An Eight-Dimensional Systematic Evaluation of Optimized Search Algorithms on Modern Processors

Lars-Christian Schulz
University of Magdeburg
Magdeburg, Germany
lschulz@st.ovgu.de

David Broneske
University of Magdeburg
Magdeburg, Germany
dbronesk@ovgu.de

Gunter Saake
University of Magdeburg
Magdeburg, Germany
saake@ovgu.de

ABSTRACT

Searching in sorted arrays of keys is a common task with a broad range of applications. Often searching is part of the performance critical sections of a database query or index access, raising the question what kind of search algorithm to choose and how to optimize it to obtain the best possible performance on real-world hardware. This paper strives to answer this question by evaluating a large set of optimized sequential, binary and k-ary search algorithms on a modern processor. In this context, we consider hardware-sensitive optimization strategies as well as algorithmic variations resulting in an eight-dimensional evaluation space.

As a result, we give insights on expected interactions between search algorithms and optimizations on modern hardware. In fact, there is no single best optimized algorithm, leading to a set of advices on which variants should be considered first given a particular array size.

PVLDB Reference Format:
Lars-Christian Schulz, David Broneske, Gunter Saake. An Eight-Dimensional Systematic Evaluation of Optimized Search Algorithms on Modern Processors. *PVLDB*, 11 (11): 1550-1562, 2018.
DOI: https://doi.org/10.14778/3236187.3236205

1. INTRODUCTION

Searching in sorted data is one of the most fundamental operations in computing. A typical search problem consists of a sequence of key/value pairs sorted by keys and a query for either a single key or a range of keys. The task is then either to retrieve the value belonging to the given key, or to retrieve all key/value pairs in the given key range.

Since this problem is so fundamental and its solution so universally useful, it has been studied extensively since the early days of computer science. The classical and asymptotically optimal algorithm to solve it is the well known binary search [7]. However, there is no single binary search algorithm, but many different variants with slightly different properties [10]. Moreover, binary searching can logically be extended to ternary searching and in all generality to k-ary searching. While their conceptual benefit is intuitve, these benefits are not as easily inferable on a modern, deeply pipelined, superscalar, out-of-order processor with a multi-level cache hierarchy and SIMD facilities. In this context, there is also the question what influence code optimizations like loop unrolling have on the performance of a particular algorithm. Furthermore, the algorithm, the applied code optimizations, and micro-architectural specifics of the processor can have surprising interactions. Therefore, a systematic experimental evaluation of all these factors is necessary to find optimal search algorithms. To enable reproducibility, the code can be access from our git[1].

Proceedings of the VLDB Endowment, Vol. 11, No. 11
Copyright 2018 VLDB Endowment 2150-8097/18/07.
DOI: https://doi.org/10.14778/3236187.3236205

Our Contributions. This paper aims at providing insight into the behavior of optimized search algorithms on modern processors on static arrays of tightly packed 32-bit integer keys. We evaluate search variants in an eight-dimensional space of algorithmic variations, code optimizations, and data-set properties. The algorithmic aspects include:

1. **Base algorithm.** We consider sequential, binary and k-ary search algorithms. The binary and k-ary search include variants with irregular subdivisions or strictly regular subdivisions, also often called *uniform.*
2. **Exact match vs. lower bound.** There are two possible functions when searching in sorted lists: range and exact match queries. To answer range queries, all keys in the closed interval $[a, b]$ have to be retrieved. To this end, a lower bound and upper-bound search is performed. We only consider the lower-bound search, since only minimal differences exist to the upper-bound search. Since exact match can be trivially derived once the lower bound has been located, we compare direct exact-match search algorithms and lower-bound-based exact-match search algorithms.

We apply the following four code optimization techniques:

3. **Branch Elimination.** We remove branches from the search loops by using branch predication.
4. **Loop unrolling.** We unroll the search loops.
5. **Software Prefetching.** We add prefetching to avoid or shorten the stalls caused by cache misses.
6. **Vectorization.** We use the AVX2 instruction set to parallelize the algorithms in SIMD fashion.

The generated evaluation dataset raises two dimensions:

7. **Dataset size.** We vary the array size from one to 2^{28} keys. Thus, we consider a continuous range of array sizes from arrays comfortably fitting into the processor cache to much larger ones.

[1] `https://git.iti.cs.ovgu.de/dbronesk/SearchAlgorithms`

8. **Locality of search queries.** We use two different schemes to generate search queries with different caching demands.

In addition to runtime measurements we analyze micro-architectural performance aspects like branch mispredictions and cache activity utilizing the performance counters built into modern processors. Ultimately, we arrive at a recommendation which algorithms and code tuning strategies should be used first when a search algorithm is needed.

2. OPTIMIZING SEARCH ALGORITHMS

We consider three different general methods to search on sorted data: (1) sequential searching, (2) binary searching and, as a generalization thereof, (3) k-ary searching. Section 2.1 discusses these algorithms. In Section 2.2, we improve the performance of the basic algorithms by adapting them to characteristics of modern processors.

2.1 Algorithmic Variations

In the following, we briefly discuss sequential, binary and k-ary searching for the lower bound. Additionally we introduce modifications to the traditional binary and k-ary search based on searching in perfect search trees inspired by Schlegel et at. [12]. The differences between lower bound and exact-match searching are outlined in Section 2.1.5.

2.1.1 Sequential Search

The sequential search visits each element in turn and returns with the current index when the first key not smaller than the search key—the lower bound—has been found. Its run time is linear in the number of array elements.

2.1.2 Binary Search

A binary search algorithm recursively splits the search range in two approximately equally sized partitions and examines the array element separating them, i.e., the *separator key*. The search then continues with just one of the partitions, thereby halving the search space. This way we need at most $\lfloor \log_2(size) \rfloor + 1$ iterations to localize the lower bound. This can be seen by constructing a binary tree corresponding to the search and analyzing its height [7].

A variant of the binary search splits the search range in two differently sized partitions according to a fixed ratio, so that the separator key is not exactly in the middle of the range. We will call this *offset binary search*.

2.1.3 k-ary Search

A logical generalization of binary searching is the k-ary search, not just dividing the search range in two partitions, but in k partitions for a $k > 2$. This requires probing $k - 1$ separator elements in each iteration. The separator elements are chosen to divide the search range evenly.

Since a k-ary search reduces the search space by a factor of k in each iteration, the worst case number of iterations is $\lceil \log_k(size + 1) \rceil = \lfloor \log_k(size) \rfloor + 1$. Thus, the speedup of k-ary searching is approximately $\log_2 n / \log_k n = \log_2 k$ [12].

2.1.4 Uniform Binary and k-ary Search

The binary and k-ary search algorithms discussed above divide the number of keys by the desired number of partitions. This has the disadvantage that the partitions are not exactly equally sized if the current number of keys is not divisible by the number of partitions. Therefore the number of iterations needed to complete a lower-bound search can vary depending on whether the smaller or the larger partitions are chosen during the search. Furthermore, we have to explicitly track the bounds of the current search range.

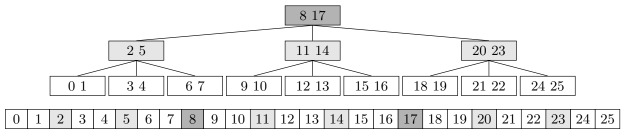

Figure 1: Perfect k-ary search tree for k = 3 containing 26 keys and the corresponding array.

It is possible to avoid these problems by restricting the algorithms to array sizes of the form $k^h - 1$, where $h > 0$ is an integer. We will call these array sizes perfect, allowing a perfect k-ary search tree to be constructed. In a perfect k-ary search tree every node contains $k - 1$ keys and every internal node has k children. Additionally, every leaf node is at the same depth [12]. Figure 1 shows a perfectly sized array and the perfect search tree for $k = 3$. As we can see, all partitions (excluding the separator keys) possibly created during a search have perfect length themselves. The separator keys contained in a node at height $h > 0$ are located at indices of the form $left + i \cdot k^{h-1} - 1$, where $left$ is the start index of the current search range. We call algorithms based on perfect search trees uniform. The uniform algorithms generally have simpler index computations than their non-uniform counterparts enabling more optimizations.

Uniform Binary Search. We first consider the uniform binary search (Algorithm 1). In contrast to the non-uniform binary search its search loop runs for exactly $\lceil \log_2(size+1) \rceil$ iterations. This is the height of the conceptual binary search tree constructed by the algorithm. The indices of the separator keys are computed as $left + 2^{height-1} - 1$. Generalization to arrays of arbitrary length is achieved by potentially letting the left and right partitions overlap in the first iteration. The following iterations can then assume both the left and right partition to be equally sized and correspond to perfect binary search trees, i.e., to have a length of $2^{height-1} - 1$. Since the right partition begins with the key to the right of the separator element, the overlapping is achieved by setting the index of the first separator element to $size - 2^{height-1}$. This way neither the left nor the right partition extend beyond the array bounds.

Algorithm 1 Lower-Bound Uniform Binary Search

```
function UNIFORMBINARYSEARCH(keys, size, searchKey)
    height ← $\lceil \log_2(\text{size} + 1) \rceil$
    mid ← size − $2^{\text{height}-1}$; left ← 0
    height ← height − 1
    while height > 0 do
        if searchKey > keys[mid] then
            left ← mid + 1
        end if
        mid ← left + $2^{\text{height}-1}$ − 1
        height ← height − 1
    end while
    return left
end function
```

Algorithm 2 Lower-Bound Uniform k-ary Search

```
function uniformKarySearch(keys, size, searchKey)
    height ← $\lceil \log_k(\text{size} + 1) \rceil$; left ← 0
    if size $\neq$ $k^{\text{height}} - 1$ then
        // imperfect array size
        height ← height − 1
    end if
    while height > 0 do
        step ← 0; offset ← 0
        for i ← $1, ..., k - 1$ do
            offset ← offset + $k^{\text{height}-1}$
            if searchKey $\leq$ keys[left + offset − 1] then
                break
            end if
            step ← offset
        end for
        left ← left + step
        height ← height − 1
    end while
    return left
end function
```

Uniform k-ary Search. The uniform k-ary search handles imperfectly sized arrays using the same idea as the binary search: The size of the array is rounded up to the next perfect size and the size of the partitions is selected accordingly, but they are allowed to overlap. Again, care must be taken that the rightmost partition does not extend outside of the array. The following iterations then proceed as shown in Algorithm 2. The algorithm uses an inner `for` loop to examine the $k-1$ separators. The index of the i-th separator is computed as $\textit{left} + \textit{offset} - 1$, where $\textit{offset} = i \cdot k^{\textit{height}-1}$, i.e., the offset of the partition to the left of the current separator from the start of the current search range.

2.1.5 Exact Match

Until now we have only described lower-bound search algorithms. There are two ways to derive exact-match variants from them: (1) Add an equality case to every key comparison and terminate successfully when the search key is found. (2) Perform a lower-bound search, then check whether the search key is at the index of the lower bound. We have implemented exact-match binary and k-ary search algorithms in both variants and compare them in Sections 4.2.4 and 4.3.3. Note that the index computations in variant (1) are slightly different from a lower-bound search, since the separator elements themselves are not included in any of the partitions.

2.2 Hardware-Sensitive Optimizations

We implemented the algorithms described above and optimized the resulting code by eliminating branches, unrolling loops and adding software prefetching. Moreover, we vectorized the sequential, binary and uniform k-ary search using AVX2 instructions. We assumed the array length to be evenly divisible by the SIMD register width and the arrays to be aligned at SIMD word boundaries. If these two assumptions are not met, additional code is needed to handle the "overhanging" elements. The vectorized implementations were again tuned like the scalar variants. In Table 1, we summarize which optimizations were applied to which algorithm. More implementation details are available in [13].

Table 1: Overview of the applied optimizations.

Algorithm	Branch Free	Loop Unrolled	Prefetch
Sequential	predication	2/4/8/16 times	–
Vectorized Seq.	mask eval.	2/4/8 times	–
Binary	predication	–	1 iter.
Vectorized Bin.	predication	–	1 iter.
Uniform Bin.	(predication)	completely	1/2 iter.
Vect. Unif. Bin.	predication	completely	1 iter.
k-ary	predication	inner loop	–
Uniform k-ary	predication	inner loop	1 iter.
Vect. Unif. k-ary	(mask eval.)	(inner loop)	–

2.2.1 Vectorization

In this paper, we use AVX2 to exploit SIMD parallelism on a single processor for all search algorithms. In each algorithm, we have to evaluate the bitmask returned by an AVX2 comparison. This mask contains set bits in each lane where the comparison evaluated to true and unset bits otherwise. One way to branch on an AVX2 comparison is the `vptest` instruction. Another, more flexible way is to create a bitmask in a general purpose register derived from the comparison result using the `vpmovmskb` instruction. We can then count the number of bits set in the mask to decide which branch to take or use the count directly without any branch. More information on the evaluation of SIMD comparison bitmasks can be found in [14].

The sequential search is vectorized by simply loading and comparing m consecutive keys in parallel each iteration, where m is the number of keys per SIMD word.

The binary search can be vectorized by viewing the array as a sequence of SIMD word sized blocks. A binary search is then performed on these blocks until the lower bound is localized to a single block [12, 15]. Once the search range is narrowed down to a single block, the position of the lower bound is determined like outlined above.

Schlegel et al. [12] propose to use SIMD to parallelize index computations and key comparisons of a k-ary search by fitting the $k-1$ separator elements and their indices in a single SIMD register. Hence, k is determined by the width of the SIMD registers and the larger of the key and index bit width. Assuming AVX2 with 32-bit indices and keys, $k = 9$. We thus implemented a vectorized branchless uniform k-ary search. It uses AVX2 gather instructions to load separators from non-continuous memory locations.

2.2.2 Branch Elimination

Modern processors are deeply pipelined, relying on branch prediction to be able to continue executing instructions immediately after they have encountered a conditional jump [6]. Consequently, mispredicted branches are costly [1]. Therefore it is worthwhile to eliminate unpredictable branches from programs. The branches in the loops of our search algorithms often fall in this category, since generally the search keys are not predictable.

One way to eliminate branches is by branch predication, i.e., both sides of a branch are executed and only the effects of the side that was actually needed are kept. The x86 architecture supports branch predication via conditional move instructions (`cmov`). The scalar sequential search can make use of conditional moves by always visiting all array elements and updating the position of the lower bound only as long as the lower bound has not yet been passed. Branches from the inner search loop over the $k-1$ separator elements in the k-ary search can be eliminated similarly. We also used branch predication in the binary and vectorized binary search, where either the left or the right boundary of the search range is updated. Note that the vectorized binary search retains one branch in the search loop to handle cases where the lower bound is located in the current SIMD word. Also, exact-match search algorithms (not based on a lower bound) always retain one branch in the search loop allowing them to terminate when an exact match is found.

Sometimes, a branch can be avoided. This is the case in the vectorized sequential search, where the number of set bits in a comparison result mask is directly used to update the location of the lower bound [14]. In the uniform binary search, there is no need to manually eliminate branches, because the compiler predicated the single branch in the search loop by itself. Similarly no branch elimination was necessary in the vectorized uniform k-ary search, because the starting index of the selected partition is computed directly from the comparison result without a branch.

2.2.3 Loop Unrolling

Loop unrolling improves performance by reducing the time spent executing loop control instructions. We have unrolled the loop in the scalar sequential search 2, 4, 8 and 16 times. The vectorized sequential search was unrolled 2, 4 and 8 times. We did not unroll the vectorized search further, since its code size is about twice that of the scalar variant, leading to a higher register pressure [4, 6].

Since the number of iterations in the uniform binary search only depends on the logarithm of the array size, we have unrolled all possible iterations of the search loop in the scalar and vectorized variant. Our implementations of scalar k-ary searching employ an inner loop over the $k-1$ separator elements. Although, this loop was automatically unrolled by the compiler, we also manually unrolled these loops.

2.2.4 Software Prefetching

A processor may start loading cache lines not yet accessed by the program, expecting them to be accessed soon due to the principle of locality [6]. This is called prefetching. Prefetching can be triggered by the hardware itself, or by using special instructions in software [1].

Due to its perfectly sequential memory access, the sequential search does not need extra software prefetching. In contrast, the scalar and vectorized binary search keep track of the next two possible separator elements, so that they can be prefetched one iteration before they are needed. The regular spacing of the separators in the uniform binary search simplifies keeping track of the possible next separator elements, thus reducing the amount of index computations needed for software prefetching. We have implemented variants looking one iteration ahead like for the non-uniform binary search, but also variants looking two iterations ahead and therefore prefetching four separators each search step. Notably, prefetching stops in the last iterations, because the cache lines containing the next separators are already loaded.

Moreover, we added software prefetching to the branchless uniform k-ary search by splitting the search loop into two variants. The first prefetches the $k-1$ separators in each of the k partitions while iterating over them, the second is unmodified. We use the unmodified loop for the last $\lceil \log_k(m) \rceil$ iterations, where m is the number of keys per cache line.

3. METHODOLOGY

We compare the search algorithm variants from the last section by the time needed to search for a single key in a tightly packed array of 32-bit signed integer keys. By averaging over a batch of 10,000 searches we obtained reproducible runtimes. Additionally, the caches were prepopulated by executing 10,000 search runs before we started the measurement. The behavior of the algorithms when starting from an empty cache are explored in Section 5.1.

The largest arrays used in the evaluation contain 2^{28} keys yielding a size of 1 GiB with 32-bit keys. By going up to this size, we can capture any changes in runtime from cases where the whole array fits into L1 cache up cases where nodes of the conceptual search trees spanned by the binary and k-ary search algorithms do not fit into L3 cache.

How many nodes in these search trees are available in the caches is critical for performance. Therefore, we consider two different schemes to select the keys to search for. The first scheme (Scheme 1) uniformly draws random keys from the array. Thus the whole search tree is evenly accessed. The second scheme (Scheme 2) works by randomly selecting 128 keys from the array and then randomly drawing from these keys for 2000 searches. After every 2000 searches, 128 new keys are selected. In doing so, only a small subset of the paths through the search tree is accessed. The evaluation is mostly based on Scheme 1. When we do not explicitly state that Scheme 2 was used to generate the search keys, Scheme 1 was used. We will only refer to Scheme 2 when there is a significant difference to Scheme 1.

Note that the distribution of the keys in the arrays has little influence on the evaluation, because the search keys are selected from random array positions and not from an underlying key distribution. In this evaluation, the arrays simply contain uniformly distributed keys.

The experiments were conducted on a single thread with an otherwise idle system. This means, the search algorithms had almost the entirety of the processor cache size and bandwidth exclusively available. This limits the applicability of our results, since in a real world scenario other code would pollute the caches. The processor used was an Intel Xeon E5-2630 v3. This processor has 32 KiB of L1 instruction cache, 32 KiB of L1 data cache and 256 KiB L2 cache per core. Additionally, it has 20 MiB of shared L3 cache. To analyze the micro-architectural resource usage, we utilized the performance monitoring counters available in the evaluation system. The specific performance events used are stated in footnotes when referring to special performance counters. See the Intel Software Developer's Manual for a description of the counters and their settings [2].

All algorithms were implemented in C++ and compiled with GNU C++ compiler version 5.4.0 on the highest optimization level (`-O3`). In some cases we had to resort to inline assembler to apply branch elimination. Automatic loop vectorization was disabled.

Goal of the Evaluation. Our goal is to determine the best search algorithm and set of optimizations minimizing the time required to locate an integer key in a sorted array. To this end, we first conduct an intra-algorithm evaluation for the sequential, binary and k-ary search to analyze the impact of software optimization techniques on their runtime. In doing so, we determine the best implementation variant of each algorithm depending on the size of the input array.

While the exact runtimes and array size class boundaries depend on the specific hardware, the same algorithm variants should perform well on all similar systems. We then use the knowledge gained in the intra-algorithm evaluation to conduct an inter-algorithm comparison, where we determine the best implementation variants from the complete set of best sequential, binary and k-ary search algorithms.

4. INTRA-ALGORITHM EVALUATION

In the following, we evaluate the effectiveness of branch elimination, loop unrolling, software-controlled prefetching and vectorization in sequential, binary and k-ary searching.

4.1 Sequential Search

For the sequential search we consider branch elimination, loop unrolling and vectorization. We exclude software prefetching, since the sequential access pattern of these algorithms is already handled by the hardware prefetcher. Since the algorithm behavior of branch elimination and loop unrolling is similar to other work, we exclude performance graphs for brevity and refer to [13].

4.1.1 Branch Elimination

We compared the average search time for the branching search implementation with their branchless counterparts, for scalar and vectorized searches. Since the branchless variants always iterate over the whole array, they get slower compared to the branching ones the larger the array becomes. However, on small array the cost of a mispredicted branch exiting the search loop outweighs the benefits of earlier termination. This break-even point is at 28 keys for the scalar and about 135 keys for the vectorized implementation.

4.1.2 Loop Unrolling

Unrolling the loop of the scalar sequential search variants does not yield consistent improvements. There is a slight speedup for some array sizes below 100 to 200 keys of up to 5%, but otherwise the runtimes are worse. Repeating the same experiment on a different processor (Intel Core i7-3610QM) we observed speed improvements of over 20% in the scalar branching search and some slight improvements in the four times unrolled scalar branchless search. Furthermore, we found the compiler has a significant influence on the effectiveness of loop unrolling in the sequential search.

We obtained more consistent results for the vectorized sequential search algorithms, where the branching variant generally profits by about 5% from loop unrolling on arrays of more than 200 keys and is never slower than a not unrolled implementation. Similarly, the branchless implementation unrolled four or eight times is accelerated by about 10% on arrays larger than 250 keys. However, the two times unrolled variant is slower than the not unrolled base version.

Given the strong influence of compiler and hardware, loop unrolling in the sequential search requires careful tuning to be effective.

4.1.3 Vectorization

In Figure 2, we compare the fastest scalar and vectorized search functions. From Figure 2a we conclude that the branchless vectorized implementation offers the best performance for arrays with up to about 135 elements. For arrays with more elements, the branching vectorized search is better. Figure 2b shows the speedup obtained by vectorization. At an array size of 100 elements, the branching vectorized search processes about 30% more keys per second than the scalar implementation. The difference is greater in the branchless search, where vectorization increases the search speed by a factor of 2.3 for 100 elements. At an array size of 4096 keys, the speedup factor has converged to 3.5 for the branching and 4.9 for the branchless search.

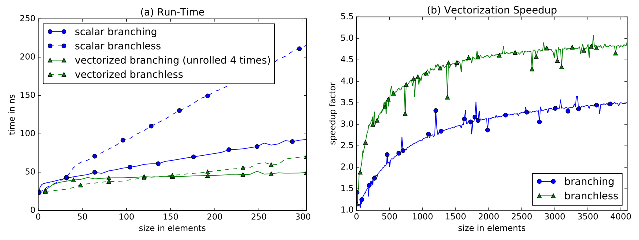

Figure 2: Comparison of scalar and vectorized (AVX2) sequential search implementations.

4.1.4 Best Sequential Search Algorithms

For the sequential search, vectorized implementations are superior to scalar ones. For arrays of 135 keys or less, the branchless version is to be preferred. Loop unrolling does not significantly improve it. On larger arrays the branching implementation becomes superior. In contrast to the branchless variant, it profits from unrolling the search loop at least four times. Table 2 summarizes these results.

Table 2: Best sequential search algorithms.

Keys	Best Lower-Bound Sequential Search
≤ 135	Vectorized branchless sequential search
> 135	Vectorized branching sequential search unrolled 4 times

4.2 Binary Search

Next we evaluate the interaction of the uniform and non-uniform binary search with branch elimination, software prefetching, loop unrolling and vectorization.

4.2.1 Branch Elimination

The runtime of the binary search, its branchless version and the uniform binary search, which is also branchless, are plotted in the first row of Figure 3. The size of the L1, L2 and L3 caches are marked with dashed lines. As can be seen, the branchless algorithms are faster than the branching implementation for arrays up to a size of 2^{17} elements. This is twice the size of the L2 cache. For larger arrays, their performance declines.

The increased runtime of the branchless implementations can be explained with significantly more stalled clock cycles[2] than in the branching implementation for arrays of more than 2^{17} keys (Figure 3c). The branching search has less stalled clock cycles, because the processor speculates on the outcome of the conditional branches and can continue with the next iteration before the necessary separator elements are actually loaded. Such speculation is not possible in the

[2]Performance event `RESOURCE_STALLS.ANY`

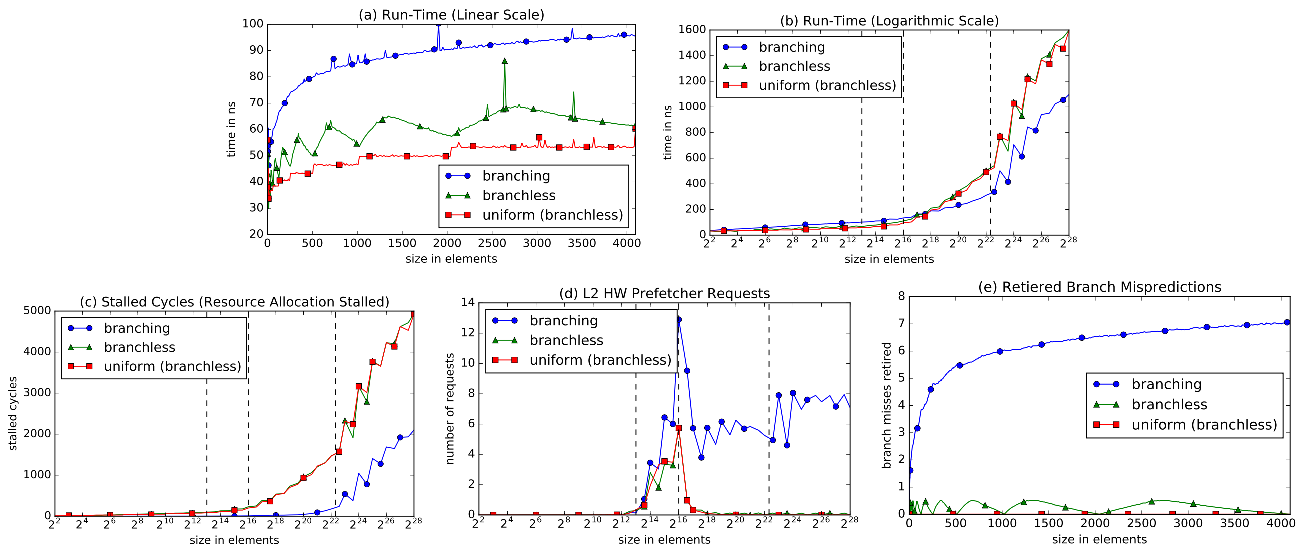

Figure 3: Branch elimination applied to the binary search (vertical lines represent respective cache sizes).

branchless implementation. Additionally, the branching binary search makes use of hardware prefetching. Figure 3d shows the number of L2 hardware prefetcher requests[3]. As we can see, for more than 2^{17} keys the prefetcher is only active in the branching implementation.

The linearly scaled runtime graph in Figure 3a shows differently shaped curves for the three algorithms on smaller arrays. The branching, non-uniform search closely follows the logarithmic curve we expect. The uniform binary search performs exactly one additional iteration for each doubling of the array size, resulting in steps at powers of two. In contrast to the former two, the branchless non-uniform search oscillates with minimums at power of two element counts and maximums about one third of the way to the next power of two. These oscillations are explained by the number of retired branch mispredictions [4,5] plotted in Figure 3e. At power of two array sizes, the non-uniform binary search runs for exactly $\log_2$(array size)+1 iterations, independently from which key is searched for. If the array size is not a power of two, either $\log_2$(array size) or $\log_2$(array size) + 1 iterations are needed. This makes the jump at the end of the search loop harder to predict than in the uniform variant.

4.2.2 *Prefetching*

For the evaluation of the software prefetching algorithms, we used the `T0` locality hint, i.e., the `prefetcht0` instruction, loading into all cache levels. We found this generally gives the best performance.

Since hardware prefetching did not work for larger arrays in the branchless implementations, software controlled prefetching might yield an improvement. The left column of Figure 4 shows our results for the non-uniform binary search. As we can see, adding software prefetching only marginally improves the runtime on arrays below the 2^{17} elements mark. On larger arrays however, the branchless binary search is significantly faster if prefetching is employed. The difference with software prefetching in the branching implementation is less apparent, but becomes more visible once the array size surpasses 2^{26} keys, since DRAM accesses are by several factors more expensive.

Remember that the uniform binary search algorithms conditionally call the prefetch intrinsic to avoid redundant instructions. We found that this significantly reduces the execution time for arrays of less than 2^{18} keys with almost no change in runtime on even larger arrays. The middle column of Figure 4 shows, that prefetching one iteration in advance (prefetch 2 keys) increases the performance of the uniform binary search if the array has more than about 2^{16} keys. For smaller arrays however, prefetching adds a few nanoseconds of overhead to the runtime. Naturally, this overhead is even larger in the implementation prefetching two iterations in advance (prefetch 4 keys). Nevertheless, prefetching two iterations in advance becomes slightly faster than prefetching just one iteration in advance for arrays larger than 2^{17} keys. Figure 4c shows the runtime difference between prefetching two and four keys in more detail. Interestingly, the largest speed difference occurs around the size of the L3 cache between 2^{20} and 2^{23} keys, where prefetching four keys is about 25% faster than prefetching just two keys.

In Figure 4f, we show the number of L3 cache misses[6] caused by the different implementations. The uniform and the non-uniform branchless binary search have the least number of cache misses, but are both slow on large arrays. As expected, prefetching two keys each iteration in the uniform and non-uniform branchless search causes more cache misses, but it also significantly improves the performance of the algorithms. The uniform search prefetching four keys each iteration causes yet more cache misses for a slight performance improvement on large arrays. The unoptimized branching binary search causes the most cache misses, yet it is never the fastest implementation. Therefore the other algorithms should be preferred.

[3]Performance event `L2_RQSTS.ALL_PF`

[4]Branch instructions whose outcome was mispredicted, but that were not executed due to false speculation themselves.

[5]Performance event `BR_MISP_RETIRED.ALL_BRANCHES`

[6]Performance event `LONGEST_LAT_CACHE.MISS`

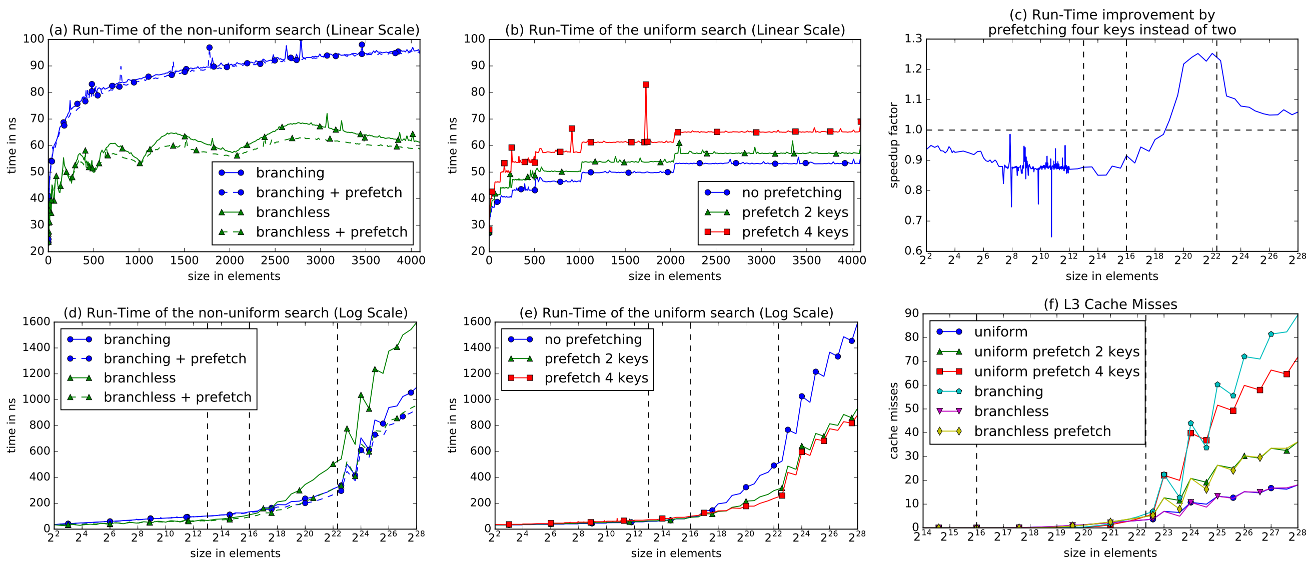

Figure 4: Software prefetching applied to the non-uniform and uniform binary search.

4.2.3 Loop Unrolling

In Figure 5, we show the speedups obtained by loop unrolling. In the non-prefetching uniform binary search, loop unrolling has the most noticeable benefit for arrays of between 2^{18} and 2^{26} keys. If prefetching (four keys per iteration) is used, there are improvements of about 5% on arrays of less than 2^{18} keys. We have excluded the variant prefetching two keys per iteration from this comparison, since the compiler did not generate branch-free code for the unrolled version. These results differ from our earlier evaluation [13], where loop unrolling had no benefit using a different compiler. We conclude that the effectiveness of loop unrolling in a high-level language like C++ depend on the compiler.

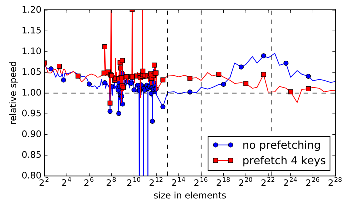

Figure 5: Speedup achieved by unrolling the uniform binary search.

4.2.4 Exact-Match Search

In this section, we compare direct exact-match search implementations with implementations in terms of a lower-bound search. The former have an equality test in the search loop terminating the search if the search key has been seen. In contrast, the latter have the same search loop as a lower-bound search and delay the equality test to the end. Since the search keys are drawn from the array being searched in (see Section 3), the search key is always found and the direct exact-match implementation can make use of its early exit condition. Consequently, the direct exact-match searches need almost precisely one iteration less than the lower-bound-based exact-match search.

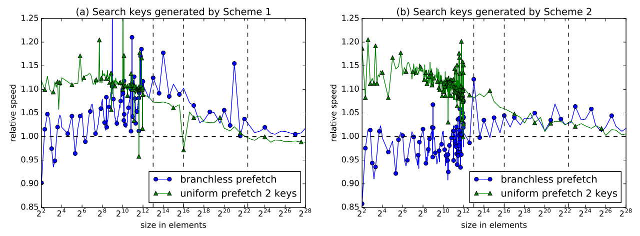

Figure 6: Speedup obtained by a lower-bound-based exact-match search over a direct implementation.

Figure 6 shows the speedup obtained for a lower-bound search to find an exact match instead of using a direct implementation. On the left side, the search keys were drawn from the whole array (Scheme 1), and on the right side only from a small randomly determined subset (Scheme 2). In case of the (non-uniform) branchless binary search, the lower bound based is almost always faster than the direct implementation. On the other hand, the lower-bound-based uniform exact-match search becomes inferior to the corresponding direct variant on large arrays, but only in Scheme 1. In all cases, the performance advantage of the lower-bound-based search algorithms diminishes on very large arrays.

In conclusion, removing the potentially expensive equality test from the search loop is often worth more than saving one search iteration, but cannot be recommended for arrays much larger than the processor cache.

4.2.5 Vectorization

We compared the runtimes of the vectorized binary search using the `ptest` instruction and the alternative implementation using a combination of `pmovmskb` and `test/cmp`. We found little to no difference between them.

The effect of optimizing the vectorized binary search by removing branches and adding software prefetching can be seen in Figure 7. Note that in contrast to the scalar binary search there is also a branching variant of the vectorized

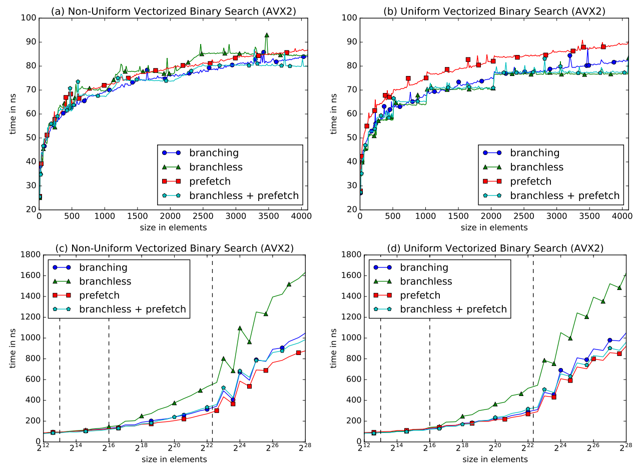

Figure 7: Branch elimination and prefetching in the vectorized (AVX2) binary search.

uniform binary search. Just eliminating a branch in the search loop again yields an algorithm becoming very slow on large arrays. A branching implementation with software prefetching offers the best performance for both the uniform and non-uniform algorithm. On smaller arrays of less than 2^{15} keys, the optimizations do not make a large difference, with the exception of the branching uniform search with prefetching, where a branchless implementation should be preferred. An additional unrolling of the vectorized binary search did not yield any benefits.

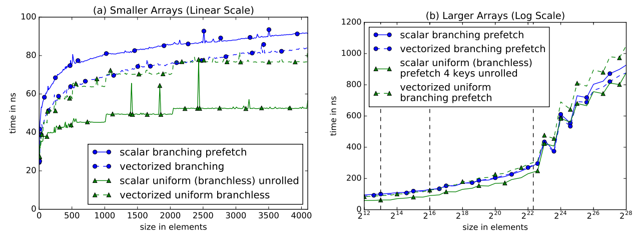

Figure 8: Comparison of scalar and vectorized binary search implementations.

Figure 8 shows the runtimes of the fastest scalar and SIMD lower-bound search algorithms on smaller ($\leq$ 4096 keys) and larger ($>$ 4096 keys) arrays. In both cases, a vectorized non-uniform binary search is faster than the fastest scalar implementation. The opposite is true for the uniform binary search, where an optimized scalar implementation is clearly superior to the vectorized version. An optimized scalar uniform binary search is preferable over vectorization. We conclude, that the additional complexity introduced by using SIMD increases the runtime more than it is decreased by slightly reducing the number of iterations.

4.2.6 *Cache Utilization and Offset Binary Search*

The binary search algorithms evaluated so far exhibit an unfortunate interaction with the way the processor caches are indexed. Consider Figure 9 showing the runtime of an optimized (branching, prefetching) non-uniform and an optimized (branchless, prefetching 4 keys, unrolled) uniform binary search. In Figure 9a, the search keys were randomly selected from the whole array (Scheme 1 described in Section 3), and in Figure 9b, the keys were drawn from a small subset of array elements (Scheme 2). Once the number of keys surpasses the size of the L3 cache (20 MiB $\approx 2^{22.3}$ keys), the runtime grows rapidly and the graphs become very spiky. The peaks occur at powers of two and are more pronounced if the better cacheable Scheme 2 is used to select the search keys, hinting at a problem with caching.

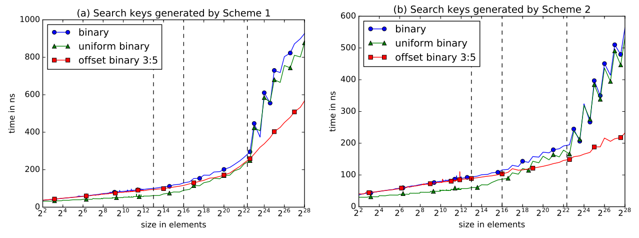

Figure 9: Inefficient cache usage of the binary search on large arrays.

The problems with the binary search on larger arrays, especially when the size is a power of two, is explained by Figure 10. It shows the number of unique memory addresses accessed, not including prefetching, falling into each set of the L1 cache. The data was collected over 10,000 search runs with randomly selected search keys on an array of 2^{24} keys. The diagram only shows the regular non-uniform binary search and the offset binary search, but the uniform binary search behaves almost identical to the regular non-uniform variant. As we can see, the access distribution of the regular binary search is very spiky and therefore many separator elements compete for the same set in the cache. This results in an inefficient usage of the available space.

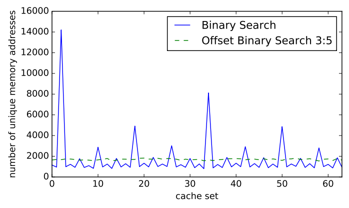

Figure 10: Number of unique memory accesses falling into each L1 cache set.

The aliasing of many separator elements to the same cache sets arises, because of the formula used to index them, in our case of an 8-way set associative L1 cache *cache line address* mod 64, and the search algorithm selecting separator keys approximately spaced powers of two apart. The effect is worst for the uniform binary search, since it directly uses powers of two to calculate the separator indices and the non-uniform binary search on arrays with a power of two size, since then most keys fall into the same cache set.

We recorded virtual addresses to generate the graphs in Figure 10. For the L1 cache this is correct, because our test machine actually indexes it with virtual addresses. The situation is more complex for the L2 and L3 cache, since higher cache levels are typically indexed with physical addresses. If the array is in a continuous range of physical memory, the

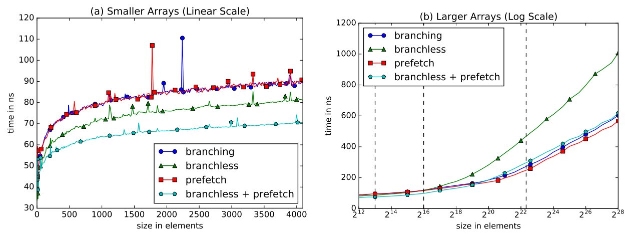

Figure 11: Runtime of the 3:5 offset binary searches.

plots would look like in Figure 10. Otherwise, the location of pages in physical memory and the distribution of search keys will influence the aliasing effect, giving the binary search an unreliable performance.

Fortunately there is a way to remedy the inefficient cache usage of the binary search. By not dividing the search ranges in two (almost) equally sized halves but in an unequal ratio, we can avoid powers of two based memory accesses. The runtime of an offset binary search using a split ratio of 3:5 is plotted in Figure 9. If the search keys are generated by Scheme 1, it becomes the fastest binary search algorithm for more than 2^{23} keys. In Scheme 2 it becomes the fastest algorithm even earlier at about 2^{18} keys.

The algorithm plotted in Figure 9 employs software prefetching. As we can see in Figure 11 this optimization provides the best performance on large arrays. We have not found a significant difference between equal split ratios in the left and right partitions and mirrored ratios. Since mirroring the split ratios adds additional complexity to the implementation, we do not recommend it.

4.2.7 Best Binary Search Algorithms

In Table 3, we summarize which binary search algorithm performs best depending on the size of the arrays to search in. Note that the break-even point between offset and regular binary searching strongly depends on the search key distribution. Distributions exhibiting a high locality of reference encounter caching issues earlier and favor the offset binary search more. For example, the offset binary search already becomes faster than the regular variants for 2^{18} keys if Scheme 2 is used to generate the search keys.

Table 3: Best lower-bound binary search algorithms.

Keys	Best Lower-Bound Binary Search
< 40	Different algorithms, unrolled branchless uniform binary search works consistently well
$40 - 2^{13}$	Unrolled branchless uniform binary search
$2^{13} - 2^{23}$	Unrolled branchless uniform binary search prefetching keys two iterations in advance
$> 2^{23}$	Offset binary search (ratio 3:5) prefetching separator keys one iteration in advance

4.3 k-ary Search

The k-ary search aims at reducing the number of search iterations by using not 1 (binary search) but $k-1$ separator elements (cf. Section 2.1.3). In Figure 12, we compare uniform and non-uniform variants of the scalar k-ary search. We show results with k from 3 to 9, since the performance quickly declines for larger k. The k plotted in Figure 12 are the ones yielding the best performance for each variant.

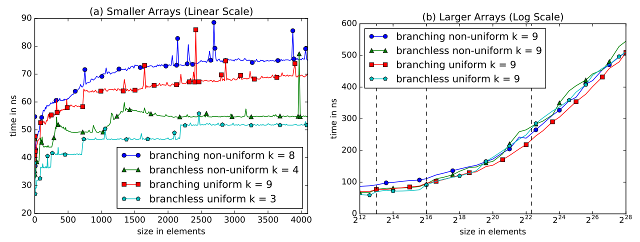

Figure 12: Runtimes of non-uniform and uniform k-ary search with and without branches in the loop.

For arrays of up to 2^{17} keys, the branchless uniform ternary search is the fastest algorithm. On larger arrays, the branching uniform 9-ary search becomes faster. Similar to the binary search, the branching implementations profit from speculative execution and hardware prefetching, giving them better performance on larger arrays.

Note that the cache aliasing effects observed in the binary search also apply to the k-ary search, so powers of two are a bad choice for k on larger arrays. On smaller arrays of less then 2^{16} keys however, the branching non-uniform k-ary search works well with $k = 4$ or 8, because they lead to simpler index computations.

4.3.1 Loop Unrolling

Concerning loop unrolling, we only considered the inner loop over the $k-1$ separator elements. In all cases, this loop was automatically unrolled by the compiler, but we also tried to manually unroll these loops. For the uniform k-ary search this did not improve runtimes. However, the manually unrolled non-uniform binary search is slightly faster than the automatic unrolling by the compiler. The graphs in Figure 12 show the automatically unrolled uniform and the manually unrolled non-uniform implementations.

4.3.2 Prefetching

We augmented the branchless uniform k-ary search with software prefetching. Each iteration, the keys potentially needed for the next search step are prefetched. Figure 13 shows the results. It is clear, that for larger k there is simply too much prefetching. In fact, $k = 3$ is the only case, where prefetching improves the runtime. The improvements only occur for arrays larger than about 2^{16} keys, but are quite substantial. The comparison between the 3-ary search with software prefetching and the 9-ary search without additional prefetching is interesting, because they both load eight search keys in each iteration: Two keys are loaded for comparisons and six are prefetched in the prefetching 3-ary search, whereas in the non-prefeching 9-ary search eight keys are directly loaded for comparisons. As we can see $k = 3$ with software prefetching is faster than $k = 9$ just relying on the hardware prefetcher. This could be due to a limit in line fill buffers on modern processors [1].

4.3.3 Exact-Match Search

We compared exact-match search implementations, terminating the search as soon as the search key has been seen, and implementations based on the lower bound moving the equality tests out of the search loop, like we did for the binary search in Section 4.2.4. For this evaluation

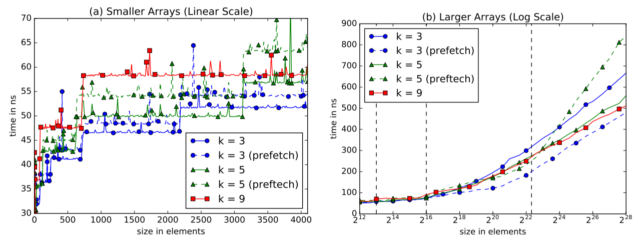

Figure 13: Scalar branchless uniform k-ary search with software prefetching.

in Figure 14, we tested $k = 3$ and 5, since these give consistent performance across all implementations and array sizes. Note, that the search keys are drawn from the array being searched in so that all searches are successful (see Section 3).

Remember that the lower-bound-based exact-match binary search needs about one iteration more than the direct exact-match implementation. For the k-ary search we expect this difference to be lower, because the conceptual search tree has up to k times more leaves than internal nodes. This means, the search is more likely to only discover the search key in the last iteration, the greater k is. For the uniform k-ary search we measured about 0.5 iterations difference if $k = 3$ and 0.25 iterations difference if $k = 5$. The differences for the non-uniform k-ary search are slightly larger with on average 0.6 and 0.4.

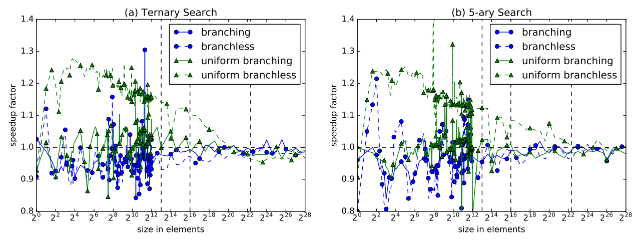

Figure 14: Speedup obtained by a lower-bound-based exact-match search over a direct execution.

A noticeable performance improvement obtained by using a lower-bound-based exact-match search occurs in the uniform branchless k-ary search. On arrays of up to 2^{20} keys, the lower-bound-based approach is significantly faster than the direct implementation. However, for the non-uniform k-ary search, the direct implementation is slightly faster. For the branching uniform k-ary search, there is no preference.

4.3.4 Vectorization

In Figure 15, we compare the vectorized k-ary search using AVX2 with the corresponding scalar branchless uniform k-ary search. We include runtimes on arrays with the usual 32-bit keys (int32), but also with 64-bit keys (int64). The k of the scalar search is chosen to match the vectorized search with 256-bit AVX registers. In all cases, the vectorized search is slower than the scalar variant, due to its expensive fetching of separator keys into registers. Note, a linearized k-ary tree improves its performance drastically [13].

4.3.5 Best k-ary Search Algorithms

The fastest k-ary search algorithm in our evaluation is the scalar branchless uniform k-ary search. On arrays of 2^{14} or fewer it obtains its top performance with $k = 3$. If the array is larger, software prefetching should be used with the same algorithm and the same k. The runtime plots of these algorithms can be found in Figure 13. In Table 4, we summarize our results.

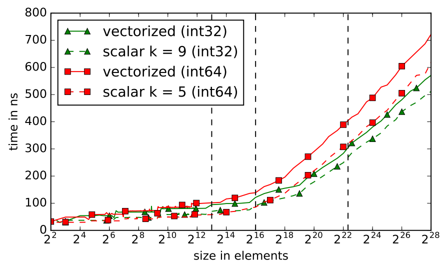

Figure 15: Comparison of the vectorized and the scalar branchless uniform k-ary search.

Table 4: Best lower-bound k-ary search algorithms.

Keys	Best Lower-Bound k-ary Search
$\leq 2^{14}$	Branchless uniform 3-ary search
$> 2^{14}$	Branchless uniform 3-ary search with prefetching

5. INTER-ALGORITHM EVALUATION

In this section, we want to determine the best combination of search algorithm and optimizations for searching in sorted arrays. To this end, we compare the runtimes of the best sequential, binary and k-ary search algorithms determined in the previous sections.

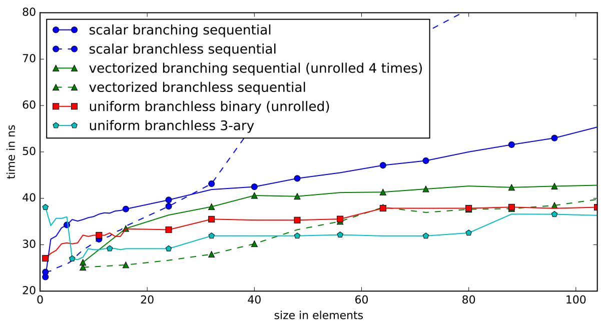

Figure 16: Fastest sequential, binary and k-ary search algorithms on small arrays.

In Figure 16, we consider small arrays of less than 100 keys, where sequential search algorithms might be superior to the more complicated binary and k-ary algorithms. This is the case for the vectorized branchless sequential search, which is the fastest algorithm for up to 48 elements. For larger arrays, the uniform branchless ternary search is faster.

Figure 17 shows the runtimes of the fastest binary and k-ary search implementations on arrays of up to 4096 keys. Again, the uniform branchless ternary search performs best. Other promising algorithms are the unrolled branchless uniform binary search and the uniform branchless 5-ary search. Since the runtime of the uniform implementations increases in steps, both can be sometimes slightly faster than the ternary search.Between 2^{13} keys (32 KiB, size of the L1

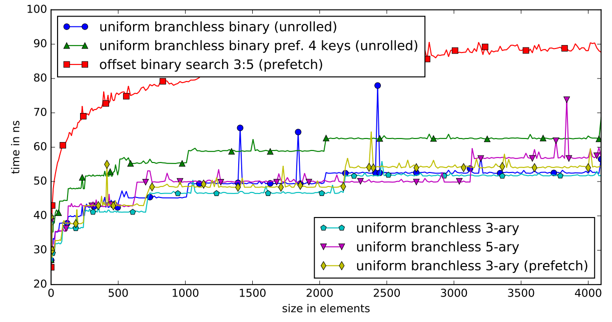

Figure 17: Binary and k-ary search variants on arrays of up to 4096 keys.

cache) and 2^{16} keys (256 KiB, size of the L2 cache) the uniform branchless ternary search with prefetching becomes faster than the variant without prefetching. As shown in Figure 18, it is the fastest algorithm for more than 2^{16} keys. Neither the optimized offset binary nor the uniform 5-ary search, that have a very similar runtime, reach its performance. It is noteworthy that between 2^{16} and 2^{23} keys (approx. size of the L3 cache) the unrolled branchless uniform binary search with prefetching is slightly faster than the offset binary and uniform 5-ary search, but on larger arrays it does not use the cache efficiently and performs much worse.

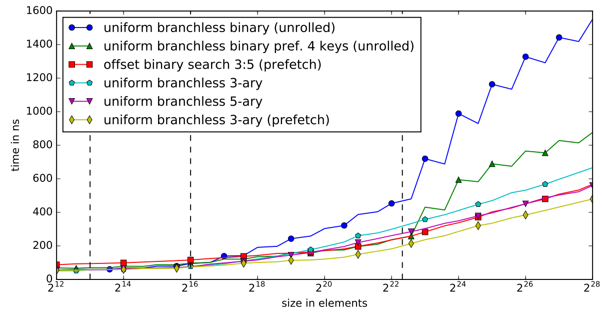

Figure 18: Fastest search algorithms on large arrays.

In Figure 19, we look at the number of L3 cache misses caused by the best performing search algorithms. Since the number of last level cache misses corresponds to the main memory bandwidth consumed by the algorithm it might be preferable to select an algorithm with few L3 cache misses. Looking at the cache misses we can differentiate between the offset binary search with prefetching and the uniform 5-ary search, that have almost the same runtime. Since the 5-ary search reaches this runtime with fewer cache misses than the prefetching offset binary search, it is the better algorithm.

There are some noteworthy differences considering search keys generated by the better cacheable Scheme 2 instead of Scheme 1: For up to 2^{16} keys, uniform binary search variants are faster than the uniform branchless ternary search. The unrolled branchless binary search is the fastest implementation. Furthermore, the prefetching uniform branchless ternary search has no advantage over the non-prefetching k-ary search variants. For large arrays of more than 2^{17} keys, the unrolled uniform k-ary search algorithms work best, especially the unrolled uniform 9-ary search. Binary search

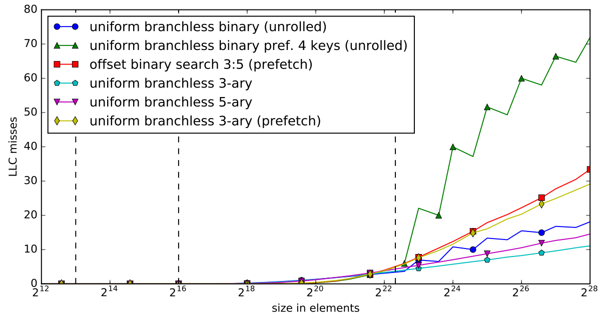

Figure 19: Number of last level cache misses caused by optimized binary and k-ary search algorithms.

variants are still far behind on large arrays. Table 5 summarizes which variants perform best on different array sizes. It also names variants almost as fast as the best algorithm, which might be worth considering.

Table 5: Best lower-bound search algorithms.

Keys	Best Lower-Bound Search
≤ 48	• Vectorized branchless sequential search
$49 - 2^{16}$	• Branchless uniform 3-ary search Alternatives: • Unrolled branchless binary search (esp. Scheme 2) • Branchless uniform 5-ary search
$2^{16} - 2^{23}$	• Branchless uniform 3-ary search with prefetching Alternatives: • Unrolled branchless uniform binary search prefetching separator keys two iterations ahead • Branchless uniform 5-ary search
$> 2^{23}$	• Branchless uniform 3-ary search with prefetching Alternatives: • Branchless uniform 5-ary search • Branchless uniform 9-ary search (for Scheme 2)

5.1 Cache Warm-Up

Until now, we only considered runtimes after 10,000 warm-up iterations. Now, we describe the behavior of the best binary and k-ary search algorithms recommended in Table 5 during the warm-up phase. Note that the search keys were generated using Scheme 1 to capture the worst case of random accesses over the whole array. If there is more locality to the accesses, there can be much shorted warm-up periods.

For less than 2^{13} keys (32 KiB, size of the L1 cache), there is only a very short warm-up phase of less than 50 iterations. As the array becomes larger, the number of warm-up iterations increases. For up to 2^{18} keys the implementations employing prefetching reach their stable performance faster than other algorithms. Figure 20a shows the runtime of a single search after a certain number of preceding search runs starting from a cold cache. Even though the non-prefetching 3-ary and 5-ary search reach an almost identical runtime to the prefetching 3-ary search, they only achieve it after about three times as many searches.

On larger arrays, the algorithms exhibit a similar number of warm-up iterations, but there is still a difference in the number of cache misses. Figure 20b shows the average number of L3 cache misses during a single search. Implementations utilizing software prefetching or enabling more hardware prefetching like the 5-ary search, initially have a

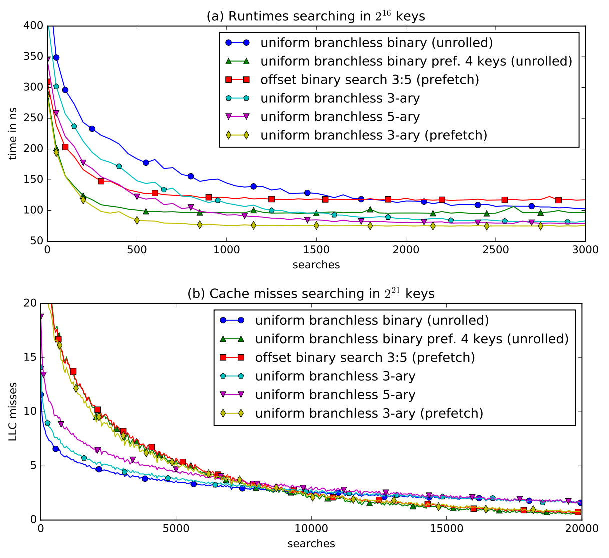

Figure 20: Runtime and LLC misses over the total number of search runs.

higher number of cache misses, but reach almost no cache misses after 15,000 search runs.

Arrays of 2^{23} and more keys are larger than the L3 cache (20 MiB), so LLC misses become unavoidable. Due to the inefficent cache usage of the non-offset binary search especially with power of two array sizes, it has is almost no warm-up phase anymore and immediately reaches a stable number of LLC misses. The other algorithms reach stable runtimes after about 10,000 searches, though slight improvements can be observed after even more searches.

Nevertheless, the general algorithm performance is not affected by the warm-up phase with the uniform branchless ternary search being the best algorithm for 2^{16} keys.

5.2 Hardware Differences

Since our results are specific to the used hardware, the break-even points have to be re-determined for different CPU architectures. However, compared to our previous evaluation [13] using an Intel Core i7 3610QM, the overall algorithm behavior is the same, although the clock frequency of the machines differs by a factor of 1.47. Still, they share the same L1 and L2 cache size while their L3 cache size also differs by a factor of 3.33.

Comparing our former results to the results in this paper, there are two interesting observations: (1) comparing corresponding experiments, the plots share the same peaks to a large extent, and (2) peaks or break-even points are shifted by a factor of around 1.5 for the sequential search or by factors between 4 and 8 for binary and k-ary searches.

The first observation indicates, that the peaks are no outliers or only specific to a machine but an algorithmic artifact due to its overall design (e.g., access pattern) and will probably persists over processor generations. The second one indicates that the sequential scan is mainly accelerated by the higher clock frequency. The increased cache size seems to have a rather marginal impact. Instead, binary and k-ary search seem to additionally benefit from bigger L3 cache sizes. This seems logical, because there will be more separator elements cached reducing costly DRAM accesses.

6. IMPLICATIONS TO THE DATABASE DOMAIN

A key result of our evaluation is that the best search algorithm clearly depends on the expected size of the array to be searched in. From this result, we see direct implications to two different levels of database research: (1) hybrid searches in sorted lists and (2) index structures.

Hybrid Search Algorithms. In an abstract sense, the smaller the remaining array to be searched in, the less separator keys should be examined to find a promising partition and vice versa. Given this observation and rather stable break-even points, it is a good idea to adapt a search algorithm to use a mix of the proposed optimized search algorithms. The idea is to start a ternary search as long as the remaining partition is large enough. Afterwards, the search follows a binary search till the remaining partition is in the 10s of keys such that a sequential search is more beneficial. Notably, given a specific array size, a code generator could wire the steps of the search algorithms to cause minimal overhead.

Index Structures. There is a multitude of index structures that rely on sorted node entries and searching the right node entry to follow is an important operation. Improving this search by choosing the right search algorithm within the node will add a constant speedup to the overall tree search. There are two kinds of trees: (1) those having rather homogeneous node sizes (e.g., B-Tree [5, 11]) and (2) having variable-length node sizes (e.g., Elf [3], Learned Index [8], Adaptive Radix Tree [9]). For the former, a search algorithm for the whole tree can be determined. However for the latter, it is necessary to switch between the search algorithms depending on the size of the node.

7. CONCLUSION

In this paper, we evaluated a large number of optimized search algorithms based on sequential, binary and k-ary searching. We showed that a combination of algorithmic variations and hardware-sensitive optimizations yields significant performance improvements over a standard implementation. Which combination of algorithm and optimizations offers the best performance is first and foremost dependent on the dataset size. Vectorized sequential searching is best suited to the smallest arrays, while uniform binary and ternary searching work best for medium sizes. For the largest arrays, software prefetching and k-ary search algorithms with larger k become worthwhile. Another important factor in choosing the best algorithm is the distribution of the search keys. If a small subset of keys is accessed disproportionally often, less keys need to be held in the processor caches. Therefore, the boundaries of an algorithm's preferred array sizes shift and different cache usage patterns become advantageous. What exact array sizes and search key distributions are favored by which class of algorithms and optimizations will depend on hardware factors like the available SIMD bandwidth and size of the processor caches. Nevertheless, our results should provide a guideline to optimize search algorithms taking the properties of modern hardware into account.

Acknowledgements. We thank the reviewers for their valuable feedback. This work was partially funded by the DFG (grant no.: SA 465/50-1 & SA 465/51-1).

8. REFERENCES

[1] Intel 64 and IA-32 Architectures Optimization Reference Manual, Nov. 2016.

[2] Intel 64 and IA-32 Architectures Software Developer's Manual, Dec. 2016.

[3] D. Broneske, V. Köppen, G. Saake, and M. Schäler. Accelerating multi-column selection predicates in main-memory - the Elf approach. In *International Conference on Data Engineering (ICDE)*, pages 647–658. IEEE, 2017.

[4] D. Broneske and G. Saake. Exploiting capabilities of modern processors in data intensive applications. *it - Information Technology*, 59(3):133–140, 2017.

[5] G. Graefe and P.-A. Larson. B-tree indexes and CPU caches. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 349–358. IEEE, 2001.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth edition, 2011.

[7] D. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, second edition, 1998.

[8] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 489–504, 2018.

[9] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[10] R. Lesuisse. Some lessons drawn from the history of the binary search algorithm. *The Computer Journal*, 26(2):154–163, 1983.

[11] J. Rao and K. A. Ross. Making b+-trees cache conscious in main memory. In *ACM SIGMOD Record*, volume 29, pages 475–486. ACM, 2000.

[12] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 52–60. ACM, 2009.

[13] L.-C. Schulz. Searching in sorted lists on modern processors. Bachelor's thesis, University of Magdeburg, 2017. Available online at `http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisSchulz17.pdf`.

[14] S. Zeuch, F. Huber, and J.-C. Freytag. Adapting tree structures for processing with SIMD instructions. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2014.

[15] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 145–156. ACM, 2002.