:

$$\begin{cases} P(Ih=0) & = 1 - 1/m \\ P(Ih=1) & = 1/m \\ P(Ih>1) & = 0 \end{cases}$$

In order to determine the probabilities of a system with N=1, we only have to add random variable Lh(0), that represents the list length when N=0, to variable Ih, which determines the change that the list associated with entry h of table will suffer after a new insertion. Since both Lh(0) and Ih are random variables, the result of this addition Lh(1) will also be a random variable. The PMF of Lh(1) will be given by the convolution between PMF of Lh(0) and PMF of Ih.

Initially we have the following PMF of Lh(0):

$$\begin{cases} P(Lh(0)=0) & = 1 \\ P(Lh(0)>0) & = 0 \end{cases}$$

After the convolution of Lh(0) with Ih we have the PMF of Lh(1):

$$\begin{cases} P(Lh(1)=0) = P(Lh(0)=0)\times P(Ih=0) \\ \qquad = 1\times(1-1/M) \\ \qquad = 1-1/M \\ P(Lh(1)=1) = P(Lh(0)=0)\times P(Ih=1) \\ \qquad = 1\times 1/M \\ \qquad = 1/M \\ P(Lh(1)>1) = 0 \end{cases}$$

After the convolution of Lh(1) with Ih we have the PMF of Lh(2):

$$\begin{cases} P(Lh(2)=0) = P(Lh(1)=0)\times P(Ih=0) \\ \qquad = (1-1/M)\times(1-1/M) \\ P(Lh(2)=1) \\ \quad = P(Lh(1)=1)\times P(Ih=0) + P(Lh(1)=0)\times P(Ih=1) \\ \quad = \frac{1}{M}\times\left(1-\frac{1}{M}\right)+\left(1-\frac{1}{M}\right)\times\frac{1}{M} \\ P(Lh(2)=2) = P(Lh(1)=1)\times P(Ih=1) \\ \qquad = (1/M)\times(1/M) \\ P(Lh(2)>2) = 0 \end{cases}$$

It is noticed that, when N grows, the individual probabilities for the possible values of list length decrease, since all PMF present longer tails. It is exactly for big values of N that the probability of occurrence becomes smaller than the threshold probability for relevant events.

When a PMF is uniform or even partially uniform, it is possible to accelerate the computation of the convolution by taking advantage of the existing symmetries. For example, in the case of a PMF perfectly uniform, all entries of the hash table present the same behavior. Therefore, it is possible to calculate for just a single entry the PMF of its chained list length, and to consider this valid result for all entries. In the same way, a PMF where an entire section of values presents the same probability, it is possible to accomplish the calculations for just a representative of this section and to use the results for all entries of this section. In the numeric experiences of this paper hash functions were used with large uniform sections. This technique resulted in significant reduction of the computation time.

4.1. Computation of the PMF

A difficulty found in the implementation of the tool is related to the resolution of the data type double. For example, in Java the data type double corresponds to the format for floating point with double precision (64 bits) defined in the standard IEEE 754. In this representation, the granularity is around 2×10^{-16}, that is to say, it is not capable of representing small, but essential numbers accurately for the correct calculation of the PMF.

In order to overcome this difficulty, we used Java class `java.math.BigDecimal`, which implements decimal numbers with arbitrary precision. An object `BigDecimal` consists of an integer number with arbitrary precision (base value) and an integer of 32 bits that defines its scale. If the scale is zero or positive, it represents the number of digits on the right side of the decimal point. If the scale is negative, it means that the base value is multiplied by 10 to the power of the value of the scale. The number represented by an object `BigDecimal` is given by: $\texttt{baseValue} \times 10^{-\texttt{scale}}$.

The class `BigDecimal` provides arithmetic operations, manipulation of scales, rounding, comparisons, etc. The class allows the complete control on rounding. If no way of rounding is specified and the exact result cannot be represented, an exception is raised.

In the case of the calculation of a PMF, some way of rounding is necessary for two reasons. Firstly, values with infinity number of digits are possible since that the probability `P(Ih=1)=1/M` itself can generate this, in particular because M is usually chosen to be a prime number. Secondly, even if it is possible to represent all values of interest with a finite number of digits, the number of digits grows in such a way that the computation time becomes prohibitive.

Rounding in PMF is possible, but it demands special cares. Every PMF presents as its fundamental property the fact that the summation of all the probabilities of the individual values results an exact value 1. For example, with `M=3` and hash function with perfectly uniform distribution:

$$\begin{cases} P(Ih=0) & = 1 - 1/3 = 2/3 \\ P(Ih=1) & = 1/3 \\ P(Ih>1) & = 0 \end{cases}$$

Clearly the sum of the probabilities is 1. However, in case the rounding is always made upward, with 10 digits, the property is lost:

$$\begin{cases} P(Ih=0) & = 0.6666666667 \\ P(Ih=1) & = 0.3333333334 \\ P(Ih>1) & = 0 \end{cases}$$

The property of the sum being always 1 could be maintained through compensations in the rounding, just as:

$$\begin{cases} P(Ih=0) & = 0.6666666667 \\ P(Ih=1) & = 0.3333333333 \\ P(Ih>1) & = 0 \end{cases}$$

But, in this case, a serious mistake is included in the model. The probability of the list associated with entry `h` having one element was artificially reduced. That is to say, the results that would be obtained starting from this analysis would be artificially optimist, what is not acceptable in a real-time system that requires guarantees for the worst-case.

In the sense of preserving the worst-case analysis, at the same time that rounding is made possible, we opted in this study to always do rounding followed by compensations that maintain the property of always having the sum to be `1`, by always increasing the probability of the worst possible behavior. Although this solution introduces a small pessimism in the analysis, it preserves the guarantees of behavior in the worst-case originating from this analysis.

Another point related to the rounding is the number of decimal digits to preserve in the rounded values. Once the rounded values are probabilities and we work here with the concept of a threshold of relevant probability, we opted for working with a number of decimal digits that allows the exact representation of probabilities 10,000 times smaller than the smallest relevant probability. For example, when the smallest relevant probability is 10^{-16}, the rounding mechanism preserves `20` decimal digits. Consider the PMF below:

$$\begin{cases} P(X=0) & = 0.367 \\ P(X=1) & = 0.322 \\ P(X=2) & = 0.211 \\ P(X=3) & = 0.100 \\ P(X>3) & = 0 \end{cases}$$

A rounding to `2` decimal digits would generate the PMF shown below. It is observed that in both cases the sum of the probabilities is `1`, and that the rounded PMF includes a small pessimism when increasing the probability of `X` to assume value `3`. However, the probability of `X` to assume larger values than `3` continues zero. In the case of a hash table with `N` records, we will always have that the probability of an entry of the table to have associated with it more than `N` records to be zero.

$$\begin{cases} P(X=0) & = 0.36 \\ P(X=1) & = 0.32 \\ P(X=2) & = 0.21 \\ P(X=3) & = 0.11 \\ P(X>3) & = 0 \end{cases}$$

It is necessary to notice that the rounding has as affect a displacement of the probabilities to the right, in the sense that columns of the left lose probability for columns of the right. This way, a small pessimism is inserted in the behavior of the system, what invalidates the result for optimistic behaviors. But it maintains the validity of the result for the worst-case behavior, that what we are interested in this work.

5. Numeric Example

For the numeric example it will be adopted the threshold probability of value 10^{-16}, according to the discussion presented in the previous section. It will be assumed for the hash table the values M=1000 for the number of entries of the table and N=750 for the maximum number of records in the table at any moment of the useful life of the system. Collisions are resolved with chaining.

The quality of a hash function is associated with its capacity to do a perfect hashing of the keys it receives. This way, an ideal hash function distributes the keys evenly among the entries of the table. In practice, hash functions present a behavior not as good as the ideal one.

At first, any hash function is vulnerable to a situation of great number of collisions. In [Carter and Wegman 1979], Universal Hashing is proposed as a form to improve the performance in the average case of the hash function. The hash function to be used is chosen in a random way at the beginning of the execution, independently of the keys, from a group of designed hash functions.

Even universal hashing doesn't guarantee an ideal behavior during execution. The numeric method described in this paper allows the designer to include in the analysis the imperfections of the hash function used. This way, it is possible to foresee the behavior of the hash table lists even when the hash function is not perfect, what comes to be most frequent case. Three types of functions will be considered:

- Perfectly uniform hash function, where every entry has the same probability of receiving a given key:
  $P(h=x)=1/M \quad case\ 1<x\le M.$
- Quasi-uniform hash function, where half the entries receive 2/3 of all keys, while the other half of the entries receive 1/3 of the keys, within each half, there is a uniform distribution:
  $$\begin{cases} P(h=x)=4/(3\times M) & case\ 1\le x\le M/2 \\ P(h=x)=2/(3\times M) & case\ M/2<x\le M. \end{cases}$$
- Non-uniform hash function, in which 3 entries receive 30% of all keys, while all other M-3 entries of the table receive 70% of all keys, it is assumed that M>3:
  $$\begin{cases} P(h=1) & =0.1 \\ P(h=2) & =0.1 \\ P(h=3) & =0.1 \\ P(h=x) & =(0.7/M), \quad 4\le x\le M \end{cases}$$

When the 2-choice approach is used, two hash functions H1 and H2 are necessary. If the hash functions are uniform, then H1 is equal to H2. When the hash functions are quasi-uniform or non-uniform, it is necessary to decide whether the entries with larger probability will be the same ones in both functions or they will be different in H1 and H2. The computations were done for both situations. Functions H1 and H2 are said to be not correlated when the entries of larger probability for each one are different. They are said to be correlated when the entries of high probability are the same for both functions.

5.1. Results

For a simple hash table with a uniform hash, the relevant maximum length of the chained list is 16 records, although 750 records were inserted in the hash table. The probabilities calculated for this case are presented in Table 1.

The fundamental results of all computations done are presented in Table 2. When the hash function is not perfectly uniform, but quasi-uniform (in the terms defined in this paper), the relevant maximum length for the chained lists increases to just 17, that is to say, a single extra record. When using the non-uniform hash function however, with a high concentration of probability in a few entries, the relevant maximum length of the chained list increases to 151. That is, almost ten times greater than the uniform case, which is the one usually considered in the literature. In this case, the real-time predictability of the system would be seriously compromised, was the system designed for a uniform hash function.

Table 1. Probabilities for a simple hash table.

Records	Probability
0	0.47676059996147070471
1	0.35333048416116927176
2	0.13075322758731027045
3	0.03221456331861269999
4	0.00594473290489221967
5	0.00087643690653154137
6	0.00010753384311034233
7	0.00001129378589414100
8	0.00000103647269260709
9	8.443815743448E-8
10	6.18267536145E-9
11	6.18267536145E-9
12	2.501022747E-11
13	1.40297537E-12
14	7.298059E-14
15	3.53786E-15
16	0.6000E-16
17	6.12E-18
18	1E-20
19 to 749	1E-20
750	1E-20
751	0

Table 2. Relevant maximum length of the chained lists.

Table	Hash Function	Correlated	Relevant max. length	Records to scan
Simple	Uniform	N/A	16	16
Simple	Quasi-Uniform	N/A	17	17
Simple	**Non-Uniform**	**N/A**	**151**	**151**
2-choice	Uniform	N/A	3	6
2-choice	Quasi-Uniform	Not Correlated	3	6
2-choice	Quasi-Uniform	Correlated	3	6
2-choice	Non-Uniform	Not Correlated	8	16
2-choice	**Non-Uniform**	**Correlated**	**10**	**20**

N/A = Not Applicable

The quality of a hash function is associated with its capacity to do a perfect scattering of the keys it receives. This way, an ideal hash function distributes the keys evenly among the entries of the table. Actually, hash functions present a behavior that is worse than the ideal one. It is necessary to live with the fact that the hash function used will not spread the keys perfectly among the entries of the table.

The 2-Choice approach increases the robustness of the system. Firstly, the relevant maximum length of the chained lists drops from 16 to 3 if the hash function is perfectly uniform. In the case of quasi-uniform hash functions, be they correlated or not, the relevant maximum length for a chained list remains 3. When the 2-choice approach is applied, it is necessary to scan two chained lists in the worst-case scenario. Therefore, a relevant maximum length of 3 for the lists indicates that it will be necessary to scan `3+3=6` records for each search operation in the table. Nevertheless, this number is much smaller than the 16 or 17 records of the simple table.

In the case of non-uniform hash functions, as well as, in the case of the simple hash table, the relevant maximum length of the chained lists increases. However, even in the case of correlated hash functions, it remains in 10, what results in scanning `10+10=20` records of the 2-choice table, instead of the 151 records for the simple hash table. The probabilities calculated for the case of the 2-choice table with correlated non-uniform hash functions, when considering an entry of high probability, are presented in Table 3.

Table 3. Probabilities for a 2-choice hash table.

Records	Probability
0	0.00900853291867478015
1	0.02367672240017487672
2	0.04528701275583918460
3	0.06499467606307381824
4	0.08976225758159696449
5	0.12795405064570978937
6	0.19983082784151369890
7	0.30209411092767075707
8	0.13556813347534509603
9	0.00182367409035382306
10	1.30004720397E-9
11	1E-20
12 to 749	1E-20
750	1E-20
751	0

The experiences indicate clearly that it is possible to adopt a probabilistic approach for the worst-case behavior of hash tables. Although there is a non-null probability of a same entry of the hash table to receive all the 750 records, what would make the search operation a very slow one, this non-null probability is so small (smaller than 10^{-20} in our case) that it can be considered irrelevant for many applications. More realistic maximum lengths for chained lists can be used, still with a good margin of safety.

The largest risk of this approach is hash functions that present pathological behavior for the group of records observed in practice. For example, the concentration of 30% of all records in only 3 entries of the hash table made the relevant maximum length of a list of the table to jump from 16 or 17 to 151 in the case of the simple hash table. In this case, the 2-choice table represents a safety mechanism regarding design faults of the hash function. In the same situation, and considering two correlated hash functions, the relevant maximum length was 10, what would mean that a search operation needs to scan 20 entries (two lists).

It was necessary about 3.5 hours to compute all the cases presented in this paper, in a Pentium of 2GHz and 500Mbytes of main memory, being the tool programmed in Java. The time to compute the probabilities depends on the formats of the hash functions and on the table type. For example, the case of the 2-choice table with correlated non-uniform hash functions took about 40 minutes in that computer.

6. Final Remarks

This paper discussed the use of hash table real-time applications, where the response time of the tasks is an important constraint and, sometimes, it must be guaranteed for critical applications. It

was shown that it is possible to define a minimum probability for events to be considered relevant such that, for any practical propose, an event with smaller probability than the threshold probability can be considered as an impossible event.

The paper described a software tool built to numerically calculate the function of mass probability of the length of the chained lists of a hash table that resolves collision through chaining. This tool uses several techniques to control the precision of the data representation and to reduce the computation time. Simple hash tables were considered together with tables based on the 2-choice approach.

The experiences showed that it makes sense to adopt a probabilistic approach for the behavior of the hash tables in the worst-case. Although there is a non-null probability of the same entry of the hash table to receive all the inserted records, this non-null probability is so small that it can be considered irrelevant. This way, more realistic maximum length of chained lists can be used, still with good safety margins. It is up to the designer of the software to define the minimally relevant probability of its system. For example, one could relate it with Safety Integrity Levels [Redmill 2000].

A subject not considered in this work is the time to calculate a second hash function for each operation on the table. This is the price to pay for the 2-choice approach. If the computational cost of the second hash function is high, it may compensate the gain of the 2-choice approach in search time. However, it may be advantageous to use a 2-choice table, due to the robustness that it adds to the table in the occurrence of hash functions with pathological behavior.

Another open question is the precise characterization of the PMF of the hash functions used, also taking in consideration the real data of the application. Any available information on the values of actual keys would allow a better characterization of its PMF, which results in a more precise numerical analysis.

References

Azar, Y., Broder, A. Z., Karlin, A. R., Upfal, E. (1994) “Balanced Allocations”, Proc. of the 26th Annual ACM Symp. on the Theory of Computing (STOC 94), pages 593-602.

Carter, J. L. and Wegman, M. N. (1979) “Universal classes of hash functions”, Journal of Computer and System Sciences, 18(2), pages 143–154.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. (1990) “Introduction to Algorithms”, The MIT Press, Cambridge, United States.

Friedman, S., Krishnan, A., Leidenfrost, N., Brodie, B. C., Cytron, R. K., and Niehaus, D. (2003) “Hash tables for embedded and real-time systems”, Technical Report 2003-15, Dept of Comp. Science & Engineering, Washington University in Saint Louis.

Gonnet, G. H. (1981) “Expected Length of the Longest Probe Sequence in Hash Code Searching”, Journal of the ACM, 28(2), pages 289-304.

Liu, J. (2000) “Real-Time Systems”, Prentice-Hall, United States.

Karp, R. M., Luby, M., Heide, F. M. (1996) “Efficient PRAM Simulation on a distributed memory Machine”, Algorithmica, 16, pages 245-281.

Mitzenmacher, M., Richa, A., and Sitaraman, R. (2000) “The power of two random choices: A survey of the techniques and results”, In Handbook of Randomized Computing, Ed. Kluwer.

Parson, D. (2004) “Incremental Reorganization of Open Hash Tables”, WiP Session of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 25, 2004.

Redmill, F. (2000) "Understanding the Use, Misuse and Abuse of Safety Integrity Levels", Proceedings of the Eighth Safety-critical Systems Symposium, 8-10 February, 2000.