Effect of cache lines in array-based hashing algorithms

Ákos Dudás[1], Sándor Kolumbán[2], and Tamás Schrádi[3]
Department of Automation and Applied Informatics,
Budapest University of Technology and Economics,
1117 Budapest, Magyar Tudósok körútja 2. QB-207 Hungary
{[1]akos.dudas,[2]kolumban.sandor,[3]schradi.tamas}@aut.bme.hu

Abstract— *Hashing algorithms and their efficiency is modeled with their expected probe lengths. This value measures the number of algorithmic steps required to find the position of an item inside the table. This metric, however, has an implicit assumption that all of these steps have a uniform cost. In this paper we show that this is not true on modern computers, and that caches and especially cache lines have a great impact on the performance and effectiveness of hashing algorithms that use array-based structures. Spatial locality of memory accesses plays a major role in the effectiveness of an algorithm. We show a novel model of evaluating hashing schemes; this model is based on the number of cache misses the algorithms suffer. This new approach is shown to model the real performance of hash tables more precisely than previous methods. For developing this model the sketch of proof of the expected probe length of linear probing is included.*

Keywords: hash table; hashing; linear probing; cache-aware; performance

1. Introduction

Hashing algorithms are a popular choice in a great variety of applications for fast storage and retrieval of items. Over the years there have been many hashing algorithms presented in the literature. These algorithms are usually compared based on the expected probe lengths [1], [2], [3], [4], that is, the number of steps the algorithm has to take before an item can be inserted into the table. (This is equivalent to the number of steps it takes to find an item. Both will be referred to as probe length throughout this paper.)

It has been our observation [5], and also the suggestion of others [4], [6], that the expected probe length does not model the true performance correctly. Measuring the wall-clock execution times of the algorithms and using the expected probe length-based raking of hashing schemes we can arrive at two contradicting conclusions; in this paper we propose a solution that unify the expected probe length-based comparison and the physical capabilities of the hardware, resulting in a more precise efficiency estimation.

The fact is, that the true performance of array-based hashing schemes is effected by the physical hardware it is executed on. The expected probe length-based efficiency analysis has the implicit assumption that every probe in the probe sequence has the same cost; this is not necessarily true though. Modern CPUs have fast (but small) integrated caches that mask the latency of the main system memory. Accessing data in these caches is by one or two orders of magnitude faster than reading from the main memory. These caches speed up temporally, and which is more relevant for us, spatially local memory reads.

Algorithms that exploit these caches are called cache friendly [7]. What we propose in this paper is basically a new efficiency model that rewards cache friendly algorithms. We focus our attention to hashing schemes that use arrays; the basic idea however it generally applicable, and not just to hash tables but to other data intensive algorithms as well.

The rest of the paper is organized as follows. Section 2 presents the related literature of hash tables and their performance and complexity. The expected probe lengths of two hashing schemes are presented in Section 3 followed by the comparison of the hash tables based in the expected probe lengths and wall-clock execution times. To resolve the contradictory results new efficiency model is presented in Section 4. We conclude in Section 5 with the summary of our results.

2. Related works

Hash tables [8] store and retrieve items identified with a unique key. A hash table is basically a fixed-size table, where the position of an item is determined by a hash function. If the position calculated by the hash function is already occupied, a collision occurs. This collision can be resolved by storing the items externally (e.g. using a linked list), the approach of bucket hash tables; or a secondary hash function can be applied which calculates a new address for the item. Repeating this process until a free slot is found the algorithm traverses a probe path. The shorter this probe path is, the faster the hash table is.

In linear probing [8] the secondary hash function calculates the next address by adding a constant (usually one) to the previous address. This, of course, is not a true hash function. However, this "laziness" is what makes linear probing cache friendly [7].

There is a theoretical hashing scheme, which produces perfectly random address generation. The idea is that there

is a uniformly chosen permutation from the space of permutations over the possible addresses for each element. The initial hash function returns the first element of the permutation, the secondary hash function iterates through the permutation. This is called uniform hashing [9]. It has been shown, that uniform hashing is optimal over all open-address hashing schemes in its expected probe length. It was proven that double hashing, a practical realization of a hash scheme [4], is asymptotically equivalent to uniform hashing when the number of available addresses is large [10], [11]. It is generally thought that these sophisticated methods are superior to linear probing.

Black et al. [6], however, has shown that linear probing can have better performance than double hashing. Moreover, by tuning the parameters of double hashing to make it approximate linear probing, its performance increased as well.

Heileman and Luo [4] also conducted similar examinations and they ended up with another conclusion. According to their results, the cache friendliness of linear probing cannot compensate the disadvantage of the longer probe sequences in case of realistic data sets. They also suggested that the relation between the size of the data and the cache size is what lies behind the seemingly contradictory results. In Section 4 we will confirm their hypothesis.

3. Expected probe length based comparison

The expected probe length is used to measure the efficiency of hash tables. The formula is known for uniform hashing; in this section we show the sketch of calculating it for linear probing.

3.1 The expected probe length of uniform hashing

The expected probe length for uniform hashing [8] in an α-filled table is

$$\mathbb{E}(L^{\alpha}_{uni}) = -\frac{\ln(1-\alpha)}{\alpha} \tag{1}$$

where $\mathbb{E}$ denotes the expected value, L_{uni} is the expected number of steps needed to find a uniformly chosen element in a hash table built with uniform hashing and α is the load factor of the hash table (i.e. the ratio between the number of elements in the table and the number of all the slots).

3.2 The expected probe length of linear probing

In order to calculate the expected probe length (i.e. number of steps in takes to insert an item), first we need to understand how a single item is inserted into an α-filled hash table using linear probing. The expected probe length in a given state of the hash table is the average of probe lengths needed to insert the elements of the tables. We describe a table configuration using the following two notions in this paper. A *cluster* is a group of adjoint occupied slots. A *closed cluster* is formed from an empty slot and the cluster that precedes it. If, for a certain empty slot there is no preceding cluster (i.e. the empty slot is preceded by another empty slot), then this empty slot forms a closed cluster by itself. Figure 1 gives a graphical representation of these notions. Closed clusters cover the whole table, while clusters obviously do not.

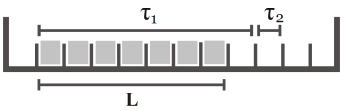

Fig. 1: A cluster (L) and two closed clusters (τ_1 and τ_2).

During insertion the probe length depends on the length of the closed cluster in which its initial address hits, since the insertion has to iterate over the items in this closed cluster. Suppose that the initial address for the new element is part of a closed cluster with length τ. The expected number of steps needed to insert the element is $\frac{\tau+1}{2}$, since the initial address is considered to be uniformly distributed over the whole table, and consequently, it is uniform restricted to the given closed cluster as well. If we knew the probability of the event that the initial address falls into a closed cluster with length τ then we would be able to calculate the expected probe length required to insert a new element.

The followings are just the sketch of how we can calculate the expected probe length for linear probing. It is out of the scope of this paper to show every step; this is merely the general idea.

Suppose there are X_i elements initially hashed to the i-th slot, $i = 1, \ldots, N$. Suppose that a closed cluster starts at the j-th position. In this case the number of elements in this closed cluster is at least X_j, and the j-th slot can hold only one. So $X_j - 1$ elements will be held in the rest of the closed cluster. The $j+1$-th slot adds X_{j+1} elements to the cluster, but it will hold one as well, so the number of elements in the cluster after the $j+1$-th slot is $X_j - 1 + X_{j+1} - 1$, and so on. When this sum reaches 0, then the cluster is closed.

We can define the stochastic process S the following way: $S_0 = 1$, $S_{i+1} = S_i + X_{j+i} - 1$. The stopping time $\tau^{(k)}$ (2) is a good approximation of the distribution of the length of the closed clusters:

$$\tau_0^{(k)} = \inf(i : S_i = 0 | S_0 = k) \tag{2}$$

The explicit formula for the distribution of the length of closed clusters can be found as

$$p(k) = \mathbb{P}(\tau = k) = \frac{(\alpha k)^{k-1}}{k!} e^{-\alpha k} \tag{3}$$

We also need the expected value of this distribution, which is

$$\mathbb{E}(\tau) = \frac{1}{1-\alpha} \tag{4}$$

The expected number of steps to find a uniformly chosen element in the table is the average of the steps needed to insert them. There are M elements in the table. When inserting the i-th element, there were $i-1$ already inserted. In other words, the i-the element was inserted into an $\alpha' = \frac{i-1}{N}$-filled hash-table. The average of these step counts gives us

$$\mathbb{E}(L_{lin}^{\alpha}) = 1 + \frac{1}{2}\frac{\alpha}{1-\alpha} \tag{5}$$

where $\mathbb{E}(L_{lin}^{\alpha})$ denotes the expected probe length for linear probing in an α-filled table.

3.3 Evaluation using probe lengths and wall-clock execution times

Figure 2 shows the expected probe length for linear probing and uniform hashing for various load factors. It is obvious that uniform hashing has a smaller expected probe length. Based on this fact, one could say that linear probing is to be neglected while choosing hashing algorithms for practical purposes.

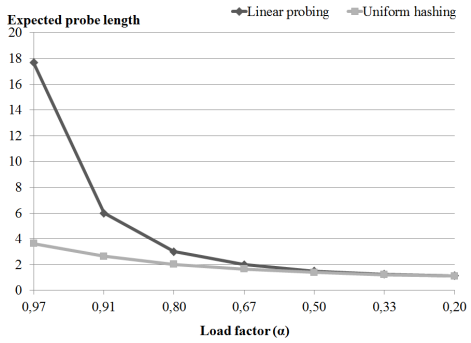

Fig. 2: The expected probe length of linear probing and uniform hashing for different load factors.

Our experimental results, on the other hand, show different results. Figure 3 plots the measured wall-clock execution times of building a table using linear probing and double hashing. Linear probing has shorter lookup execution than double hashing. This is the exact opposite of the previous result.

To resolve this contradiction a new complexity function is required, one that approximates the true performance of hash tables. The problem with the probe length based ranking is that it assumes that every probe has the same cost; instead, the characteristics of the execution environment have to be considered and integrated into the cost function.

4. Cache-line aware algorithm complexity

This section presents a simple model of memory hierarchy which is then incorporated into the cost function of the steps of the probe sequence.

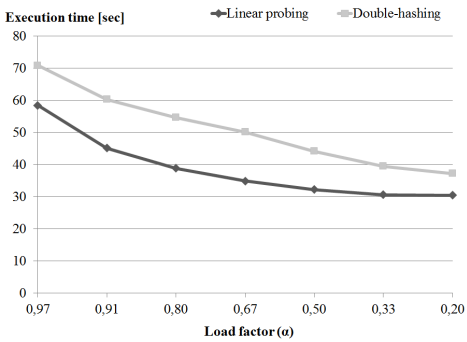

Fig. 3: The measured wall-clock execution times of linear probing and uniform hashing for different load factors.

4.1 Caches

Open hash tables span over a large block of allocated memory. This block is split into slots, which are identified by a number (memory address; i.e. indexes of the array). The address of neighboring slots are sequential, therefore in linear probing, after slot i is visited, whose address is j, the next slot, $i+1$, will have the address $j+1$. This is not true for uniform hashing; the addresses of the probe sequence will be scattered across the table. Let us explain why this is important.

In current computer systems CPUs have caches, which are fast access, but limit space storages integrated into the CPU or onto the same die. The cache stores a small fraction of the data that is stored in the system memory, therefore a small portion of the hash table also resides in the cache. Whenever the CPU requests data, it is first checked in the cache. If found, the main system memory is not queried as the cache returns the data. However, if the data is not in the cache (this event is called a cache miss), the data is loaded from the main system memory; this operation is by one or two orders of magnitude slower than reading from the cache.

An other important factor is cache lines. The memory is partitioned into small blocks, called cache lines. Whenever data is loaded into the cache, an entire cache line is loaded; the one the requested data is inside. This means, that with a single memory access it is not just the requested data that is loaded, but some neighboring addresses are read as well - at no additional cost. If the next read is in this very same cache line, it will be served by the low latency cache.

If accesses to the memory is temporarily or spatially local, the cache speeds up the algorithm by not requiring the system to read data from the system memory. When an algorithm exploits these effects, it is called cache friendly. It allows the algorithm to have lots of data requests at fraction of the cost.

4.2 Cache-line based memory model

The cost difference between accessing data from the main memory and from the cache is often neglected in

performance models. Let physical memory requests have a cost of one. Altering the usual memory model, where every access has a uniform cost, we propose a new model. In our model the blocks of the memory are grouped into lines of equal lengths. These lines correspond to the cache lines of the real system. The characteristics of these lines is that they are read as one.

In this memory model, the true performance of an application is determined not by the total number of memory accesses, but by expected number of read lines. In case of hashing algorithms this is equal to the number of probed lines, which the probe sequence accesses. In other words, the number of produced cache misses is the determining factor.

Suppose that the parameters of the memory architecture and the hash table are such that an integer number of hash table slots fit in a cache line. This parameter will be denoted with B. Figure 4 shows a scenario where three hash table slots fit into a single cache line ($B = 3$). Items with the same color are hashed to the same position by the primary hash function.

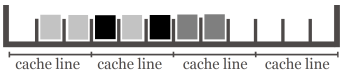

Fig. 4: The segmentation of the memory into cache lines of length $B = 3$. Items with the same color have the same initial address.

As an example, the second item from the left can be found with cost of one if linear probing is used, since the first checked slot and the final position are both in the same cache line, and the probe sequence examines no slots outside of this line.

4.3 Cache-aware cost function for uniform hashing and linear probing

Given a relatively large hash tables that does not fit into the cache (usual size of caches is 2-4-8 MB) but instead is stored in the main system memory, the CPU cache has considerable impact. A typical hash table entry consists of a unique integer id and a data pointer. This means that the number of hash slots that fit in a cache line (B) is about 2-8 entries (assuming a cache line is 64 bytes) and the number of lines that can be stored in the cache memory is negligible compared to the number of lines that are covered by the hash table.

For uniform hashing the probe sequence is a random permutation of the positions in the table. Thus, it is a fair approximation that all probes fall into one of the not cached lines. This means that every probe has a cost of one when uniform hashing is used. In other words every probe produces a cache miss. This can be formalized as

$$\mathbb{E}(C^{\alpha}_{uni}) = \mathbb{E}(L^{\alpha}_{uni}) = -\frac{\ln(1-\alpha)}{\alpha} \qquad (6)$$

where C^{α}_{uni} is the cost of a probe sequence that finds a random element in an α-filled table built with uniform hashing. The value is independent of B.

To verify this formula, Figure 5 shows the calculated values against our measured values.

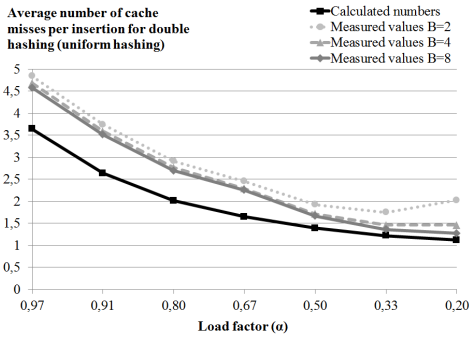

Fig. 5: The theoretical and experimental number of cache misses for double hashing (uniform hashing).

In case of linear probing the first address of a probe sequence will not be among the list of cached addresses. This means that the first probe will request a new line to be read from the memory. But the following probes have a high probability of being served from the cache, since the neighboring slots of the first probed one were cached when the first probe was performed. If the initial address is given then each step will produce a cache miss with probability of zero or one, depending on whether it is in the same cache line, or in the next one, respectively. Since the initial address of a probe sequence is uniform over the slots of the line it falls into, each of the remaining $L^{\alpha}_{lin} - 1$ steps produce a cache miss with probability $\frac{1}{B}$. Thus, we can say that

$$\mathbb{E}(C^{\alpha}_{lin}) = 1 + \frac{1}{B}(\mathbb{E}(L^{\alpha}_{lin}) - 1) = 1 + \frac{1}{B}\frac{1}{2}\frac{\alpha}{1-\alpha} \qquad (7)$$

where C^{α}_{lin} is the cost of the probe sequence that finds a uniformly chosen element in an α-filled table built with linear probing.

To verify this formula as well, Figure 6 plots the measured and the calculated values for various Bs.

4.4 Cache-aware cost model and true performance

Finally, let us compare the true performance of the hash tables, measured in terms of wall-clock execution time, with the proposed cost model.

Figure 7 shows how the expected number of produced cache misses look like (left), and the what are the corresponding measured execution times (right). The number of slots that fit in a cache line is $B = 2$, $B = 4$ and $B = 8$.

It is clear that for any given value of B there is a load factor α_B under which linear probing has lower cost and over which uniform hashing algorithms are better. From

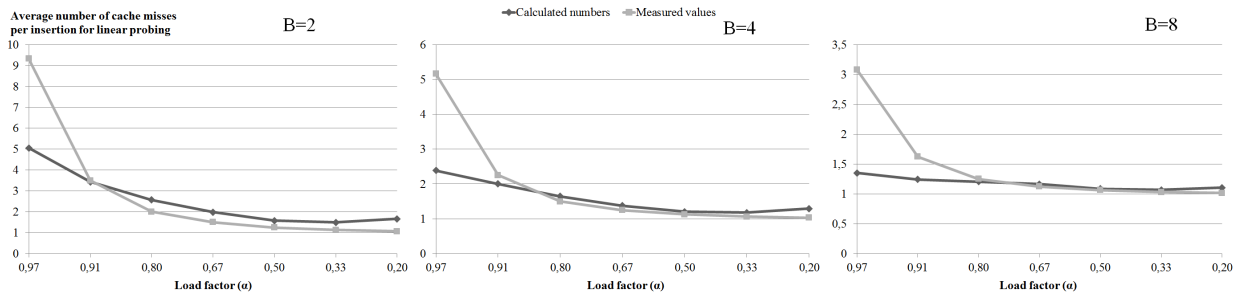

Fig. 6: The theoretical and experimental number of cache misses for linear probing.

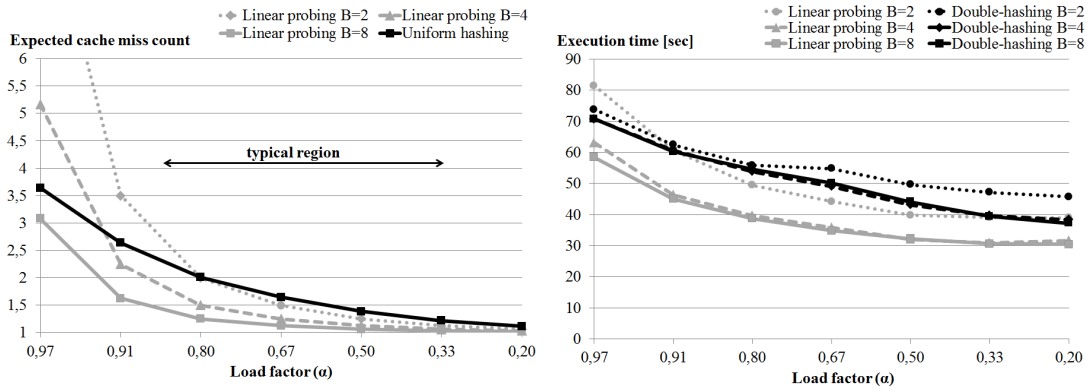

Fig. 7: The expected number of cache misses for linear probing and uniform hashing for different load factors for $B = 2, 4, 8$.

equations (6) and (7) this load factor can be obtained. In the typical operation region of $\alpha \in [0.3\ 0.8]$, linear probing has lower expected cache miss count even when $B = 2$. This is confirmed by the execution times as well.

In general, when choosing a hashing algorithm, one should consider the parameters of the hash table and memory architecture, namely parameter B should be determined and the operational load factor should be decided.

Generally, our conclusion is that the simple algorithmic step count based raking of algorithms, especially for algorithms that intensively use memory, is not sufficient. The physical capabilities of the machine that executed the algorithms should be taken into consideration, and with integrating the memory model into the cost function, a better efficiency comparison can be derived.

5. Conclusion

Hashing algorithms are usually ranked by their expected probe lengths. It has been our observation, and also published in the literature, that this is not always true. Based on previous works we know that in case of open-address hashing the performance of the algorithm is greatly effected by its memory characteristics.

We have shown that the expected probe path based efficiency comparison is not fair for linear probing, which is generally though of as an inferior choice of hashing scheme. Under real-life circumstances, however, it is able to outperform more sophisticated hash tables, such as double hashing.

Incorporating the effect of cache lines into the cost function of hashing algorithms we have presented a novel model of evaluation. This approach models the true performance of these hash tables more precisely.

Acknowledgment

This project is supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002) and by the fund of the Hungarian Academy of Sciences for control research, the Hungarian National Research Fund (grant number T68370).

The authors express their thanks to Sándor Juhász for his help as scientific advisor.

References

[1] G. H. Gonnet, "Expected Length of the Longest Probe Sequence in Hash Code Searching," *Journal of the ACM*, vol. 28, no. 2, pp. 289–304, Apr. 1981.

[2] M. V. Ramakrishna, "Hashing practice: analysis of hashing and universal hashing," *ACM SIGMOD Record*, vol. 17, no. 3, pp. 191–199, Jun. 1988.

[3] A. Pagh, R. Pagh, and M. Ruzic, "Linear probing with constant independence," *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing - STOC '07*, p. 318, 2007.

[4] G. L. Heileman and W. Luo, "How Caching Affects Hashing," in *Proc. 7th ALENEX*, 2005, pp. 141–154.
[5] S. Juhász and A. Dudás, "Adapting Hash Table Design to Real-life Datasets," in *Proc. of the IADIS European Conference on Informatics 2009, part of the IADIS Multiconference of Computer Science and Information systems 2009*, Algarve, Portugal, 2009, pp. 3–10.
[6] J. R. Black, C. U. Martel, and H. Qi, "Graph and Hashing Algorithms for Modern Architectures: Design and Performance," pp. 37–48, 1998.
[7] A. Binstock, "Hashing Rehashed," *Dr. Dobb's Journal*, vol. 4, no. 2, 1996.
[8] D. E. Knuth, *The art of computer programming, Vol 3*. Addison-Wesley, Nov. 1973.
[9] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *J. Comput. System Sci.*, vol. 18, no. 2, pp. 143–154, May 1979.
[10] L. J. Guibas, "The Analysis of Hashing Techniques That Exhibit k-ary Clustering," *Journal of the ACM*, vol. 25, no. 4, pp. 544–555, Oct. 1978.
[11] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a Sparse Table with 0(1) Worst Case Access Time," *Journal of the ACM*, vol. 31, no. 3, pp. 538–544, Jun. 1984.