Proceedings WAE'98, Saarbrücken, Germany, August 20-22, 1998
Ed. Kurt Mehlhorn, pp. 1-3

Graph and Hashing Algorithms for Modern Architectures: Design and Performance

John R. Black, Jr. blackj@cs.ucdavis.edu
Charles U. Martel martel@cs.ucdavis.edu
Hongbin Qi qi@cs.ucdavis.edu

Department of Computer Science, University of California, Davis [1]
Davis, CA 95616, USA

ABSTRACT

We study the effects of caches on basic graph and hashing algorithms and show how cache effects influence the best solutions to these problems. We study the performance of basic data structures for storing lists of values and use these results to design and evaluate algorithms for hashing, Breadth-First-Search (BFS) and Depth-First-Search (DFS).

For the basic data structures we show that array-based lists are much faster than linked list implementations for sequential access (often by a factor of 10). We also suggest a linked list variant which improves performance. For lists of boolean values, we show that a bit-vector type approach is faster for scans than either an integer or character array. We give a fairly precise characterization of the performance as a function of list and cache size. Our experiments also provide a fairly simple set of tests to explore these basic characteristics on a new computer system.

The basic data structure performance results translate fairly well to DFS and BFS implementation. For dense graphs an adjacency matrix using a bit-vector is the universal winner (often resulting in speedups of a factor of 20 or more over an integer adjacency matrix), while for sparse graphs an array-based adjacency list is best.

We study three classical hashing algorithms: chaining, double hashing and linear probing. Our experimental results show that, despite the theoretical superiority of double hashing and chaining, linear probing often outperforms both for random lookups. We explore variations on these traditional algorithms to improve their spatial locality and hence cache performance. Our results also suggest the optimal table size for a given setting.

More details on these experiments can be found at: http://theory.cs.ucdavis.edu/

1. Introduction

Helping programmers create efficient code is an important goal of the study of algorithms. Many major contributions have been made in algorithm design, but some of these results need to be revisited to account for characteristics of modern processors. Design and analysis which focuses solely on instruction count may lead to faulty designs and misleading analysis. Caching as well as other features of the machine architecture may change the best strategies for designing algorithms. The importance of these effects is increasing as processor speeds outstrip memory access time with the cache-miss penalty now over 100 machine cycles on high-performance processors.

By measuring and analyzing the performance of fundamental data structures and algorithms we hope to provide a basic foundation for the design and evaluation of more complex algorithms. In this paper we study two fundamental topics using this approach: graph algorithms and hashing.

[1] This work was supported by NSF grant CCR 94-03651.

We show that significant performance improvements can be gained by using data structures and algorithms which take architectural features into account. We first study the performance of basic data structures in isolation and then use the indicated structures for Breadth-First-Search (BFS) Depth-First-Search (DFS) [7] and hashing. We also consider ways to predict performance and give indications of the effectiveness of these predictions by measurements of both actual run time and other statistics such as cache misses and number of instructions executed.

Our results show that the data structures for graphs found in almost all the standard texts often have much worse performance than simple alternatives. Since the extra overhead for using the standard data structure can be a factor of ten or more, it is well worth considering the performance effects we study when efficiency is important.

The Dictionary problem, where keys may be inserted, deleted and looked up, is one of the most fundamental uses of computers, and hashing is often the method of choice for solving it. Thus it is important to find the best practical hashing schemes and to understand the empirical behavior of hashing. While hashing algorithms have been studied extensively under traditional cost models, there has been little prior work focusing on their cache effects.

The desire to understand how different algorithms perform in practice has led to a recent increase in the experimental study of algorithms. There have been a number of experimental studies of graph algorithms which focus on important problems such as shortest paths [9, 5], minimum spanning trees (MST) [17], network flow and matching [1, 4, 10, 19], and min-cut algorithms [6]. These experiments provide valuable insight into the performance of different algorithms and can suggest new algorithmic choices. The authors of these studies reasonably spend most of their effort on the higher level algorithm details, so these papers have typically had a very limited discussion of the basic representation issues we discuss in this paper. In addition, we hope our results will help future experimental studies by suggesting good supporting data structures.

When doing experiments it is important to understand which variables can affect results. In many experimental papers (including all of those listed in the prior paragraph), the cache characteristics of the machines used are not even listed. In addition, in some of these studies different data structures were used for different algorithms which were being compared (for example linked lists for one and arrays for another). This shows the general lack of focus on these issues by experimenters.

1.1. Related Work

Because of its importance, compiler writers have spent considerable effort on generating code with good locality [3], however substantial additional improvements can be gained by proper algorithm design. Moret and Shapiro discuss cache effects on graph algorithms in their MST paper [17]. They comment that data caching and performance is affected by the method used to store a graph. They attempt to normalize for machine effects by using running times relative to the time needed to scan the adjacency structure. More recently, several researchers have focused on designing algorithms to improve cache performance by improving the locality of the algorithms [16, 13, 14]. Lebeck and Wood focused on recoding the SPEC benchmarks, and also developed a cache-profiler to help in the design of faster algorithms. LaMarca and Ladner came up with improved heap and later sorting algorithms by improving locality. They also developed a new methodology for analyzing cache effects [15]. While we didn't directly use their analysis tools since they were for direct-mapped caches and somewhat different access patterns, our analysis does use some of their ideas.

These papers show that substantial improvements in performance can be gained by improving data locality. Our results are similar in spirit to these but tackle different data structures and target different algorithms. Also, our approach focuses more on trying to understand the basic effects architectural features can have by studying them in simple settings.

A recent hashing paper [20] develops a collision resolution scheme which can reduce the probes compared to double hashing for some very specialized settings. However, since they only look at probes rather than execution time they don't address the effects we study here.

1.2. Result Summary

We start with the most basic data structures and compare arrays and linked lists (LL) for storing a sequence of integers when the basic operation is to scan consecutive elements in the list (with scans of an adjacency list structure or linear probing/chaining in mind). In this case reading all elements of an array can be 10 times faster than the equivalent LL scan. In addition, the performance gap is larger for more recent machines compared with older ones, so this disparity may grow in the future. By studying the architecture and compiler effects which slow down the LL implementation we develop LL variants which are much faster (though still slower than an array).

For lists of boolean values (with an adjacency matrix for an unweighted graph as our target) we compare integer, character and bitpacked arrays (where all the bits in a 32 bit word are used to store individual data elements). We show that even though the bitpacked array scans have higher instruction counts (due to overhead for extracting the bits), they outperform the integer and (usually) character arrays when the non-boolean list is too large to fit in cache. The bit matrix is often twice as fast as alternatives for large lists.

We study algorithms for Depth-First-Search (DFS), Breadth-First-Search (BFS), and hashing. For the graph algorithms we show that using a bitpacked array always outperforms an integer array (often by a factor of 20+ for BFS). In addition, the bitpacked array outperforms an adjacency list except for sparse graphs. Our experiments also confirm the substantial gain from using an array rather than a linked list to represent an adjacency list. Our results suggest that the best data structure depends largely on graph size and average node degree but not on graph topology.

In our hashing experiments we show that among traditional schemes Linear Probing is a clear winner over double hashing and chaining for both successful and unsuccessful search when the access pattern is uniform and multiple table entries fit in a single cache line. We suggest alternatives to double hashing and chaining which reduce cache misses and improve performance. We are able to model some of these settings to predict performance and the optimal size of a hash table.

2. Experimental Setting

We ran our experiments on five platforms: two DECstations, a 5000/25 and a 5000/240 (henceforth referred to as DEC0 and DEC1), two types of DEC Alphas: an older one with a 21064 processor and a newer one with a 21164 processor (Alpha0 and Alpha1) and a Pentium II (Pentium). We list the cache characteristics of the machines below since those are used directly in our analysis. DEC0 and DEC1 have a 64K byte cache and use 16-byte cache blocks. Both Alphas have an 8K byte on-chip direct mapped L1 data-cache and a 96K 3-way set-associative L2 cache (21064 is off-chip, 21164 is on-chip) and both use 32-byte cache blocks [8]. The Pentium runs at 266 Mhz, has an L1 cache of 16K for instruction and 16K for data, both 4-way associative with 32-byte line size. The L2 cache is 512K bytes and has a direct 133 MHz bus. There is also a prefetch buffer which fetches cache block $k+1$ whenever the current access is to block k.

Our programs were all written in C and compiled under highest optimization using cc. We did try alternative compilers but found there wasn't too much variation in results. In reporting the results we focus primarily on the Alpha1 and Pentium II results since they are the most relevant for current (and likely future) machines. However, we also report results on the older Alpha0 and the DECstations to show the changes which occur with the move to newer architectures. Fortunately the best design choice varies little between the five machines, though the degree of benefit varies considerably.

3. Basic Data Structure Results

The following experiments study basic algorithms and data structures. This allow us to study the performance effects closely, and also to suggest simple and robust structures for others to use.

Finally, it also gives a set of benchmark routines which test the relative performance on different machines.

3.1. Arrays versus Linked Lists

We start by comparing two data structures for holding the integers 1 through n when our goal is to scan these numbers. The main data structures we consider are a length n array of 32-bit integers (ARRAY) and a linked list of n nodes each of which contains a 32-bit integer and a pointer to the next node (LL) (32-bit pointers on the DECstations and Pentium, 64-bit pointers on the two Alphas). Each LL node is allocated by a call to `malloc()`.

Once the data structures are allocated and initialized, we repeatedly process the elements in order and add up all the numbers in the list. This is used as a simple surrogate experiment to represent a sequence of consecutive memory reads (but no memory writes). The goal with respect to graph algorithms, is to represent the operation of scanning all neighbors in a node's adjacency list where the list might be represented by an array or linked list.

Table 1 summarizes our basic data structure timing results. We list the per-element time in nanoseconds and the range of values of n (the number of items in the data structure) for which this time holds (when times were similar over a range we report only the median value). LL-12 is a linked list with 12 integers and one pointer per node.

Times in Nanoseconds per Element										
	DEC0		DEC1		Alpha0		Alpha1		Pentium	
Integer List Results (Time followed by n = list size)										
	Time	n	Time	n	Time	n	Time	n	Time	n
Array	114	to 16K	71	to 16K	19	to 2K	10	to 2K	12	to 64K
	410	> 16K	114	> 16K	35	3-128K	13	3-16K	26	128K
					60	> 128K	17	32-128K	36	≥ 256K
							30	> 1M		
LL	144	to 4K	90	to 4K	34	to 256	10	to 256	24	to 1024
	1339	> 7K	400	> 7K	141	512-16K	34-70	to 4K	49	2K-8K
					387	> 16K	121-167	to 65K	119	32K
							229	> 128K	160	≥ 64K
LL-12	131	to 8K	81	to 8K	27	to 1K	14	to 1K	27	to 16K
	358	at 16K	115	at 16K	39	at 2K	17	2-16K	30	to 64K
	529	> 16K	141	> 16K	50	to 64K	26	to 128K	49	128K
					86	> 128K	34	> 128K	58	> 128K
Boolean list results										
Char	114	to 64K	71	to 64K	41	to 8K	27	to 8K	9	to 256K
	184	> 64K	81	> 64K	45	to 512K	28	to 128K	15	≥ 512K
					51	> 512K	30	to 1M		
Bit	149	to 32K	93	all	35	all	12	all	15	all
	157	> 64K								

Table 1. Timings of Basic Data Structures for 5 Processors

For example, the top left entry shows that for an array of integers, the per-element time on DEC0 is 114 nanoseconds per-element for lists of up to 16K integers (which occupy 64K bytes), but jumps to 410 nanoseconds per element when the list is much above 16K elements and exceeds the 64K byte cache.

The integer list results show ARRAY is strictly better than LL and can be more than 10 times faster on the Alpha1 (13ns versus 128ns at $n = 8$K, 17ns versus 229ns at $n = 128$K) or Pentium (12ns versus 160ns for $n = 64$K). This performance gap is almost entirely explained by cache effects. When both data structures are small enough to fit entirely in the fastest (L1) cache they have comparable running times (except on the Pentium where LL is twice as slow even on small lists). However, as soon as the size exceeds that of the L1 cache for LL, its performance jumps. This is particularly serious since the L1 cache is exceeded for quite small lists.

A close look explains this effect quite clearly. Recall that the Alpha and Pentium L1 cache use 32-byte blocks. Each cache miss brings eight 4-byte integers into the cache, and on the Pentium a prefetch starts for the next eight. The LL uses more space per data element, particularly since `malloc()` allocates 32 bytes on the Alphas and 16 bytes on the DECstations and Pentium, even though only 12 and 8 are needed. Thus, each node access is in a separate cache line (except 2 per line on the Pentium) and only one data element is brought into the cache on a miss. On all machines the better spatial locality of the array structure greatly reduces its total cache misses when the array is too large to be contained in the L1 cache.

For an array, on the Alpha the total number of L1 cache misses is roughly $n/8$ when the array exceeds the L1 cache size, while LL has almost exactly n cache misses. The 32-bytes/node for LL also explains why its performance degrades significantly when n goes to 512 on the Alpha and its memory use exceeds the size of the L1 cache, while the Array performance does not drop until n goes above 2048 (since 512×32 bytes $= 2048 \times 4$ bytes $=$ 8K $=$ size of L1 cache). There is also a second (expected) drop in performance when the data structure exceeds the size of the L2 cache (at 96K bytes) on the Alphas.

On the Pentium we also see the LL performance drop when the size exceeds the L1 cache size (1024×16 bytes), and again when its size exceeds the L2 cache size of 512K $=$ 32K $\times$ 16. However, the ARRAY performance does not drop when its size exceeds the L1 cache size, but only when it reaches the L2 cache size (at 128K $\times$ 4 bytes). This is a strong statement on the effectiveness of the prefetching for sequential access. Each time we enter a new cache line, those data are already in the prefetch buffer.

Despite the large difference in running times, profiling shows that the array and linked list scans use the same number of instructions. Thus the time difference is due to memory effects.

We also stored multiple data values in a single linked list node. This packing of data values can greatly improve the performance of the LL structure. When two data items are stored in each node, the number of cache misses is roughly cut in half since the LL nodes have the same size as in the single data-item case.

On Alpha1 two-item-packing cuts the scan time in half (when the data exceed the L1 cache size). In fact up to four 4-byte integers can be put in a node without increasing the 32-byte size allocated by `malloc()`. Using nodes with four data-items per node results in an almost 4-fold speedup. We also tried using 12 data items per node (12 is the maximum number for a 64 byte allocation by `malloc()`). This resulted in a 7-fold speedup compared to the single node LL, but this still makes it almost 50% slower than ARRAY.

Our experiments using Atom [21] to study the cache misses on Alpha1 shows very much what we predicted. When n exceeds the cache size there are almost exactly $n/8$ cache misses for ARRAY reflecting the 8-word cache block brought in by each miss. For LL there are almost exactly n cache misses since each node uses an entire 32-byte cache line. Similarly, when we pack four data items per node, we now get four data items per cache line and $n/4$ cache misses.

3.2. Boolean Arrays of Integers, Characters and Bits

We study the best way to store a boolean array. Our motivation is for an adjacency matrix of an unweighted graph (as in BFS, topological sorting, and matching). We tested: an array of 4-byte integers (INT), an array of 1-byte characters (CHAR), and an array of bits (BIT) with 32 boolean values packed into a 4-byte integer.

As in the prior section, our test is to step through the array sequentially and add up all the values (in this case the bits). Profiling this code shows that the bit scan takes 50% more instructions than for the integer array. However, the improved locality more than makes up for this extra work. We could have also sped up the bit extraction by exploiting word-level parallelism (e.g. by table lookup for 8 or 16 bit blocks), but we avoided this since it may not generalize to other uses of the boolean array.

3.3. Results

The boolean list results of Table 1 show the timing effects for our settings. INT had the same results as Array (top table line). The only difference is adding ones instead of integers 1 through n. For the bit array the time per element is almost constant on each of the machines for all array sizes. This is not surprising since a cache line brings in 128 (DEC) or 256 (Alphas/Pentium) bits so any cache miss penalty is amortized over so much work it has a negligible effect. For CHAR there is only a small variation in per-element cost on the Alphas and DEC1, and a 50% change in cost on the DECstation and the Pentium. The higher CHAR cost on the Alphas reflects their extra cost for byte-level operations. We expect only a small change in cost as n varies since 32 data elements are being brought in with a cache line on the Alphas and Pentium.

Thus BIT is an attractive solution for large boolean arrays, and has its biggest gains over an integer array on the newer machines. Two other plus factors for BIT: (1) accesses to BIT will be less likely to evict other data items in the cache, and (2) accesses may be able to exploit word-level parallelism to reduce running time (as we do in our BFS and DFS programs). However, if an application only uses a few bits of the word in a window of time, then the performance may be worse than in our tests.

4. Graph Algorithm Results

Our graph experiments focused on the main graph traversal algorithms: DFS and BFS. We show that the performance predicted by the analysis above is exhibited by these algorithms.

We ran experiments on graphs from Knuth's Stanford GraphBase [12] and internally generated random graphs. GraphBase generates a variety of graphs which are both standard and available. We considered a range of graphs, but there was little difference in performance between random and structured graphs of the same size and density. This suggests that our results are largely independent of topology and rest primarily on the graph's density as discussed below. Thus we present here a representative subset taken from random undirected unweighted graphs with edges uniformly distributed over the vertices.

We focused on five data structures to represent our graphs: three adjacency lists, and two were adjacency matrices. The three adjacency list implementations were (1) a linked list, called "LL" above, (2) an array-based adjacency list [9], and (3) a "blocked" linked list where each node of the list contains multiple data items. The two adjacency matrix implementations were (1) a two-dimensional array of integers, and (2) a two-dimensional array of bits.

Results were collected on all platforms, but we focus on the Alpha1 results, noting significant differences where they occur.

In BFS when we visit a vertex we check all of its neighbors to determine which are unvisited (and add these to a queue). This operation is close to the scans discussed in section 3, so we expect the performance to generally reflect the results we saw for those basic data structures. In DFS we check neighbors until we hit an unvisited vertex, then change vertices, so we see less spatial locality than with BFS. DFS is further from the basic experiments, but still close enough for those results to provide good indications of performance.

4.1. Overall Results

For an n node graph with average vertex degree d our experiments showed that d and n/d were the overriding factors affecting the relative performances of the algorithms on the various data structures. Because DFS and BFS each look at an edge only once, the issue of whether the graph fits in cache is less important than if the algorithm examined edges multiple times. The degree d affects the spatial locality (when a block of data is brought into cache, how many are neighbors of the current node), and when d is small the fraction of time spent on edge scanning is also smaller, so the total speedup of the algorithm due to improving the edge scans is reduced. The ratio n/d is critical for the classic tradeoff between adjacency matrix structures and adjacency list structures.

We did experiments on a range of graphs, but present here a representative set of data for a family of random graphs, all having 1,000 nodes and between 6,000 and 20,000 edges and generated by the SGB. We present figures for these relatively small graphs for several reasons: (1) the SGB allows a standard widely-available source for graphs, and we were not able to produce much larger random graphs with it on our platforms, (2) we were able to run experiments on all data structures at this size (we could not test an integer-based adjacency matrix for a 20,000 node graph since we don't have 160M of memory to store it), and (3) experiments on larger graphs exhibited consistent performance with these smaller graphs: using our internal graph generators (which are fast and memory-efficient) we tested random graphs with 5,000, 10,000, and 20,000 nodes; these graphs had n/d values in the neighborhood of their tradeoff points (see below). For example, on the Alpha1 platform, we tested graphs of 20,000 nodes and from 5.6 million to 6.4 million edges.

On all platforms the relative speedups between competing structures remained constant and the n/d tradeoff point discussed below also remained constant, for a given platform. On the Alpha1 the tradeoff constant was about 33, while on the Pentium it was about 29.

By holding the number of nodes constant and varying the number of edges in the experiments presented, we were able to find which of the various data structures performed best for each of the graph densities. A plot of the running times on Alpha1 is presented in Figure 1.

The Y axis is the time, in seconds, for a BFS traversal. The integer adjacency matrix results are omitted since they would plot a horizontal line at 0.055 level, well off the top of our figure.

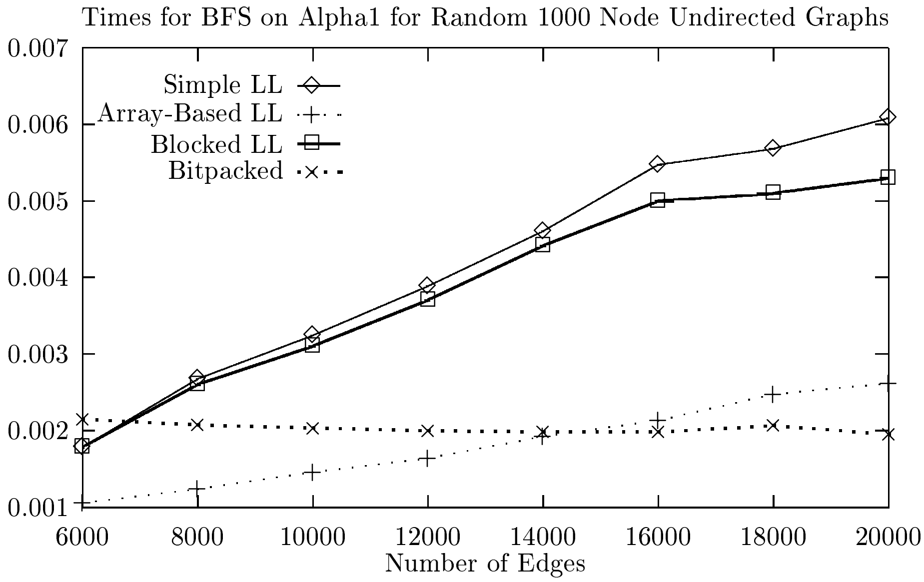

Figure 1

4.2. *Bitpacked versus Integer Adjacency Matrices*

Our bitpacked representation does vastly better than an integer array for all graphs tested, typically running in 1/20 the time or less. There are several reasons for this large disparity.

A principal cause is cache effects as discussed in section 3.2. The bitpacked representation also allows us to use several optimizations which exploit the inherent parallelism of word-level operations. For example, when visit the neighbors of a vertex, we can mark these as visited by a bitwise-OR of the adjacency bits into the bit array which tracks the visited vertices. We potentially mark 32 neighbors as visited in parallel.

The time required for packing and unpacking bits can be reduced substantially by a few tricks, which we also employed. It is not always necessary to shift and mask bits: we used techniques which extract a piece of the bit array (typically 4 or 8 bits) and then use a table lookup to quickly finish the operation.

4.3. *Simple LL versus Array-Based and Blocked LLs*

In Figure 1, we see that an array-based adjacency list always beats the LL structures, and a blocked LL beats a simple LL. The gains in adopting the array-based or blocked LLs are not as large as they were in section 3, since a smaller portion of the overall running time of this test is involved with scanning the adjacency lists. In section 3 the only task was to scan the list and add up numbers.

4.4. *LLs versus Bitpacked Adjacency Matrix*

The last two sub-sections gave clear indications regarding the correct choice of data structure when deciding between the pair being compared. In this setting, however, the better performer depends on the graph. This is completely expected: for sparse graphs with large n/d, the LL implementations need to scan only d items on average to process a given vertex whereas any adjacency matrix representation will require a scan of n objects. However, as we see from Figure 1, as n/d falls, the times for the LL structures increase linearly while the adjacency matrix times remain roughly constant. For our experiments the break-even point for the simple LL and the blocked LL is when n/d is about 70 (this is 7000 on the horizontal axis of Figure 1) and for the array-based LL the break-even point occurs at around $n/d = 33$. These numbers held constant over all graphs tested on Alpha1. On the other platforms there were similar constant trade-off points.

We should note that the algorithms tested here, BFS and DFS, were quite simple and perhaps atypical of many graph algorithms in that each data object was read only once. Although we did not experiment with other algorithms, we suspect that algorithms which access objects more than once, particularly with good spatial locality, would benefit even more from the nice behavior of the bitpacked adjacency matrix. This requires further exploration.

5. Hashing

Chaining, double hashing and linear probing [7, 11] are the three most classic hashing algorithms. Traditionally, chaining and double hashing are considered superior to linear probing because they disperse the keys better and thus require fewer probes. Our experiments show, however, that at least for uniform accesses, linear probing is fastest for insertions, successful searches and unsuccessful searches. This is true unless the table is almost 100% full or can be stored entirely in the L1 cache. Since it is rarely a good idea to have the hash table that full, linear probing seems to be the clear winner in the settings we considered.

5.1. *Experimental Setting*

We did experiments on both Alphas and the DECstations, but we focus here on the results for Alpha1 (the results for the other machines were generally similar). We used 8-byte keys on the

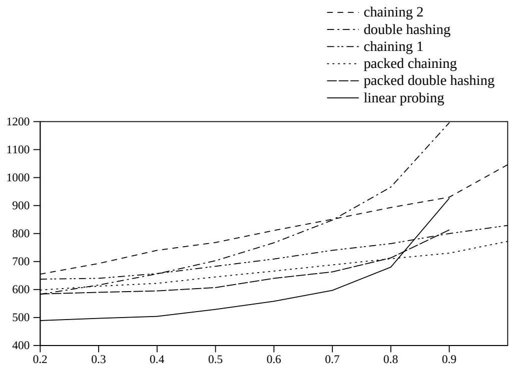

Figure 2. Time expense of insertions on the ALPHA. The X-axis is the load factor. The Y-axis is the average time in nanoseconds to insert a key. Insertions start with an empty table and end at the load factor on the X-axis. The table has 4M key slots.

Alpha1 and our tables contained only keys (plus 8-byte pointers for chaining). We chose hash(key) = key mod T as the hash function, where T is the table size. Table size is the maximum number of keys a table can hold. We distinguish table size from table space, which is the memory space a table occupies. To get a table with n keys we generate n random 32 bit integers and insert them into an empty table using the appropriate hashing scheme. Since we used random keys, the modulo function sufficed as our hash function.

5.1.1. Comparing Different Schemes

The performance of a hashing scheme will largely be determined by the number of probes and the number of cache misses. Double hashing (DH) and Chaining (C) do a good job of randomly distributing keys through the table. This reduces their number of probes, but also gives them very poor spatial locality. Successive probes looking for a key are almost certain to be in different cache blocks. Linear probing (LP) has excellent spatial locality but uses more probes. Figure 2 shows clearly that LP beats both DH and C for random insertions as long as the table is less than 80% full. The same results hold for successful search since the same probe sequences are used, and LP is also the best scheme for unsuccessful search, though its margin is somewhat smaller. Linear probing continued to be the best scheme over a wide range of table sizes where the number of keys did not fit into L1 cache but the table fit in main memory. When the key set size was small enough to fit entirely in L1 cache LP lost its advantage. LP's performance is well explained by its fewer cache misses which is detailed in the next section and by the analysis in section 5.3.2.

5.2. Analysis of the Number of Cache misses

We used Atom [21] to simulate our algorithms' cache behavior. We simulated a direct-mapped single-level cache which is the same as the DECstation cache. We chose to simulate the DECstation cache because a single-level cache would make our experimental results easier to analyze.

The number of cache misses roughly tracked the timing performance in Figure 2 and the analysis we do in section 5.3.2. Linear probing is a bit better than the cache-miss curves suggest and chaining somewhat worse. This may be due to the simpler address calculations in linear probing or due to easier optimizations of non-pointer based code by the compiler.

5.3. New Hashing Schemes

We designed variants on DH and C to improve their spatial locality. In *packed double hashing PDH*, we hash a key to a table entry which contains multiple key slots. The number of key slots in a table entry is set so that a table entry has exactly the same size as a cache block. When a key x is hashed to entry i, the slots in that entry are examined sequentially. If x is not found and the entry has no empty slots, we compute a second hash function $h_2(x)$ as in double hashing, and examine entries $(i + increment) \mod T$, $(i + 2increment) \mod T$, ..., until x or an empty slot is found (note that now T represents the number of table entries, each of which contains multiple key slots).

In *Packed Chaining (PC)*, instead of storing one key and one pointer in a table entry or list node, we store multiple keys and one pointer, so that a table entry or list node has the same size as a cache block. When a key is hashed to a table entry, the key slots in that table entry are examined sequentially. If the the key or an empty slot is not found, we next check successive nodes in the associated linked list.

Figure 2 also reports the performance of PDH and PC. We see that they both improve on their normal versions, but they still perform less well than LP for moderate load factors. Measurements show that PDH and PC have the fewest cache misses of any of the schemes, however, they use more probes than linear probing for load factors below 0.8. However, these schemes do have the advantage that their performance is more stable as the load factor increases compared to LP.

Our successful and unsuccessful search results assume uniform access patterns. If the access pattern is skewed (as is true in many real applications) the number of cache misses will decrease and therefore chaining and double hashing may perform better. Minimizing probes will also be more important if key comparison is more expensive, and locality for linear probing will be reduced if other data is in the hash table or the keys are larger.

5.3.1. Optimal Table Size

In addition to the collision resolution scheme, the other main design choice in hashing is the table size. In a cache setting there is a tradeoff since a larger table will reduce the number of probes but may also increase the percentage of probes which are cache misses. We see in Figure 2 that the performance of all the hashing schemes monotonically decrease as the load factor increases. This same trend was seen for successful and unsuccessful search and for a wide range of key set sizes which did not fit in L1 cache. However, as in Figure 2, the performance was usually fairly flat for load factors in the 0.1 to 0.4 range.

It is also worth noting that since each scheme is best at relatively low load factors, and linear probing is a clear winner at these low load factors, linear probing looks to be a clear winner when one can predict the number of keys, and the desired hash table size is bigger than the L1 cache but fits in main memory.

5.3.2. Theoretical Analysis

The expected number of probes for uniform hashing is well analyzed [11], but adding a cache complicates things. However an approximate analysis helps explain the performance in Figure 2.

Assume we have n keys in a table of size T, and cache capacity C in units of table entries. Let $\alpha = n/T$ be the load factor and P the cache miss penalty. We assume a memory system with a single cache, where B is the number of table entries which fit into a cache block (so $B = 4$ on the Alpha1 if we store 8-byte keys in the table).

If T is much larger than C we assume that each access to a new cache block is a miss. Thus every probe for DH and C is a miss, and for LP if we do k probes in a single lookup, the first probe is a miss and subsequent probes have a $1/B$ chance of hitting a new block. Thus the expected number of misses is $1 + (k - 1)/B$.

We now consider DH and LP using the classic results for their expected number of probes for successful search. When $n/T = .5$ the expected number of probes (and cache misses) for DH is about 1.386 and 1.5 probes for LP. Thus if $B = 4$ as in our experiments, the expected number of cache misses for LP is $1 + .5/4 = 1.125$. When $n/T = .8$, DH takes 2 probes and LP 3, so LP has 1.5 expected misses compared to DH's 2. Thus it is not surprising LP is faster at both these load factors. A similar analysis can be done for chaining as well.

We can analyze *unsuccessful* searches for Double Hashing more exactly since each probe can be viewed as hitting a random location in the hash table, and each location in the hash table is equally likely to be in cache (these properties are not true for any other setting). For $T > C$, the probability that each location probed is not in the cache is approximated by $\frac{(T-C)}{T}$ (*). The expected number of probes for a random unsuccessful search is $\frac{1}{1-\alpha}$ [11]. So the expected cost of a random unsuccessful lookup is $\frac{1}{1-\alpha}\left(1 + \frac{T-C}{T}P\right)$

To study the behavior of this function with respect to T we take its derivative which is

$$\frac{PC - (P+1)n}{(T-n)^2}$$

The most interesting feature of the derivative is that it is always negative when $n > C$. Therefore if the keys do not fit in the cache, the expected cost keeps decreasing as we make the table bigger. This is true regardless of P, the cache miss penalty. We can extend this analysis to a two level cache as well, which shows that if n is larger than the size of the $L2$ cache it is optimal to keep increasing the table size (presumably up to the point where paging effects start and the models break down). If the key set is bigger than the L1 cache but smaller than the L2 cache, the models suggest setting T to the size of the L2 cache.

To test the predictions of the models, we used a key set which was larger than the $L2$ cache and varied the table size. The expected time for a random unsuccessful search did decrease as the table size increased, and at approximately the rate suggested by the models.

Unfortunately, other settings are more complex to model exactly. Consider random successful searches in double hashing. The expected number of probes for a random successful search is well known, but the probability that a probe will be a cache hit is more complicated than in the prior case. First, only those cache blocks which contain at least one key will ever be accessed during a successful search. Thus equation (*) is immediately invalid if we perform only successful searches. In addition, cache blocks which contain different number of keys have different probabilities of being in the cache. Consider the case where there is room for 4 keys in a cache block. A block B_4 with four keys is approximately four times as likely as a block B_1 with only one key to be in the cache, since it is almost four times as likely a key in B_4 was hit recently than the key in B_1.

6. Conclusions

We show that experimental studies of basic data structures provide useful insight into performance. This can be used to choose (and design) the proper data structures for larger applications.

There is considerable interesting followup work suggested by our results. For graphs the most immediate is to apply our design suggestions to other graph algorithms. Matching and unit Network flow are the most direct uses of our work since they use multiple scans of the arcs in an unweighted graph, possibly with multiple calls to BFS/DFS. Yet another extension is to apply a similar investigation to other basic data structures.

For hashing there are several important additional areas to study. First, it is important to consider various data sizes associated with the keys and larger keys such as strings. Second, it would be good to consider skewed access patterns, which occur quite often in real applications. Third, it would be good to study hashing when other memory intensive operations are being used.

References

[1] R. Ahuja, M. Kodialam, A. Mishra and J. Orlin. Computational testing of maximum flow algorithms. Sloan working Paper, MIT, 1992.

[2] Eli Biham. A fast new DES implementation in software. Technion, Computer Science Dept. Technical Report CS0891-1997.

[3] S. Carr, K. Mckinley, and C. Tseng. Compiler optimizations for improving data locality. In *Sixth ASPLOS*, 252-262, 1994.

[4] B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi. Augment or push? A computational study of bipartite matching and unit capacity flow algorithms. Technical Report 97-127, NEC Research Institute, Inc., August 1997.

[5] B. Cherkassky, A. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming,* Vol. 73, 129-174, June 1996.

[6] C. Chekuri, A. Goldberg, D. Karger, M. Levine, and C. Stein, Experimental study of minimum cut algorithms. Technical Report 96-132, NEC Research Institute, Inc., October 1996.

[7] T. Cormen, C. Leiserson, R. Rivest. Introduction to Algorithms. McGraw-Hill, 1990.

[8] Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual. Digital Equipment Corporation, Maynard, MA, 1997.

[9] G. Gallo and S. Pallottino. Shortest paths algorithms. *Annals of Operations Research,* vol. 13, pp. 3-79, 1988.

[10] D. Johnson and C. McGeoch Ed. Network Flows and Matching. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1993.

[11] Donald Knuth. Sorting and searching, the art of computer programming, Volume 3. Addison-Wesley Publishing Company, 1973.

[12] D. Knuth. The Stanford GraphBase. ACM Press, 1994.

[13] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *Journal of Experimental Algorithms*, Vol.1, 1996.

[14] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. In the *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 370-9, 1997.

[15] A. LaMarca and R. Ladner. Cache Performance Analysis of Algorithms. Preprint, 1997.

[16] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: a case study. *Computer*, 27(10):15-26, 1994.

[17] B. Moret and H. Shapiro. An Empirical Assessment of Algorithms for Constructing a Minimum spanning tree. *DIMACS Series in Discrete Math and Theoretical CS,* vol. 15, 99-117, 1994.

[18] R. Orni and U. Vishkin. Two computer systems paradoxes: serialize-to-parallelize and queuing concurrent-writes. Preprint 1995.

[19] J. Setubal. Sequential and parallel experimental results with bipartite matching algorithms. Technical Report EC-96-09, Institute of Computing, University of Campinas, Brasil, 1996.

[20] B. Smith, G. Heileman, and C. Abdallah. The Exponential Hash Function. *Journal of Experimental Algorithms,* Vol.2, 1997.

[21] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *ACM Symposium on Programming Language Design and Implementation*, 196-205, 1994.