A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation

Richard E. Ladner* Ray Fortna† Bao-Hoang Nguyen‡

April 9, 2002

Abstract

An experimental comparison of cache aware and cache oblivious static search tree algorithms is presented. Both cache aware and cache oblivious algorithms outperform classic binary search on large data sets because of their better utilization of cache memory. Cache aware algorithms with implicit pointers perform best overall, but cache oblivious algorithms do almost as well and do not have to be tuned to the memory block size as cache aware algorithms require. Program instrumentation techniques are used to compare the cache misses and instruction counts for implementations of these algorithms.

1 Introduction

The performance of an algorithm when implemented is a function of many factors: its theoretical asymptotic performance, the programming language chosen, choice of data structures, the configuration of the target machine, and many other factors. One factor that is becoming more and more important is how well the algorithm takes advantage of the memory hierarchy, its *memory performance*. Data to be processed by the algorithm can stored in different levels of the memory hierarchy: the registers on the processor chip, first level cache, second level cache, main memory, and secondary memory on disk. Each successive level of the memory hierarchy is slower and larger than the preceding level. When a datum is required by the processor it must be transferred from its current location in the hierarchy to the processor. Because of the time delay in moving the datum to the processor, typically surrounding data is also transferred down the memory hierarchy in a *block* that contains the required datum. This block transfer amortizes the transfer time of all the data in the hope that not just the one datum is required,

*Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195. Research supported in part by NSF grant CCR-9732828. E-mail: `ladner@cs.washington.edu`.

†Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195. Research supported in part by NSF REU grant CCR-9732828.

‡Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195.

but that surrounding data will be required soon. Typical block sizes are 1024 bytes from disk to main memory, 32 bytes from main memory to the level two and level one caches, and 4 bytes from the level one cache to the registers.

Most algorithms are not designed with memory performance in mind and probably shouldn't be. However, there are cases where an algorithm is in the "inner loop" where good memory performance is necessary. In these cases designing for good memory performance is needed to achieve optimal performance. It is not difficult to find examples where a "memory sensitive" main memory algorithm can achieve a 50% reduction in running time over a similar "memory insensitive" algorithm. The reduction in running time can be attributed to the reduction in level two cache misses, where a cache miss is an access to a datum that is in main memory but not in the level two cache.

In this paper we concentrate on the memory performance of algorithms where the data resides in main memory and not in secondary memory. In particular, we examine the classic technique of *binary search*, an algorithm to locate an item among a static set of items. Classic binary search is so well known that it does not need any introduction. However, a quick analysis shows that its memory performance is poor. Suppose several items can fit into a memory block. In classic binary search the items are stored in a sorted array. The query item is compared with the middle item in the array. If it is equal, the search is completed. If it is smaller, the subarray to the left of the middle is searched in the same way. If it is larger, the subarray to the right is searched in the same way. The important point is that in the two latter cases the next item accessed is likely to be far from the middle of the array, so it is not in the same memory block. Thus, memory blocks are poorly utilized in classic binary search. Can binary search's memory performance be improved? The answer is a resounding yes, and there are several strategies to do so.

In this paper we examine two strategies for improving the memory performance of binary search. The first is the *cache aware* approach where items that are accessed together are stored together in the same memory block. Knowledge of the memory block size is needed to accomplish this. In this approach the items can be stored without the use of explicit pointers, but the layout of the items in memory does not constitute a sorted array. A disadvantage of cache aware search is that, because the items are organized into memory blocks, the algorithm does not achieve the perfect binary splitting into equal size subproblems. Cache aware algorithms have been studied in a number of different contexts [7, 8, 5, 10].

The second approach to improving the memory performance of binary search is the *cache oblivious* [9, 4, 1, 2, 3] approach where the items are organized in a universal fashion so that items that are accessed closely in time are stored near each other. The method is called cache oblivious because knowledge of the memory block size is not needed to achieve the organization. The advantage of the cache oblivious approach is that the organization of the data yields good memory performance at all levels of the memory hierarchy. One disadvantage of the cache oblivious approach is that it might not perform as well as the cache aware approach because it cannot take advantage of knowledge of the memory block size. Another disadvantage of the cache oblivious approach is that, although items can be accessed without explicit pointers, the computation to find the next item may be prohibitively expensive. This means that explicit pointers must be used, which increases the memory footprint of the data structure, which may hurt memory performance.

We take an experimental approach in comparing cache aware and cache oblivious search. We first implemented in C classic binary search, cache aware search, and cache oblivious search. There are two versions of each implementation, one with implicit pointers and one with explicit pointers. As is normally done we did execution time studies on a wide range of data set sizes on several platforms. More interesting is our use of program instrumentation tools to count the number of machine instructions executed by each program and to simulate the number of cache misses that occurs for each implementation. The former metric is called *instruction count* and the latter metric is called *cache performance*. We simulated a direct mapped cache with several different memory block sizes.

We summarize our main results as:

1. In terms of execution time, both cache aware search with implicit pointers and cache oblivious search with explicit pointers perform comparably, and are both significantly faster the classic binary search.

2. cache aware search with implicit pointers has slightly better cache performance than cache oblivious search with explicit pointers.

3. cache aware search with implicit pointers has slightly worse instruction count performance than cache oblivious search with explicit pointers.

In summary, the cache oblivious approach is almost as effective as the cache aware approach to reducing cache misses for static search and has the advantage that it does not need to be tuned to the memory block size.

2 Organization

In sections 3 and 4 we present cache aware search and cache oblivious search, respectively. In section 5 we present the instrumentation tool ATOM [12] and how it is used for measuring cache misses and instruction counts. In section 6 we present our experimental results. In section 7 we present our conclusions.

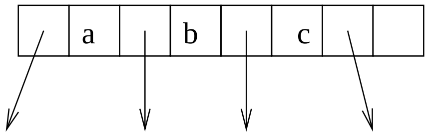

Figure 1: Node of a cache aware 4-ary search tree stored in a 32 byte memory block.

3 Cache Aware Search

In this section, we present the cache aware approach for improving memory performance of binary search. The basic idea is to store in the same memory block those items that are most likely accessed together in time. In this way when an item moves from main memory to the cache, other items that are likely to be accessed soon are moved to the cache in the same block. Hence, cache misses are avoided. A simple way to achieve this is to use a k-ary tree where a node contains $k-1$ items and k pointers to subtrees. To achieve the effect we want we choose k so that all the items and pointers fit in a memory block.

There are several ways to implement k-ary trees, one which employs explicit pointers and one that uses implicit pointers. In the former memory must be allocated to the pointers, while in the latter the address of the child of a node is calculated and no storage is wasted on pointers. Suppose that we know the cache line size is 32 bytes, and assume that an item and a pointer each occupies four bytes, we can store at most four pointers and three items. This means that our tree would be a 4-ary tree. Figure 1 depicts this example.

However, there are still four bytes in the memory block left unused. In order to make the node cache-align, these four bytes need to be padded in our structure. Hence, we lose some more memory for padding besides the memory used for pointers. A big disadvantage of the explicit pointer structure is the size of its memory footprint is increased by the inclusion of pointers and padding. An advantage of the explicit pointer structure is the speed in following pointers rather than calculating them.

Using implicit pointers helps to alleviate memory footprint problem. By not storing the explicit pointers, we can use the whole memory block to store keys, so the parameter k is larger. For example, for 32 byte memory block we now can store 8 items instead of 3 in a memory block and have no padding. Interestingly, the utilization of the memory blocks for explicit and implicit pointers is about the same. If binary search is done within a node, then in the explicit pointer case 2 items and one pointer are touched most commonly. In the implicit pointer case 3 or 4 items are touched most commonly. The big win of implicit pointers is that the height of the tree, which bounds the number of cache misses, is much less.

For a k-ary tree, we layout the nodes in a contiguous piece of memory, starting from the root node going down, and from left to right for nodes at the same height. If the nodes are stored in an array, the root is stored at index 0 and the j-th child ($1 \leq j \leq k$) of the node stored at index i is stored at index $ik + (k-1)(j+1)$. This simple calculation replaces the explicit storage of pointers. The layout of the nodes of a 3-way cache aware search tree with implicit pointer is described in figure 2.

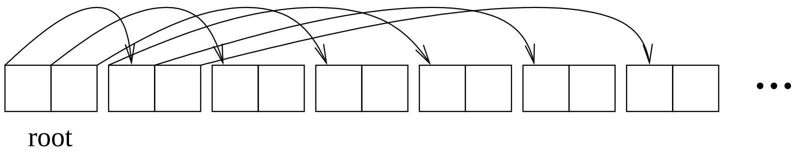

Figure 2: Memory layout of a 3-way cache aware search tree with implicit pointers.

4 Cache Oblivious Search

Cache oblivious algorithms operate under the same principles as cache aware algorithms. Both types of algorithms try to "cluster" data together in memory so that the locality of memory references is increased. The cache aware algorithm described above accomplishes this by "clustering" nodes of a binary search tree into nodes that fit into a memory block. The cache oblivious algorithm described by Prokop [9] approximates the same behavior, but does so without any knowledge of the cache parameters. Figure 3 shows how the cache oblivious algorithm lays out the data in memory to accomplish this.

Given a binary search tree of h (assuming h is a power of 2) levels, the memory layout algorithm works as follows. Cut the tree in half vertically, leaving one subtree above the cut and $2^{h/2}$ subtrees below the cut, giving a total of $2^{h/2}+1$ subtrees all of the same size. The top subtree is then placed in a contiguous block of memory followed by the $2^{h/2}$ subtrees from the left most to the right. The algorithm is then recursively applied to the to the top subtree followed by the bottom subtrees in left to right order. The algorithm terminates when it is applied to a subtree of one level, at which point it will add the single node into the array. As the algorithm recurses through each of the subtrees, it will eventually reach a tree which will occupy contiguous memory of size about the same as a memory block. This is similar to the behavior of the cache aware algorithm, the only difference in the two layouts is that the cache aware algorithm ensures that each "cluster" starts at the beginning, and spans only one memory block. The cache oblivious algorithm on the other hand cannot ensure that the cluster starts on a memory block boundary. Instead it guarantees that each cluster will span at most two memory blocks. This is illustrated below in Figure 4.

The cache aware algorithm knows the cache parameters and can therefore align the array in memory as to ensure that all the clusters are cache aligned. By virtue of the fact that

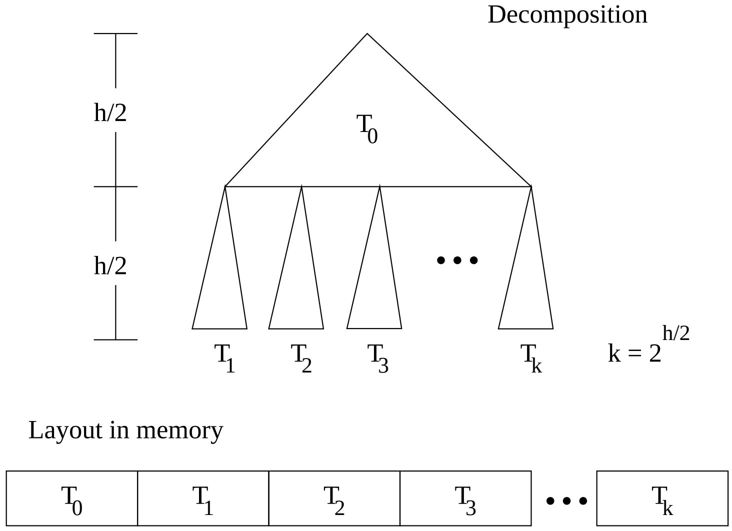

Figure 3: cache oblivious search tree decomposition and layout in memory.

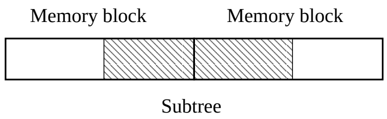

Figure 4: A subtree of memory block size spans at most two memory blocks.

the cache oblivious algorithm knows nothing of the cache parameters there is no way for it to ensure that a "cluster" does not begin somewhere in the middle of a memory block and thus ending in another memory block. Because of this fact, the cache aware algorithm will inherently have better cache performance than the cache oblivious algorithm. However, the cache oblivious algorithm does have the advantage that it does not have to be "hand tuned" for each cache size. Its properties ensure that each "cluster" will only span at most two memory blocks no matter the memory block size, where as the cache aware algorithm must be adjusted for each memory block size to ensure that its properties hold. As was the case with the cache aware implementation, the cache oblivious algorithm can be implemented with both explicit and implicit pointers. Implicit pointers have the benefit of reducing the memory footprint of a single node, and thus increase the overall cache performance. However the computation of the implicit pointers at run time impacts the instruction count of the algorithm and can have a negative effect on performance.

In recent work Bender *et al* [2] and Brodal *et al* [3] have used the cache oblivious static search tree as the basis of a cache oblivious dynamic search structures that allow for insertions and deletions. In particular, Brodal *et al* have discovered a very elegant and efficient way to calculate the pointers in the cache oblivious static search tree with implicit pointers. For our cache oblivious search tree with implicit pointers we use a more computation intensive algorithm for computing pointers which is described in the next paragraph. Hence, our cache oblivious search tree with implicit pointers have high instruction counts and execution times. This could be remedied by using the Brodal *et al* calculation of pointers.

As described in figure 3, the layout of the cache oblivious search tree in memory is determined by recursively "cutting" the tree in half height-wise and placing the nodes in contiguous memory starting with the top half of the tree. It is not surprising that traversing the tree also involves recursively cutting the tree. The algorithm works as follows. Initially the algorithm begins it search at the root of the tree, the first level of the tree, and the initial "cut" is located at h, where h is the height of the tree. If the number of levels that separate the current node from the next cut is greater than or equal to 2, a new cut is placed halfway between the current level of the search and the level of the next cut. If the current node is at the same level as the next cut an inter-cut traversal is done. Otherwise an intra-cut traversal is done. This process is repeated at the new node until the search succeeds or fails. An intra-cut traversal is defined as follows. Let i be the index of the current node in the search. If the difference between the level of the current node and the level of the next cut is ℓ, where $\ell \leq 2$ the left child of the current node is located at $i+1$ while the right child is located at $i+2^\ell$. An inter-cut traversal is done in the case that by moving to the next node in the traversal, we cross over an existing cut. In this case, the next node is located in memory after all the nodes above the cut and after all the nodes between the cut and the next cut in the subtrees to the left of the next node. If the number of levels in the tree above it is d and the number of trees to the left of it is s then j child ($1 \leq j \leq 2$) of the node indexed at i is located at

$$(2s+j-1)(2^\ell-1)+2^{d+1}-1.$$

The quantity $2^{d+1}-1$ is the number of nodes above the cut and the quantity $2^\ell-1$ is the number of nodes in each of the subtrees to the left of the next node. There are either $2s$ or $2s+1$ subtrees to the left of the next node depending on whether the traversal goes left or right respectively. A stack can be used to maintain the current cut by simulating the recursive construction described in figure 3. The values of d and s can be maintained easily and the value of ℓ can be calculated from the cut value and d.

5 Program Instrumentation

Program instrumentation is a general way to understand what an executing program is doing. Program instrumentation can be achieved automatically or manually. For example, when a compiler is called with the debugger on, then the executable code is augmented automatically to allow the user to view values of variables or other quantities. The semantics of the program should be the same whether or not the debugger is turned on. Manual instrumentation is where the programmer inserts instructions into the source code to measure some quantity or

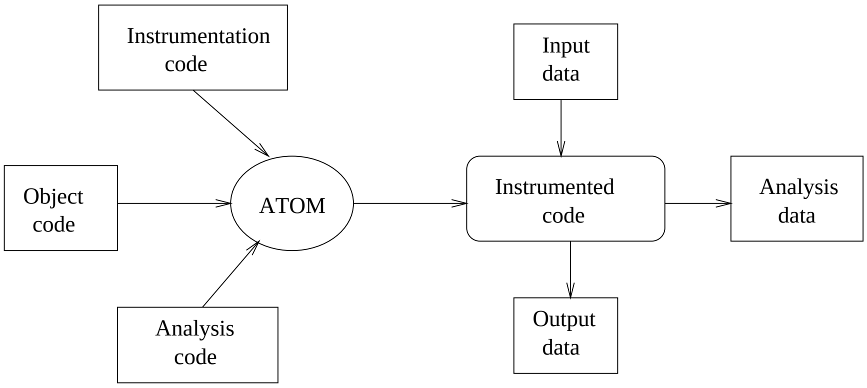

Figure 5: Schematic of the ATOM system.

print out some intermediate values. For this study we employed the system ATOM [12] which enables the user to build customized tools to automatically instrument and analyze programs that run on DEC alphas. Other program instrumentation tools that are useful for measuring memory performance are Cacheprof [11] and Etch [6].

The programmer provides three pieces of code to ATOM: (i) the unlinked object code of the program to be instrumented, (ii) *instrumentation code* that tells atom what "sensors" to insert into the object code and where to place them, and (iii) *analysis code* that processes the sensor data from the executing program to provide information about the execution. ATOM takes the three pieces and produces an instrumented program executable. When the executable is then run, it has the same semantics as the uninstrumented program, but during the execution the sensors gather data that is processed by the analysis code. Figure 5 gives a picture of the ATOM system. A simple example of the use of ATOM is an *instruction counter*. The instrumentation code inserts an increment-counter instruction after every instruction in the object code. The analysis code sets the counter to zero initially and outputs the final count on termination. We employ such an instruction counter in our study.

A second, more sophisticated example, is a *trace driven cache simulator*. In this case the instrumentation code inserts instructions after each load and store to sense the memory address of the operand. The analysis code is a cache simulator that takes the address as input and simulates what a memory system would do with the address. In addition, the analysis code keeps track of the number of loads and stores to memory and how many accesses are misses. Figure 6 shows the instrumentation code for a cache simulator and figure 7 shows the analysis code for a very simple one level, direct mapped cache simulator. We used a trace driven cache simulator similar to this one for our study.

In the instrumentation code, Figure 6, the nested for loops identify each procedure in the object code, then each basic block within the procedure, then each instruction within the basic block. If the instruction is a load, then code is inserted before the instruction which calls `LoadReference` in the analysis code passing `EffAddrValue`, the operand, as a parameter. If the

```
Instrument() {
  Proc *p;  Block *b; Inst *i;
  AddCallProto("LoadReference(VALUE)");
  AddCallProto("StoreReference(VALUE)");
  AddCallProto("PrintResults()");
  for (p = GetFirstProc(); p != NULL;  p = GetNextProc(p)) {
    for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b)) {
      for (i = GetFirstInst(b); i != NULL; i = GetNextInst(i))  {
        if (GetInstClass(i) == ClassLoad ||
          GetInstClass(i) == ClassFload) {
            AddCallInst(i,InstBefore,"LoadReference",EffAddrValue);
          xs}
        if (GetInstClass(i)== ClassStore ||
          GetInstClass(i)== ClassFstore) {
            AddCallInst(i,InstBefore,"StoreReference",EffAddrValue);
          }
        }
      }
    }
  }
  AddCallProgram(ProgramAfter,"PrintResults");
}
```

Figure 6: Instrumentation code for a trace driven cache simulator.

instruction is a store, then code is inserted before the instruction which calls `StoreReference` in the analysis code passing `EffAddrValue`, the operand, as a parameter. At the end of the program code is inserted which calls `PrintResults`.

The analysis code implements a direct mapped cache with size `CACHE_SIZE` in bytes and block size `BLOCK_SHIFT` in bits. The analysis code maintains the array `tags` which stores the memory addresses that currently reside in the cache. In addition it maintains counters for the number of load and store references and load and store misses. For our study we use a similar cache simulator, but we add the load and store values into one value for both references and misses. We chose to simulate a one level cache because the cache miss penalty for the level two cache is typically much greater than that for the level one cache. Having just one number representing the cache performance is reasonable compromise considering that level one cache misses tend to have a low order effect on performance on large data sets.

ATOM is a powerful tool but it must be used properly to obtain accurate results. First, the analysis code is interleaved with the code to be analyzed. This means that the instrumented code can be considerably slower than the uninstrumented code. This is not a serious problem with either instruction counting or cache simulation because the analysis code is quite efficient. For cache simulation it is important not to use dynamic memory in the analysis code. Use of dynamic memory would cause a difference in the addresses used by the instrumented and uninstrumented codes, and distort the results. This is not a problem in our case because we used static memory to allocate the `tags` array and counters. Finally, the trace driven cache simulator is just that. It does not measure cache misses caused by swapping, TLB misses, or

```
void generalreference(long address, int isLoad) {
  int index = (address & (CACHE_SIZE-1)) >> BLOCK_SHIFT;
  long tag = address >> (BLOCK_SHIFT + INDEX_SHIFT);
  int returnval;

  if (tags[index] != tag) {
     if (isLoad){
loadmisses++;
tags[index] = tag;
     }
     else  {
storemisses++
     }
  }
  if (isLoad) {
     loadreferences++;
  }
  else {
     storereferences++;
  }
}
void LoadReference(long address) {
   generalreference(address, 1);
}
void StoreReference(long address) {
   generalreference(address, 0);
}
```

Figure 7: Analysis code for a trace driven cache simulator for a one level, direct mapped cache.

instruction cache misses.

6 Experimental Results

In order to better understand the alternative static search algorithms we implemented in C six algorithms: classic binary search, cache aware search, and cache oblivious search each with explicit and implicit pointers versions. All studies were for data sets the range from 128 to 2,097,152 and for larger data sets when possible. All items and pointers used are 4 bytes. In order to compare our static search algorithms we employed program instrumentation for trace driven cache simulation and instruction counts. These studies were done using ATOM on a Compaq Alpha 21164. In addition, we performed two execution studies one on Windows and one on Linux. In Table 6 we list the computer configurations and compilers. All the caches in the two platforms are 4-way set-associative and all block sizes are 32 bytes. In all the studies each data point represents the median of 10 trials where a trial consisted of n random successful lookups where n is the number of items. The median of 10 is computed as the average of the 5th and 6th ranked trials to avoid the effect of outliers. For a given n, the 10 measured trials were preceded by n unmeasured successful lookups to warm up the cache.

	Windows	Linux
Operating System	Windows 2000 “Professional”	Linux Mandrake 7.2
Processor	533 MHz Intel Celeron	350 MHz Intel Pentium II
Memory	64 MB	128 MB
Memory Block	32 B	32 B
L2 Cache	128 KB	512 KB
L1 Cache	32 KB	32 KB
Compiler	MSVC 6.0	gcc 2.95.2
Options	Release Build Highest Option	-O3 (highest setting)

Table 1: Computer configurations and compilers used in the execution time studies.

Figure 8 gives the results a cache simulation using ATOM where we simulated a direct mapped cache of size 8,192 bytes and a memory block size of 32 bytes. In the x-axis we plot the number of items on a log scale and in the y-axis we plot the number of cache misses. We see that the cache aware search with implicit pointers has the fewest cache misses, while classic binary search has the most. All the algorithms that use implicit pointers have fewer cache misses than their explicit pointer counterparts showing the effect of the larger memory footprint for the explicit pointers. Most important is that both cache oblivious and cache aware search algorithms have much better memory performance than classic binary search.

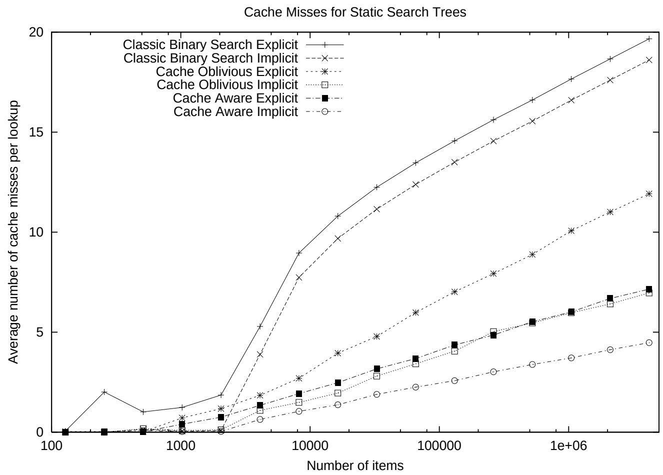

Figure 8: Cache misses per lookup for static search algorithms.

Figure 9 gives the results of instruction counting using ATOM for the algorithms. In the x-axis we plot the number of items on a log scale and in the y-axis we plot the number of Compaq Alpha 21164 instructions executed per lookup. We see immediately the high price in instruction count that is paid for our version of cache oblivious search with implicit pointers. The instruction count penalties for implicit pointers for classic binary search and cache aware search are small. The explicit pointer versions of classic binary search and cache oblivious search execute the fewest instructions per lookup. Cache aware search with explicit pointers has slightly more instructions per lookup because it does not achieve perfect binary splitting into equal size subproblems.

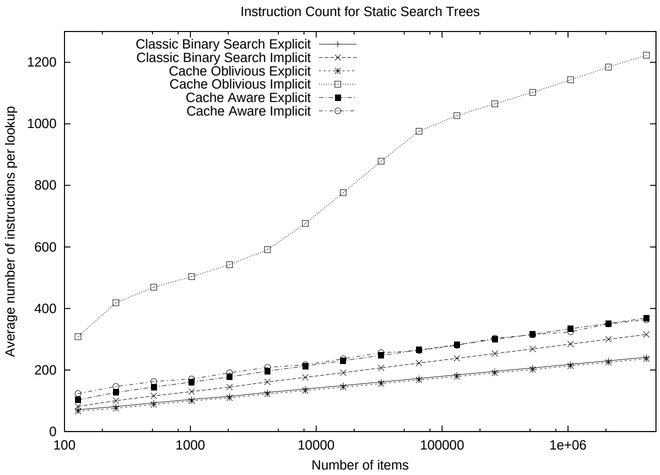

Figure 9: Instruction count per lookup for static search algorithms.

In our execution time studies shown in Figures 10 and 11 in the x-axis we plot the number of items on a log scale and in the y-axis we plot the time per lookup measured in microseconds. Each trial was measured using `time.h` from the Standard C library.

Figure 10 gives the results of an execution time study using Windows. Cache aware search with implicit pointers is the fastest, but cache oblivious search with explicit pointers is not far behind. Cache oblivious search with implicit pointers is the slowest of all because of the high cost of computing pointers.

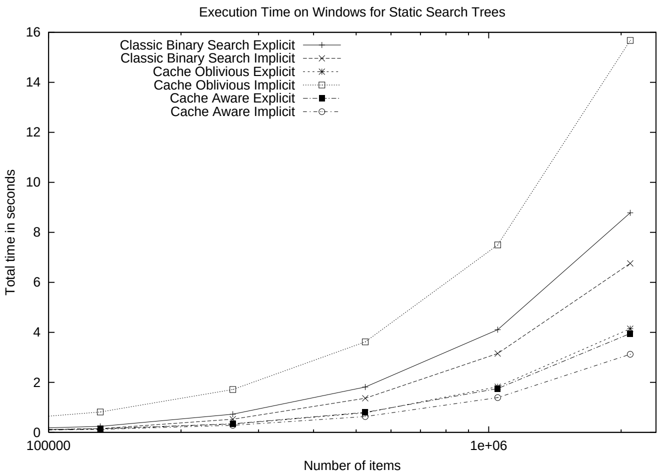

Figure 10: Execution time on Windows for static search algorithms.

Figure 11 gives the results of an execution time study using Linux. Again, Cache aware search with implicit pointers is the fastest, but cache oblivious search with explicit pointers is not far behind. Again, Cache oblivious search with implicit pointers is the slowest of all because of the high cost of computing pointers. Inexplicably, Cache aware search with explicit pointers showed consistently poor performance under Linux. We looked at a number of possible causes for the poor performance but were not able to pin down a reason for it. We believe that the Linux behavior perhaps demonstrates the perils of cache aware programming. The cache oblivious algorithms performed consistently on both platforms.

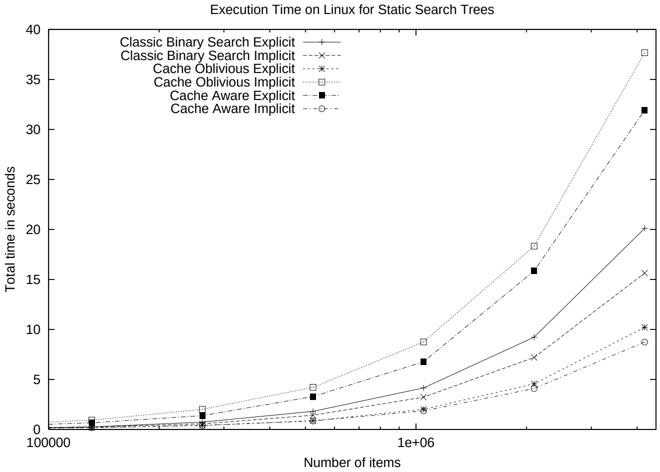

Figure 11: Execution time on Linux for static search algorithms.

7 Conclusion

Both cache aware and cache oblivious search perform better than classic binary search on large data sets. Cache aware search algorithms have the disadvantage that they require knowledge of the memory block size. Cache oblivious search algorithms have only slightly worse memory performance than cache aware search, but in our study only the explicit pointer version of oblivious search has comparable overall performance. As mentioned earlier Brodal *et al* [3] have found a way to compute the implicit pointers efficiently in the cache oblivious algorithm. The cache oblivious search algorithms do not require knowledge of the memory block size to achieve good memory performance. Finally, program instrumentation tools like ATOM let us obtain a deeper understanding of the performance of these algorithms.

Acknowledgments

The research was supported by NSF Grant No. CCR-9732828 and by Microsoft. Ray Fortna and Bao-Hoang Nguyen were undergraduate students at the University of Washington at the time this paper was written. Ray Fortna was supported by a NSF REU.

Thanks to the anonymous referees for suggesting several improvements to the paper.

References

[1] Bender, M.A.; Demaine, E.D.; Farach-Colton, M. (2000): Cache-Oblivious B-Trees. FOCS'00, Proceedings of 41st Annual Symposium on Foundations of Computer Science, 399–409

[2] Bender, M.A.; Duan, Z.; Iacono, J.; Wu J. (2002): A locality-preserving cache-oblivious dynamic dictionary. SODA'02, Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 29–38

[3] Brodal, G.S.; Fagerberg, R.; Jacob, R. (2002): Cache oblivious search trees via binary trees of small height. SODA'02, Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 39–48

[4] Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S. (1999): Cache-oblivious algorithms. FOCS'99, 40th Annual IEEE Symposium on Foundations of Computer Science, 285–297

[5] Rao, J., Ross, K.A. (1999): Cache Conscious Indexing for Decision-Support in Main Memory. VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, 78–89

[6] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B (1997): Instrumentation and optimization of Win32/Intel executables using Etch. The USENIX Windows NT Workshop. See www.usenix.org/publications/library/proceedings/usenix-nt97/romer.html.

[7] LaMarca, A., Ladner, R.E. (1996): The Influence of Caches on the Performance of Heaps. *Journal of Experimental Algorithmics*, Vol. 1.

[8] LaMarca, A., Ladner, R.E. (1999): The Influence of Caches on the Performance of Sorting. *Journal of Algorithms*, Vol. 31, 66–104.

[9] Prokop, H. (1999) Cache-Oblivious Algorithms. Master's Thesis, MIT Department of Electrical Engineering and Computer Science, June 1999.

[10] Sen, S., Chatterjee, S. (2000): Towards a Theory of Cache-Efficient Algorithms. SODA'00 Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, 829 –838

[11] Seward, J. Cacheprof. See www.cacheprof.org.

[12] Srivastava A., Eustace, A. (1994): ATOM: A System for Building Customized Program Analysis Tools. PLDI'94, Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation, 196–205