Cache Conscious Static Search

Gabriel Deal

May 17, 1999

Abstract

A cache conscious static search data structure is presented. The advantages and disadvantages of this data structure over traditional binary search are investigated. Cache simulations and execution timings demonstrate the performance gains in using this cache conscious design.

1 Introduction

Modern computers have a memory hierarchy that starts with a small amount of very fast memory and the progresses through increasingly larger and slower memory levels (a simplified view of this is registers and then the cache at the top of the hierarchy followed by main memory and then disk drives). Since processor speeds are increasing more swiftly than that of memory, the cost of accessing main memory is becoming more expensive and so it is becoming increasingly important to utilize the cache as efficiently as possible. Many algorithms, designed before caches were a concern, can now be modified to run faster by utilizing the cache better. The binary search, first introduced in 1946 according to Knuth [1], is one such algorithm. This paper introduces a cache conscious static search (CCSS) algorithm that yields a speedup over the traditional binary search.

2 Related Work

In [2] Sanders introduced a priority queue tuned for cached memory. Acharya et al. present a data structure for cache efficient trie searches [3]. LaMarca and Ladner improve the cache performance of several fast sorting algorithms in [4]. In [5] LaMarca and Ladner present a general method of predicting cache performance and use it to analyze a cache efficient priority queue using d-heaps. The idea behind the cache efficiency of the d-heap and the CCSS are so similar that at first glance the speedups seem like they should be very close. But while the CCSS and d-heap both reduce instruction count and increase cache hits, the d-heap takes fuller advantage of locality, since utilization of the keys in a node is 100% in d-heaps while only approximately $log_2 d$ keys in a node are read in the CCSS.

3 The CCSS Data Structure

The organization of the CCSS data structure is similar to a B-tree with full nodes (with the possible exception of one partially-full node), but, unlike B-trees, this data structure uses implicit pointers as in d-heaps (figure 1). Like B-trees, each node in a CCSS tree can contain a set number S of keys (the keys are a fixed size) and can have up to $S+1$ children. With this organization, in a tree with S keys per node in an array with starting index 0 (each index corresponds to one key), the first child of the node whose first key is in array position k will start at position $(k+1)+S(k+1)-1$, and the ith child ($i = 0 .. S-1$) of that node will start at position $(k+1)+S(k+1)-1+Si$. It is important that the array starts at index 0 so the nodes will be aligned on cache blocks.

To initialize the CCSS tree, first sort the data, then perform an in-order traversal of the tree, writing data to each key after all keys less than it have been written (example code in figure 2). This produces a

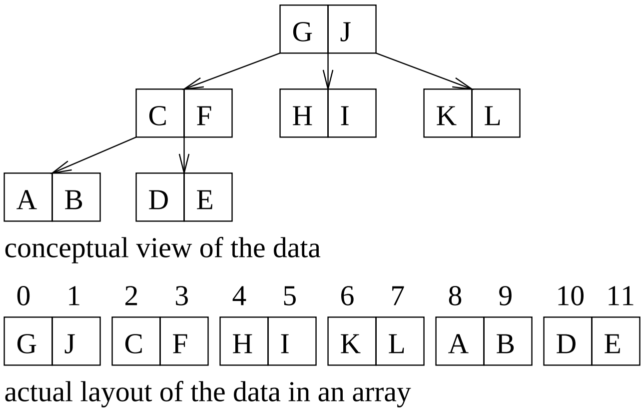

Figure 1: An example tree with two keys per node and the corresponding layout of the tree in an array.

```
void
filltree(src, targ, i_ptr, node, key_size, order, num_elements)
   char  *src;          /* sorted source array */
   char  *targ;         /* array to create CCSS data structure in */
   int   *i_ptr,        /* index into 'src' */
         node,          /* index into 'targ' for the current node +1 */
         key_size,      /* size of a key in bytes */
         order,         /* number of keys in a node */
         num_elements;  /* total number of keys to read */
{
   int   kids,          /* index into 'targ' for the children nodes */
         cnt,           /* count how many children we have visited */
         stop;          /* number of children to visit */

   if(node > num_elements)
      return;

   if(num_elements - node + 1 < order)
      stop = num_elements - node + 1;
   else
      stop = order;

   kids = node * (order + 1);
   for(cnt = 0 ; cnt < stop ; cnt++)
   {
      filltree(src, targ, i_ptr, kids + cnt * order);
      strncpy(
         targ + (node + cnt - 1) * key_size,
         src + (*i_ptr) * key_size,
         key_size
      );
      (*i_ptr)++;
   }
   filltree(src, targ, i_ptr, kids + cnt * order);
}
```

Figure 2: C function to fill the CCSS data structure.

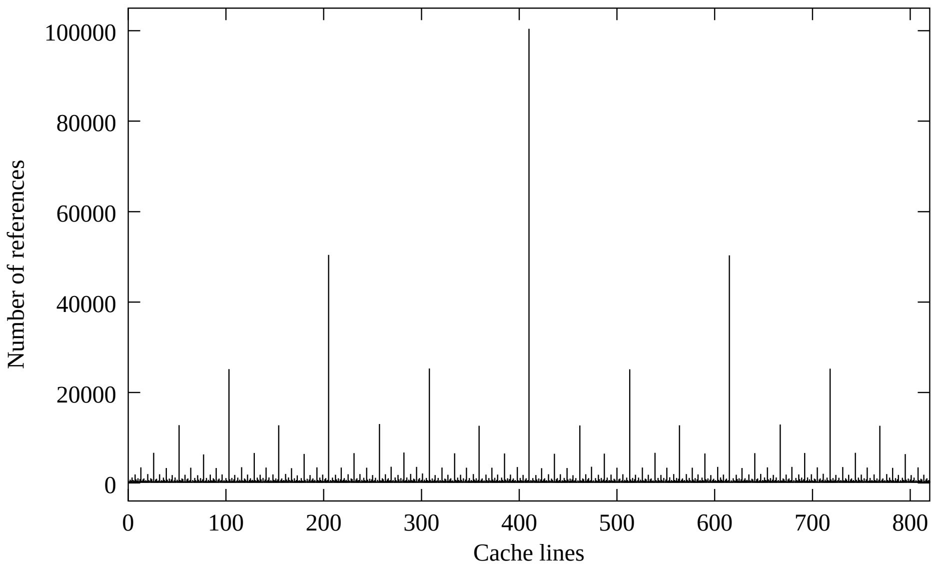

Figure 3: Reference pattern in a traditional binary search on a data set with 6560 elements in it.

complete tree (all levels completely full except the bottom one whose elements will all be flush with the left side of the tree) as long as no key is written to an index greater than the total number of keys to write.

Searches on the tree are carried out in the same manner as in a B-tree, the keys (call the keys $k_0..k_{S-1}$) in the node are searched (with a binary search) and either the key is located in this node or we find keys k_i and k_{i+1} such that $k_i < key < k_{i+1}$. In the latter case we calculate the position of the ith child node and continue searching there or halt if child i does not exist.

The fixed number of keys in each node makes it possible to optimize search code for any fixed node size. Instead of using general binary search within a node, the search can be performed explicitly without using loops. This increases the code size and is less general, but it has the advantage of reducing the instruction count by eliminating extra assignments to variables and division operations. Unless otherwise noted, all timings and simulations use optimized search code.

The source code for the implementation used in this paper, along with the simulation code, is available at http://www.yellowleaf.org/gmd/software/cache/.

4 Comparison of the Binary Search Versus the CCSS

When searching a CCSS tree with nodes the size of a cache line, each search within a single node will cause at most one cache miss on the data in the tree (assuming nodes are aligned on cache lines). So the number of misses per search is bounded above by the height of the tree. Whereas, except for the last few references when the search is narrowed down to a small portion of the array, a binary search can cause a cache miss on every reference in a search. So the CCSS algorithm can be expected to improve the miss rate by a constant factor over the binary search when searching data that is not initially in the cache. And when the data set is larger than the cache, the CCSS will also generate fewer cache misses per search regardless of whether the search data is hot in the cache or not.

Additionally, certain data set sizes can cause the traditional binary search algorithm to generate significantly more cache misses than sets with only a few more or less elements in them. For instance, if the data set D is twice the size of the cache, the two possible locations for the second reference in the search (the keys near array positions $\frac{1}{4}D$ and $\frac{3}{4}D$) will map to the same cache line, so subsequent searches will have a 50% chance of a cache miss on the second key they reference. This alignment property does not just exist for a few special cases, to illustrate this, figure 3 displays the total number of references made to each key during a traditional binary search. (In figure 3 and all other figures, the queries are uniformly distributed throughout the data set, so the likelihood that a key will be the target of a search is the same for all keys.)

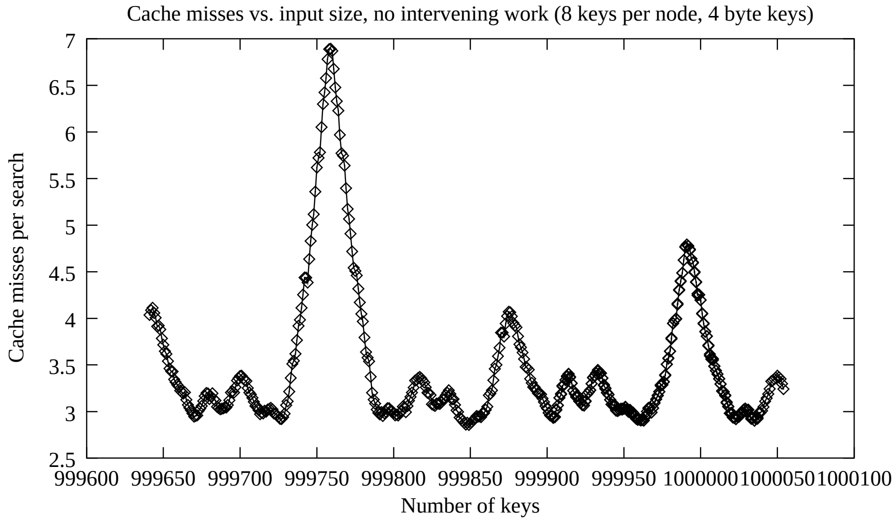

Figure 4: A close-up of the cache misses in a binary search.

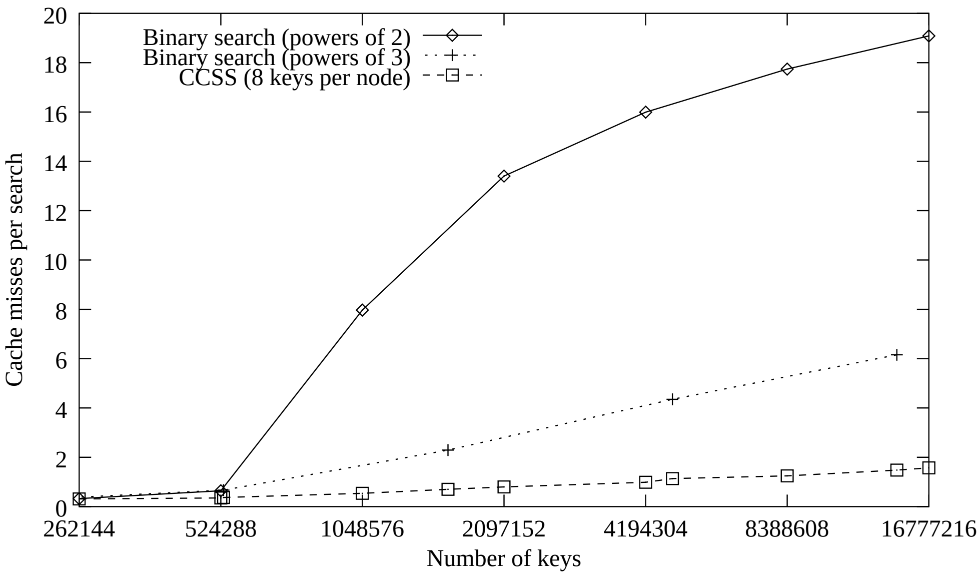

Figure 5: Comparison between a binary search on a data set with the number of elements being a power of two vs. a set with the elements being a power of three.

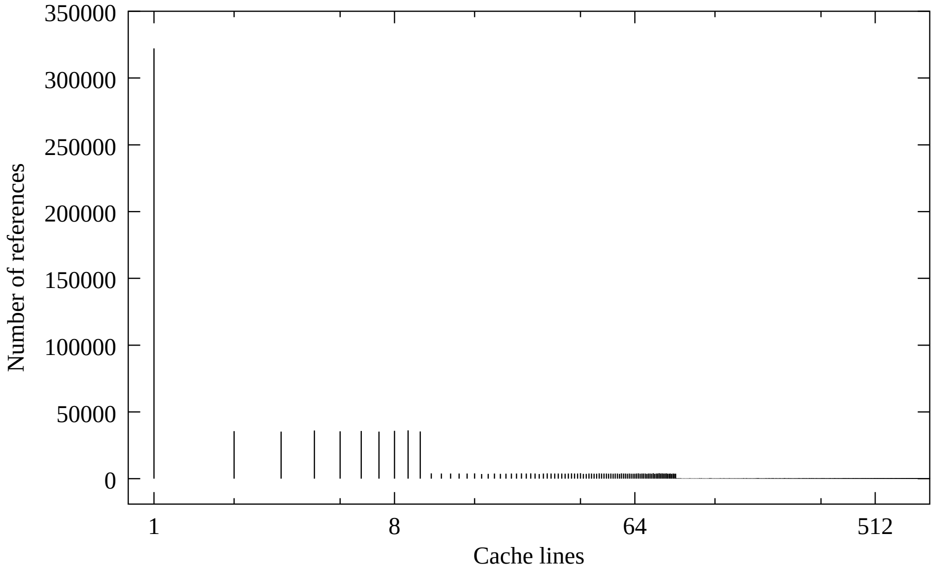

Figure 6: Reference pattern in a CCSS tree with 6560 elements (4 levels). This figure maps one node to one cache line and vice versa, so the effect of the cache line mappings wrapping around the cache are not shown.

To see the alignment effect, imagine how figure 3 will map to the cache as the number of elements increases; when the size of the data exceeds the size of the cache, the rightmost side of figure 3 will map to the same area of the cache that the leftmost side does, so at times keys with very high reference counts will map to the same cache lines, and then when the number of keys increases even more, these keys will no longer be aligned and the miss rate will decrease. Figure 4 is a cache simulation of a traditional binary search showing this sensitivity to the input size. Figure 5 contrasts extreme worst case cache performance with the normal behavior, the top line is the number of cache misses for data sets when the number of elements is a power of two, the middle line is for powers of three, and the lower line is the CCSS with points in both the powers of two and three.

In the CCSS the cache conscious layout of the data in the array causes the number of references to the array to decrease monotonically as we observe the reference rate at increasing indices (figure 6). Moreover when the search data exceeds the size of the cache, the new data that maps to the same cache lines as the heavily used data at the head of the array will be the least frequently used data in the data set, thus minimizing the new potential conflict misses. So the resonances exhibited by the traditional binary search do not occur with the CCSS.

The CCSS has its disadvantages. Since the CCSS may not divide the area it searches in half each time, due to an unbalanced tree (reference counts for the special case of complete trees are close to the count for the binary search) or the search pattern within groups of keys, it has a higher reference count despite its lower miss rate (figure 7). For poorly balanced trees with a small number of keys, the instruction count is also higher for the CCSS for the same reasons (although an optimized CCSS can drop the instruction count below that of the binary search, figure 8). When initializing the CCSS tree, if the data is not already sorted, then it must be sorted in an array and then read into the tree, doubling the amount of total storage space normally required. These, in addition to added programming complexity, are the main disadvantages of the CCSS.

5 Experimental Methodology

I did my cache simulations with DEC's Atom [6] simulator, simulating a 2MB direct mapped cache with 32 byte cache lines. The data in the tree was the even integers 0 to $2(N-1)$ where N is the total number of elements in the data set, I generated the keys to search for using rand() seeded by the $RANDOM environment

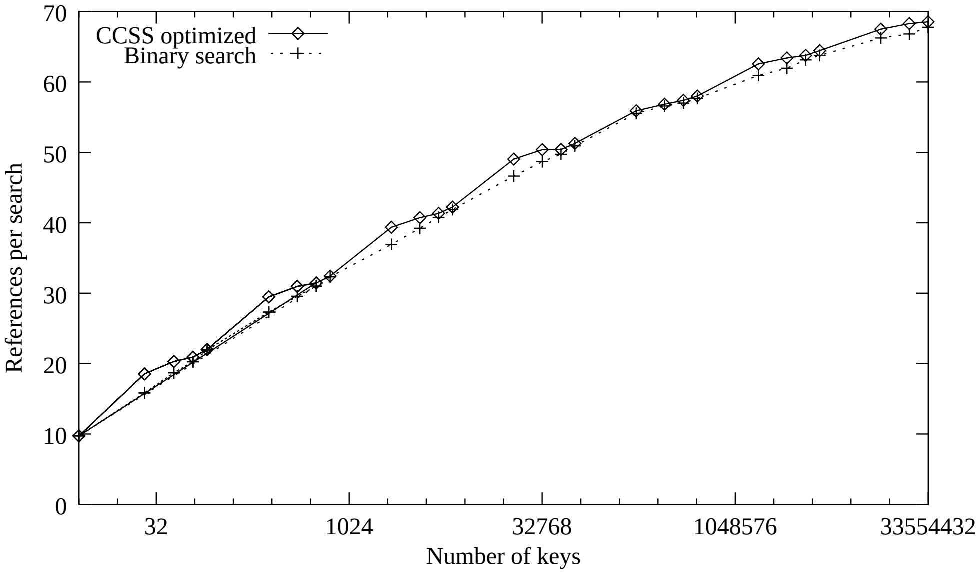

Figure 7: References per search, 8 keys per node, 4 byte nodes.

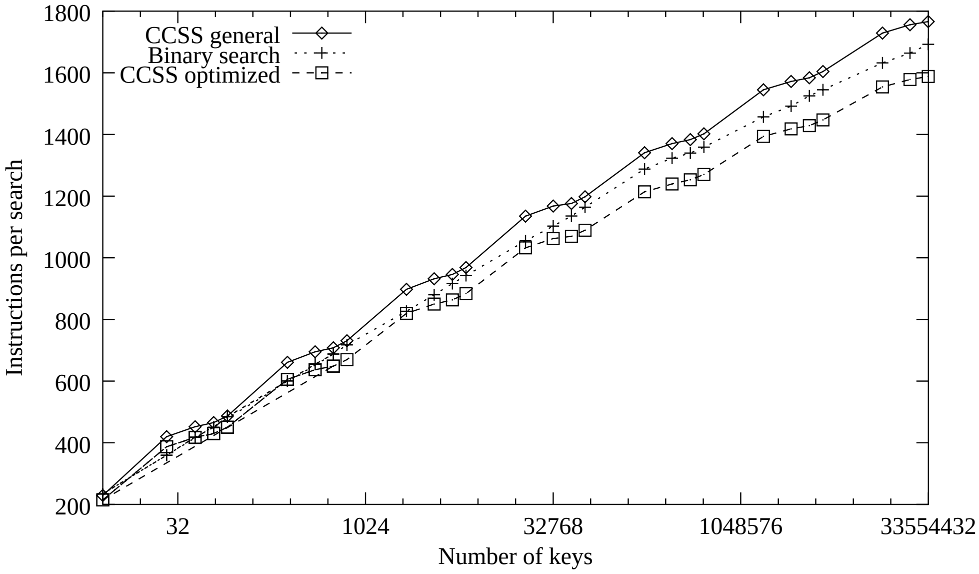

Figure 8: Instructions per search, 8 keys per node, 4 byte nodes.

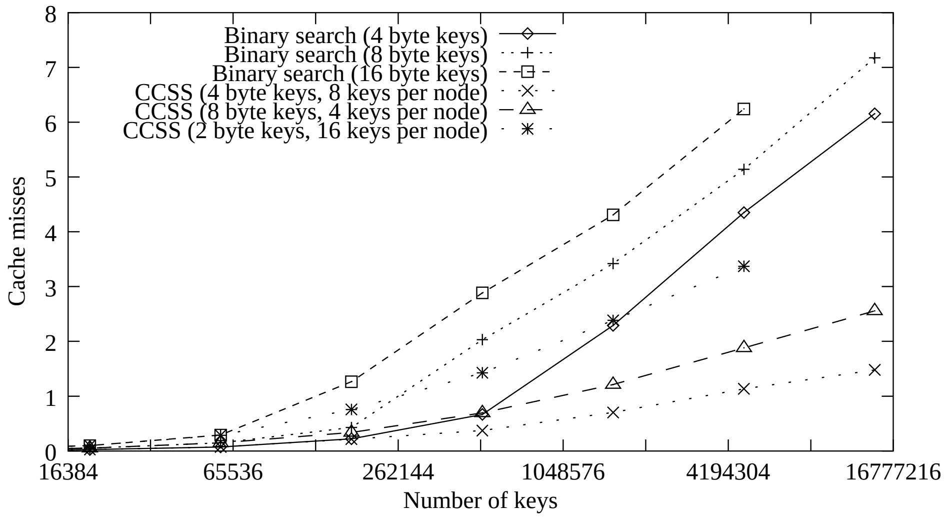

Figure 9: Cache misses versus input size, data points are powers of ten on the x-axis to avoid worst binary search cases.

variable provided the UNIX shell "bash". Each data point in the simulation graphs was generated from the average value of 100,000 separate searches on data that was in the data set (all searches were successful). All points in the execution timings were the average of 10 points (minus any obvious outliers), each of these 10 points was generated by the average time over 1,000,000 successful searches. To generate the "outside work", I allocated an array the size of the cache and between each search I read one byte from N contiguous 32-byte sections of the array, then for the work after the next search I started reading where I left off last time. The cache simulations only recorded reference and miss information on addresses within the search data (and within the work data when it was used), stack pointers and any other memory references were ignored.

The execution timings were done on an AlphaStation 500 with the DECchip 21164 and a 2MB on-board cache, and a Ultra 2 with a 1MB cache.

6 Experimental Results

I used three different key/node size combinations, 8 keys per node and 4 byte keys, 4 keys per node and 8 byte keys, and 2 keys per node and 16 byte nodes. I chose these because they all fit within one cache line. Of these three combinations, 8 keys per node and 4 byte keys should perform the best since it packs the highest number of keys within a cache line, so when a cache miss occurs on a node with this configuration, the next three references will all be within that node (line) and thus will be cache hits. Figure 9 compares the cache miss behavior of the binary search and the CCSS for each of the three different key/node size configurations. Note that the CCSS with four times the data of the binary search (16-byte key CCSS versus 4-byte key binary search) still incurs less misses per search when operating on a large data set. To only take into account the improvement in cache misses, figure 10 shows the speedup of an unoptimized CCSS versus the binary search. Despite a higher instruction and reference count, the unoptimized CCSS surpasses the speed of the binary search just before the data set size exceeds the size of the cache. Figure 10 shows the speedup of an optimized CCSS, here (depending on the key/node configuration) the benefits show up even for small data set sizes. NOTE: FIX!

The previous timings and simulations were generated with no outside work between the searches, figure 12 shows the effect that adding outside work between searches has on the miss rate. The data set remains constant ($2 * \frac{2^{21}}{4} = 1,048,576$ elements, twice the cache size) while the amount of outside work increases along the x-axis. Figure 12 shows the CCSS holds the miss rate to about $\frac{1}{3}$ that of the binary search, which is as expected since a CCSS with 8 keys per node will access in average slightly more than three keys before

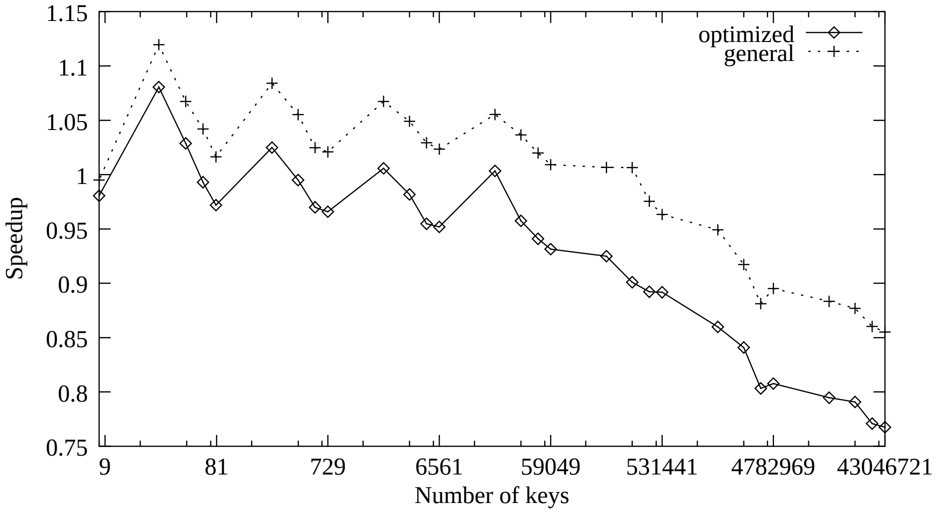

Figure 10: Speedup versus input size. The data points cycle through trees with a $\frac{1}{4}$ full bottom level, to $\frac{1}{2}$ full, to $\frac{3}{4}$ full, and then completely full. Timings from Ultra 2, 8 keys per node, 4 byte keys.

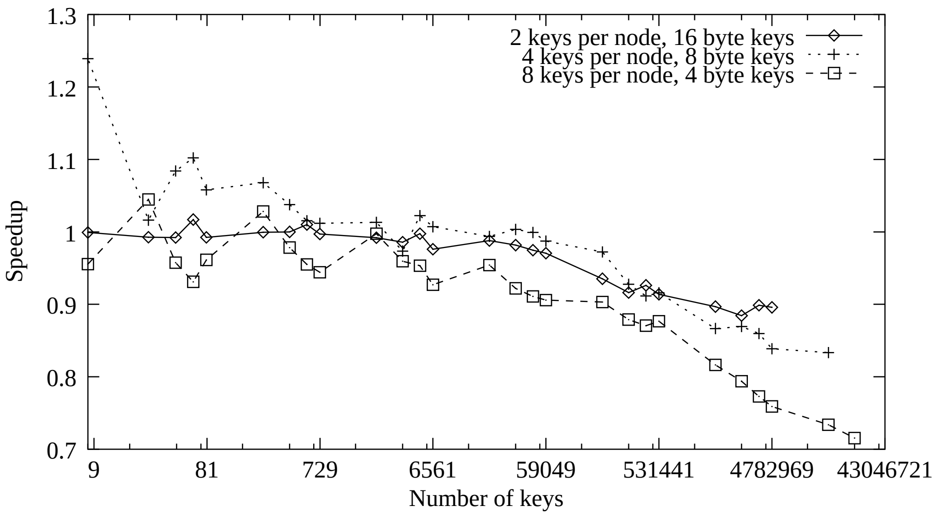

Figure 11: Speedup versus input size. Same data points as in figure 10. Timings from AlphaStation.

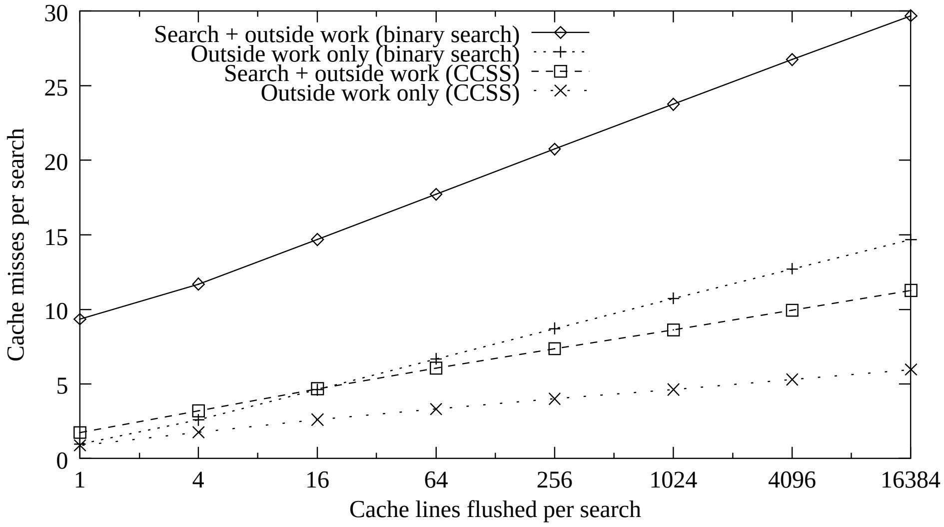

Figure 12: Cache misses versus outside work, 8 keys per node, 4 byte keys, input size is twice the cache size. Both search type has two lines here, one that shows the total misses caused by the outside work and the search itself, and another that just shows the misses caused by the outside work.

moving to a new node. Compare figure 13 to the values at $x = 1,048,576$ in figure 11, the speedup is more dramatic. The speedup figures include the time to perform the outside work, so as the work grows it dwarfs the time to search and drives the speedup toward one, when in actuality the differences in search times should be diverging even more.

7 Conclusion

The CCSS has many very nice properties: lower instruction count for larger (and some smaller) data sets, consistent performance over all data set sizes, the same memory footprint as the binary search, and dramatically improved cache performance. When used only occasionally on any size of cold data, it will incur a smaller number of cache misses, thus not only running faster but interfering less with the cache lines of the algorithm that called it. In the less likely event that the CCSS is called repeatedly on a data set larger than the cache, its performance also outstrips that of the binary search. Additionally when the CCSS tree is balanced (so the number of references per search is the same as in the binary search), an optimized search will perform marginally better than the binary search whether or not the data is hot in the cache due to the lower instruction count. The CCSS also has a reference pattern that makes it ideal when employing coloring [7] since all the frequently-used data is contiguous in a compact area.

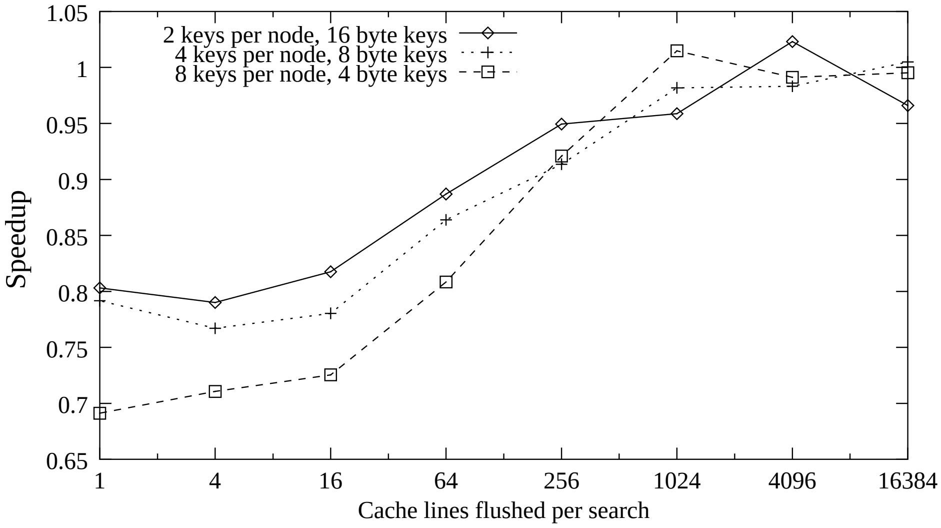

Figure 13: The time per search includes the time for the outside work. Timings from the AlphaStation. Only used 100,000 searches per data point in this graph.

References

[1] D.E. Knuth. *The Art of Computer Programming, vol III – Sorting and Searching.* Addison-Wesley, 1973.

[2] Peter Sanders. Fast priority queues for cached memory. ALANEX '99, Workshop on Algorithm Engineering and Experimentation.

[3] H. Zhu A. Acharya and K. Shen. Adaptive algorithms for cache-efficient trie search. ALANEX '99, Workshop on Algorithm Engineering and Experimentation.

[4] A. LaMarca and R.E. Ladner. The influence of caches on the performance of sorting. *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, 1997.

[5] A. Lamarca and R.E. Ladner. The influences of caches on the performance of heaps. *The Journal of Experimental Algorithmics*, 1, 1996.

[6] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation*, pages 196–205, 1994.

[7] J.R. Larus T.M. Chilimbi, M.D. Hill. Cache-conscious structure layout. *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999.