University of Magdeburg

School of Computer Science

Bachelor Thesis

Searching in Sorted Lists on Modern Processors

Author:

Lars-Christian Schulz

July 26, 2017

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

Department of Technical & Business Information Systems

M.Sc. David Broneske

Department of Technical & Business Information Systems

Schulz, Lars-Christian:
Searching in Sorted Lists on Modern Processors
Bachelor Thesis, University of Magdeburg, 2017.

Abstract

Sorted lists are one of the simplest, but also one of the most commonly used, data structures. An important operation on sorted lists is searching for a specific element or a range of elements. This work revisits the basic sequential, binary and k-ary ($k > 2$) search algorithms in the context of modern CPU architectures. To this end, we provide multiple implementation variants, including vectorized search functions using the x86 SIMD instruction set extensions SSE, AVX and AVX2, and apply software optimization techniques to them. The optimizations include branch elimination, loop unrolling and software controlled prefetching. We have evaluated the implementations on a modern x86 CPU and identified the implementation variants and optimizations preferred by the processor. In doing so, we found three different list size ranges favoring different algorithms. In particular sequential searching is suited to very small lists and binary searching to larger datasets still fitting in the processor cache. If cache misses are common, we found k-ary searching to offer the best performance.

Acknowledgments

I would like to thank my advisors Prof. Dr. Gunter Saake and David Broneske for giving me the opportunity to write this thesis at the Department of Technical & Business Information Systems. I especially thank David Broneske, whose feedback was very valuable.

I am grateful to Prof. Dr. Thomas Leich for giving me the opportunity of an internship at the METOP GmbH.

Finally, I thank my family for their support and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Listings

1. Introduction

Sorted list are one of the simplest ways to store sets of elements. Nevertheless, they allow for efficient element retrieval, while being more compact than search trees storing explicit pointers or hash tables. They work especially well, if iteration over sorted ranges of elements is required, since the elements are already continuously stored in the correct order.

This thesis deals with searching as one of the most common operations on sorted lists. Specifically, we are concerned with arrays of tightly packed elements of basic types. With basic types we refer to the elementary integral and floating point types a 64-bit processor can store in its registers. The task of searching in such arrays arises in the common case of a program working with sets of pointers, IDs or keys. Since arrays are often sorted especially to facilitate efficient repeated searching, the employed search algorithm will run many times and can have a significant impact on overall execution time. Therefore it is worthwhile to investigate optimized search algorithms as an alternative to standard library search functions that must maintain generality and thus are probably not optimal for every case. This is especially true, if we consider the complex behavior of modern superscalar processors and the ever increasing gap between processor speed and memory latency, that make selecting the right optimizations difficult and machine dependent.

In this work, we compare variants and optimizations of sequential, binary and, as a generalization of binary searching, k-ary search algorithms. We pay close attention to the superscalar, out-of-order and speculative nature of modern high-performance CPUs and employ single instruction multiple data (SIMD) precessing.

If one gives up the simplicity and convenience of a simple sorted array, implicit search trees embedded in an array are an interesting and still very compact alternative. We examine one possible search tree linearization scheme to determine the possible gains in search speed.

Since searching in sorted arrays is a sub-algorithm in many more complex operations, understanding the behavior of these elementary search algorithms is necessary to optimize more complex programs. An important example are database operations, where recent research aims at automatically tuning high level operations to the underlying hardware [BBHS14].

Goal of this Thesis

The goal of this thesis is to examine the performance characteristics and hardware utilization of elementary search algorithms operating on sorted arrays and to some extent on linearized k-ary search trees. To this end, we:

1. Implement classical sequential, binary and k-ary searching in C++, as well as advanced variants including single instruction multiple data (SIMD) enabled algorithms.
2. Apply hardware sensitive optimizations like branch elimination, loop unrolling and software controlled prefetching to the implementations.
3. Evaluate and compare the run-time and system resource usage of the various search algorithms and the differences introduced by different implementation variants and optimizations.

Structure of the Thesis

This thesis begins with a background chapter, giving a summary of the characteristics of modern CPUs relevant to searching and introducing the examined search algorithms. The bulk of the thesis is divided in three chapters, each presenting our implementations, optimizations and evaluation of sequential, binary and general k-ary searching. In Chapter 6, we compare the fastest implementations from the previous chapters with each other to determine the overall best algorithms. The thesis is completed by a chapter on related work and the conclusion.

2. Background

In this chapter we discuss selected characteristics of modern processors relevant to program optimization and introduce various search algorithms operating on ordered lists.

2.1 Processor Architecture

In the following sections, we briefly introduce the concepts of a CPU cache, branch prediction, 'single instruction, multiple data' (SIMD) processing and some related implementation techniques relevant to the search algorithms treated in Section 2.2.

2.1.1 Memory Hierarchy and Caches

Over the past decades, processor performance grew exponentially, while memory technology gradually fell behind. A modern CPU like the Intel Core i7-7700 has a theoretical maximum memory bandwidth of 37.5 GB/s when both memory channels are used with DDR4-2400 memory.[1] The same processor can fetch 16 bytes of instructions each clock cycle [Int16a], yielding a theoretical bandwidth requirement of 57.6 GB/s at a the rated clock speed of 3.6 GHz for instructions alone on a single core. Clearly the main memory is not fast enough to support even a single core. An even more important limit of DRAM main memory is its latency. A modern processor with a clock rate of 3 GHz can access memory approximately every 300 ps, whereas the CAS latency[2] of a typical DDR4-2400 module is 14.2 ns[3]. This is worse than older DDR2-800 memory with a CAS latency of 12.5 ns[4], indicating the physical limits to main memory latency. In short, the DRAM based main memory can not directly be connected to the processor if it should run with speeds in the GHz range.

[1] Datasheet, Vol. 1: 7th Gen Intel Processor Family for S Platforms

[2] The time until the transfer of the requested data starts after issuing a READ command

[3] Assuming timings of 17-17-17 (CL-tRCD-tRP)

[4] Assuming timings of 5-5-5 (CL-tRCD-tRP)

The solution to this problem lies in faster but much smaller memories embedded into the processors. These memories are called caches and their effectiveness depends on two locality properties of typical memory accesses [SSH11]:

Temporal Locality Recently used data will probably be referenced again in the near future.

Spatial Locality If a particular storage location is referenced, adjacent storage locations are likely to be referenced soon.

From a software optimization standpoint, this means the memory access pattern of a program should be designed to exhibit these properties as much as possible.

Cache Organization

Recent CPUs typically have three general data caches, called the L1, L2 and L3 cache. The last cache level is also refereed to as LLC (last level cache). On x86 processors the L1 cache is most often divided in an instruction and a data cache. We will mostly be concerned with the data cache (L1D).

Each cache level can either be strictly inclusive, exclusive or neither of those. A strictly inclusive cache contains everything the caches further up the hierarchy contain, e.g. if the L3 cache is inclusive, everything in the L1 and L2 cache is also in the L3 cache. Inclusiveness has the advantage of simplifying cache coherence. Its main disadvantage is the eviction of cache lines from multiple caches when a new line is loaded into the higher cache levels [JBB+10]. A strictly exclusive cache contains nothing that is already present in a higher cache, thus increasing the effective cache capacity [ZDJ04]. The last possibility is a cache guaranteeing neither inclusiveness nor exclusiveness [ZAF07]. Note that the inclusive/exclusive property can be different for each cache level. Recent Intel processors since the Sandy Bridge microarchitecture have an inclusive L3 cache and non-inclusive L1 and L2 caches [Int16a].

Caches deal with data in units of cache lines. Cache lines are always loaded as complete blocks aligned to their size. A typical cache line size is 64 bytes. The question where in the cache a cache line should be placed leads to three different possible mapping schemes from main memory addresses to positions in the cache. The first option is to allow any memory block to be stored at any location in the cache. This is called a fully associative cache. The second option is to define a function mapping each memory block to exactly one possible location in the cache. This is called a direct mapped cache. Since modern computers are binary, functions of the form $block\ address \bmod 2^n$, where n is an integer and 2^n is the size of the cache in lines, are preferred. However, other mapping functions are also possible. The general data caches in modern CPUs use a scheme in between the first two called a set associative cache: Each cache line can be placed in one of a small set of possible locations. [HP11]

To access a set associative cache the address of a memory location is broken up in three parts [HP11]:

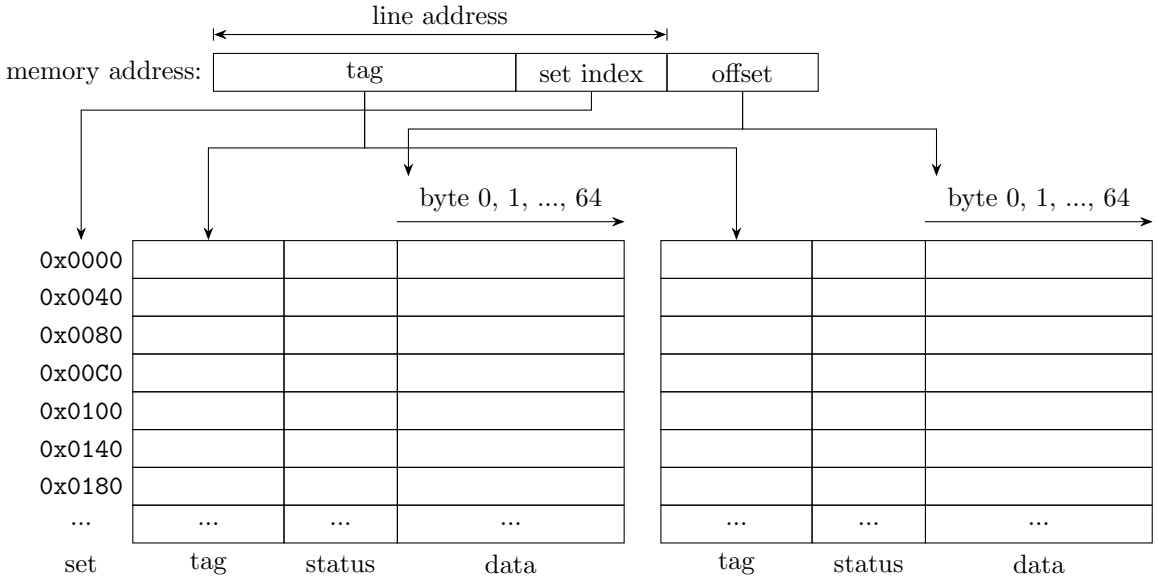

Figure 2.1: 2-way Set Associative Cache

1. The **offset** of the specific memory location in the cache line.
2. The **set index** identifying the set that the cache line can reside in.
3. The **tag** used to check whether a cache line is present in its set.

In Figure 2.1 we show how these fields are used to access a 2-way set associative cache. An entry in a set associative cache contains the tag and the actual data. Additionally the status of the cache location is stored, indicating among other things whether the data is valid.

To simplify programming and isolate different applications running simultaneously, programs access memory using virtual addresses. Concerning caches, this raises the question whether to use physical or virtual addressing. Using virtual addresses has the advantage of lower latency, since the cache lookup can begin before the address translation is complete. If virtual addresses are used for the index, care must be taken to keep the cache consistent, usually by using physical addresses for the tags. Note that some of the least significant bits of physical and virtual addresses are equal. For example, if the page size is 4 KiB, the least significant 12 bits are shared between physical and virtual addresses. Thus, if a cache with a line size of 64 bytes has at most 64 sets, no address translation is necessary to find the set index. This is often the case for L1 caches. Lower cache levels do not profit as much from virtual indexing and might be physically indexed. [CD97]
For programs this means both the position of data in virtual and in physical memory can influence how an algorithm utilizes the caches.

The L1D cache is directly connected to the processor core, this means it has to support not only line sized accesses, but also word sized and smaller ones. Since the processor

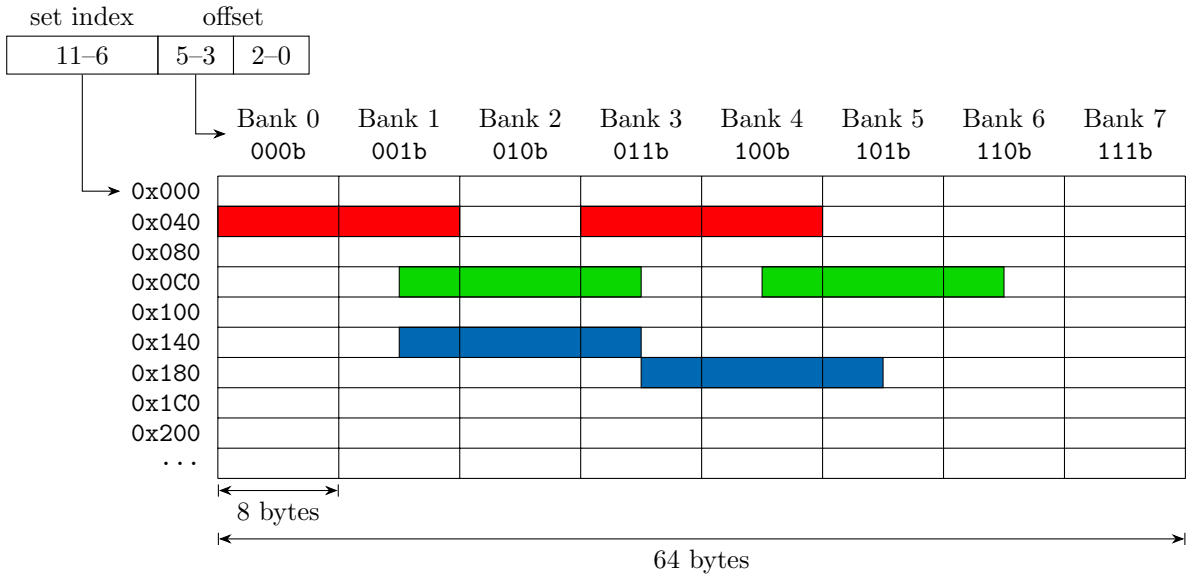

Figure 2.2: A 4 KiB cache with eight interleaved memory banks and a cache line size of 64 bytes. Each bank has a width of 8 bytes, so bits 5–3 of the memory address select the bank and bits 2–0 are the offset within a bank.

can execute multiple instructions simultaneously, the L1D cache should support multiple non-consecutive accesses in parallel. For these reasons the L1D cache is often organized in banks and/or has multiple ports. Consider the Intel Sandy Bridge microarchitecture as an example: It has a 32 KiB 8-way set associative L1D cache per core organized in eight banks. It supports two 16-byte loads, together accessing up to six banks per clock cycle. If two loads access the same bank and are not targeting the same cache line, a *bank conflict* occurs, delaying the second load. Figure 2.2 shows a possible organization for a single way in the L1D cache of a Sandy Bridge processor. The memory addresses are interleaved across eight banks. Two 16-byte loads from addresses 0x040 and 0x058 (red) can occur in the same clock cycle. Also two 16-byte loads from addresses 0x0CC and 0x0E4 can be served simultaneously (green), together accessing 6 banks. Parallel 16-byte loads from addresses 0x14C and 0x15C are not possible (blue), since they both access bank 3 on different cache lines.

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. This optimization is called prefetching. Prefetching can be triggered by the hardware itself, or by using special instructions in software. The Intel Core microarchitecture employs two automatic prefetching methods for the L1D and L2 cache: First, ascending accesses to recently loaded data automatically fetch the next cache line. Second, loads with a regular stride are assumed to continue with this stride and the corresponding cache lines are prefetched. Such prefetching strategies are common in modern processors.

A different way to use prefetching is to allow the software to explicitly request moving data to the caches. The SSE instruction set extension contains the four instructions `prefetchnta`, `prefetcht0`, `prefetcht1` and `prefetcht2` to facilitate this. All of these instructions load at least 32 bytes from memory into a cache. `prefetcht0` loads into all cache levels, `prefetcht1` loads into level 2 and 3, and `prefetcht2` loads into level 3. `prefetchnta` is used to prefetch data while minimizing cache evictions, for example by only loading into the L1D cache. Note that these instructions are only hints and may be ignored by the processor [Int16b].

Sometimes the programmer knows that temporal locality does not apply to a a specific memory access. The x86 ISA provides load and store instructions with a 'non-temporal' hint to communicate this to the processor.

2.1.2 Branch Prediction and Predication

Processing an instruction generally takes multiple clock cycles. To achieve a throughput of one instruction per cycle processors employ pipelining, i.e. various stages of instruction execution are processed in parallel for multiple instructions. For example, the next instruction can already be fetched and decoded while the preceding instruction is still being executed by an ALU. To keep all pipeline stages utilized, new instructions are fetched each clock cycle. Now consider what happens if a conditional branch is in the instruction stream. When the processor has decoded the branch instruction, instructions immediately following it might already be in the pipeline, regardless of whether the branch will be taken or not. Even worse, the processor does not yet know if the branch will be taken and whether it should start fetching new instructions from the branches destination. A simple processor might now stop putting new instructions into the pipeline and abort all instructions already in the pipeline after the branching one, creating what is know as a pipeline bubble. More advanced processors apply a heuristic to predict whether the branch will be taken and speculatively execute the instructions that are most likely needed next. As soon as the processor knows the actual outcome of the branch instruction it either keeps or discards the effects of the speculatively executed instructions. This means software should avoid hard to predict conditional jumps. [Tan05]

A powerful technique to avoid conditional jumps altogether is branch predication. Here the processor provides instructions that are always executed, but only take effect if their corresponding predicate is `true`. x86 processors provide the `cmovcc` instructions for this, where `cc` is the condition. The `cmovcc` instructions perform a move between two registers or from memory to a register. Another set of instructions useful to avoid branches have the mnemonic `setcc`, where again `cc` defines a condition. They set a byte in a register or in memory to zero or one, depending on the bits in the processor status register. The available condition codes specify a combination of the processor flags zero, carry, overflow, parity and sign.

An example of branch removal applied to database scans can be found in Broneske et al. [BBS15]. A simple scan loop using a branch is implemented in Listing 2.1,

an alternative using no branch is shown in Listing 2.3. Listing 2.2 shows a possible implementation of the branching loop in x86 assembler. The if-statement was translated to a conditional jump. In contrast, the assembler implementation of the branchless search loop (Listing 2.4) does not contain a conditional jump, but a `setl` (set byte if less) instruction retrieving information from the status register.

Listing 2.1: Scan loop in C

```
1 for (int i = 0; i < N; ++i) {
2     if (array[i] < value)
3         result[pos++] = i;
4 }
```

Listing 2.2: Scan loop in x86-64 assembler (MASM syntax)

```
1  ; rcx = &array, rdx = &result and r8d = value
2  ...
3  continue:
4    cmp  dword ptr [rcx], r8d
5    jge  short branch
6    mov  dword ptr [rdx], eax
7    add  rdx, 4
8  branch:
9    inc eax
10   add rcx, 4
11   cmp eax, N
12   jl short continue
13 ...
```

Listing 2.3: Branchless scan loop in C

```
1 for (int i = 0; i < N; ++i) {
2     result[pos] = i;
3     pos += (array[i] < value);
4
5 }
```

Listing 2.4: Branchless scan loop in x86-64 assembler (MASM syntax)

```
1  ; rcx = &array, rdi = &result and r8d = value
2  ...
3  continue:
4    mov  dword ptr [rdi + edx*4], eax
5    cmp  dword ptr [rcx], r8d
6    setl r9b
7    add  edx, r9d
8    inc eax
9    add rcx, 4
10   cmp eax, N
11   jl short continue
12 ...
```

2.1.3 SIMD Instruction Sets

Many programs exhibit data parallelism. Take for example a program adding two vectors. Traditionally it would loop over all element pairs, add them and store the result. To increase execution speed of such programs one can let multiple loop passes overlap, e.g. already load the next values while the addition of the current values is not yet complete. This is what modern superscalar processors do and can be viewed as an extension to pipelining. Another way to approach the problem is with a processor that has multiple execution units working on different data, but all executing the same instructions every clock cycle. Such an execution model is called 'single instruction, multiple data' (SIMD). On general purpose processors SIMD is typically achieved by using special instructions operating on separate registers. The SIMD instruction sets of x86 processors are called MMX, SSE and AVX [Int16b]. For this work we made use of SSE extensions up to SSE4.1, AVX and AVX2. A 64-bit processor supporting SSE has 16 architectural registers with a width of 128-bit. If it additionally supports AVX these registers are extended to 256-bit. When using SSE or AVX instructions, it is important to be aware of the data alignment in memory, because there are different instructions to load data from aligned and from unaligned locations. The aligned load instructions are faster, so data should be aligned to SIMD words whenever possible.

SIMD Comparisons

Integer comparisons in SSE and AVX are always signed. To compare unsigned integers the operands have to be biased by subtracting the smallest signed integer of the same length from them. This operation transforms the smallest possible unsigned integer zero to the smallest possible signed integer. Since singed integers are stored in two's complement the same effect can be achieved by using an addition, because the smallest signed integer is its own negation. Yet a different option is to use the XOR operation, in other words a carry-less addition. It behaves identically to a regular addition since a carry can only be generated in the most significant bit. A different view is to see that all these operations toggle the most significant bit, which is set for negative numbers in two's complement. [Gie16]

The results of an SSE or AVX comparison is a full vector containing ones in all bits for the lanes where the comparison evaluated to `true` and zero in all bits for the lanes where the result was `false`. For example, when comparing the 128-bit vectors (1, 2, 3, 4) and (2, 2, 2, 2) using the greater-than predicate, the result is (0x00000000, 0x00000000, 0xFFFFFFFF, 0xFFFFFFFF). There are several options to branch depending on such a mask. The first is the `ptest` instruction (since SSE4.1). It computes the bitwise AND of its vector operands and sets the zero flag according to the result. Additionally the carry flag is set, if all bits in the bitwise AND of the first operand with the bitwise NOT of the second operand are zero. The second option is the `pmovmskb` instruction. It takes the most significant bit of each byte from a vector register and packs them into a general purpose register. In our example `pmovmskb` would create the 16-bit result 0x00FF. We can then evaluate this results using the usual `test` instruction. Additionally there

are the slightly different `movmskps` and `movmskpd` instructions to operate on single and double precision floating point numbers, respectively. These instructions extract the most significant bit of each floating-point element. It is advisable to use the correct instructions for integer and floating-point data, since mixing them can lead to additional latency [Int16a].

When comparing sorted values, we are often interested in the number of lanes with positive or negative results. The `popcnt` instruction counts the number of bits set to one in a general purpose register. Combined with `pmovmskb` this gives the desired result [ZHF14]. Since `pmovmskb` is only available on more recent processors, it is worth considering `bsf` and `bsr` as alternatives. `bsf` returns the index of the least significant set bit and `bsr` returns the index of the most significant set bit. Because the result of comparing two sorted vectors is a possibly empty sequence of zeros followed by a possibly empty sequence of ones, or first ones and then zeros depending on the predicate, these instructions can be used instead of a population count [SGL09]. Additionally knowing the lane index of the first positive result will be useful. This can be accomplished with `bsf` or with the much newer `tzcnt` instruction. `tzcnt` counts the number of trailing zeros. `bsf` and `tzcnt` compute the same result in all but one case: If the input is zero, `bsf` is undefined whereas `tzcnt` returns the width in bits of its operand.

2.2 Search Algorithms

In this section we will introduce several search algorithms operating on sorted lists. We assume the lists contain either integer or floating-point data supported directly on a 64-bit x86 processor (see table Table 2.1 on page 19 for all possible types). As such, the elements of the lists will often be used as keys to identify bigger datasets. To search efficiently, the keys are stored in consecutive memory locations and are tightly packed. The sorted order is given by the less-than relation. Floating-point numbers must not have the value Not-a-Number (NaN), because NaN is not ordered with respect to other floating-points values. Additionally we assume proper alignment for SIMD loads.

If the list does not contain the key being search for, the search terminates unsuccessfully. For an exact match search it is enough to simply report the absence of the search key in this case. In other situations it is useful to know either the index of the first element not smaller then the search key or the index of the first key greater than the search key. These indices are commonly called the lower and upper bound. The lower and upper bound delimit the range of keys comparing equal to the search key. If keys are unique and the search key is not in the list, the lower and upper bound are identical. Lower and upper bound search algorithms are available in the C++ standard library[5] as `std::lower_bound` and `std::upper_bound`. In this work we will focus on search algorithms for a simple exact match and search algorithms for the lower bound.

[5] ISO/IEC 14882:2014

2.2.1 Sequential Search

A sequential search operates by visiting each element in turn. Algorithm 1 shows a sequential search for an exact match. It loops over all elements in the list until the first element not smaller than the search key is found. If this element is equal to the search key, its index is returned in line 5. Otherwise all remaining elements must be greater than the search key and the search terminates in line 7. The only remaining case is a search key greater than all elements in the list. In this case the loop continues until the list is exhausted and the search ends unsuccessfully in line 11.

Algorithm 1 Exact Match Sequential Search

```
1:  function sequentialSearch(list, size, searchKey)
2:      for i = 0, 1, ..., size − 1 do
3:          if list[i] ≥ searchKey then
4:              if list[i] == searchKey then
5:                  return i
6:              else
7:                  return size                    ▷ search key is not in list
8:              end if
9:          end if
10:     end for
11:     return size                                ▷ search key is not in list
12: end function
```

The lower bound search (Algorithm 2) is even simpler. It is a direct application of the definition of the lower bound. The index of the first element greater than or equal to the search key is returned in line 4. If no such element exists, the lower bound is one past the last element of the list (line 7).

Algorithm 2 Lower Bound Sequential Search

```
1:  function lowerBoundSequentialSearch(list, size, searchKey)
2:      for i = 0, 1, ..., size − 1 do
3:          if list[i] ≥ searchKey then
4:              return i
5:          end if
6:      end for
7:      return size
8:  end function
```

The worst case run time of Algorithm 1 is $\mathcal{O}(size)$, because all elements of the list have to be visited if the search key is greater than all elements of the list. The same argument holds for Algorithm 2, so its worst case run time is also $\mathcal{O}(size)$.

lane	0	1	2	3
search key	2	2	2	2
block to compare	1	2	3	4
result of greater-than	0xFFFFFFFF	0x00000000	0x00000000	0x00000000

Figure 2.3: Result of comparing the search key 2 with the vector (1, 2, 3, 4) as 32-bit integers in a 128-bit SSE register. The lane index of the lower bound is 1.

SIMD Sequential Search

It is possible to improve the performance of the sequential search by utilizing SIMD instructions. Algorithm 3 works by iterating over the list in blocks of complete SIMD words. These words are loaded together and compared to the search key in parallel (line 3). The function createMask refers to the usage of an appropriate instruction to create a mask in a general purpose register (see Section 2.1.3). This mask will contain a possibly zero length string of one bits followed by a possibly zero length string of zero bits. If the mask contains any zero bits, the lower bound is in the current block and its offset from the beginning of the block is obtained by counting the number of leading one bits divided by the bit-width of a lane (see Figure 2.3). Line 9 handles the case where the search key is greater than all elements in the list.

Algorithm 3 Lower Bound Sequential Search using SIMD

```
Require: size is a multiple of the SIMD register width in lanes
 1: function lowerBoundSequentialSearchSIMD(list, size, searchKey)
 2:     for all SIMD word sized element blocks in list do
 3:         vector compResult = broadcast(searchKey) > load(block)
 4:         if vecCompResults contains at least one zero then
 5:             int mask = createMask(compResult)
 6:             return start index of current block + countPositiveResults(mask)
 7:         end if
 8:     end for
 9:     return size
10: end function
```

Algorithm 3 has the same asymptotic time complexity as the scalar variants, but will need less iterations by a constant factor. This factor is one over the number of SIMD lanes used. An exact match search algorithm with the same time complexity can be constructed by checking if the list element at the index found by Algorithm 3 is the search key. If so, we have an exact match, otherwise the search key is not in the list.

2.2.2 Dichotomic Search

To improve over the linear time complexity of the previous algorithms we will use the Divide-and-Conquer principle. More specifically the algorithms in this section recursively divide the list in two disjunct partitions and use an element at the border

separating the partitions to decide in which of them the search should continue. Search algorithms that decide between two distinct alternatives are called dichotomic. The main question is now how to decide where to split the search range. In the following, we review the binary search as a prominent example of dichotomic searching and the lesser know Fibonaccian search.

2.2.2.1 Binary Search

A binary search operates by recursively checking the middle element of the search range and then continuing either with the left or right sub-range. We start with the exact match search in Algorithm 4. The current sub-range is represented by the variables *left* and *right*. They form the closed interval $[left, right]$. The middle of the interval is computed in line 4. Note the formula used: While $(left + right)/2$ would be equivalent when computed using real numbers, the formula used in the algorithm avoids an integer overflow that could happen in the sub-expression $left + right$. We will call the element at index *mid* the separator element. In each iteration the separator element is checked against the searched key. If they are equal the search terminates successfully (line 6). Otherwise the search continues with the partition that still can contain the key. The separator is not part of any partition, because the check in line 5 has already ruled it out. If the search interval becomes empty, i.e. $left > right$, the search terminates unsuccessfully in line 13.

Algorithm 4 Binary Search

```
1:  function BINARYSEARCH(list, size, searchKey)
2:      int left = 0, right = size − 1                    ▷ both inclusive
3:      while left ≤ right do
4:          int mid = left + (right − left)/2
5:          if searchKey == list[mid] then
6:              return mid
7:          else if searchKey < list[mid] then
8:              right = mid − 1                           ▷ go to left partition
9:          else
10:             left = mid + 1                            ▷ go to right partition
11:         end if
12:     end while
13:     return size                                       ▷ search key is not in list
14: end function
```

Algorithm 5 is a lower bound binary search. Here the search range represented by *left* and *right* is the half-open interval $[left, right)$. Accordingly the loop is exited when $left \geq right$. Note that for the loop invariant $left \leq mid < right$ to hold, the division in line 4 has to be truncating. Algorithm 5 excludes the separator element from both partitions, just like the exact match search. For the algorithm to work it is import to favor the left partition if the separator is equal to the search key. This is because the

search loop is terminated by setting *left* to $mid + 1$ in line 8 when *mid* is $right - 1$. Then we have $left = right$, i.e. the search interval is empty. Left now points to a separator element from an earlier iteration that has been disregarded by choosing the left partition. Consider the example in Figure 2.4. In iteration 4 *left* is 4 and *right* is 5, leaving just the 8 in the search range. The final iteration will then set *left* to 5, because the search key 9 is greater than the last separator 8 (line 8). Now the search terminates and both *left* and *right* hold the correct answer. There are two cases that the above argument did not cover:

1. The search key is smaller than all elements in the list: In this case line 6 is executed in every iteration, eventually reducing *right* to zero. Then *left* and *right* hold the correct lower bound.

2. The search key is greater than all elements in the list: In this case line 8 is executed in every iteration, until $left = right = size$. Again *left* and *right* hold the correct lower bound.

Algorithm 5 Lower Bound Binary Search

```
1:  function lowerBoundBinarySearch(list, size, searchKey)
2:      uint left = 0, right = size          ▷ left is inclusive, right is exclusive
3:      while left < right do
4:          uint mid = left + (right − left)/2          ▷ left ≤ mid < right
5:          if searchKey ≤ list[mid] then
6:              right = mid                              ▷ go to left partition
7:          else
8:              left = mid + 1                           ▷ go to right partition
9:          end if
10:     end while
11:     return left                                      ▷ left = right
12: end function
```

Algorithms 4 and 5 both need $\lfloor \log_2(size) \rfloor + 1$ iterations in the worst case, yielding an asymptotic run time of $\mathcal{O}(\log(size))$. This can be seen by constructing a binary tree corresponding to the search and analyzing its height [Knu98]. A simple way to think about the above formula is to consider the number of possible outcomes of the search. If there are n elements in the list, the result is an index in $[0, n]$, so there are $n+1$ outcomes the search has to distinguish. With m binary decisions one can at best distinguish 2^m cases. So we need at least $\lceil \log_2(n+1) \rceil = \lfloor \log_2(n) \rfloor + 1$ binary decisions. Note that the asymptotic run time of binary search is optimal for all comparison based search algorithms [Knu98]. It is possible to create an alternative exact match search algorithm that works like a lower or upper bound search and performs the comparison for equality after the search loop. Such an algorithm was first published by H. Bottenbruch [Bot62]. This has the advantage of eliminating a branch from the search loop, but has the

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Iteration 1	0	2	4	6	8	10	12	**14**	16	18	20	22	24	26	28	30
Iteration 2	0	2	4	**6**	8	10	12	14	16	18	20	22	24	26	28	30
Iteration 3	0	2	4	6	8	**10**	12	14	16	18	20	22	24	26	28	30
Iteration 4	0	2	4	6	**8**	10	12	14	16	18	20	22	24	26	28	30
Result	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30

Figure 2.4: Example of lowerBoundBinarySearch searching for the key 9. The interval $[left, right)$ is shaded and the separator elements are printed in boldface.

drawback that the loop will always run until the search interval is narrowed down to a single element.

Other variants of the binary search, called uniform binary search, do not store the upper boundary *right* of the search interval explicitly. Instead, they only use the start of the search interval, *left*, and its current length. This can lead to simpler index calculations, but care must be taken for the algorithm to work on list with arbitrary lengths [Knu98][Les83]. We have implemented an algorithm of this kind in Section 4.1.2 on page 39.

SIMD Binary Search

A straightforward usage of SIMD in a binary search is to examine a complete SIMD word of separator elements in each iteration [ZR02][SGL09]. So for example, instead of loading a separator element from index 8, we might load four elements from indices 8, 9, 10 and 11 into a SIMD register and parallelly compare them to the search key.

Algorithm 6 is a binary search using SIMD loads and comparisons. The list indices *left*, *right* and *mid* refer to SIMD word sized blocks instead of individual elements, leaving the loop condition and the calculation of the middle the same as in the scalar search. In line 6 a complete block of separator elements is loaded, it is then compared for equality in line 7. If we have a match, indicated by set bits in the result mask, the search terminates successfully. Otherwise the search branches to the left or right partition. If the loop is exited, because the search interval became empty, the search terminates unsuccessfully in line 21. There is one additional branch compared the the scalar search: The search can terminate early if a word of separators contains keys both smaller and larger than the element searched for in line 18, because we have already checked for equality.

Algorithm 6 Binary Search using SIMD

Require: size is a multiple of the SIMD register width in lanes

```
1:  function binarySearchSIMD(list, size, searchKey)
2:      int simdWords = number of SIMD words in list
3:      int left = 0, right = simdWords − 1                          ▷ both inclusive
4:      while left ≤ right do
5:          int mid = left + (right − left)/2
6:          vector separators = load(block of elements starting at index mid)
7:          vector eqCompResult = broadcast(searchKey) == separators
8:          if any bit in eqCompResult is set then
9:              int i = getFirstPositiveResult(createMask(eqCompResult))
10:             return start index of current block + i
11:         end if
12:         int mask = createMask(broadcast(searchKey) > separators)
13:         if all bits in mask are clear then
14:             right = mid − 1          ▷ all separators are larger than the search key
15:         else if all bits in mask are set then
16:             left = mid + 1           ▷ all separators are smaller than the search key
17:         else
18:             return size                                  ▷ search key is not in list
19:         end if
20:     end while
21:     return size                                          ▷ search key is not in list
22: end function
```

The SIMD binary search algorithm we have discussed loops over blocks as if they were single elements and retains the loop structure of its scalar counterpart. Therefore we can analyze its run time by substituting the number of blocks for *size* in the equations for the scalar algorithms [SGL09]. A list of *size* elements is broken into $size/n$ blocks, where n is the number of SIMD lanes used. So the worst case number of iterations is $\lfloor \log_2(size/n) \rfloor + 1$. Since $\log_2(size/n) = \log_2(size) - \log_2(n)$, we expect a speedup of about $log_2(n)$ iterations. Of course the asymptotic time complexity is still $\mathcal{O}(\log(size))$.

2.2.2.2 Fibonaccian Search

Instead of splitting the search range in equally sized halves, other ratios are possible. This section deals with an algorithm approximately partitioning the range in the golden ratio $\varphi \approx 1.618$. Since indices are discrete, the Fibonacci numbers, defined as $F_n = F_{n-1} + F_{n-2}$ with $F_0 = 0$ and $F_1 = 1$, are used as an approximation.

The Fibonaccian search technique was inspired by a numerical algorithm for finding the extremum of a unimodal function, called Fibonacci search [Knu98]. This algorithm was discovered by J. Kiefer [Kie53].

In Algorithm 7 we present an iterative version of the Fibonaccian search. The variable *left* is used to keep track of the start of the current search range, while the size

of the range is given by the Fibonacci numbers. The Fibonacci numbers can either come from a static list or be computed during the search. To do this efficiently, it is advisable to maintain two variables containing consecutive Fibonacci numbers, allowing easy computation of the preceding numbers in the series. The check for $i \geq size$ in line 6 is necessary when operating on lists with a size not equal to $F_n - 1$, where F_n is a Fibonacci number. It lets the search continue in the left partition if the separator element is out of bounds. This is effectively the same as extending the list with sentinel values greater than all keys in the list.

Algorithm 7 Fibonaccian Search

```
1:  function FIBONACCIANSEARCH(list, size, searchKey)
2:      Set n, so that F_n is the greatest Fibonacci number still smaller then size.
3:      uint i = F_n
4:      uint left = 0
5:      while n ≥ 0 do
6:          if i ≥ size or searchKey < list[i] then
7:              i = left + F_{n-1}                          ▷ go to left partition
8:              n = n − 1
9:          else if searchKey > list[i] then
10:             left = i
11:             i = i + F_{n-2}                             ▷ go to right partition
12:             n = n − 2
13:         else
14:             return i
15:         end if
16:     end while
17:     return size                                         ▷ search key is not in list
18: end function
```

To see how the search operates, consider Figure 2.5. In the first iteration, the element at index 13 (F_7) is examined, since 13 is the greatest Fibonacci number still in the index range of the list. Since we are looking for the key 12, we continue to the left. Now the key at index 8 (F_6) is checked. There are 8 keys in the range from index 0 to index 7 and 5 keys in the range from index 8 to 12, forming a ratio of $8/5 = 1.6 \approx \varphi$. Note how the separators selected by the Fibonaccian search are always closer to the previous separator than in a binary search.

The Fibonaccian search has an asymptotic time complexity of $\mathcal{O}(\log(\text{size}))$, just like the binary search [Knu98].

2.2.3 k-ary Search

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

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Iteration 1	0	2	4	6	8	10	12	14	16	18	20	22	24	**26**	28	30
Iteration 2	0	2	4	6	8	10	12	14	**16**	18	20	22	24	26	28	30
Iteration 3	0	2	4	6	8	**10**	12	14	16	18	20	22	24	26	28	30
Iteration 4	0	2	4	6	8	10	12	**14**	16	18	20	22	24	26	28	30
Iteration 5	0	2	4	6	8	10	**12**	14	16	18	20	22	24	26	28	30

Figure 2.5: Example of a Fibonaccian search searching for the key 12. The interval still considered by the search is shaded and the separator elements are printed in boldface.

the search range evenly. Algorithm 8 does this in line 4. The for-loop then checks each separator element first for equality with the search key. The search is terminated successfully, if the search key has been found. Otherwise the search continues with the segment to the left of the smallest separator larger then the search key (lines 9 to 12). If the search key is larger than all separator elements, the rightmost segment is selected in line 14.

A lower bound variant can be constructed, by removing the equality test in lines 6 to 8 and performing a less-equal comparison in line 9 to decide whether the lower bound is in the current segment. The behavior and termination of the while-loop in line 3 is then identical to the lower bound binary search and the algorithm can return *left* as the lower bound after the search loop.

Algorithm 8 and its lower bound counterpart correspond to multi-way search trees in the same way the binary search corresponds to binary search trees, so the only difference in run-time comes from a different base for the logarithm. Since a k-ary search reduces the search space by a factor of k in each iteration, the worst case number of iterations is $\lceil \log_k(size+1) \rceil = \lfloor \log_k(size) \rfloor + 1$. This yields the same asymptotic time complexity as for a binary search, namely $\mathcal{O}(\log(size))$. The speedup gained from k-ary searching over binary searching is approximately $\frac{\log_2 n}{\log_k n} = \log_2 k$ [SGL09].

SIMD k-ary Search

Schlegel et al. propose using SIMD to parallelize the index computations and key comparisons of a k-ary search [SGL09]. This should lower the time needed for each iteration of the search, while the number of iterations stays the same as in a scalar implementation. An exact match SIMD k-ary search has the following steps:

1. Calculate the indices of $k-1$ separator elements.

2. Load the separators into a SIMD register.

Algorithm 8 k-ary Search

```
1: function karySearch(list, size, searchKey)
2:     uint left = 0, right = size                ▷ left is inclusive, right is exclusive
3:     while left < right do
4:         Divide [left, right) in k segements, separated by k − 1 separator elements.
5:         for each separator element s do
6:             if searchKey == s then
7:                 return index of s
8:             end if
9:             if searchKey < s then
10:                Set left, right to the segment left of s.
11:                Continue with the next iteration of the while loop.
12:            end if
13:        end for
14:        Set left, right to the last segment.
15:    end while
16:    return size                                ▷ search key is not in list
17: end function
```

Data Type	k in 128-bit register	k in 256-bit register
signed/unsigned 8-bit integer	17	33
signed/unsigned 16-bit integer	9	17
signed/unsigned 32-bit integer	5	9
signed/unsigned 64-bit integer	3	5
single-precision floating-point number	5	9
double-precision floating-point number	3	5

Table 2.1: Data type and corresponding k for the SIMD k-ary search

3. Compare the separators to the search key. If the search key is present, terminate successfully.

4. Continue the search in the partition the search key belongs to. If the next partition is empty, terminate unsuccessfully.

The separator element's offsets from the start of a search range with length p are given by $i\lceil(p+1)/k\rceil$ with $1 \leq i < k$. Schlegel et al. suggest precomputing the separators' distance to avoid the division in this formula. Selecting the next partition in step 4 can be implemented using the `popcnt` instruction as in the other algorithms. Step 2 requires a loop or a load instruction targeting multiple non-continuous memory locations. Such an instruction is available in the AVX2 instructions set, but not on earlier x86 processors. Table 2.1 shows the available key data types on a x86 64-bit processor and the resulting k when using SSE (128-bit) or AVX (256-bit) registers.

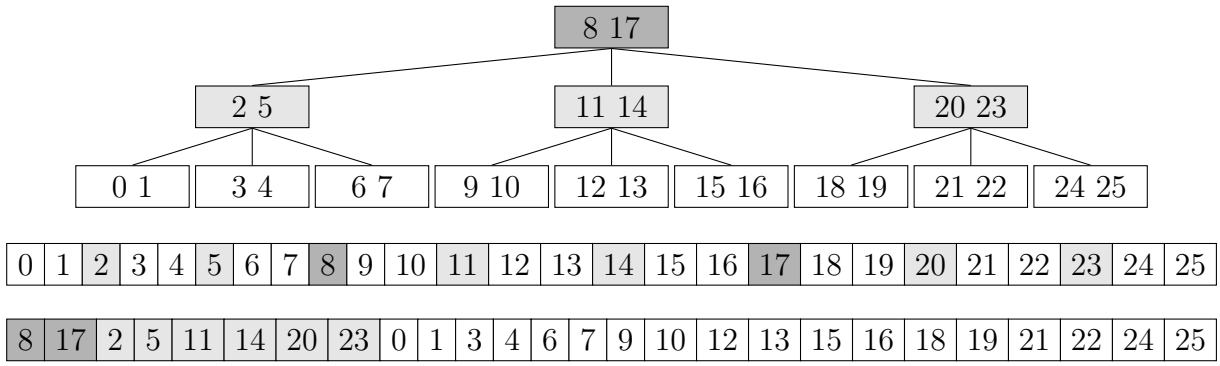

Figure 2.6: Perfect k-ary search tree for k = 3 containing 26 keys, the corresponding ordered list (upper row) and the linearized tree (lower row)

2.2.4 Linearized k-ary Search Trees

The SIMD k-ary search has the disadvantage of loading the separator elements from far scattered memory locations. If we allow a transformation reordering the sorted list before the search, this is avoidable.

Following Schlegel et al., we will first restrict our considerations to list with a size of $k^h - 1$ elements, where $h > 0$ is an integer. From the elements of such a list, a perfect k-ary search tree can be constructed. The nodes of a k-ary search tree contain up to $k - 1$ keys and accordingly have up to k child-nodes. The tree is perfect if "every node—including the root node—has precisely $k - 1$ entries, every internal node has k successors, and every leaf node has the same depth" [SGL09]. Figure 2.6 shows an example with $k = 3$ and the corresponding ordered list (upper row). By performing a level-order traversal of the search tree, a linearized form of it is constructed (lower row). In this linearized tree, the separator elements needed in the same iteration are located in adjacent memory cells. The linearized search tree corresponding to the example is shown in the lower row of Figure 2.6.

Support for lists with a length not corresponding to a perfect k-ary search tree is accomplished by constructing complete trees. A k-ary search tree of height h is complete, if "(1) removing the leafs at depth $h - 1$ yields a perfect tree of height $h - 1$ and (2) the leafs at depth $h - 1$ grow from left to right" [SGL09]. Figure 2.7 shows an example with 11 keys.

It is not necessary to explicitly construct the search tree to compute the linearized tree representation, since a formula giving the permutation from the ordered list to the linearized tree exists. For the derivation of this formula, the rightmost entry in the last level of the search tree has a special significance. It is referred to as the fringe entry and is always the last element in the linearized tree. The fringe entry in Figure 2.7 is 10. All elements smaller than or equal to the fringe entry have the same position in the linearized tree as they had if the tree was perfect. In the example these are the keys 0 to 10. They have the same indices in the perfect linearized tree of Figure 2.6 and in the complete linearized tree of Figure 2.7. All elements larger then the fringe entry are

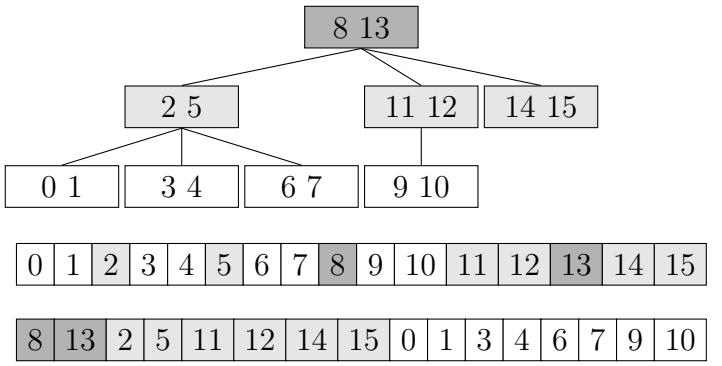

Figure 2.7: Complete k-ary search tree for k = 3 containing 16 keys, the corresponding ordered list (upper row) and the linearized tree (lower row)

at the same positions they would have if the tree would be made perfect by omitting all leafs from the last—incomplete—level. If the keys 0 to 10 would be removed from the tree in Figure 2.7, it would be a perfect k-ary search tree with 8 keys. The formula yielding the one-based index j of a key in the linearized tree given its one-base index i in the sorted list is [SGL09]:

$$j = g_n(i) = \begin{cases} f_H(i) & i \leq f_H^*(n) \\ f_{H-1}(i - o_H^*(n) - 1) & \text{otherwise} \end{cases} \tag{2.1}$$

Where n is this number of keys in the sorted list and $H = \lceil \log_k(n+1) \rceil$ is the height of the search tree. $f_H^*(n)$ is the index of the fringe entry in the sorted list and f_H yields the index of a key in the linearized tree given its index in the sorted array. $o_H^*(n)$ is the zero-based offset of the fringe element in its level of the search tree. The functions f, f^* and o^* can all assume the conceptual search tree to be perfect, because of the arguments given above. They are defined as:

$$f_h(i) = k^{d_h(i)+o_h(i)} \tag{2.2}$$

$$f_h^* = k^{h-d_j^*(j)-1} \left\lfloor \frac{k}{k-1} o_h^*(j) + 1 \right\rfloor \tag{2.3}$$

$$o_h^*(j) = j - k^{d_h^*(j)} \tag{2.4}$$

Furthermore, the functions d_h, o_h and d_h^* are defined as:

$$d_h(i) = \sum_{x=1}^{h-1} \text{sign}(i \bmod k^{h-x}) \tag{2.5}$$

$$o_h(i) = \left\lfloor \frac{k-1}{k} \cdot \frac{i}{k^{h-d_h(i)-1}} \right\rfloor \tag{2.6}$$

$$d_h^*(j) = \lfloor \log_k(j) \rfloor \tag{2.7}$$

2.2.4.1 Searching in Linearized Trees

An exact match search algorithm using zero-based indices is show in Algorithm 9. It uses the variable *left* to mark the current node in the search tree. Observe, how the load of the separators in line 5 is from one continuous block of memory. Determining the next node to search in line 14 does not require conditional branching, since the mask evaluation in line 13 already yields the index of the tree branch to take. The start index of the corresponding child node is then easily computed. $(left + 1) \cdot k - 1$ gives the start index of the next level of the tree, and $branch \cdot (k - 1)$ yields the offset of the next node in the level. The search terminates unsuccessfully when *next* would lie beyond the bounds of the linearized tree.

Algorithm 9 Exact Match Search on a Linearized k-ary Search Tree using SIMD

```
1:  function karySearchLinearizedTree(linTree, size, searchKey)
2:      uint left = 0, next = 0
3:      while next < size do
4:          left = next
5:          vector separators = load(linTree + left)
6:          vector eqCompResult = broadcast(searchKey) == separators
7:          if any bit in eqCompResult is set then
8:                                          ▷ searchKey is equal to one of the separators
9:              int i = getFirstPositiveResult(createMask(compResult))
10:             return left + 1
11:         end if
12:         vector gtCompResult = broadcast(searchKey) > separators
13:         int branch = countPositiveResults(createMask(gtCompResult))
14:         next = (left + 1) · k + branch · (k − 1) − 1          ▷ 0 ≤ branch < k
15:     end while
16:     return size                                          ▷ search key is not in list
17: end function
```

2.2.4.2 Range Scans

To process range queries, it is necessary to iterate over the keys in sorted order. Fortunately, it is possible to efficiently iterate over the keys in a k-ary search tree with only $\mathcal{O}(\log_k(size))$ extra space. To do so, we first need to localize the start of the range in the tree. During this search the index of the child node the search continues in is stored. The first child has index 0, the second 1, and so on. For example, if we search for the key 7 in the tree depicted in Figure 2.7, we would store the sequence $B = [0, 2, 1]$, since 6 is less than 8 and greater than 5. The last 1 is stored, because we would continue with the middle child of the node containing 6 and 7 if the tree would extend further down. Additionally the offset from the beginning of the linearized tree to the next separator element to look at in each tree level is stored. In the example of searching for the key 7, this would mean storing the sequence $O = [1, 4, 14]$, since the next element to compare

in the root is 13 at index 1, the next element to compare in the next level is 11 at index 4, and so on. To advance to the next element in the sequence Algorithm 10 is used [SGL09].

Algorithm 10 Advance to the next element in sorted order when iterating over a linearized tree

```
1: d = depth of the tree
2: while B[d] == k − 1 do
3:     B[d] = 0
4:     d = d − 1
5: end while
6: Now O[d] is the next index of the next key.
7: B[d] = B[d] + 1
8: O[d] = O[d] + 1
```

For the example, this means we start with:

$$B = [0, 2, 1] \quad O = [1, 4, 14] \quad d = 2$$

The while-loop is not entered, because $B[2] < 2$. Now we find the next key in sorted order at index $O[2] = 15$. After $B[2]$ and $O[2]$ have been incremented, we have:

$$B = [0, 2, 2] \quad O = [1, 4, 15] \quad d = 2$$

To locate the next key in sorted order, the while-loop traverses the tree up, resulting in:

$$B = [0, 0, 0] \quad O = [1, 4, 15] \quad d = 0$$

This means, the next key is at index O[0] = 1. Then $B[0]$ and $O[0]$ are incremented, yielding:

$$B = [1, 0, 0] \quad O = [2, 4, 15] \quad d = 0$$

Algorithm 10 is applied until the last element of the query range has been reached.

3. Sequential Search

In the following, we present basic implementations of the sequential search algorithms and apply software optimization techniques such as branch elimination and loop unrolling. Then, the performance of the various implementations is evaluated.

3.1 Implementation

In this section, we present implementations of the sequential search algorithms from Section 2.2.1. All search functions in this and in later chapters are C++ templates. They all have the type of the keys to search in, `T`, as a common parameter. Unless noted otherwise, `T` can be any built-in signed or unsigned 8, 16, 32, or 64-bit integral type, or one of the floating-point types float and double. The type `IndexType` is assumed to be unsigned.

3.1.1 Scalar Sequential Search

The scalar algorithms in Listing 3.1 and Listing 3.2 are literal transcriptions of Algorithms 1 and 2. The only difference between the exact match and lower bound search is an additional if-statement to check for an exact match after the lower bound has been found (lines 6 and 7 in Listing 3.2).

Listing 3.1: Lower Bound Sequential Search

```
1  template <typename T>
2  IndexType lowerBoundSequentialSearch(
3    const T *keys, IndexType size, T searchKey) {
4    for (IndexType i = 0; i < size; ++i) {
5      if (keys[i] >= searchKey) {
6        return i;
7      }
8    }
9    return size;
10 }
```

Listing 3.2: Sequential Search

```
1  template <typename T>
2  IndexType sequentialSearch(
3    const T *keys, IndexType size, T searchKey) {
4    for (IndexType i = 0; i < size; ++i) {
5      if (keys[i] >= searchKey) {
6        if (keys[i] == searchKey)
7          return i;
8        else
9          return size; // not found
10     }
11   }
12   return size; // not found
13 }
```

3.1.2 Vectorized Sequential Search

To allow for different key data types and to support both 128-bit and 256-bit registers, the SIMD implementations make use of template functions that are defined as an appropriate intrinsic for the given types. All these functions are listed in Section A.3 in the appendix. The type `Vector` is defined as `__m128i`, when compiling for SSE, and as `__m256i` when compiling for AVX (see Listing A.4).

The sequential search algorithms using SIMD share a common prolog shown in Listing 3.3. It gets the number keys fitting in one SIMD word using the template function `keys_per_simd_word` (Listing A.5) and calculates the length of the array in SIMD words (line 4). We assume the length of the array is a multiple of `KEYS_PER_WORD`. Line 7 loads the search key into a SIMD register, replicating it over all lanes. As noted in Section 2.1.3, we need a key transformation to supported unsigned types. This transformation is provided by the template function `adjustForSignedComparison` (Listing A.8). Additionally a vector containing all ones is prepared by comparing a vector register for equality to itself in line 6 (see `getAllOnesVector` in Listing A.6). This vector is used as the second operand for the `ptest` instruction to test whether a comparison result contains at least one zero. `ptest` flips the bits of its first operand, in our case the comparison's result mask, and then calculates the bitwise AND with the second operand, in our case all ones. If the result of this operation is zero, the carry flag is set to one, otherwise to zero. This means, the mask contains at lest one zero, if the carry flag is not set. In the source code, this is expressed with the template function `_mm_testc` (Listing A.10) calling the appropriate intrinsic function.

When the vector containing the lower bound has been found, its location in the SIMD word is found by creating a mask with the template function `createMask` (Listing A.9) and evaluating it with `countPositiveResults` (Listing A.9) in lines 12 and 13 of Listing 3.4. `countPositiveResults` will yield a value in the range $[1, \texttt{KEYS\_PER\_WORD}]$. If the comparison evaluated to `true` for one key, the search key is still smaller than the first key in `values`, i.e. the second key in `values` is the lower bound. The same reasoning holds for the other possible outcomes of `countPositiveResults`. To get the

Listing 3.3: Prolog of the SIMD sequential search

```
1  #define PROLOG() \
2    assert((intptr_t)keys % sizeof(Vector) == 0); \
3    constexpr IndexType KEYS_PER_WORD = keys_per_simd_word<Vector, T>(); \
4    const IndexType simdWords = size / KEYS_PER_WORD; \
5    assert(size % KEYS_PER_WORD == 0); \
6    const Vector vecOnes = getAllOnesVector<Vector>(); \
7    const Vector vecSearchKey = \
8      adjustForSignedComparison<T>(_mm_set1<Vector>(searchKey));
```

array index of the lower bound, we add `KEYS_PER_WORD` times `i`, the array index of the first key in `values`, in line 13.

Listing 3.4: Lower Bound Sequential Search SIMD

```
1  template <typename T>
2  IndexType lowerBoundSequentialSearchSIMD(
3    const T *keys, IndexType size, T searchKey) {
4    PROLOG();
5    for (IndexType i = 0; i < simdWords; ++i) {
6      Vector values = loadVector(reinterpret_cast<const Vector*>(
7        keys + KEYS_PER_WORD * i));
8      Vector compResult = _mm_cmpgt<T>(vecSearchKey,
9        adjustForSignedComparison<T>(values));
10     if (!_mm_testc<T>(compResult, vecOnes)) {
11       // compResult contains at least one zero
12       int mask = createMask<T>(compResult);
13       return KEYS_PER_WORD * i + countPositiveResults<T>(mask);
14     }
15   }
16   return size;
17 }
```

An exact match search is constructed from the lower bound variant by replacing lines 10 to 14 with the code in Listing 3.5. Again, the only difference to the exact match search is an additional if-statement at the end.

Listing 3.5: Equality test for an exact match vectorized sequential search

```
1  if (!_mm_testc<T>(compResult, vecOnes)) {
2    // compResult contains at least one zero
3    int mask = createMask<T>(compResult);
4    IndexType result = KEYS_PER_WORD * i + countPositiveResults<T>(mask);
5    if (keys[result] == searchKey)
6      return result;
7    else
8      return size; // not found
9  }
```

3.2 Optimizations

We will present the elimination of branches and loop unrolling as possible optimizations to the functions introduced in the previous section. We will only show these optimizations applied to the lower bound search, since an exact match search can be constructed from a lower bound search by adding an if-statement after the search loop. This if-statement simply has to check if the lower bound is the search key and return the appropriate result (like in Listing 4.3). The last if-statement outside the loop of an exact match search implemented this way can be replaced by a conditional move just as the branch in the search loop. Furthermore, loop unrolling does not change anything for this last if-statement, since it is outside of the loop.

3.2.1 Branch Elimination

We eliminate the if-statement in the for-loop of the scalar search (Listing 3.1) by letting the algorithm always run through the complete array, even when the lower bound has already been found. Instead of possibly exiting the loop, the function shown in Listing 3.6 records the index of the last seen array element still smaller than the search key plus one in the variable `j`. Equivalently this is the number of elements that are smaller than the search key. The conditional operator replacing the if-statement is compiled to a `cmov` instruction, therefore there is no conditional jump left in the loop body.

Listing 3.6: Branchless Lower Bound Sequential Search

```
1 template <typename T>
2 IndexType lowerBoundSequentialSearchBranchless(
3   const T *keys, IndexType size, T searchKey) {
4   IndexType j = 0;
5   for (IndexType i = 0; i < size; ++i) {
6     j = keys[i] < searchKey ? (i + 1) : j;
7   }
8   return j;
9 }
```

For the SIMD sequential search we simply drop the if-statement and accumulate the number of keys smaller than the search key in the variable `j` (line 7 in Listing 3.8).

Listing 3.7: Branchless Vectorized Lower Bound Sequential Search

```
1  template <typename T>
2  IndexType lowerBoundSequentialSearchSIMD_Branchless(
3    const T *keys, IndexType size, T searchKey) {
4    PROLOG();
5    IndexType j = 0;
6    for (IndexType i = 0; i < simdWords; ++i) {
7      SEARCH_STEP(); // defined in Listing 3.8
8    }
9    return j;
10 }
```

Listing 3.8: Loop body of the branchless vectorized search

```
1 #define SEARCH_STEP() do{ \
2   Vector values = loadVector(reinterpret_cast<const Vector*>( \
3     keys + KEYS_PER_WORD * i)); \
4   Vector compResult = _mm_cmpgt<T>(vecSearchKey, \
5     adjustForSignedComparison<T>(values)); \
6   int mask = createMask<T>(compResult); \
7   j += countPositiveResults<T>(mask); \
8 } while (0)
```

3.2.2 Loop Unrolling

In Listing 3.9 the search loop of the scalar search has been unrolled four times. The switch-statement in lines 5 to 10 handles the remainder for arrays with a length not divisible by 4. After the switch, $size - i$ is a multiple of four, so the for-loop does not access memory locations beyond the end of the array.

Listing 3.9: Unrolled Lower Bound Sequential Search

```
1  template <typename T>
2  IndexType lowerBoundSequentialSearchUnrolled4(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType i = 0;
5    switch (size % 4) {
6      case 3: if (keys[i] >= searchKey) return i; ++i;
7      case 2: if (keys[i] >= searchKey) return i; ++i;
8      case 1: if (keys[i] >= searchKey) return i; ++i;
9      case 0: break;
10   }
11   for (; i < size; i += 4) {
12     if (keys[i] >= searchKey) return i;
13     if (keys[i + 1] >= searchKey) return i + 1;
14     if (keys[i + 2] >= searchKey) return i + 2;
15     if (keys[i + 3] >= searchKey) return i + 3;
16   }
17   return i;
18 }
```

Furthermore we have unrolled the branchless search in Listing 3.10. This implementation combines the switch and the loop in a way known as “Duff’s device” [Duf88].

In the previous function, all search steps depend on the same variables `i` and `j`. It might be possible to increase performance by removing this dependency and thus making better use of instruction level parallelism. The function in Listing 3.11 utilizes a dedicated variable for each unrolled iteration. The variables `j0` to `j3` accumulate the number of elements smaller than the search key, each for a fourth of the array. Thus, the lower bound for the whole array is their sum.

We have unrolled the SIMD search analogously to the scalar search functions in Listing 3.12, Listing 3.13 and Listing 3.14.

Listing 3.10: Branchless Unrolled Lower Bound Sequential Search

```
1  template <typename T>
2  IndexType lowerBoundSequentialSearchBranchlessUnrolled4(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType j = 0, i = 0;
5    if (size == 0) return 0;
6    switch (size % 4) {
7      do {
8        case 0: j = keys[i] < searchKey ? i : j; ++i;
9        case 3: j = keys[i] < searchKey ? i : j; ++i;
10       case 2: j = keys[i] < searchKey ? i : j; ++i;
11       case 1: j = keys[i] < searchKey ? i : j; ++i;
12     }
13     while (i < size);
14   }
15   return j;
16 }
```

Listing 3.11: Branchless Unrolled Lower Bound Sequential Search with independent search steps

```
1  template <typename T>
2  IndexType lowerBoundSequentialSearchBranchlessUnrolledIndependent4(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType j0 = 0, j1 = 0, j2 = 0, j3 = 0;
5    IndexType i0 = 3, i1 = 2, i2 = 1, i3 = 0;
6    if (size == 0) return 0;
7    switch (size % 4) {
8      do {
9      case 0: j0 = keys[i0] < searchKey ? (j0 + 1) : j0; i0 += 4;
10     case 3: j1 = keys[i1] < searchKey ? (j1 + 1) : j1; i1 += 4;
11     case 2: j2 = keys[i2] < searchKey ? (j2 + 1) : j2; i2 += 4;
12     case 1: j3 = keys[i3] < searchKey ? (j3 + 1) : j3; i3 += 4;
13     }
14     while (i0 < size);
15   }
16   return j0 + j1 + j2 + j3;
17 }
```

Listing 3.12: Unrolled Vecorized Lower Bound Sequential Search

```
#define SEARCH_STEP(offset) do { \
  Vector values = loadVector(reinterpret_cast<const Vector*>( \
    keys + KEYS_PER_WORD * (i + offset))); \
  Vector compResult = _mm_cmpgt<T>(vecSearchKey, \
    adjustForSignedComparison<T>(values)); \
  if (!_mm_testc<T>(compResult, vecOnes)) { \
    int mask = createMask<T>(compResult); \
    return KEYS_PER_WORD * (i + offset) + countPositiveResults<T>(mask); \
  } \
} while(0)
template <typename T>
IndexType lowerBoundSequentialSearchSIMD_Unrolled4(
  const T *keys, IndexType size, T searchKey) {
  PROLOG();
  IndexType i = 0;
  switch (simdWords % 4) {
    case 3: SEARCH_STEP(0); ++i; case 2: SEARCH_STEP(0); ++i;
    case 1: SEARCH_STEP(0); ++i; case 0: break;
  }
  for (; i < simdWords; i += 4) {
    SEARCH_STEP(0); SEARCH_STEP(1); SEARCH_STEP(2); SEARCH_STEP(3);
  }
  return size;
}
```

Listing 3.13: Branchless Unrolled Vectorized Lower Bound Sequential Search

```
#define SEARCH_STEP() do{ \
  Vector values = loadVector(reinterpret_cast<const Vector*>( \
    keys + KEYS_PER_WORD * i)); \
  Vector compResult = _mm_cmpgt<T>(vecSearchKey, \
    adjustForSignedComparison<T>(values)); \
  int mask = createMask<T>(compResult); \
  j += countPositiveResults<T>(mask); \
} while (0)
template <typename T>
IndexType lowerBoundSequentialSearchSIMD_BranchlessUnrolled4(
  const T *keys, IndexType size, T searchKey) {
  if (size == 0) return 0;
  PROLOG();
  IndexType j = 0, i = 0;
  switch (simdWords % 4) {
    do {
      case 0: SEARCH_STEP(); ++i; case 3: SEARCH_STEP(); ++i;
      case 2: SEARCH_STEP(); ++i; case 1: SEARCH_STEP(); ++i;
    } while (i < simdWords);
  }
  return j;
}
```

Listing 3.14: Branchless Unrolled Vectorized Lower Bound Sequential Search SIMD with independent search steps

```
1  #define SEARCH_STEP(i, j) do{ \
2    Vector values = loadVector( \
3      reinterpret_cast<const Vector*>(keys + KEYS_PER_WORD * i)); \
4    Vector compResult = _mm_cmpgt<T>(vecSearchKey, \
5      adjustForSignedComparison<T>(values)); \
6    int mask = createMask<T>(compResult); \
7    j += countPositiveResults<T>(mask); \
8  } while (0)
9  template <typename T>
10 IndexType
11   lowerBoundSequentialSearchSIMD_BranchlessUnrolledIndependent4
12   (const T *keys, IndexType size, T searchKey)
13 {
14   if (size == 0) return 0;
15   PROLOG();
16   IndexType j0 = 0, j1 = 0, j2 = 0, j3 = 0;
17   IndexType i0 = 3, i1 = 2, i2 = 1, i3 = 0;
18   switch (simdWords % 4) {
19     do {
20       case 0: SEARCH_STEP(i0, j0); i0 += 4;
21       case 3: SEARCH_STEP(i1, j1); i1 += 4;
22       case 2: SEARCH_STEP(i2, j2); i2 += 4;
23       case 1: SEARCH_STEP(i3, j3); i3 += 4;
24     } while (i0 < simdWords);
25   }
26   return j0 + j1 + j2 + j3;
27 }
```

3.3 Evaluation

To evaluate the proposed optimizations, we executed the functions on randomly generated arrays and measured the average time for one search over 10,000 runs. The test data is of a signed integral type with 32-bits and is uniformly distributed over the complete range of the data type. The search keys were randomly drawn from the same distribution as the test data.

3.3.1 Evaluation Environment

We used the Microsoft C++ compiler version 19.00.24215.1 for x64 with full optimizations enabled (`/Ox`) to compile the evaluation program. The test system was equipped with an Intel Core i7 3610QM processor, with a clock rate fixed at 2.3 GHz. 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 6 MiB of shared L3 cache. The L1I, L1D and L2 caches are 8-way set associative, and the L3 cache is 12-way associative. The cache line size of all these caches is 64 bytes. Two 4 GiB DDR3-1600 memory modules in a dual

channel configuration were used as main memory. The memory timing were 9-9-9-24 (CL-tRCD-tRP-tRAS).

3.3.2 Branch Elimination

We compared the average search time for the basic, branching, search implementation with their branchless counterparts, for scalar and SIMD searches. The results in Figure 3.1 show the linear graphs expected for sequential search algorithms. The different slope of the branching and branchless graphs is the most noticeable difference. The average run-time of the branching functions increases slower than the run-time of the branchless implementations. This is, because the branchless implementations always search trough the whole array, whereas the branching versions can terminate earlier. Since the search keys in our experiment are uniformly distributed, the branching search on average only goes trough half the array. The advantage of the branchless search becomes apparent for small arrays. Here, the branchless search is superior to the branching one, since no branch misprediction can occur in the search loop. However, the branchless search becomes slower than the branching one when the array size surpassed a certain threshold. After this threshold, the cost of always iterating over the complete array outweighs the gains made by eliminating a hard to predict branch. For the scalar search the threshold is at about 20 keys, and for the vectorized search at about 50.

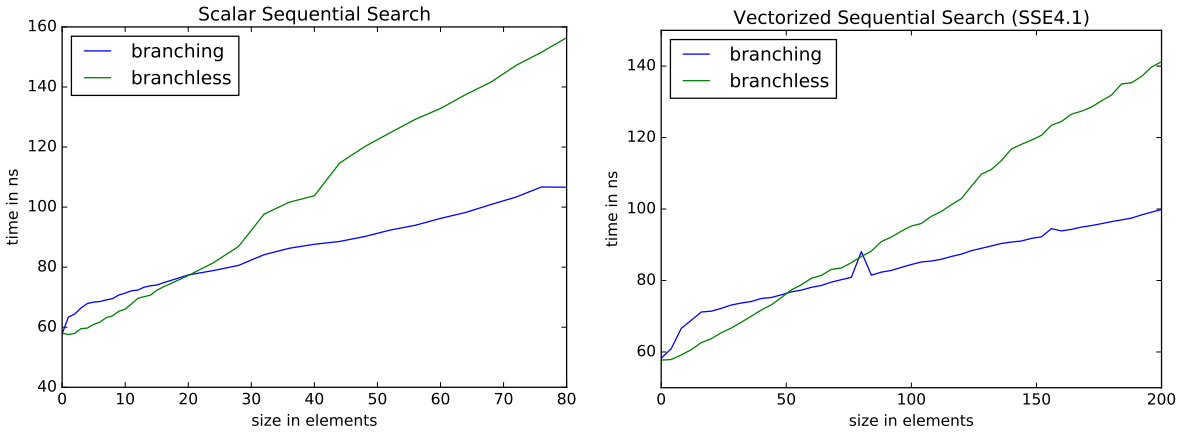

Figure 3.1: Scalar and vectorized (SSE4.1) sequential search with and without a branch in the search loop

3.3.3 Loop Unrolling

Loop unrolling tries to improve the performance of an algorithm by reducing the time spend executing loop control code, since multiple iterations are executed for each check of the loop condition. The results of unrolling the search loop 2, 4, 8, and in case of the scalar search 16 times, are shown in Figure 3.2. As can be derived from all graphs in both the left and the right plot starting at approximately the same point, all variants

retain their base cost for very small arrays. The slope of the graphs corresponding to unrolled implementations generally is shallower, indicating the effectiveness of loop unrolling. The gains are more noticeable in the scalar plot. The scalar search function is significantly improved by unrolling the loop up to 8 times. Unrolling 16 times does not yield a clear benefit for array sizes suitable to a sequential search. The situation is similar for the vectorized search, where the improvements diminish after 4 unrolled iterations.

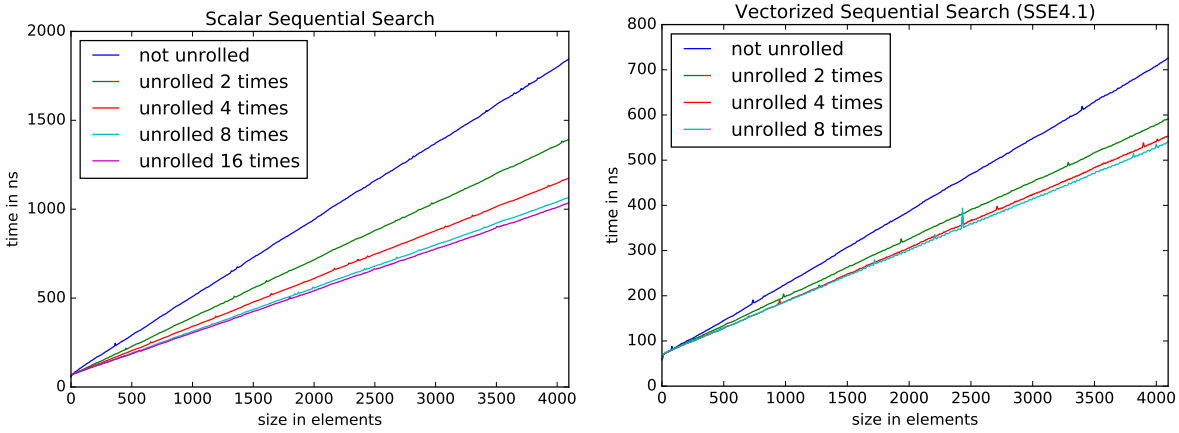

Figure 3.2: Unrolled sequential search

Additionally, we have unrolled the loops in the branchless sequential search functions (Figure 3.3). Here the results are much less clear. The SIMD search functions do not noticeably profit from loop unrolling, and the scalar functions only show improvements at element counts where a branching search is already better.

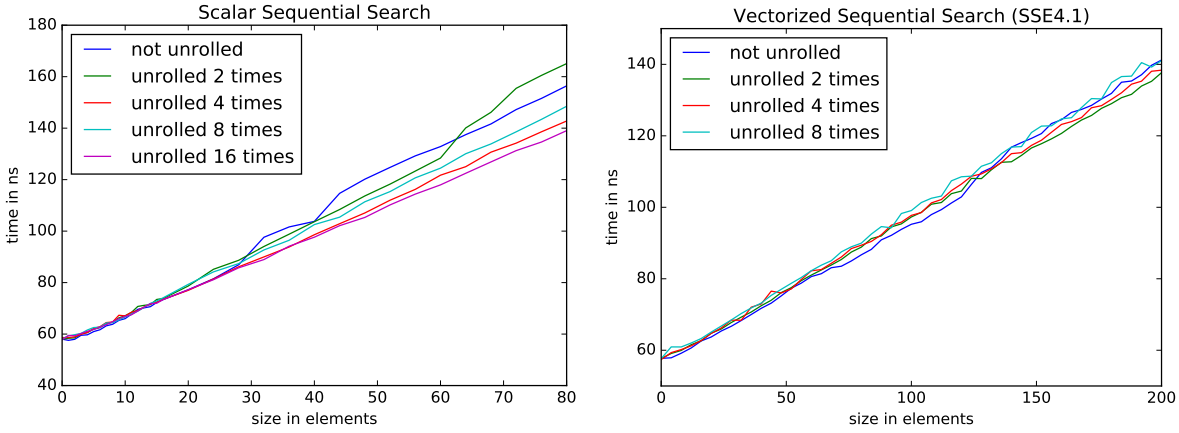

Figure 3.3: Unrolled branchless sequential search

A useful feature of loop unrolling is that it can improve instruction level parallelism, simply by providing more adjacent instructions, that can overlap in execution. To facilitate this we created unrolled search algorithms with no data dependencies between

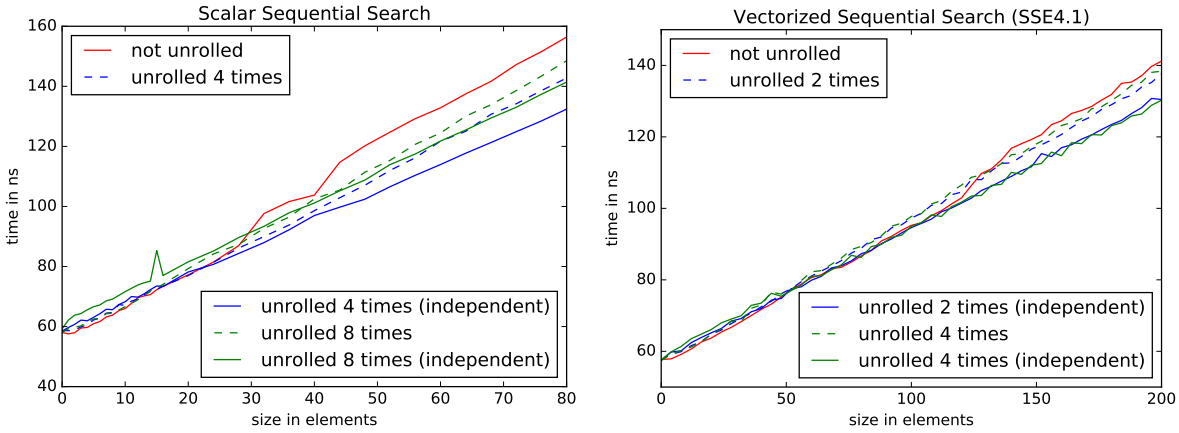

Figure 3.4: Unrolled branchless sequential search with independent search steps

the unrolled iterations. Figure 3.4 shows the result of our evaluation of this technique. Removing data dependencies is effective for both the scalar and vectorized search functions. For arrays with more than approximately 20 keys, the 4 times unrolled scalar search with independent iterations is faster than the simple 4 times unrolled version, making it the fastest algorithm in the plot. Something similar happens in the plot of the vectorized searches, where the 2 and 4 times unrolled versions clearly are the best for arrays with more than about 130 keys. However, for less then 20 keys the simple unrolled versions are still the faster scalar searches, with the basic branchless implementation with no loop unrolling at all being the fastest. Since the branching implementation gets faster than the branchless version at about the same point the unrolled search with independent search steps sets itself apart, we do not recommend it. The same is true in the SIMD case, so we do not see any advantages in unrolling the loop in the branchless implementation.

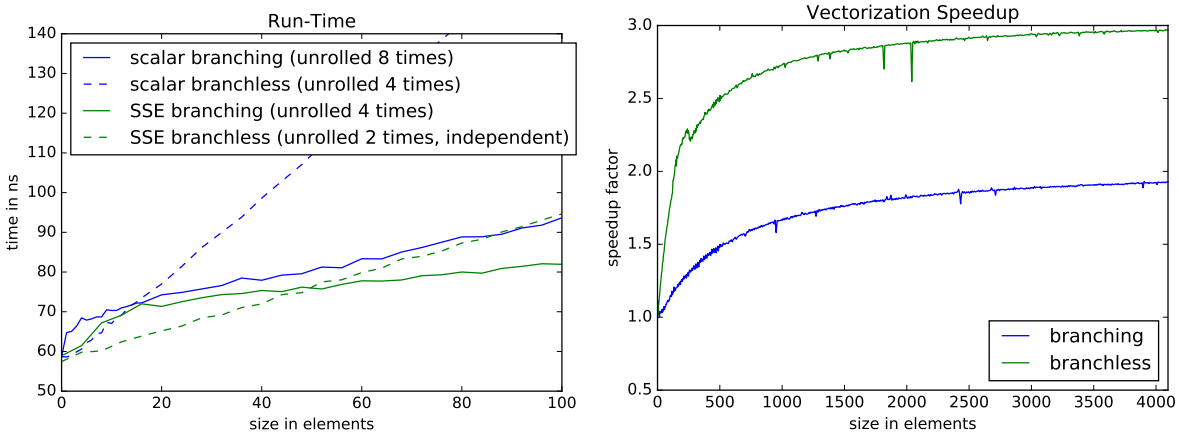

Figure 3.5: Comparison of scalar and vectorized (SSE4.1) sequential search implementations

3.3.4 Vectorization

In Figure 3.5 we are comparing the best scalar and vectorized search functions. From the left graph we conclude, that the branchless SSE implementation offers the best performance for arrays with up to about 50 elements. For arrays with more elements, the branching vectorized search is better. The right graph shows the speedup obtained by vectorization. At an array size of 100 elements, the branching vectorized search processes about 10% more keys per second than the scalar implementation. The difference is greater in the branchless search, where vectorization gives a speed-up of about 75% for 100 elements. At an array size of 4096 keys, the speedup factor has converged to about 1.9 for the branching and 3 for the branchless search.

3.4 Summary

We have implemented the sequential search algorithm both to search for an exact match and to search for the lower bound. To utilize the SIMD capabilities of current CPUs, we have not only implemented a traditional scalar search function, but also a vectorized variant using the SSE instruction set extensions. Then we introduced optimized versions of these functions, applying the principles of branch elimination and loop unrolling to them. Finally, we evaluated the implementation variants with randomly generated test data. In doing so, we found the functions using SSE to be always faster than the scalar implementations. Branch elimination provides an additional speed-up for small arrays. How small an array needs to be for the branchless functions is likely dependent on the machine and needs to be evaluated from case to case. Loop unrolling proved to be valuable for the branching search functions, but the branchless versions were only significantly improved for array sizes where the branching implementations already performed better.

4. Binary Search

The binary search is generally considered the default choice for searching in sorted lists. In this chapter, we analyze the behavior of different scalar and vectorized implementations and experiment with branch elimination, loop unrolling and software-controlled prefetching.

4.1 Implementation

In the following, we present implementations of the binary search algorithms from Section 2.2.2.1. Additionally we show an implementation of the so called uniform binary search. We adapted both of these variants of binary searching to SIMD processing using the idea outlined in Section 2.2.2.1. While binary searching normally divides in two equally sized partitions, we have also implemented functions using other split ratios. We will refer to this as offset binary search. Finally we discuss an implementation of the Fibonaccian search, that itself is a dichotomic search using a split ratio other than 1:1.

4.1.1 Scalar Binary Search

We start with the basic binary search in Listing 4.1. The implementation is nearly identical to the pseudocode presented in Algorithm 4 on page 13. Refer to Section 2.2.2.1 on page 13 for an explanation of its operation. In the same section we have described a binary search algorithm to search for the lower bound (Algorithm 5 on page 14), its implementation is shown in Listing 4.2.

Listing 4.1: Binary Search

```
1  template <typename T>
2  IndexType binarySearch(const T *keys, IndexType size, T searchKey) {
3    int left = 0, right = (int)size - 1; // both inclusive
4    while (left <= right) {
5      int mid = left + (right - left) / 2;
6      if (searchKey == keys[mid])
7        return (IndexType)mid;
8      else if (searchKey < keys[mid])
9        right = mid - 1;
10     else
11       left = mid + 1;
12   }
13   return size; // not found
14 }
```

Listing 4.2: Lower Bound Binary Search

```
1  template <typename T>
2  IndexType lowerBoundBinarySearch(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType left = 0, right = size; // left inclusive, right exclusive
5    while (left < right) {
6      IndexType mid = left + (right - left) / 2;
7      if (searchKey <= keys[mid])
8        right = mid;
9      else
10       left = mid + 1;
11   }
12   return left; // left == right
13 }
```

Note that the lower bound search is more general than the exact match search. The result of a lower bound search just has to be checked against the array one last time to create the behavior of an exact match search. We have implemented a binary exact match search in Listing 4.3 using the lower bound search to demonstrate this. `binarySearchByLowerBoundSearch` has the advantage of only needing a single if-statement in the loop body, therefore reducing the potential for branch mispredictions. On the other hand, it always loops until the search interval becomes empty, even when an examined separator element already is the key being searched for. In contrast, the function `binarySearch` terminates as soon as the search key has been tested. If we assume all searches to be successful and the keys to be chosen with equal probability, we can on average expect about one iteration less than the worst case of `binarySearchByLowerBoundSearch`. This is, because the number of potential nodes in a binary tree doubles with each level. Note that the if-statement in line 5 could be implemented as a conditional move, if the array is augmented with a sentinel key not equal to the search key, thus elimination the `lowerBound < size` test.

Listing 4.3: Exact match binary search implemented in terms of a lower bound binary search

```
1 template <typename T>
2 IndexType binarySearchByLowerBoundSearch(
3   const T *keys, IndexType size, T searchKey) {
4   IndexType lowerBound = lowerBoundBinarySearch(keys, size, searchKey);
5   if (lowerBound < size && keys[lowerBound] == searchKey)
6     return lowerBound;
7   else
8     return size; // not found
9 }
```

4.1.2 Scalar Uniform Binary Search

Listing 4.4 shows an alternative binary search implementation using the variables `left` and `dist` to keep track of the search interval instead of `left` and `right`. `left` is the index of the first key in the interval, and `dist` is used to compute the offset of the separator element from the beginning of the search interval. The function in Listing 4.4 performs a lower bound search, an exact match variant is obtained by replacing line 10 with the commented out if-statements.

Listing 4.4: Lower Bound Uniform Binary Search

```
1  template <typename T>
2  IndexType lowerBoundUniformBinarySearch(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType height = ceilLog2(size + 1);
5    IndexType dist = 1 << (height - 1);
6    IndexType left = 0;
7    IndexType mid = size - dist; // avoid out-of-bounds access
8    dist = dist >> 1;
9    while (height--- > 0) {
10     if (searchKey > keys[mid]) left = mid + 1;
11     // For an exact match search:
12     // if (searchKey == keys[mid]) return mid;
13     // else if (searchKey > keys[mid]) left = mid + 1;
14     mid = left + dist - 1;
15     dist = dist >> 1;
16   }
17   return left;
18 }
```

Arrays with perfect length

The search effectively assumes the arrays to correspond to perfect binary search trees, i.e. to have a length of $2^n - 1$ with an integer n. This simplifies the index computations, since the search sub-intervals themselves have a length of $2^n - 1$ with decreasing n, and `dist` is always a power of two. In line 4 we use the function `ceilLog2`, defined in the appendix (Listing A.2), to compute the height $\lceil \log_2(\texttt{size + 1}) \rceil$ of the conceptual

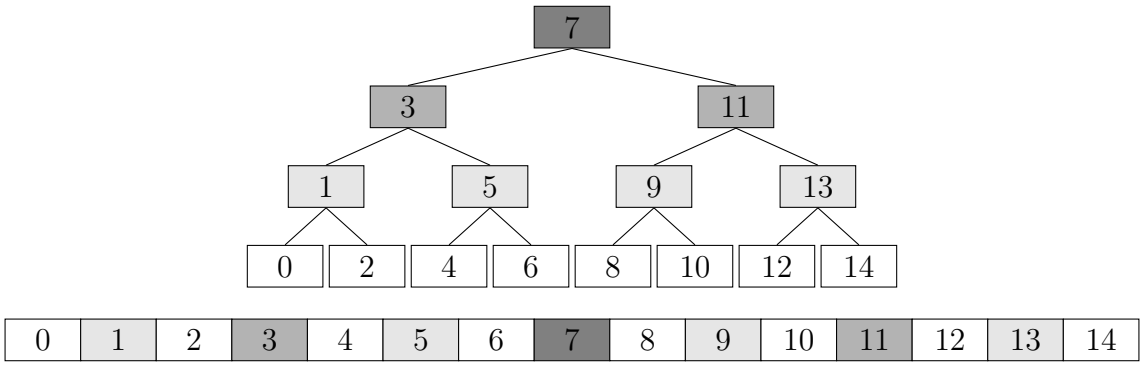

Figure 4.1: Perfect binary search tree containing 15 keys and the corresponding ordered list.

search tree corresponding to the array. Figure 4.1 shows a perfect binary tree with 15 keys and the corresponding flat array. If we define the height h of a node in the tree as the number of levels below it, the separator elements are at indices of the form $left + 2^h - 1$, where *left* is the start index of a sub-interval. This means we need two variables to describe the current position in the conceptual search tree: The current start offset from the beginning of the array `left`, and the current height `height`. Instead of using `height` in the index computations, we use $\texttt{dist} = 2^{\texttt{height}}$.

Generalization to arbitrary array lengths

The generalization to array with arbitrary lengths is achieved by letting the first two partitions overlap. 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 $\texttt{dist} - 1$. 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 $\texttt{size} - \texttt{dist}$ in line 7. The number of keys in both the left and right partition is $2^{\texttt{height}} - \texttt{size} - 1$. Note that the initial overlap means the search can perform a redundant comparison after the first iteration. Figure 4.2 shows an example of a search on an array with 10 keys. After the key 2 has been examined in the first iteration, the search proceeds with the right partition of perfect size. Figure 4.3 shows an example, where the search continues with the left partition after the first iteration. Since the initial array size was not perfect, the search interval considered in iteration 2 includes the keys 3, 4, 5 and 6, that already could have been eliminated in the first iteration. Nevertheless, the search still terminates after four iterations.

The important advantage of `lowerBoundUniformBinarySearch` over `lowerBoundBinarySearch` is that the former always needs exactly $\lceil \log_2(\texttt{size + 1}) \rceil$ iterations, regardless of the search key. This will allow us to unroll the loop in Section 4.2.

Iteration 1	0	1	**2**	3	4	5	6	7	8	9
Iteration 2	0	1	2	3	4	5	**6**	7	8	9
Iteration 3	0	1	2	3	**4**	5	6	7	8	9
Iteration 4	0	1	2	3	4	**5**	6	7	8	9

Figure 4.2: Example of `lowerBoundUniformBinarySearch` searching for the key 5. The search interval is shaded and the separator elements are printed in boldface.

Iteration 1	0	1	**2**	3	4	5	6	7	8	9
Iteration 2	0	1	2	**3**	4	5	6	7	8	9
Iteration 3	0	**1**	2	3	4	5	6	7	8	9
Iteration 4	**0**	1	2	3	4	5	6	7	8	9

Figure 4.3: Example of `lowerBoundUniformBinarySearch` searching for the key 0. The search interval is shaded and the separator elements are printed in boldface.

4.1.3 Vectorized Binary Search

Our SIMD implementations of the binary search technique use the same prolog as the sequential searches. Its definition is in Listing 3.3.

Listing 4.5 shows a binary search using SIMD loads and comparisons. The array indices `left`, `right` and `mid` refer to SIMD word sized blocks instead of individual elements, leaving the loop condition and the calculation of the middle the same as in the scalar search. In lines 9 to 11 a complete block of separator elements is loaded. It is then compared for equality in lines 14 to 20. The resulting mask is evaluated using the function `getFirstPositiveResult` defined in Section A.3.2 of the appendix. It returns `true` and sets `i` to the index of the first positive result in the mask, if at least one bit in the mask is set. Consider the 128-bit SSE mask 0x0000 0000 0000 0000 FFFF FFFF 0000 0000. If we are working with 32-bit keys, `getFirstPositiveResult` would set `i` to 2, since the first set bit lies in the second 32-bit word (counted starting from the least significant bit). Therefore, the search terminates successfully, if the body of the if-statement in line 17 is entered. The index of the found key is computed from the separator word position `mid` in the array and the offset `i` in the current separator array. If the equality test fails, a greater-than comparison is performed in line 22. Its result mask is evaluated using the `ptest` instruction. If no bit in the mask is set, all separators are greater than the search key and the search continues to the left. If all bits are set, all separators are smaller than the search key and the search continues with the right partition. There is one additional branch compared the scalar search: The search can

terminate early if a word of separators contains keys both smaller and greater than the search key in line 31, because we have already checked for equality. If the loop is exited because the search interval became empty, the search terminates unsuccessfully in line 34.

Listing 4.5: Vectorized Binary Search

```
1  template <typename T>
2  IndexType binarySearchSIMD(
3    const T *keys, IndexType size, T searchKey) {
4    PROLOG();
5    // indices correspond to SIMD words, not to individual keys
6    int left = 0, right = simdWords - 1; // both inclusive
7    while (left <= right) {
8      int mid = left + (right - left) / 2;
9      Vector separators = adjustForSignedComparison<T>(
10       loadVector(reinterpret_cast<const Vector*>(
11         keys + KEYS_PER_WORD * mid)));
12
13     // compare for equality
14     Vector compResult = _mm_cmpeq<T>(vecSearchKey, separators);
15     int mask = createMask<T>(compResult);
16     unsigned long i = 0;
17     if (getFirstPositiveResult<T>(&i, mask)) {
18       // search key is equal to one of the separators
19       return KEYS_PER_WORD * mid + i;
20     }
21
22     compResult = _mm_cmpgt<T>(vecSearchKey, separators);
23     if (_mm_testz<T>(compResult, vecOnes)) {
24       // no bit in compResult is set
25       right = mid - 1; // all keys are greater than the search key
26     }
27     else if (_mm_testc<T>(compResult, vecOnes)) {
28       // all bits in compResult are set
29       left = mid + 1; // all keys are smaller than the search key
30     } else {
31       return size; // the search key is in not in the array
32     }
33   }
34   return size; // not found
35 }
```

Alternatively the SIMD mask generated in line 22 can be evaluated using the instructions `pmovmskb` and `test`. Listing 4.6 shows the alternative code. First, the mask is transferred to a general purpose register in line 2 using the function `createMask`, which in turn compiles to a `pmovmskb`. Then regular if-statements operating on `int` values are used for the evaluation. The constants `NONE` and `ALL` are defined as 0 and 0xFFFFFFFF, respectively (see Listing A.9). This implementation has the advantage of not requiring the `ptest` instruction, which is only available since SSE4.1. Additionally the combination of `pmovmskb` and `test` might execute faster on some processors.

Listing 4.6: Alternative mask evaluation for `binarySearchSIMD`

```
1 compResult = _mm_cmpgt<T>(vecSearchKey, separators);
2 mask = createMask<T>(compResult);
3 if (mask == MASK<Vector, T>::NONE)
4   right = mid - 1;
5 else if (mask == MASK<Vector, T>::ALL)
6   left = mid + 1;
7 else
8   return size;
```

The SIMD lower bound search is listed in Listing 4.7. First note the loop condition, together with both indices—`left` and `right`—being inclusive it terminates the loop when a single block is left in the search range. This is important, because the last iteration requires special handling. The decision between the left and right partition is the same as in the previous algorithm (lines 15 to 22), but the case where the mask contains both ones and zeros is different (lines 23 to 28). If the algorithm reaches line 26, the mask contains a switch from one bits, indicating elements not smaller than `searchKey`, to zero bits, indicating elements smaller than `searchKey`, i.e. the lower bound is in the current block. The search then terminates using the `countPositiveResults` function to obtain the correct index. This means, that the function can terminate before the last iteration, in contrast the the scalar lower bound search (Listing 4.2).

Lines 35 to 40 deal with the last iteration, if the algorithm has not exited earlier. `countPositiveResults` is used to obtain the index of the first element not smaller than the search key, i.e. the lower bound. If all bits in the mask are set, this index is outside of the current block. This allows the lower bound to be one past the last valid index and is similar to how the scalar lower bound search operates in its last iteration. Like the scalar algorithm, the SIMD algorithm favors the left partition in case of equality: Line 18 is executed when all separators in a block are equal to the key being searched for.

Figure 4.4 depicts an example of the algorithm's operation. Note how the correct lower bound is not part of the search range any more in the second iteration. This is the case where the special handling of the last iteration in lines 35 to 40 becomes important. Since the search key is greater than all the separator elements, it is correct to return the index after the last separator, i.e. 4.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Iteration 1	0	2	4	6	**8**	**10**	**12**	**14**	16	18	20	22	24	26	28	30
Iteration 2	**0**	**2**	**4**	**6**	8	10	12	14	16	18	20	22	24	26	28	30
Iteration 3	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30

Figure 4.4: Example of `lowerBoundBinarySearchSIMD` searching for the key 7. The interval [`left`, `right`] is shaded and the separator elements are printed in boldface.

Listing 4.7: Vectorized Lower Bound Binary Search

```
template <typename T>
IndexType lowerBoundBinarySearchSIMD(
  const T *keys, IndexType size, T searchKey) {
  if (size == 0) return 0;
  PROLOG();
  // indices correspond to SIMD words, not to individual keys
  int left = 0, right = (int)simdWords - 1; // both inclusive
  while (left < right) {
    int mid = left + (right - left) / 2;
    Vector separators = loadVector(reinterpret_cast<const Vector*>(
      keys + KEYS_PER_WORD * mid));
    Vector compResult = _mm_cmpgt<T>(
      vecSearchKey, adjustForSignedComparison<T>(separators));

    if (_mm_testz<T>(compResult, vecOnes)) {
      // all keys are greater than the search key or separators
      // contains the first key equal to the search key
      right = mid - 1;
    }
    else if (_mm_testc<T>(compResult, vecOnes)) {
      // all keys are smaller than the search key
      left = mid + 1;
    } else {
      // the search key is in separators
      // or not in the array at all
      int mask = createMask<T>(compResult);
      return KEYS_PER_WORD * mid + countPositiveResults<T>(mask);
    }
  }

  // If mask is 0, the first key in the current word is
  // the first key equal or greater than the search key.
  // If mask is 0xFFFF, the first key not smaller than
  // the search key is to the right of the current word.
  Vector values = loadVector(reinterpret_cast<const Vector*>(
    keys + KEYS_PER_WORD * left));
  Vector compResult = _mm_cmpgt<T>(
    vecSearchKey, adjustForSignedComparison<T>(values));
  int mask = createMask<T>(compResult);
  return KEYS_PER_WORD * left + countPositiveResults<T>(mask);
}
```

4.1.4 Vectorized Uniform Binary Search

In the same way we constructed the vectorized search in the last section, a SIMD uniform binary search can be derived from the scalar uniform search. Listing 4.8 shows how we do this for the lower bound algorithm. An exact match variant can be constructed analogously.

Listing 4.8: Vectorized Lower Bound Uniform Binary Search

```
1  template <typename T>
2  IndexType lowerBoundUniformBinarySearchSIMD(
3    const T *keys, IndexType size, T searchKey) {
4    if (size == 0) return 0;
5    PROLOG();
6    IndexType height = ceilLog2(simdWords + 1);
7    IndexType dist = 1 << (height - 1);
8    IndexType left = 0;
9    IndexType mid = simdWords - dist;
10   dist = dist >> 1;
11
12   while (height-- > 1) {
13     Vector separators = loadVector(
14       reinterpret_cast<const Vector*>(
15         keys + KEYS_PER_WORD * mid));
16     Vector compResult = _mm_cmpgt<T>(
17       vecSearchKey, adjustForSignedComparison<T>(separators));
18
19     if (_mm_testc<T>(compResult, vecOnes)) {
20       // all bits in mask are set
21       left = mid + 1; // all keys are smaller than the search key
22     }
23     else if (!_mm_testz<T>(compResult, vecOnes)) {
24       // any bit in mask is set
25       int mask = createMask<T>(compResult);
26       return KEYS_PER_WORD * mid + countPositiveResults<T>(mask);
27     }
28     mid = left + dist - 1;
29     dist = dist >> 1;
30   }
31   // handle last SIMD word, see lowerBoundBinarySearchSIMD
32 }
```

4.1.5 Offset Binary Search

Normally a binary search splits the search interval in equally sized halves, but other division ratios are also possible. In Listing 4.9 we present a binary search partitioning the search range in the ratio `a` : `b`. The only difference to Listing 4.2 is the modified calculation of the separator element's location in line 7. Note that the multiplication in that calculation can overflow the index type, so care must be taken to choose a suitable type. Note that the division in line 7 can be implemented with a single shift if $\texttt{a} + \texttt{b}$ is a power of two, so such parameters might be preferable.

Uncommenting lines 10 and 13 yields a slightly different offset binary search partitioning in the ratio `b` : `a` if the search key is to the left of the current separator element, and in the ratio `a` : `b`, if it is to the right. If we choose $\texttt{a} < \texttt{b}$, this guarantees that the next separator element has the same distance to the current separator, regardless of the side

the search continues in. The Fibonaccian search discussed in the next section has the same property.

Listing 4.9: Lower Bound Offset Binary Search

```
1  template <typename T, IndexType a = 1, IndexType b = 2>
2  IndexType lowerBoundOffsetBinarySearch(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType left = 0, right = size;
5    IndexType numerator = a;
6    while (left < right) {
7      IndexType mid = left + (numerator * (right - left)) / (a + b);
8      if (searchKey <= keys[mid]) {
9        right = mid;
10       // numerator = b; (optional)
11     } else {
12       left = mid + 1;
13       // numerator = a; (optional)
14     }
15   }
16   return left;
17 }
```

Note that `lowerBoundOffsetBinarySearch` needs more iterations than the non-offset binary search if the larger partition is chosen often. However, since the search range is reduced by a constant factor smaller than one in each iteration, the asymptotic complexity stays logarithmic.

4.1.6 Fibonaccian Search

While the term binary search most often refers to a dichotomic search splitting the search interval in the middle, the Fibonaccian search is still very similar to the classical binary search. Therefore, its implementation will be useful for our evaluation, especially concerning the different memory access patterns. Additionally the Fibonaccian search is related to the offset binary search presented above. If we try to construct an offset binary search with precalculated subdivisions, we see that we need a sequence of integers with $a_n = a_{n-1} + a_{n-2}$ to locate the separator elements, since we want to evenly split a range of a_n keys into two sub-ranges, that themselves can be split evenly in the same way. In that sense, the Fibonaccian search is to the offset binary search what the uniform binary search is to the (non-uniform) binary search.

First we define some helper functions related to the Fibonacci numbers. The functions `prevFib` and `prevPrevFib` in Listing 4.10 both take pointers to two consecutive Fibonacci numbers p and q with $p > q$. `prevFib` calculates the previous pair of Fibonacci numbers (p', q') as $(q, p - q)$, and `prevPrevFib` calculates the previous pair of the previous par of Fibonacci numbers (p'', q'') as $(p-q, q-(p-q))$, as such it performs the same operation as two applications of `prevFib`. The third helper function `findFibonacciNumbers`, calculates three consecutive Fibonacci numbers $a < b < c$ with c being the greatest Fibonacci numbers still smaller than the parameter `size`. Additionally it sets `n` to the index of c in the Fibonacci sequence as defined in Section 2.2.2.2.

Listing 4.10: Helper functions for the Fibonaccian search

```
1  inline void prevFib(IndexType *p, IndexType *q) {
2    auto temp = *q;
3    *q = *p - *q;
4    *p = temp;
5  }
6
7  inline void prevPrevFib(IndexType *p, IndexType *q) {
8    *p = *p - *q;
9    *q = *q - *p;
10 }
11
12 inline std::tuple<IndexType, IndexType, IndexType>
13 findFibonacciNumbers(IndexType size, int *n) {
14   IndexType a = 0, b = 1, c = 1, d = 2;
15   *n = 2; // c is the n-th Fibonacci number
16   while (d < size) {
17     a = b;
18     b = c;
19     c = d;
20     d = b + c;
21     *n += 1;
22   }
23   return std::make_tuple(a, b, c);
24 }
```

The exact match search in Listing 4.11 is an implementation of Algorithm 7 on page 17 using the variables `p`, `q` and the functions defined above to efficiently generate the Fibonacci numbers. Note, how the search needs to obtain suitable start values using `findFibonacciNumbers` in line 7. If repeated searches on arrays with the same length are needed, this computation is only necessary once, and the results could be passed in as arguments to the search function.

We have constructed a lower bound Fibonaccian search in Listing 4.12 by omitting the last branch for the case of equality and by replacing the last return-statement. Note that our implementation has the property of sometimes checking the last separator element twice. For an example see Figure 4.5, where the key 8 is tested in both the fifth and sixth iteration. At the end of the loop, the lower bound can actually lie to the left of the last separator element. We can detect this by examining the variable `n`. If `n` is -1 the first branch in lines 11 to 13 was the last executed. In this case the search key is smaller than or equal to the last separator. If `n` is -2 the second branch was executed last, meaning that the search key is greater than the last separator element, so we have to add one the get the correct lower bound. We use the expression $-\texttt{n} - 1$ for this adjustment. For $\texttt{n} = -1$ it evaluates to 0, and for $\texttt{n} = -2$ to 1.

Listing 4.11: Fibonaccian Search

```
1  template <typename T>
2  IndexType fibonaccianSearch(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType i;
5    IndexType p, q; // two consecutive Fibonacci numbers (p > q)
6    int n;
7    std::tie(q, p, i) = findFibonacciNumbers(size, &n);
8    IndexType left = 0;
9    while (n >= 0) {
10     if (i >= size || searchKey < keys[i]) {
11       i = left + p;
12       prevFib(&p, &q);
13       n -= 1;
14     }
15     else if (searchKey > keys[i]) {
16       left = i;
17       i += q;
18       prevPrevFib(&p, &q);
19       n -= 2;
20     }
21     else // searchKey == keys[i]
22       return i;
23   }
24   return size; // not found
25 }
```

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Iteration 1	0	2	4	6	8	10	12	14	16	18	20	22	24	**26**	28	30
Iteration 2	0	2	4	6	8	10	12	14	**16**	18	20	22	24	26	28	30
Iteration 3	0	2	4	6	8	**10**	12	14	16	18	20	22	24	26	28	30
Iteration 4	0	2	4	**6**	8	10	12	14	16	18	20	22	24	26	28	30
Iteration 5	0	2	4	6	**8**	10	12	14	16	18	20	22	24	26	28	30
Iteration 6	0	2	4	6	**8**	10	12	14	16	18	20	22	24	26	28	30
Result	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30

Figure 4.5: Example of `lowerBoundFibonacciSearch` searching for the key 9. The interval still considered by the search is shaded and the separator elements are printed in boldface. The key 8 is tested in two iterations.

Listing 4.12: Lower Bound Fibonaccian Search

```
1  template <typename T>
2  IndexType lowerBoundFibonaccianSearch(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType i;
5    IndexType p, q; // two consecutive Fibonacci numbers (p > q)
6    int n;
7    std::tie(q, p, i) = findFibonacciNumbers(size, &n);
8    IndexType left = 0;
9    while (n >= 0) {
10     if (i >= size || searchKey <= keys[i]) {
11       i = left + p;
12       prevFib(&p, &q);
13       n -= 1;
14     } else {
15       left = i;
16       i += q;
17       prevPrevFib(&p, &q);
18       n -= 2;
19     }
20   }
21   return left + (IndexType)(-n - 1);
22 }
```

4.2 Optimizations

Like for the sequential search, we have applied branch elimination and loop unrolling to the binary search. Additionally we derived implementation variants using explicit software-controlled prefetching.

4.2.1 Branch Elimination

We replaced the if-else construct in the lower bound binary search (Listing 4.2) with two conditional moves. The conditional moves are represented as conditional operators in Listing 4.13. We did the same with the exact match variant, but here the branch for the equality test remains (Listing 4.14).

The statement `if (searchKey > keys[mid]) left = mid + 1` in the uniform binary search (Listing 4.4) already gets compiled to a conditional move, so no changes to the source code are required. Of course, the branch exiting the search loop in the exact match search must remain.

Listing 4.13: A branchless lower bound binary search is constructed by replacing lines 7 to 10 of Listing 4.2 with this code.

```
1  const IndexType midPlusOne = mid + 1;
2  right = searchKey <= keys[mid] ? mid : right;
3  left = searchKey <= keys[mid] ? left : midPlusOne;
```

Listing 4.14: Two branches in the binary search (Listing 4.1) are eliminated by replacing lines 6 to 11 with this code.

```
1 if (searchKey == keys[mid]) return (IndexType)mid;
2 const IndexType midMinusOne = mid - 1;
3 const IndexType midPlusOne = mid + 1;
4 right = searchKey <= keys[mid] ? midMinusOne : right;
5 left = searchKey <= keys[mid] ? left : midPlusOne;
```

The two branches selecting the left or right partition in the SIMD binary search algorithms in Listing 4.5 and Listing 4.7 are replaceable by conditional operators, yielding conditional move instructions, in the same way as for the scalar algorithms. Listing 4.15 and Listing 4.16 show the necessary modifications to the earlier listings. The final else-case terminating the algorithm has been replaced by a simple if-statement using `ptest` to test for a mix of zero and one bits. Listing 4.17 shows how we replaced one branch in the lower bound uniform binary search (Listing 4.4) using the same idea.

Listing 4.15: Two branches in the vectorized binary search (Listing 4.5) are eliminated by replacing lines 23 to 33 with this code.

```
1  const int midMinusOne = mid - 1;
2  const int midPlusOne = mid + 1;
3  right = _mm_testz<T>(compResult, vecOnes) ?
4    midMinusOne : right; // all zeros
5  left = _mm_testc<T>(compResult, vecOnes) ?
6    midPlusOne : left; // all ones
7  if (!_mm_testz<T>(compResult, vecOnes)
8    && !_mm_testc<T>(compResult, vecOnes)) {
9    return size; // mixed zeros and ones
10 }
```

Listing 4.16: Two branches in the vectorized lower bound binary search (Listing 4.7) are eliminated by replacing lines 14 to 28 with this code.

```
1  const int midMinusOne = mid - 1;
2  const int midPlusOne = mid + 1;
3  right = _mm_testz<T>(compResult, vecOnes) ?
4    midMinusOne : right; // all zeros
5  left = _mm_testc<T>(compResult, vecOnes) ?
6    midPlusOne : left; // all ones
7  if (!_mm_testz<T>(compResult, vecOnes)
8    && !_mm_testc<T>(compResult, vecOnes)) {
9    // mixed zeros and ones
10   int mask = createMask<T>(compResult);
11   return KEYS_PER_WORD * mid + countPositiveResults<T>(mask);
12 }
```

4.2.2 Prefetching

The access pattern of a binary search is essentially random to a typical hardware prefetcher, so it might be possible to improve performance by using explicit prefetch in-

Listing 4.17: A branch in the vectorized lower bound uniform binary search (Listing 4.7) is eliminated by replacing lines 18 to 29 with this code.

```
1 const int midPlusOne = mid + 1;
2 left = _mm_testc<T>(compResult, vecOnes) ? midPlusOne : left;
3 if (!_mm_testz<T>(compResult, vecOnes)
4   && !_mm_testc<T>(compResult, vecOnes)) {
5   int mask = createMask<T>(compResult);
6   return KEYS_PER_WORD * mid + countPositiveResults<T>(mask);
7 }
```

structions. In the source code, the `_mm_prefetch` intrinsic function is used for prefetching. It takes the memory address to load from and a hint specifying which of the four versions of the prefetch instruction to use (see Section 2.1.1).

Listing 4.18 shows a branchless lower bound binary search prefetching the next two possible separator elements one iteration ahead. In line 10 the index of the next separator element, if the left partition is chosen, is calculated. The location of the separator element of the right partition is calculated in line 15. The cache lines containing these keys are prefetched in lines 11 and 16, respectively. A third conditional move has been added in line 21 to reuse the already calculated separator index in the next iteration.

Listing 4.18: Branchless Lower Bound Binary Search with Prefetching

```
1  template <typename T, int PREFETCH_LOCALITY_HINT = _MM_HINT_T0>
2  IndexType lowerBoundBinarySearchBranchlessPrefetch(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType left = 0, right = size;
5    IndexType mid = left + (right - left) / 2;
6    while (left < right) {
7      IndexType midPlusOne = mid + 1;
8
9      // new mid if left partition is selected
10     IndexType mid1 = left + (mid - left) / 2;
11     _mm_prefetch(reinterpret_cast<const char*>(&keys[mid1]),
12       PREFETCH_LOCALITY_HINT);
13
14     // new mid if right partition is selected
15     IndexType mid2 = midPlusOne + (right - midPlusOne) / 2;
16     _mm_prefetch(reinterpret_cast<const char*>(&keys[mid2]),
17       PREFETCH_LOCALITY_HINT);
18
19     right = searchKey <= keys[mid] ? mid : right;
20     left = searchKey <= keys[mid] ? left : midPlusOne;
21     mid = searchKey <= keys[mid] ? mid1 : mid2;
22   }
23   return left; // left == right
24 }
```

Prefetching is also possible in the uniform binary search. Listing 4.19 shows a lower bound uniform binary search prefetching the next two possible separator elements one iterations in advance. To do this, it not only maintains the offset of the current search range in the variable `left1`, but also the offsets of the two partitions the current range is divided in. The starting index of the left partition is the same as for the current range, namely `left1`. The offset of the right partition is stored in `left2`. If the search continues in the left partition, `left1` remains unchanged. Otherwise the statement `left1 = left2` in line 20 is executed, selecting the right partition. In any case, the new right partition starts at `left1 + dist`, directly to the right of the next separator element (line 21). The actual prefetching is done in lines 14 to 17, with the two possible next separator elements at `left1 + dist - 1` and `left2 + dist - 1`.

Listing 4.19: Uniform Binary Search prefetching the next two possible separator elements one iterations in advance

```
1  template <typename T, int PREFETCH_LOCALITY_HINT = _MM_HINT_T0>
2  IndexType lowerBoundUniformBinarySearchBranchlessPrefetch2(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType height = ceilLog2(size + 1);
5    IndexType dist = 1 << (height - 1);
6
7    IndexType mid = size - dist;
8    dist = dist >> 1;
9    IndexType left1 = 0;
10   IndexType left2 = mid + 1;
11
12   while (height-- > 0) {
13     if (dist >= CACHE_LINE_SIZE / sizeof(T)) {
14       _mm_prefetch(reinterpret_cast<const char*>(
15         &keys[left1 + dist - 1]), PREFETCH_LOCALITY_HINT);
16       _mm_prefetch(reinterpret_cast<const char*>(
17         &keys[left2 + dist - 1]), PREFETCH_LOCALITY_HINT);
18     }
19
20     if (searchKey > keys[mid]) left1 = left2;
21     left2 = left1 + dist;
22     mid = left1 + dist - 1;
23     dist = dist >> 1;
24   }
25   return left1;
26 }
```

The function in Listing 4.20 looks even further ahead and prefetches the next four possible separator elements two iterations in advance. We use the four variable `left1`, `left2`, `left3` and `left4` to keep track of the possible partitions in the next two search steps. `left1` is the beginning of the current search range, its left sub-range, and the left sub-range of this sub-range. `left3` is the right sub-range of the current search interval, and the left sub-range of this right sub-range. `left2` and `left4` store the corresponding right sub-ranges of the two possible partitions of the current search range. Figure 4.6 illustrates this for a search range of length 15. Note that the spacing between `left1`,

`left2`, `left3` and `left4` is regular when the length of the current search range is $2^n - 1$ for some integer n, so a single variable `left` would be sufficient. An implementation with only a single `left` would free up registers and might lower the instruction count. However, this would require more code to handle the first iteration if the array size is not perfect. Because of this and for more compact source code we decided to use multiple variables.

The prefetches in Listing 4.19 and Listing 4.20 are skipped, if the distance between the elements to load is smaller than a cache line, because prefetching the same cache line multiple times would be redundant. The additional branch introduced by this is not problematic, since it is easily predictable.

Listing 4.20: Uniform Binary Search prefetching four possible separator elements two iterations in advance

```
1  template <typename T, int PREFETCH_LOCALITY_HINT = _MM_HINT_T0>
2  IndexType lowerBoundUniformBinarySearchBranchlessPrefetch4(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType height = ceilLog2(size + 1);
5    IndexType dist = 1 << (height - 1);
6
7    IndexType mid = size - dist;
8    dist = dist >> 1;
9    IndexType left1 = 0;
10   IndexType left2 = left1 + dist;
11   IndexType left3 = mid + 1;
12   IndexType left4 = left3 + dist;
13
14   while (height--- > 0) {
15     if (dist >= CACHE_LINE_SIZE / sizeof(T)) {
16       IndexType midOffset = (dist >> 1) - 1;
17       _mm_prefetch(reinterpret_cast<const char*>(
18         &keys[left1 + midOffset]), PREFETCH_LOCALITY_HINT);
19       _mm_prefetch(reinterpret_cast<const char*>(
20         &keys[left2 + midOffset]), PREFETCH_LOCALITY_HINT);
21       _mm_prefetch(reinterpret_cast<const char*>(
22         &keys[left3 + midOffset]), PREFETCH_LOCALITY_HINT);
23       _mm_prefetch(reinterpret_cast<const char*>(
24         &keys[left4 + midOffset]), PREFETCH_LOCALITY_HINT);
25     }
26
27     left1 = searchKey <= keys[mid] ? left1 : left3;
28     left3 = searchKey <= keys[mid] ? left2 : left4;
29     left2 = left1 + (dist >> 1);
30     left4 = left3 + (dist >> 1);
31     mid = left1 + dist - 1;
32     dist = dist >> 1;
33   }
34   return left1;
35 }
```

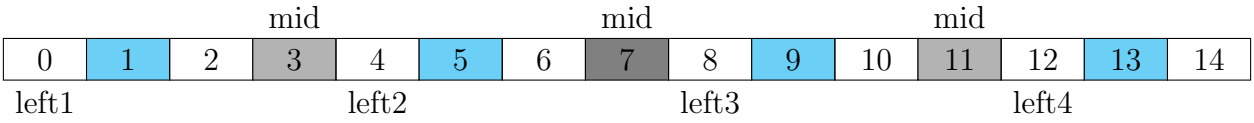

Figure 4.6: Indices used by `lowerBoundUniformBinarySearchBranchlessPrefetch4`. The current search range are the keys 0 to 14. 7 is the current separator element, 3 and 11 are the next two possible separator elements. left1, left2, left3 and left4 mark the four possible sub-ranges the search can continue with after two search steps. The cache lines holding the keys 1, 5, 9 and 13 are prefetched.

We also added explicit prefetching to the vectorized binary search and the vectorized uniform binary search. In case of the non-uniform search the necessary changes are analogous to Listing 4.18. For the vectorized uniform binary search we have implemented prefetching one iteration ahead analogously to Listing 4.19.

Listing 4.21: Unrolled Branchless Lower Bound Binary Search

```
1  #define SEARCH_STEP \
2    left = searchKey <= keys[mid] ? left : mid + 1; \
3    mid = left + dist - 1; \
4    dist = dist >> 1;
5
6  template <typename T>
7  IndexType lowerBoundUniformBinarySearchUnrolled(
8    const T *keys, IndexType size, T searchKey) {
9    IndexType height = ceilLog2(size + 1);
10   IndexType dist = 1 << (height - 1);
11
12   IndexType left = 0;
13   IndexType mid = size - dist; // avoid out-of-bounds access
14   dist = dist >> 1;
15
16   switch (height) {
17   case 31: SEARCH_STEP
18   case 30: SEARCH_STEP
19   /* ... */
20   case 2: SEARCH_STEP
21   case 1: SEARCH_STEP
22   case 0: break;
23   }
24   return left;
25 }
```

4.2.3 Loop Unrolling

The while-loops in Listing 4.1 and Listing 4.2 are difficult to unroll, since the loop condition is not determined by a simple counter variable. Instead, we have unrolled the loop of the uniform binary search. Listing 4.21 shows the result of completely unrolling the uniform binary search's loop for arrays with up to 2^{31} keys. This is possible, because

of the logarithmic complexity of the binary search and the regularity of our uniform implementation. The source code of the vectorized uniform binary search, we have unrolled in the same way as the scalar variant, is not shown here.

4.3 Evaluation

Our evaluation first deals with the lower bound scalar binary search, the lower bound scalar uniform binary search and the effects of branch elimination, software prefetching and loop unrolling on them. In Section 4.3.5 we analyze the vectorized search functions and in Section 4.3.6 we compare exact match search functions implemented either directly or based on the lower bound. The offset binary search and the Fibonaccian search are evaluated in Section 4.3.7. Finally we compare the algorithms concerning their effectiveness in utilizing the CPU caches in Section 4.3.8.

4.3.1 Evaluation Setup

We used the same compiler and hardware to evaluate the binary search, as we have used for the sequential search. See Section 3.3.1 for a description. Like in Section 3.3 we used randomly generated datasets for the evaluation. All keys are of a signed integral type with 32-bits. Our measurements are averages over 10,000 search runs. We have three different algorithms to generate the search keys:

Algorithm 1 The search keys are drawn from the same distribution as the array elements. This is the same as in Section 3.3.

Algorithm 2 The search keys are randomly selected from the array with equal probability.

Algorithm 3 5 sets of 128 keys are randomly selected from the array. Each set is used for 2000 consecutive searches, randomly sampling from the set.

Algorithm 1 was used in the sequential search chapter. We introduce Algorithm 2 in addition to Algorithm 1 to compare different exact match implementations in Section 4.3.6, because Algorithm 1 has a very small chance of producing a key actually present in the array. Since we concentrated on lower bound searching in the previous chapter, Algorithm 2 was not needed there. Algorithm 3 allows us to compare the caching behavior of different algorithms, since it generates search keys with better locality. For the sequential search this was not needed, since its access pattern always has a high degree of spatial and temporal locality. Unless noted otherwise, Algorithm 1 was used to generate the search keys in this chapter.

4.3.2 Branch Elimination

The run-time of `lowerBoundBinarySearch`, its branchless version and `lowerBoundUniformBinarySearch`, which is also branchless, are plotted in Figure 4.7. As can be seen, the branchless algorithms are faster then `lowerBoundBinarySearch` for arrays up to a size of 2^{17} elements. For larger arrays, their performance declines. In all cases, the uniform binary search is faster than the branching non-uniform implementation.

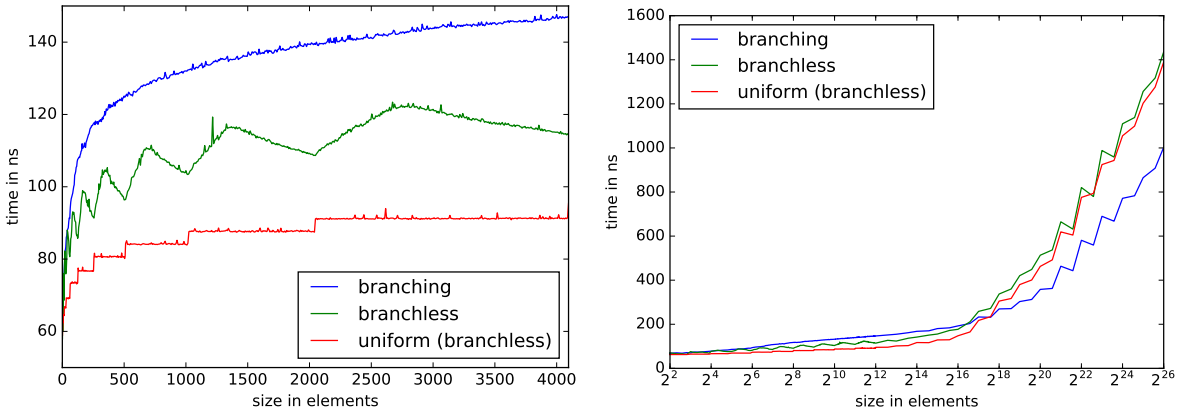

Figure 4.7: Lower Bound Binary Search Branch Elimination

The increased run-time of the branchless implementations can be explained with significantly more stalled clock cycles[1] than in the branching implementation for arrays of more than 2^{17} keys (left side of Figure 4.8). 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 branchless implementation. Additionally, the branching binary search makes use of hardware prefetching. The right side of Figure 4.8 shows the number of L2 hardware prefetcher requests[2]. As we can see, for more than 2^{17} keys the prefetcher is only active in the branching implementation.

The linearly scaled graph on the left side of Figure 4.7 shows very 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 the steps at powers of two we see in the plot. Interestingly, the branchless non-uniform search behaves very differently from the branching one. It oscillates, and has minimums at power of two element counts. Maximums occur about one third of the way to the next power of two. The origin of this behavior becomes clear, if we look at the number of retired branch mispredictions[3,4] in Figure 4.9. While the branching binary search

[1]Measured using performance counter event unit `0xA2` with unit mask `0x01`, see [Int16b]

[2]Measured using performance counter event unit `0x24` with unit mask `0xC0`, see [Int16b]

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

[4]Measured using performance counter event unit `0xC5` with unit mask `0x00`, see [Int16b]

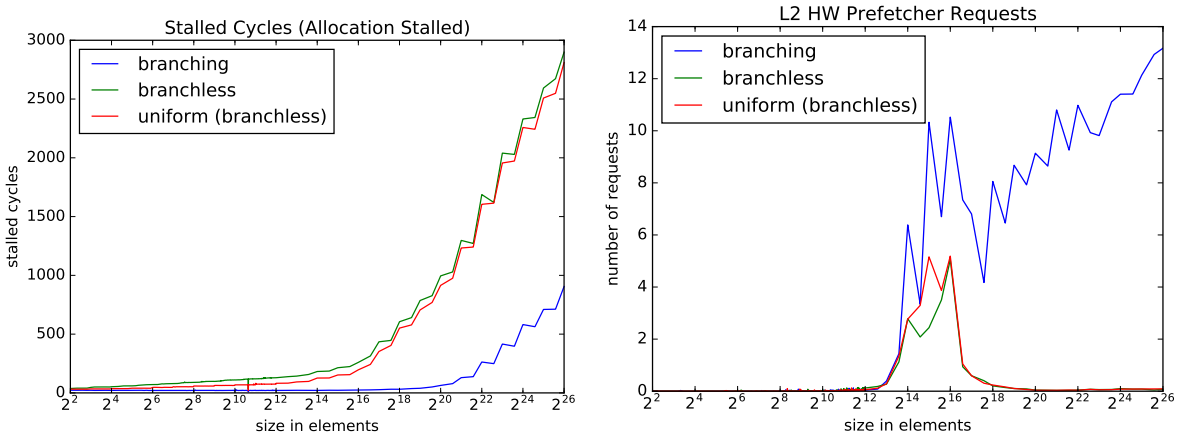

Figure 4.8: Stalled clock cycles and L2 hardware prefetcher requests

follows a logarithmic curve, and the uniform binary search sits at a constant number of 2 mispredictions, the branchless non-uniform algorithm oscillates with an amplitude of 0.5 mispredictions, reaching the level of the uniform search at powers of two. It is now clear, where the oscillations come from: At power of two array sizes the non-uniform binary search runs for exactly $\log_2(\text{array size}) + 1$ iterations, independently from which key is searched for. If the array size is not a power of two, either $\log_2(\text{array size})$ or $\log_2(\text{array size}) + 1$ iterations are needed. This makes the jump at the end of the search loop harder to predict.

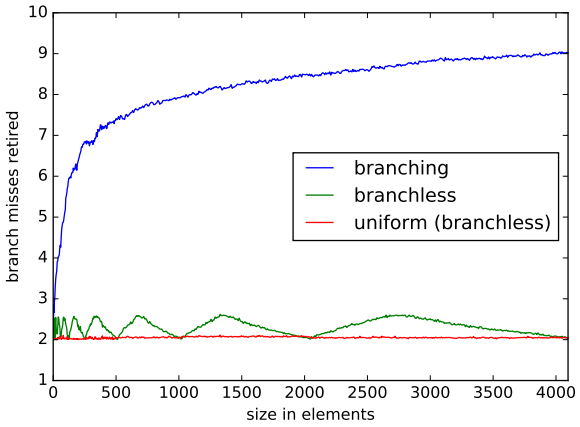

Figure 4.9: Retired Branch Mispredictions

4.3.3 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.

Because hardware prefetching did not work for larger arrays in the branchless implementations, software controlled prefetching might yield an improvement. Figure 4.10

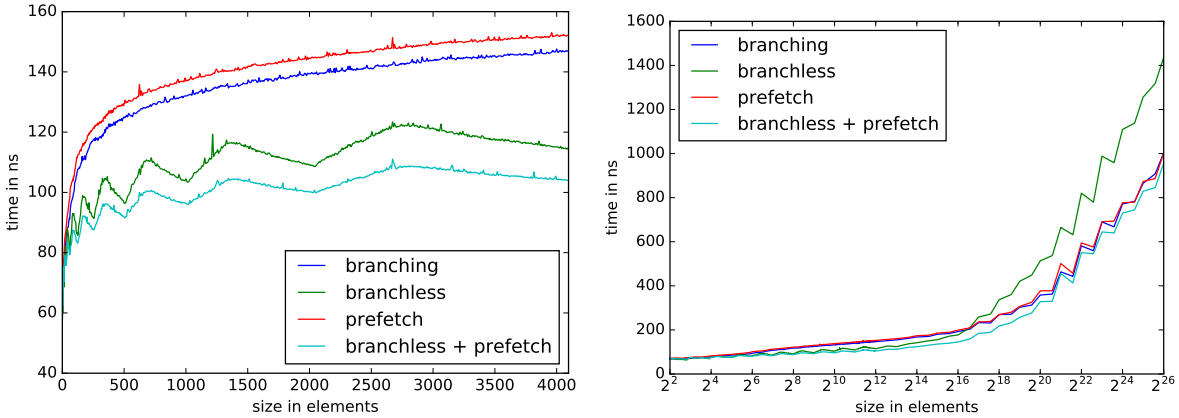

Figure 4.10: Software prefetching applied to the (lower bound) non-uniform binary search

shows our results for the non-uniform binary search. As we can see, additional software prefetching alone worsens the performance of the branching implementation. However, the branchless function is significantly improved and is now not only faster than the original branchless binary search, but is the fastest algorithm in this test.

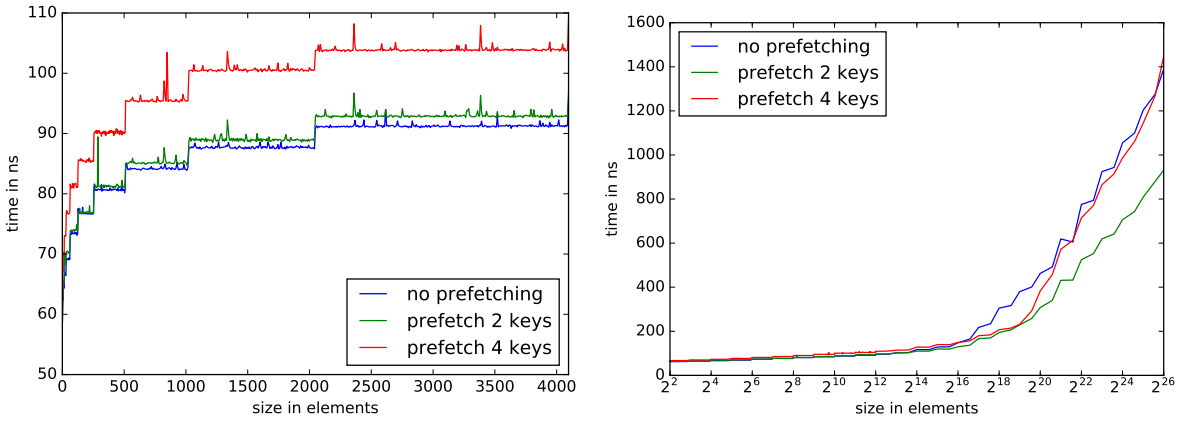

Figure 4.11: Software prefetching applied to the (lower bound) uniform binary search

Remember, that the uniform binary search algorithms conditionally call the prefetch intrinsic, to avoid redundant instructions. We found, that this slightly reduces the execution time by a small constant amount, which is especially noticeable for smaller arrays. Figure 4.11 shows, that prefetching one iteration in advance (prefetch 2 keys) increases the performance of `lowerBoundUniformBinarySearch` if the array has more than about 2^{16} keys. For smaller arrays however, prefetching adds a few nanoseconds to the run-time. Prefetching two iterations in advance (prefetch 4 keys) delays the performance drop of the uniform binary search to about 2^{19} elements, but then falls back to the run-time of the implementation with no software prefetching at all. Additionally, the overhead for small arrays is much higher as when only two keys per iteration are prefetched. Clearly this function wastes too many cycles executing prefetch instructions.

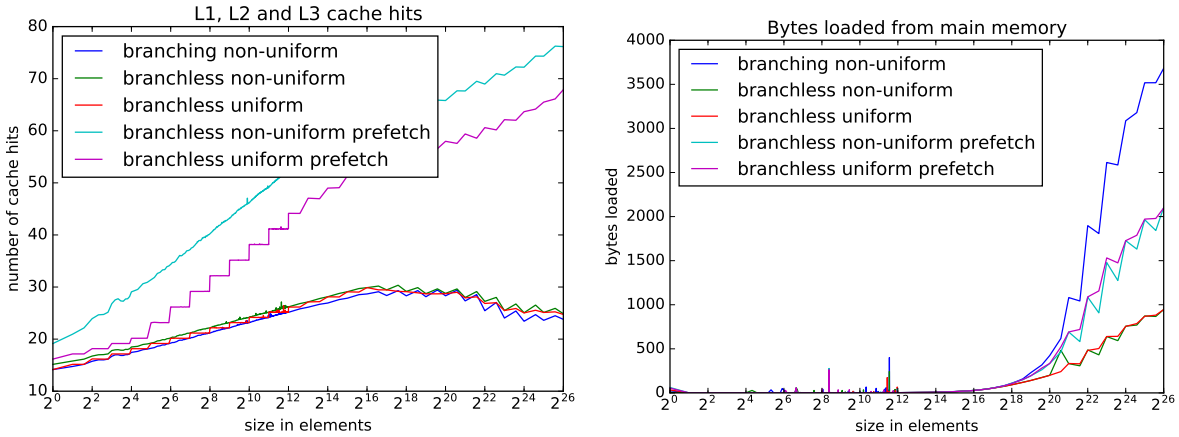

Figure 4.12: Cache hits and bytes loaded from main memory for the scalar binary search

In Figure 4.12 we show the number of L1, L2 and L3 cache hits[5] on the left and the number of bytes loaded from main memory[6] on the right. Interestingly, the non-uniform branching binary search using only hardware prefetching has about the same number of cache hits as the branchless binary and (branchless) uniform binary search for that L2 hardware prefetching is not working well. Both software prefetching search algorithms have much more cache hits. On the right side of Figure 4.12 we see, that explicit prefetching reduces the number of bytes loaded from main memory compared to the hardware prefetcher. Still, the fewest main memory accesses are performed by the branchless non-uniform and unoptimized uniform search. Therefore they require the least main memory bandwidth.

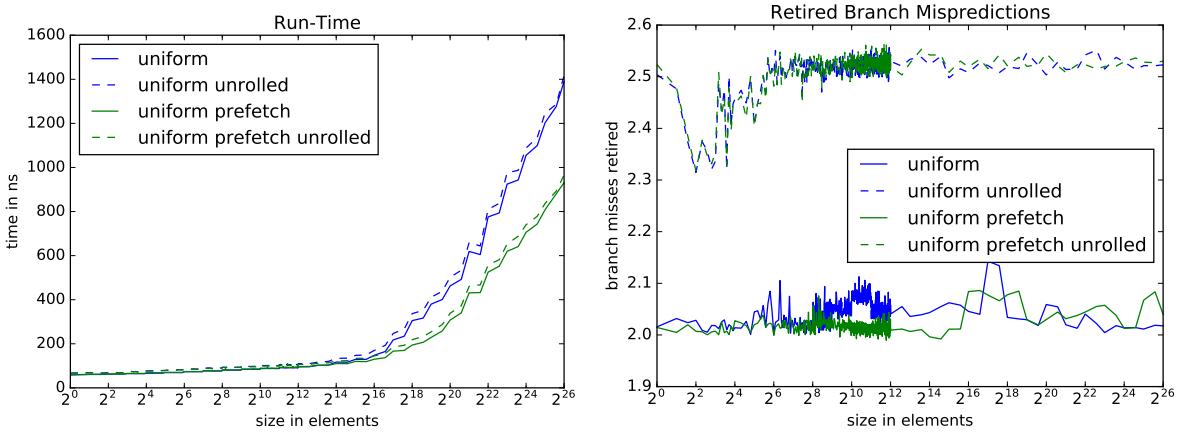

Figure 4.13: Loop Unrolling

[5]Measured using performance counter event unit `0xD1` with unit mask `0x07`, see [Int16b]
[6]Measured by the memory controller for the complete system

4.3.4 Loop Unrolling

The unrolled uniform binary search performs worse than the not unrolled version (left side of Figure 4.13). The loop overhead avoided by unrolling is too small to matter. Additionally, there are 0.5 more branch mispredictions on average (right side of Figure 4.13). We conclude that loop unrolling is not a suitable optimization for the binary search.

4.3.5 Vectorization

We compared the run-times of the vectorized binary search using the `ptest` instruction (Listing 4.5 and Listing 4.7) and the alternative implementation using a combination of `pmovmskb` and `test`/`cmp` (Listing 4.6). We found little to no difference between them, with the largest difference occurring in the exact match search if the search keys are generated using Algorithm 3, where the `pmovmskb` version is 5% faster for an array size of about 1000 keys.

The vectorized lower bound binary search is similar to the scalar implementation, in that the branchless version is significantly slower. Prefetching improves the run-time a bit, but it is still worse than the branching function. Finally, the version adding just software prefetching is slightly faster on arrays with more than 2^{16} elements (left side of Figure 4.14). In case of the uniform binary search, the branching implementation without prefetching is the fastest overall. Only for arrays with more than 2^{17} elements, the prefetching search has a similar and sometimes slightly smaller run-time (right side of Figure 4.14).

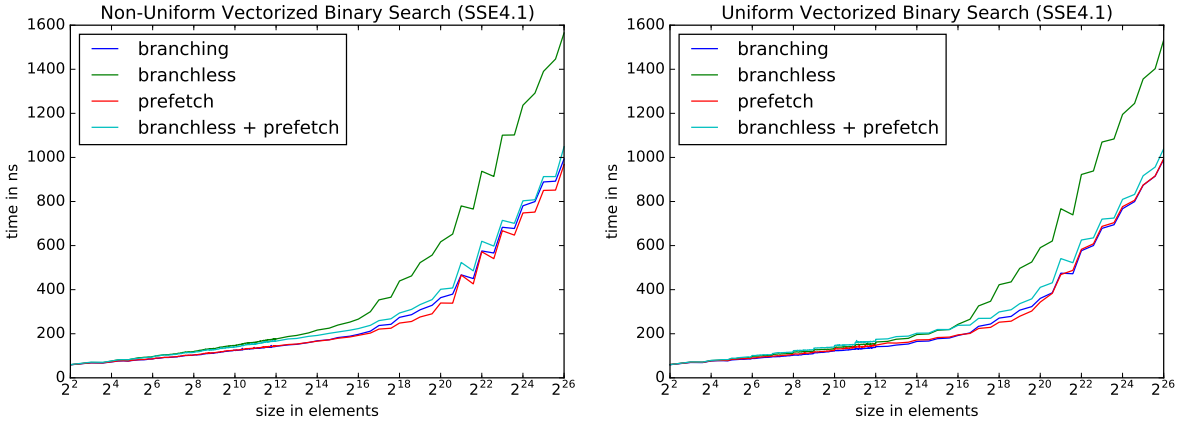

Figure 4.14: Lower Bound Vectorized Binary Search

The left side of Figure 4.15 shows the run-times of the fastest scalar and SIMD lower bound search algorithms. The vectorized implementations are slower than the scalar ones, with the exception of small arrays with less than 60 elements, where the vectorized algorithms perform similar to the non-uniform binary search. But even then, the uniform search is faster. The results are very similar for the exact match search functions

(right side of Figure 4.15). Additionally, we tested the vectorized implementations with 16 and 64-bit keys, with the same result: The scalar uniform binary search is faster than the SSE searches, with the only exception being arrays no larger than two 128-bit words, were the SSE binary search behaves more like a sequential search. We conclude, that the additional complexity introduced by using SIMD increases the run-time more than it is decreased by slightly reducing the number of iterations.

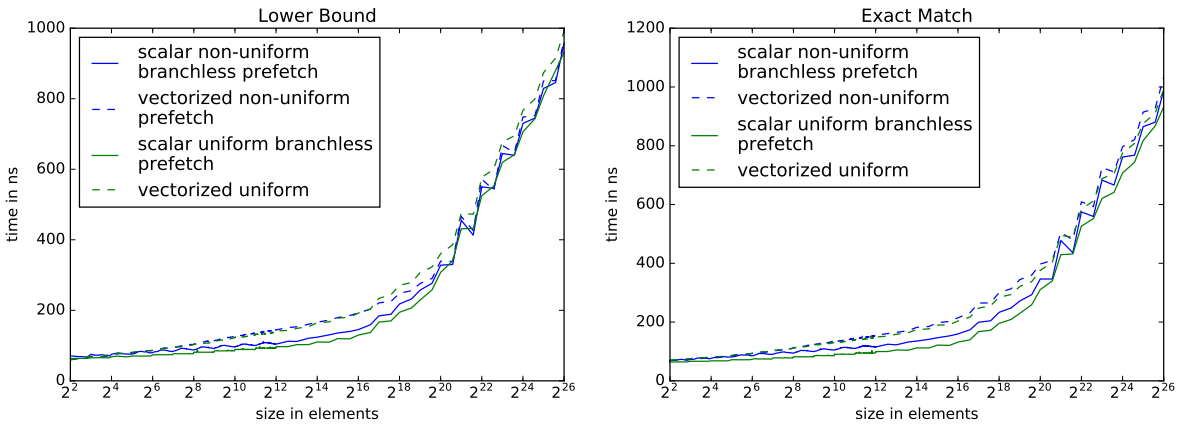

Figure 4.15: Comparison of scalar and vectorized binary search implementations

4.3.6 Exact Match Search

In this section, we compare direct exact match search implementations with implementations in terms of a lower bound search, like Listing 4.3. The direct implementations have an equality test in the search loop terminating the search as soon as the search key has been seen. In contrast, the lower bound based versions have the same search loop as a lower bound search and delay the equality test to the end.

We have measured the number of iterations exact match and lower bound algorithms needed on the evaluation dataset with keys generated by the three algorithms described in Section 4.3.1. For search keys drawn from the same distribution as the array elements, it is very unlikely for a search key to actually be in the array. Consequently both exact match and lower bound algorithms run for the same number of iterations. If the search keys are randomly drawn from the array and therefore are guaranteed to be present, exact match searches need almost precisely one iteration less. This is also the case for the third search key generation algorithm, where a small set of keys drawn from the array is repeatedly searched.

Figure 4.16 shows the run-times of the non-uniform branchless binary search with prefetching, the uniform branchless binary search prefetching 2 keys per iteration and alternative implementations based on the corresponding lower bound searches. For keys generated by Algorithm 1, we expect the lower bound implementations to be superior, because they have one branch less and run for the same number of iterations. For the non-uniform search this is indeed the case, but for the uniform search, the direct exact

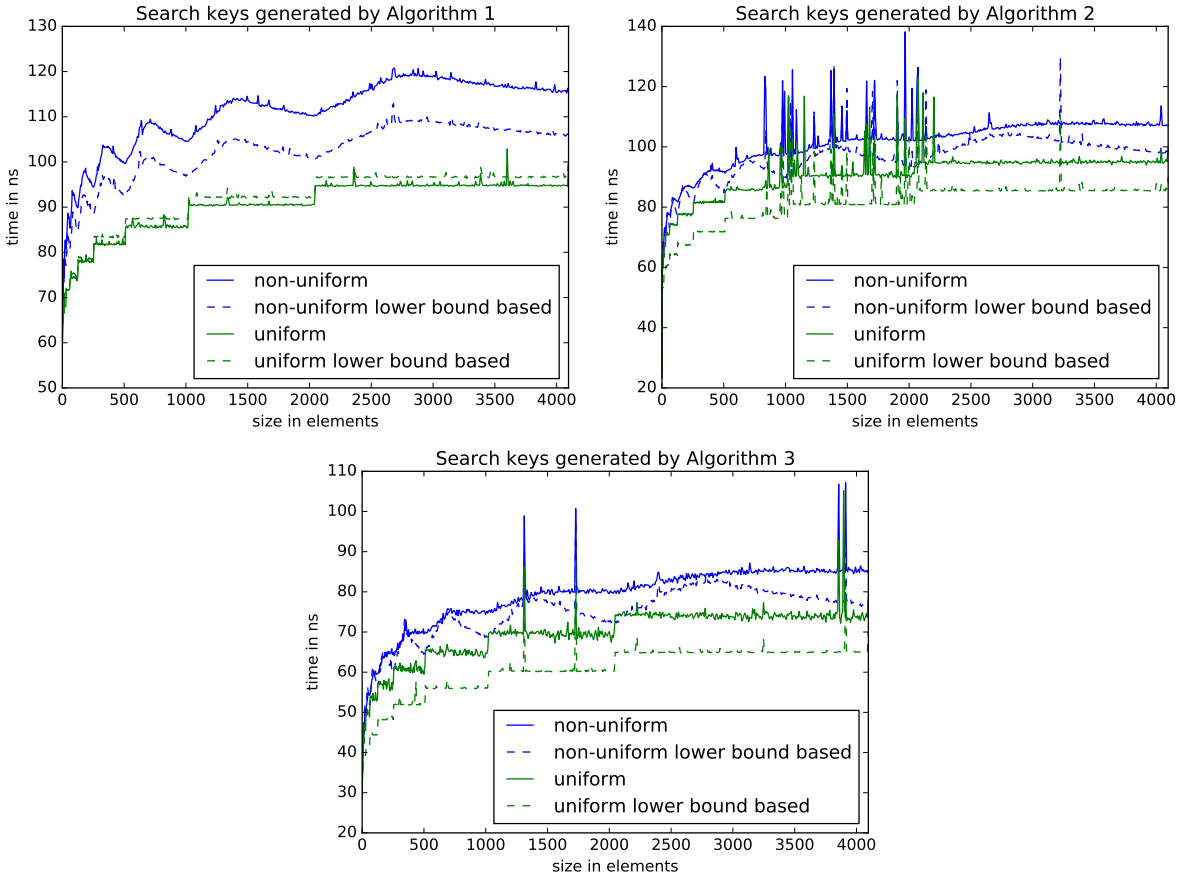

Figure 4.16: Exact match binary search and an exact match search implemented in terms of a lower bound search

match implementation is slightly faster. In the next test, with search keys generated by Algorithm 2, the lower bound based uniform search is significantly faster, since the equality test in the search loop of the direct implementation has become hard to predict. For the non-uniform search, the lower bound based version is still faster. Finally, the last test using Algorithm 3 to select the search keys, produces a result almost identical to Algorithm 2.

4.3.7 Offset Binary and Fibonaccian Search

Figure 4.17 compares the run-times of the lower bound offset binary search with equal split ratios in the left and right partitions (1:2/1:2 and 3:5/3:5), and with ratios mirrored in the left partition (2:1/1:2 and 5:3/3:5). As we can see, mirroring the split ratio depending on whether the left or right partition is chosen in a search step degrades performance. In the following, we only consider equal ratios in the left and right partitions.

We have tested the branch elimination and software prefetching optimization techniques on the lower bound offset binary search with a split ratio of 3:5 (Figure 4.18).

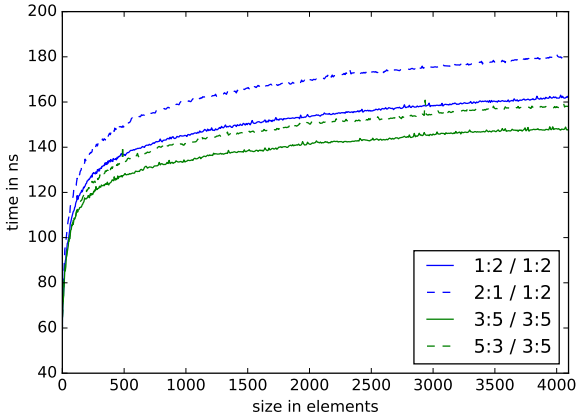

Figure 4.17: Lower Bound Offset Binary Search

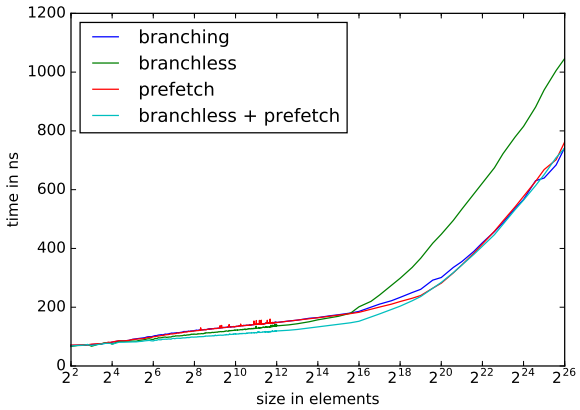

Figure 4.18: Lower Bound Offset Binary Search 1:2 Optimizations

The results for branch elimination are similar to the standard 1:1 binary search: The branchless implementation performs worst, while additional prefetching improves its run-time substantially. In contrast to the 1:1 binary search prefetching alone yields longer run-times. Also, the branchless prefetch variant looses its advantage over the branching implementations for array sizes over 2^{19}.

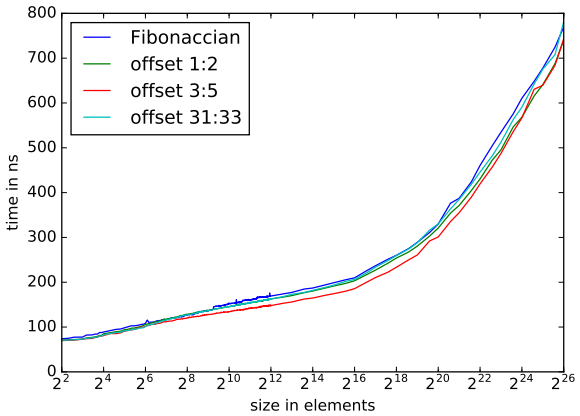

Figure 4.19: Comparison of (lower bound) Fibonaccian and Offset Binary Search.

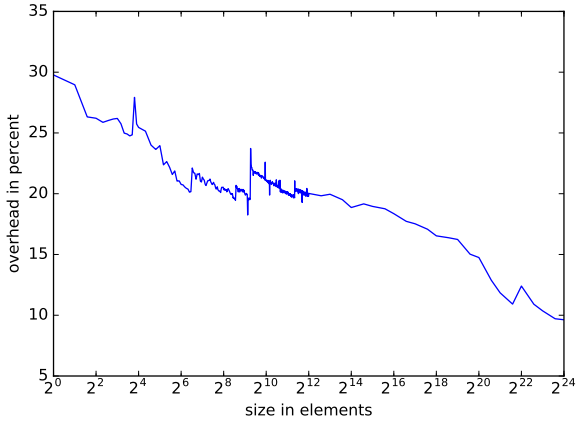

Figure 4.20: Percentage of the execution time of the lower bound Fibonaccian search spend determining the initial Fibonacci numbers

In Figure 4.19 we compare the run-times of the Fibonaccian search with the offset binary search. Generally, the Fibonaccian search is slower than the offset binary search. In part, because the Fibonaccian search needs to determine suitable Fibonacci numbers at the start of each invocation, but probably also, because our implementation mirrors the split ratio to have the separator elements nearer to their predecessors. The percentage of the execution time of `lowerBoundFibonaccianSearch` spent with finding the initial

triple of Fibonacci numbers ranges from 30% for smaller arrays to 10% for larger ones (Figure 4.20). From the split ratios we have tested for the offset binary search 3:5 is the best. Generally the behavior of the offset binary search approaches the behavior of the regular binary search for ratios closer to 1:1.

4.3.8 Cache Utilization

Figure 4.21 shows the run-times of the lower bound non-uniform, uniform and offset branchless binary search with explicit prefetching. On the left, the search keys were generated by Algorithm 1 and on the right by Algorithm 3. In both cases, the 1:1 binary search algorithms are faster than the offset binary search for smaller arrays, but the offset binary search overtakes them around an array size of 2^{17} to 2^{19} elements. Especially on the right, the non-offset searches have larger spikes, while the offset search stays on a smooth curve. The peaks in the non-offset binary search occur at powers of two. If no explicit prefetching is used the difference to the offset binary search is even greater.

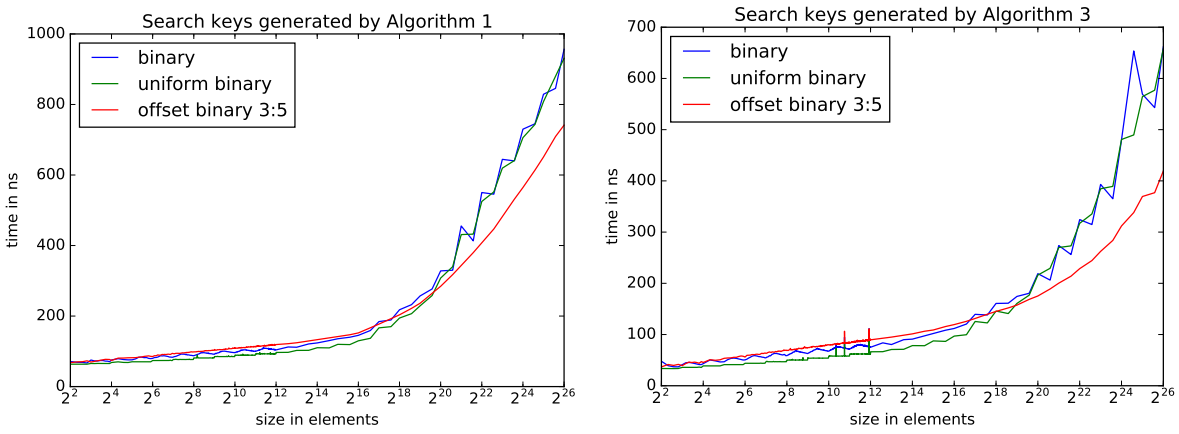

Figure 4.21: Comparison of non-offset (1:1) and offset binary search

The problems with the non-offset binary search functions on larger arrays, especially when the size is a power of two, is explained by Figure 4.22. 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. In the diagram on the left, the non-uniform binary search is plotted, but the graph is almost precisely identical for the uniform binary search. As we can see, the distribution 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. The offset binary search on the right has much more evenly distributed accesses leading to less evictions of separator keys that might be needed again soon. The corresponding plot of the Fibonaccian search would look similar to the offset binary search.

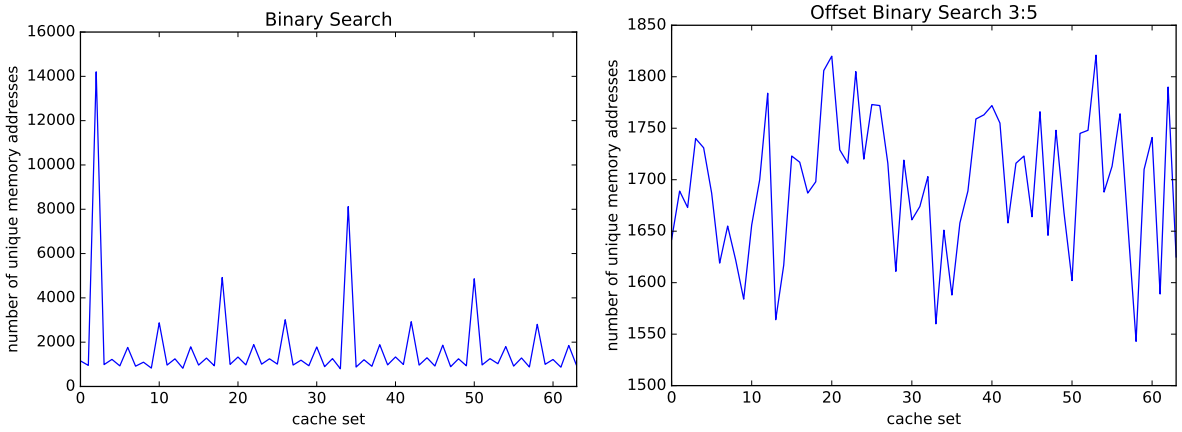

Figure 4.22: Number of unique memory accesses falling into each L1 cache set

The aliasing of many separator elements to the same cache sets comes from the formula used to index them, in our case *block 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 the most keys fall into the same cache set.

We recorded virtual addresses to generated the graphs in Figure 4.22. For the L1D 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 plots would look like in Figure 4.22. Otherwise, the location of pages in physical memory and the distribution of search keys will influence the aliasing effect, giving the non-offset binary search an unreliable performance.

The SSE4.1 instruction set extension offers a stream load instruction designed to avoid polluting the cache with loads known to not exhibit temporal locality. We tried to delay the cache aliasing effects on power of two array sizes by using this instruction in the last few iterations of the vectorized binary search. However, using streaming loads was always slower.

4.4 Summary

We have implemented the scalar exact match and lower bound binary search discussed in the background chapter and also introduced our version of a uniform binary search. Furthermore we implemented the vectorized binary search based on the idea described in the background chapter. In addition to the usual binary search, partitioning the search interval in two equally sized halves, we implemented an offset binary search with a variable split ratio and the Fibonaccian search algorithm.

Applying branch elimination to the scalar binary search proved to be detrimental to its performance, due to less hardware prefetching. However, combined with explicit

software prefetching we saw a performance increase of up to 40% for the scalar non-uniform binary search and of up to 60% for the scalar uniform binary search. Loop unrolling proved to be useless for the binary search, since the loop overhead already is low. Finally, vectorizing the binary search by loading whole SIMD words of separators did also not improve the run-time of binary searching.

We evaluated the performance difference between exact match binary search algorithms allowing an early termination as soon as the search key is found and algorithms delaying the equality test until the end of the search loop. We found, that moving the equality test to the end of the loop never significantly increases the run-time and often brings a positive effect.

While the non-offset binary search makes optimal use of the information gained with each comparison by halving the search space, we found that offsetting the location the separator keys are chosen from can significantly improve performance for larger arrays, because of issues with cache aliasing. The offsetting can be achieved by changing the split ratio from 1:1 to something asymmetric, either by simply modifying the calculation of the midpoint in the usual binary search or by using the Fibonaccian search using the Fibonacci numbers to achieve a split ratio approximating the golden ratio.

5. k-ary Search

In the last chapter, we treated dichotomic or binary search techniques choosing from two alternatives in each search step. The exact match search algorithms added what can be considered a third option of terminating the search early. In this chapter we will examine search algorithms choosing between an arbitrary fixed number k of possibilities each iteration.

5.1 Implementation

In this section, we present implementations of the k-ary search for arbitrary k. We have also developed a uniform k-ary search with simplified index computations. This algorithm is the basis for our implementation of a vectorized k-ary search. Additionally we have implemented search functions operating on the linearized complete k-ary search trees discussed in the background chapter (Section 2.2.4).

5.1.1 Scalar k-ary Search

We have implemented a lower bound k-ary search in Listing 5.1, where `k` is a template parameter. The function uses the variables `segmentLeft` and `segmentRight` to subdivide the search range [`left`, `right`) in `k` segments. `segmentLeft` holds the start index of the current segment, and `segmentRight` is the index of the current separator. The initialization in line 8 selects the first separator element. The loop in lines 9 to 18 iterates over all segments. Line 14 tests whether the search key falls in the current segment, and if so exits the loop. After the for-loop is exited [*segmentLeft*, *segmentRight*) is the new search range. The search terminates when the search range becomes empty.

An exact match variant is created by replacing line 14 with the commented out if-else construct in lines 11 to 13 and uncommenting line 22. Lines 11 to 13 check the separator elements for equality with the search key and line 22 reports an unsuccessful search if the search interval becomes empty.

Listing 5.1: Lower Bound k-ary Search

```
1  template <typename T, unsigned int k = 3>
2  IndexType lowerBound_kArySearch(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType left = 0, right = size; // left inclusive, right exclusive
5    while (left < right) {
6      // divide [left, right) in k segments
7      IndexType segmentLeft = left;
8      IndexType segmentRight = left + (right - left) / k;
9      for (IndexType i = 2; i <= k; ++i) {
10       // for exact match:
11       // if (searchKey == keys[segmentRight])
12       //     return segmentRight;
13       // else if (searchKey < keys[segmentRight]) break;
14       if (searchKey <= keys[segmentRight]) break;
15       // advance to the next segment
16       segmentLeft = segmentRight + 1;
17       segmentRight = left + (i * (right - left)) / k;
18     }
19     left = segmentLeft;
20     right = segmentRight;
21   }
22   // return size; // for exact match
23   return left; // left == right
24 }
```

5.1.2 Scalar Uniform k-ary Search

We have constructed a uniform k-ary search analogously to the uniform binary search in Section 4.1.2. For our discussion of the search function we again first assume `size` to correspondent to a perfect k-ary search tree, i.e. to be $k^n - 1$ with an integer n. The algorithm first needs to determine the height of the tree `treeHeight`, or in other words the number of iterations needed in the search loop (lines 21 to 31). This is done with the formula $\lceil \log_k(\texttt{size} + 1) \rceil$, computed by `getTreeHeight`. `getTreeHeight` is listed in Listing 5.2. It searches the first h for that $k^h \leq \texttt{size}$ using a precalculated table via the `powConstBase` template function defined in Listing A.3.

Listing 5.2: Calculate the height of a complete k-ary search tree containing `size` keys.

```
1  template <IndexType k>
2  IndexType getTreeHeight(IndexType size) {
3    IndexType height = 0;
4    while (powConstBase<IndexType, k>(height) <= size) ++height;
5    return height;
6  }
```

Listing 5.3: Lower Bound Uniform k-ary Search

```
1  template <typename T, IndexType k = 3>
2  IndexType lowerBoundUniform_kArySearch(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType treeHeight = getTreeHeight<k>(size);
5    bool isPerfect = size == (powConstBase<IndexType, k>(treeHeight) - 1);
6    IndexType depth = 0, left = 0;
7    if (!isPerfect) {
8      IndexType idealPartitionSize =
9        powConstBase<IndexType, k>(treeHeight - 1) - 1;
10     IndexType remainder = size - idealPartitionSize;
11     for (IndexType i = 1; i < k - 1; ++i) {
12       IndexType separatorIndex = (i * remainder) / (k - 1);
13       if (searchKey <= keys[separatorIndex]) break;
14       left = separatorIndex + 1;
15     }
16     IndexType separatorIndex = size - idealPartitionSize - 1;
17     if (searchKey > keys[separatorIndex])
18       left = separatorIndex + 1;
19     depth++;
20   }
21   for (; depth < treeHeight; ++depth) {
22     IndexType step = 0;
23     IndexType offset = 0;
24     for (IndexType i = 0; i < k - 1; ++i) {
25       offset += powConstBase<IndexType, k>(treeHeight - depth - 1);
26       IndexType separatorIndex = left + offset - 1;
27       if (searchKey <= keys[separatorIndex]) break;
28       step = offset;
29     }
30     left += step;
31   }
32   return left;
33 }
```

In line 5 `isPerfect` is set to `true`, if `size` corresponds to a perfect k-ary search tree, i.e. $\texttt{size} = 2^{\texttt{treeHeight}} - 1$. For perfect array sizes, the algorithm directly continues with the main search loop in line 21. The current search interval is determined by the variables `left` and `depth`. `left` stores the offset of the current search range from the beginning of the array, and `depth` is the current level in the conceptual search tree counting the root as level 0. The inner for-loop iterates over the $k - 1$ separator elements. Remember, that the offset of the i-th separator element from the beginning of the current search range is $i \cdot k^{h-1} - 1$, if the current search range has a length of $k^h - 1$. The variable `offset` accumulates the term $i \cdot k^{h-1}$, where h is given by $\texttt{treeHeight} - \texttt{depth}$. It is then used in line 26 to calculate the index of the separator key. The inner for-loop is exited in line 27, if the search key lies to the left of the current separator. `step` then contains the starting offset relative to `left` of the corresponding partition. If no separator is greater than or equal to the search key, `offset` contains the

starting offset of the last partition at the end of the loop. In line 30, `left` is updated to point to the selected partition.

Handling imperfectly sized arrays

If `size` does not correspond to a perfect search tree, the first search step is handled by lines 8 to 19. The code first calculates how many keys would need to be removed from the array in order to make it correspond to a perfect search tree in the variable `remainder`. These elements are equally distributed over the first $k-1$ partitions, while the last partition gets the perfect size. This means, the combined length of the first $k-1$ partitions is less than the length of the last partition. The following iterations can then safely assume all partitions to have the perfect size of the last one, effectively letting the initial partitions overlap. It is important that the last partition is the perfectly sized one, so that no partition reaches beyond the end of the array.

An exact match search can be derived from Listing 5.3 by adding a test for equality with the search key to every key comparison and reporting an unsuccessful search if the algorithm reaches line 32.

5.1.3 Vectorized k-ary Search

The uniform k-ary search from the last section is the basis of our implementation of a vectorized k-ary search parallelly computing separator indices and parallelly comparing them to the search key. If the AVX2 instructions set is available, loading the keys is also done in parallel. In the following implementation, parallel AVX2 loads are enabled via the preprocessor constant `USE_AVX2`.

The first part of the implementation is shown in Listing 5.4. Since the type of the keys (`T`) and of the indices (`IndexType`) can differ, we also need two types for our SIMD vectors. These are `KeyVector` and `IndexVector`, respectively. Lines 5 to 17 select the correct types. Lines 6 to 12 are used if parallel AVX2 loads should be used, otherwise lines 14 to 16 are active. If AVX2 is enabled, we require both `T` and `IndexType` to be 4 or 8 bytes wide, since the `gather` family of instructions only supports these sizes. The `IndexVector` and `KeyVector` types are determined using the `kArySearchVectorTypes` template defined in Listing 5.5. By using either 128-bit or 256-bit vectors we have the same number of keys and indices in one 128/256-bit vector. Additionally we need to make sure that `IndexType` is signed, since our code uses signed SIMD multiplications. This is done by redefining `IndexType` in line 6. Of course, care must be taken to avoid overflowing the signed type.

If parallel loads are disabled, `IndexType` is derived from the type of the keys `T` using the template `kArySearchIndexType` defined in Listing 5.6. The derived type is signed and has the same width as `T`. This way, we have the same number of keys and indices in a 128-bit vector. This also limits the array size to $2^{8 \cdot \texttt{sizeof(T)}-1} - 1$ keys. Note that we require `IndexType` and therefore also the key type `T` to be at least 2 bytes wide.

Listing 5.4: Vectorized Lower Bound k-ary Search (1)

```
1  template <typename T>
2  IndexType lowerBoundUniform_kArySearchSIMD(
3    const T *keys, IndexType size, T searchKey) {
4    if (size == 0) return 0;
5  #ifdef USE_AVX2
6    using IndexType = typename std::make_signed<::IndexType>::type;
7    static_assert(sizeof(T) >= 4 && sizeof(IndexType) >= 4,
8      "Invalid types");
9    using IndexVector = typename kArySearchVectorTypes<sizeof(T),
10     sizeof(IndexType)>::IndexVector;
11   using KeyVector = typename kArySearchVectorTypes<sizeof(T),
12     sizeof(IndexType)>::KeyVector;
13 #else
14   using IndexType = typename kArySearchIndexType<T>::type;
15   static_assert(sizeof(IndexType) >= 2, "Invalid types");
16   using IndexVector = __m128i; using KeyVector = __m128i;
17 #endif
18
19   constexpr unsigned int KEYS_PER_WORD =
20     keys_per_simd_word<KeyVector, T>();
21   constexpr unsigned int k = KEYS_PER_WORD + 1;
22   const IndexVector one = _mm_set1<IndexVector, IndexType>(1);
23   const IndexVector laneFactors =
24     generateLaneFactors<IndexVector, IndexType>();
25   const KeyVector vecSearchKey = adjustForSignedComparison<T>(
26     _mm_set1<KeyVector, T>(searchKey));
27
28   // temporary storage for sequential loads
29   IndexType indicesArray[KEYS_PER_WORD] alignas(sizeof(IndexVector));
30   #ifndef USE_AVX2
31   T separatorsArray[KEYS_PER_WORD] alignas(sizeof(KeyVector));
32   #endif
33
34   IndexType treeHeight = getTreeHeight<k>((::IndexType)size);
35   bool isPerfect = size == (powConstBase<IndexType, k>(treeHeight) - 1);
36   IndexType left = 0, depth = 0;
37
38   // handle imperfect array size, see Listing 5.7
39   // main search loop, see Listing 5.8
40 }
```

In lines 19 to 26 the constants `KEYS_PER_WORD` and `k` are defined and the search key is loaded into a SIMD register like in our other SIMD implementations. Additionally a vector containing a single one in each lane is prepared in line 22. The variable `laneFactors` is loaded with the one based index of each lane for a width given by `IndexType`. For example, if `IndexType` is 32-bit wide, `laneFactors` would be (1, 2, 3, 4) (starting with the least significant byte). The template function `generateLaneFactors` is defined in the appendix (Listing A.13). Some storage for sequentially calculating

indices is allocated in line 29. If sequential loads are used, additional space to copy the separator elements to is allocated in line 31. Lines 34 to 36 are the same as for the scalar algorithm in Listing 5.3.

Listing 5.5: kArySearchVectorTypes

```
1 template <size_t keySize, size_t indexSize> struct kArySearchVectorTypes;
2 template <> struct kArySearchVectorTypes<4, 4>
3 { using IndexVector = __m256i; using KeyVector = __m256i; };
4 template <> struct kArySearchVectorTypes<8, 4>
5 { using IndexVector = __m128i; using KeyVector = __m256i; };
6 template <> struct kArySearchVectorTypes<4, 8>
7 { using IndexVector = __m256i; using KeyVector = __m128i; };
8 template <> struct kArySearchVectorTypes<8, 8>
9 { using IndexVector = __m256i; using KeyVector = __m256i; };
```

Listing 5.6: kArySearchIndexType

```
1 template <typename T> struct kArySearchIndexType
2 { using type = typename std::make_signed<T>::type; };
3 template <> struct kArySearchIndexType<float> { using type = int32_t; };
4 template <> struct kArySearchIndexType<double> { using type = int64_t; };
```

The code in Listing 5.7 is executed, if `size` does not correspond to a perfect k-ary search tree. It works the same as in the scalar search. The most important difference is that the indices of the separator elements are calculated up front in lines 2 to 7. If enabled, an AVX2 `gather` instruction is used to load all separators in parallel (lines 12 and 13). Since the indices are stored in an array, they first have to be loaded in lines 10 and 11. We use the template function `_mm_gather` defined in the appendix (Listing A.12) to select the appropriate intrinsic for the data types passed to it. If `gather` is not used, the separators are copied to the previously allocated array `separatorsArray` and then loaded into a SIMD register (line 15 to 18). Note that the load loop is unrolled by the compiler. The comparison with the search key is done in lines 22 and 23. We then use the `createMask` and `countPositiveResults` functions to evaluate the result. If the search key was smaller than or equal to the smallest separator, the variable `next` is zero. In this case, the search range still starts with the first element of the array, and `left` is not updated. If the search key falls into any other partition, its starting index is the index of the separator to its left plus one. We have all the separator indices in the array `indicesArray`, so we can retrieve them by indexing the array with $\texttt{next} - 1$. So if `next` is one, the new search range begins to the right of the first separator and so on.

Listing 5.7: Vectorized Lower Bound k-ary Search (2)

```
1  if (!isPerfect) {
2    IndexType idealPartitionSize =
3      powConstBase<IndexType, k>(treeHeight - 1) - 1;
4    IndexType remainder = size - idealPartitionSize;
5    for (IndexType i = 1; i < k - 1; ++i)
6      indicesArray[i-1] = (i * remainder) / (k - 1);
7    indicesArray[k-2] = size - idealPartitionSize - 1;
8
9  #ifdef USE_AVX2 // load the separator elements (parallel)
10   IndexVector indices = loadVector(
11     reinterpret_cast<const IndexVector*>(indicesArray));
12   KeyVector separators =
13     _mm_gather<KeyVector, IndexVector, T>(keys, indices);
14 #else // load the separator elements (sequential)
15   for (IndexType i = 0; i < KEYS_PER_WORD; ++i)
16     separatorsArray[i] = keys[indicesArray[i]];
17   KeyVector separators = loadVector(
18     reinterpret_cast<const KeyVector*>(separatorsArray));
19 #endif
20
21   // compare and determine next partition (parallel)
22   KeyVector compResult = _mm_cmpgt<T>(vecSearchKey,
23     adjustForSignedComparison<T>(separators));
24   int mask = createMask<T>(compResult);
25   int next = countPositiveResults<T>(mask);
26   if (next != 0) left = indicesArray[next-1] + 1;
27
28   depth++;
29 }
```

The main search loop is listed in Listing 5.8. Like in the scalar uniform k-ary search, this part of the function can safely assume a perfect array size. In lines 2 to 9 the indices of the separator elements are computed. First all lanes of `vecDist` are set to $2^{\texttt{treeHeight}-\texttt{depth}-1}$, the distance between the separator elements in the current tree level. Then `laneFactors` is used to compute $i \cdot \texttt{dist} - 1$ in the i-th lane of `indices`. Finally, `left` is added to all lanes of `indices` to get a vector of the offsets from the beginning of the array. If parallel loads are enabled this vector is used directly to load the separator keys in lines 11 and 12. Otherwise, the indices are first stored in `indicesArray`. Then the separators are loaded sequentially like in Listing 5.7. The calculation of `next` is also identical to Listing 5.7. The search is terminated in line 28, if the leafs of the conceptual search tree are reached. The lower bound is then one of the last separators or, if the search key is greater than all of the last separators, to the right of them. Therefore the lower bound is simply $\texttt{left} + \texttt{next}$. If the search not yet terminates, the starting index of the next search range is given by $\texttt{left} + \texttt{next} \cdot \texttt{dist}$.

Listing 5.8: Vectorized Lower Bound k-ary Search (3)

```
1  for (;; ++depth) {
2    // calculate separator indices (parallel)
3    IndexType dist = powConstBase<IndexType, k>(treeHeight - depth - 1);
4    IndexVector vecDist = _mm_set1<IndexVector, IndexType>(dist);
5    IndexVector indices = _mm_sub_epi<IndexType>(
6      _mm_mullo_epi<IndexType>(laneFactors, vecDist), one);
7    indices = _mm_add_epi<IndexType>(
8      _mm_set1<IndexVector, IndexType>(left), indices);
9
10 #ifdef USE_AVX2 // load the separator elements (parallel)
11   KeyVector separators =
12     _mm_gather<KeyVector, IndexVector, T>(keys, indices);
13 #else // load the separator elements (sequential)
14   storeVector(reinterpret_cast<IndexVector*>(indicesArray), indices);
15   for (IndexType i = 0; i < KEYS_PER_WORD; ++i)
16     separatorsArray[i] = keys[indicesArray[i]];
17   KeyVector separators = loadVector(
18     reinterpret_cast<const KeyVector*>(separatorsArray));
19 #endif
20
21   // compare and determine next partition (parallel)
22   KeyVector compResult = _mm_cmpgt<T>(vecSearchKey,
23     adjustForSignedComparison<T>(separators));
24   int mask = createMask<T>(compResult);
25   int next = countPositiveResults<T>(mask);
26   if (depth == (treeHeight - 1))
27     return (::IndexType)(left + next);
28   left += next * dist;
29 }
```

To create a vectorized exact match k-ary search, the equality test in Listing 5.9 has to be added before the key comparisons in line 22 of Listing 5.7 and line 22 of Listing 5.8. Additionally, lines 7 to 9 of Listing 5.9 have to be uncommented in the main search loop to store the indices if AVX2 is used. Moreover, the function should return `size` in line 29 of Listing 5.8 to indicate an unsuccessful search. The code in Listing 5.9 works the same as in the vectorized exact match binary search (see Listing 4.5 on page 42), with the difference that the index of the found key comes from the already calculated indices vector.

5.1.4 Linearized k-ary Search Trees

We have also implemented the linearized k-ary search trees discussed in Section 2.2.4. In Listing 5.10 we present a lower bound search version returning an iterator object whose implementation is listed in Listing 5.11.

Listing 5.9: Key equality test for vectorized exact match uniform k-ary search

```
KeyVector compResult = _mm_cmpeq<T>(vecSearchKey,
  adjustForSignedComparison<T>(separators));
int mask = createMask<T>(compResult);
unsigned long i = 0;
if (getFirstPositiveResult<T>(&i, mask)) {
  // search key is equal to one of the separators
  //#ifdef USE_AVX2
  //    storeVector(reinterpret_cast<IndexVector*>(indicesArray), indices);
  //#endif
  return (::IndexType)indicesArray[i];
}
```

Listing 5.10: Searching in Linearized k-ary Search Trees

```
template <typename T> LinearizedTreeIterator<T,
  keys_per_simd_word<Vector, T>() + 1, MAX_TREE_HEIGHT>
lowerBound_kArySearchSIMDLinearizedTree(
  const T *keys, unsigned int size, T searchKey) {
  assert((intptr_t)keys % sizeof(Vector) == 0);
  constexpr IndexType KEYS_PER_WORD = keys_per_simd_word<Vector, T>();
  constexpr IndexType k = KEYS_PER_WORD + 1;
  assert(size % KEYS_PER_WORD == 0);
  const Vector vecSearchKey = adjustForSignedComparison<T>(
    _mm_set1<Vector, T>(searchKey));

  std::array<unsigned short, MAX_TREE_HEIGHT> branches;
  std::array<IndexType, MAX_TREE_HEIGHT> offsets;

  IndexType left = 0, next = 0;
  int depth = 0;
  while (next < size) {
    // load the separator elements
    left = next;
    Vector separators = loadVector(
      reinterpret_cast<const Vector*>(keys + left));

    // compare with search key
    Vector compResult = _mm_cmpgt<T>(vecSearchKey,
      adjustForSignedComparison<T>(separators));
    int mask = createMask<T>(compResult);
    int branch = countPositiveResults<T>(mask);

    // go to the next node, 0 <= branch < k
    next = (left + 1) * k + branch * (k - 1) - 1;

    // keep track of the path taken
    branches[depth] = branch;
    offsets[depth] = left + branch;
    depth++;
  }

  return LinearizedTreeIterator<T, k, MAX_TREE_HEIGHT>(keys,
    size, depth, std::move(branches), std::move(offsets));
}
```

Listing 5.10 closely follows the pseudocode given in Algorithm 9 on page 22 with some additional bookkeeping to construct an iterator allowing to efficiently visit the keys following the lower bound in sorted order. The input array to the function must be a linearized complete search tree. An array in sorted order is converted to such a tree by applying the permutation in Equation 2.1 on page 21.

The search loop in lines 17 to 36 runs over the levels of the search tree. In each iteration `left` contains the array index of the first key in the current tree node. Additionally, there is the variable `next` set to the index of the node to visit in the next tree level. `left` is updated with this value at the beginning of each iteration in line 19, but only after the loop condition `next` < `size` has been verified. Remember, that in a complete k-ary search tree some nodes above the deepest level of the tree have less than k children, for example the nodes with the keys 11 and 12 in Figure 2.7 on page 21. If the search is in such a node, the index of a child node computed in line 30 can lie beyond the end of the array, meaning that there are no further nodes to check. The determination of the next tree node to visit has already been discussed in Section 2.2.4.1.

Listing 5.11: LinearizedTreeIterator

```
1  template <typename T, unsigned int k, unsigned int MAX_TREE_HEIGHT = 16>
2  class LinearizedTreeIterator {
3  public:
4    LinearizedTreeIterator(
5      const T *_keys, IndexType _size,
6      int _treeHeight,
7      std::array<unsigned short, MAX_TREE_HEIGHT> _branches,
8      std::array<IndexType, MAX_TREE_HEIGHT> _offsets
9    ) : /* initialize members */ {
10     if (treeHeight > 0 && offsets[treeHeight-1] == size) treeHeight--;
11     traverseUp();
12   }
13
14   const T& operator*() const { return keys[getIndex()]; }
15   IndexType getIndex() const { return offsets[depth]; }
16   LinearizedTreeIterator& operator++();
17   bool isDereferenceable() const { return depth >= 0; }
18
19 private:
20   void traverseUp();
21   const T *keys;
22   const IndexType size;
23   int treeHeight;
24   int depth;
25   std::array<unsigned short, MAX_TREE_HEIGHT> branches;
26   std::array<IndexType, MAX_TREE_HEIGHT> offsets;
27 };
```

To construct the iterator in line 38 we need to store the index of the child node selected and the offset from the beginning of the array to the next separator to test in the current

node for each level of the tree, like discussed in Section 2.2.4.2. To this end we use the arrays `branches` and `offsets`.

In Listing 5.11 we show the relevant parts of the implementation of an iterator based on Algorithm 10 on page 23. The constructor in lines 4 to 12 initializes the member variables with its parameters. Note that `treeHeight` gets the value of `depth` from the search function. For perfect linearized search trees, this value is the height of the tree. If the search tree is only complete, `treeHeight` can get two different values, depending on whether the lower bound is smaller or greater than the fringe element. If it is smaller than the fringe element, `treeHeight` is the maximal height of the tree, otherwise one less. For example if we would search for 7 in the tree depicted in Figure 2.7, we would get a height of 3, and if we search for 14, a height of 2. To ensure the correct function of the iterator, `treeHeight` must be decremented if the iterator reaches the fringe element. This is done in `operator++` (line 5 of Listing 5.12). Special care must be taken if the lower bound is the fringe element. In this case `treeHeight` is initialized to the higher value and immediately needs to be decremented in the constructor (line 10 of Listing 5.11). Remember from Section 2.2.4, that `offsets[depth]` gives the index of the current key and that the fringe element is always the last key in the array containing the linearized tree. So the expression `offsets[depth] == size - 1` is `true`, if the fringe entry has been reached.

Listing 5.12: LinearizedTreeIterator::operator++()

```
1  inline LinearizedTreeIterator&
2  LinearizedTreeIterator::operator++() {
3    if (depth >= 0) {
4      if (offsets[depth] == size - 1) {
5        treeHeight--;
6      } else {
7        branches[depth] += 1;
8        offsets[depth] += 1;
9      }
10     traverseUp();
11   }
12   return *this;
13 }
```

Furthermore, the constructor calls `traverseUp`, which is listed in Listing 5.13. `traverseUp` implements lines 1 to 5 of Algorithm 10. It initialized `depth` to the leaf level in the current part of the tree (`treeHeight` $-$ 1) and then traverses up, until a node is found, where the search has not yet reached the last child using the while-loop. When `traverseUp` is first called by the constructor this loop is only entered, if the lower bound lies to the right (in sorted order) of the last tree node examined. This is always the case when the lower bound is not in one of the leafs. In other words: If the search stops between two leafs, the lower bound is in one of the inner nodes. We have already encountered this case in the vectorized binary search, where it required special handling of the last iteration (see Section 4.1.3).

Listing 5.13: LinearizedTreeIterator::traverseUp()

```
1 inline void LinearizedTreeIterator::traverseUp() {
2   depth = treeHeight - 1; // reset depth
3   while (depth >= 0 && branches[depth] == (k - 1)) {
4     branches[depth] = 0;
5     depth--;
6   }
7   if (depth >= 0 && offsets[depth] >= size)
8     depth = -1;
9 }
```

The last issue is to detect when there are no more keys left in sorted order. First assume a perfect search tree. Then the while-loop in `traverseUp` will run until `depth` is -1, if the iterator is incremented after the last key has been reached. The method `isDereferenceable` can then report the iterator to be no longer dereferenceable if `depth` is negative. If the tree is not perfect, we detect this condition with the test `offsets[depth] >= size`, i.e. the next key would lie outside of the array, and set `depth` to -1 manually (lines 7 and 8 of Listing 5.13). This is similar to the loop condition in the search function.

Implementing an exact match variant of Listing 5.10 is simply a matter of adding a test for equality in the search loop and reporting an unsuccessful search, if the algorithm leaves the loop without having found the search key. The algorithm is shown in pseudocode in the background chapter (Algorithm 9 on page 22).

5.2 Optimizations

We have applied two manual optimizations to the k-ary search. First, we have manually unrolled the inner loops of Listing 5.1 and Listing 5.3. The inner loops of Listing 5.15 and Listing 5.16 were automatically unrolled by the compiler. Second, we eliminated branches in the inner loops in the same way we did for the sequential search. We are not considering unrolling the main search loop, because this optimization technique was already detrimental in the binary search. Furthermore, we are not adding software prefetching to the k-ary search, since even the ternary search would already prefetch six potential separator elements looking one iterations ahead. Even prefetching just four keys each iteration slowed down the uniform binary search considerably.

Branch Elimination

Determining the next partition in each step of a k-ary search is in itself a search problem on $k-1$ separator keys. The scalar k-ary search algorithms we presented search sequentially through the separators, breaking out of the innermost search loop as soon as the correct partition has been determined. In Chapter 3, we have seen that it can be beneficial to instead always search through all keys and thus enable replacing conditional

jumps with conditional move instructions. In Listing 5.14 we show the (non-uniform) ternary search using a branchless sequential search on the separator elements. Note that the sequential search loop has been unrolled in this listing. We have implemented versions for $3 < k \leq 9$ analogously to Listing 5.14.

Listing 5.14: Branchless Lower Bound Ternary Search

```
1  template <typename T>
2  IndexType lowerBound_kArySearch3Branchless(
3    const T *keys, IndexType size, T searchKey) {
4    IndexType left = 0, right = size;
5    while (left < right) {
6      IndexType sepIndex1 = left + (1 * (right - left)) / 3;
7      IndexType sepIndex2 = left + (2 * (right - left)) / 3;
8      IndexType nextLeft = left;
9      IndexType nextRight = sepIndex1;
10     nextLeft = searchKey > keys[sepIndex1] ? sepIndex1 + 1 : nextLeft;
11     nextRight = searchKey > keys[sepIndex1] ? sepIndex2 : nextRight;
12     nextLeft = searchKey > keys[sepIndex2] ? sepIndex2 + 1 : nextLeft;
13     nextRight = searchKey > keys[sepIndex2] ? right : nextRight;
14     left = nextLeft;
15     right = nextRight;
16   }
17   return left; // left == right
18 }
```

Using the branchless sequential search is also possible in the uniform variant of the k-ary search. In Listing 5.15 and Listing 5.16 we show the updated inner search loops.

Listing 5.15: Branchless version of the search loop in lines 11 to 18 of Listing 5.3

```
1 for (IndexType i = 1; i < k - 1; ++i) {
2   IndexType separatorIndex = (i * remainder) / (k - 1);
3   left = searchKey > keys[separatorIndex] ? separatorIndex + 1 : left;
4 }
5 IndexType separatorIndex = size - idealPartitionSize - 1;
6 left = searchKey > keys[separatorIndex] ? separatorIndex + 1 : left;
```

Listing 5.16: Branchless version of the search loop in lines 22 to 30 of Listing 5.3

```
1 const IndexType dist = powConstBase<IndexType, k>(treeHeight - depth - 1);
2 IndexType step = 0;
3 IndexType separatorIndex = left - 1;
4 for (IndexType i = 0; i < k - 1; ++i) {
5   separatorIndex += dist;
6   step = searchKey > keys[separatorIndex] ? step + dist : step;
7 }
8 left += step;
```

The vectorized k-ary search fits all of the $k - 1$ separator keys into one SIMD word to compare them to the search key in parallel. Therefore it unconditionally loads all separator keys like the branchless k-ary search implementations in this section and does not require an inner search loop with conditional jumps.

We considered another modification to the branchless scalar k-ary search. Instead of always iterating over all separator keys like the vectorized algorithm, the search functions could stop advancing to the next separator as soon as the correct partition was determined. For example if $k = 5$ and the search key falls into the second partition, we would stay at the second separator element and repeatedly test this key instead of loading the other separators. This method theoretically has the advantage of avoiding conditional jumps, while still eliminating unnecessary loads as the branching implementation. However, our implementations of this idea were significantly slower than the branchless implementations we show in this section.

5.3 Evaluation

We used the same evaluation setup and the same search key generation algorithms as in the evaluation of the binary search to evaluate the k-ary search. See Section 3.3.1 and Section 4.3.1 for a description.

Note that we have manually unrolled the inner loops of Listing 5.1 and Listing 5.3. The uniform k-ary search with $k \leq 5$ is sped up by 1% to 3% by unrolling the inner loop, for larger k it is about 1% slower. In case of the non-uniform k-ary search unrolling improves the performance of the ternary search by about 5%, but for larger k and especially on larger arrays, unrolling can also reduce the search speed by about the same 5%. Since the inner loops in Listing 5.15 and Listing 5.16 were automatically unrolled by the compiler, we will only use the unrolled implementations in this chapter. This way, we maintain comparability between the branching and branchless implementations.

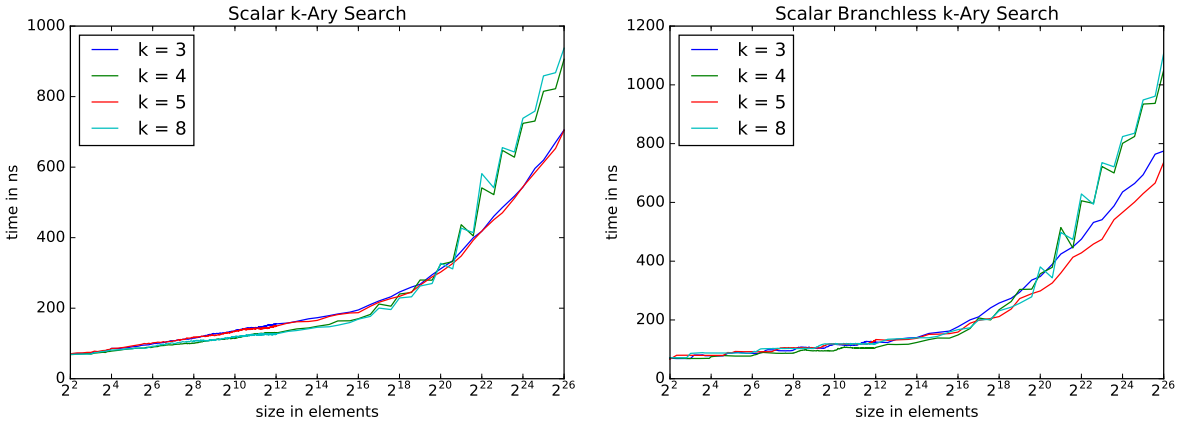

Figure 5.1: Branching and branchless non-uniform k-ary search

5.3.1 Scalar k-ary Search

We show the run-times of the scalar k-ary search for $k = 3, 4, 5$ and 8 in Figure 5.1. For smaller arrays with less than 2^{17} keys, setting k to a power of two results in the best performance, probably mostly due to the simpler index computations. The division by

k in Listing 5.1 is compiled to a single bit shift if k is a power of two. For larger arrays the cache aliasing effects discussed in Section 4.3.8 apply, so a power of two is a bad choice for k. In our experiments with $3 \leq k \leq 9$, we found 3 and 5 to give good reliable performance.

5.3.2 Scalar Uniform k-ary Search

In the uniform k-ary search powers of two have no advantage anymore. Since the binary search is theoretically optimal, we expect small k to yield the best performance. Interestingly, $k = 9$ is relatively fast in the branching uniform search on the left side of Figure 5.2, but slow in the branchless search on the right side of Figure 5.2. This is probably cause by the branchless variant always searching through all separator elements, whereas the branching variant enables the case $k = 9$ to gain some efficiency by exiting the inner search loop earlier. However, since the branchless implementation generally is faster, smaller k are more useful. Like for the non-uniform search, $k = 3$ or $k = 5$ are the best choices.

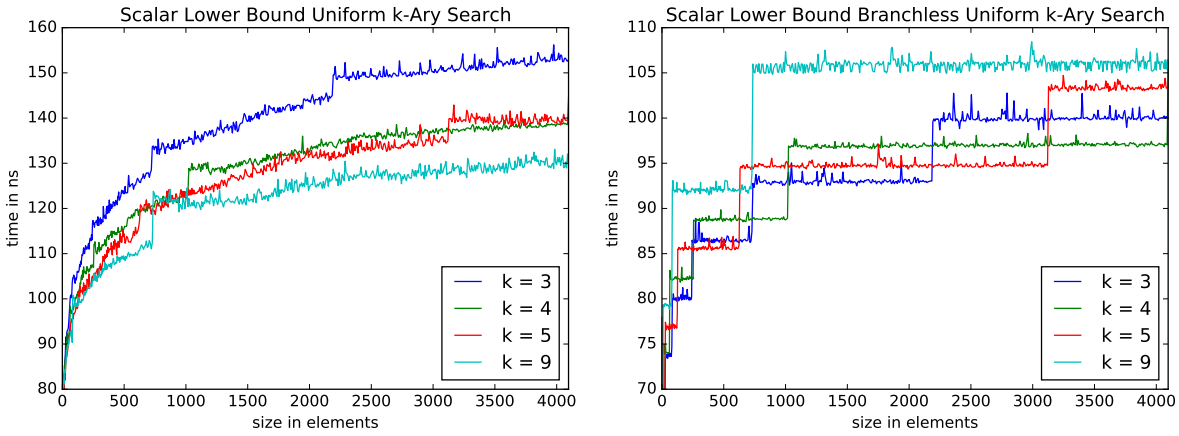

Figure 5.2: Branching and branchless uniform k-ary search

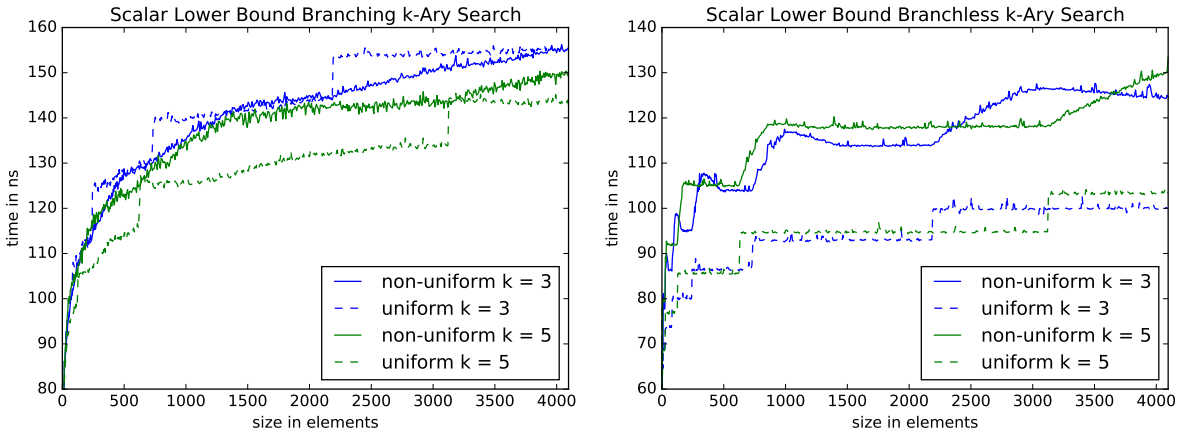

Figure 5.3: Comparison of the non-uniform and uniform k-ary search

In Figure 5.3 we compare the uniform and non-uniform k-ary search. The branching uniform k-ary search tends to be faster than the non-uniform alternative if $k = 3$ and is always faster if $k = 5$. For the branchless k-ary search, the uniform variant is clearly always preferable.

5.3.3 Branch Elimination

The branchless k-ary search is faster then the branching implementation for arrays with up to 2^{17} to 2^{19} keys, depending on k. For larger arrays, the branchless implementations fall behind. This is similar to the binary search in Section 4.3.2, albeit less pronounced. As with the binary search, the branching k-ary search has much more hardware prefetcher activity and produces more LLC misses than the branchless implementation when the array size exceeds about 2^{17} keys. There are more hardware prefetcher requests in the branchless search the larger k is, since the prefetcher can detect loads with a constant stride [Int16a]. However, this does not yield a better run-time for $k > 5$.

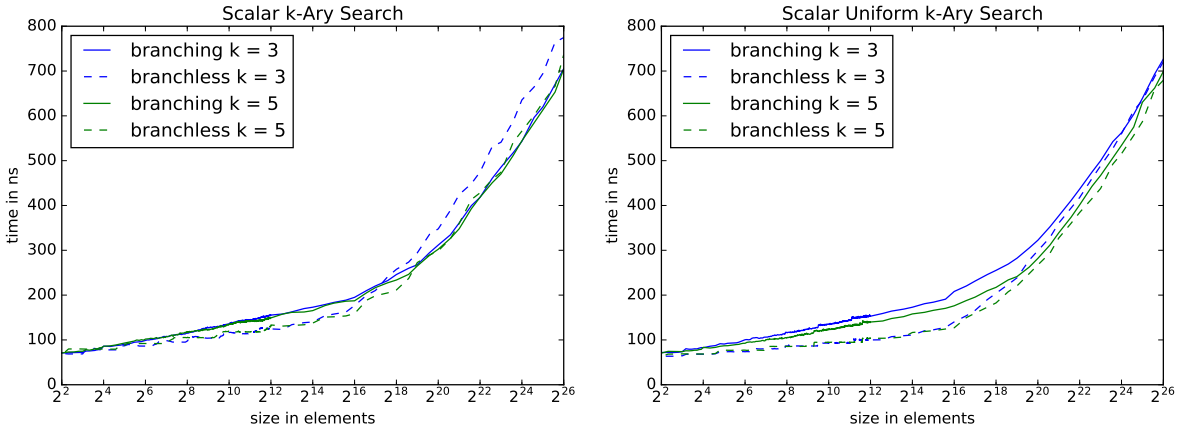

Figure 5.4: Branching and branchless scalar lower bound k-ary search

The right side of Figure 5.4 shows the branching and branchless uniform k-ary search. It behaves similar to the non-uniform search, but the points where the branching implementations become faster are at about four times as many keys, probably because the uniform search makes better use of the caches, since there are less possible separator elements at each search depth.

5.3.4 Vectorized k-ary Search

Figure 5.5 compares the vectorized k-ary search using SSE with the corresponding scalar branchless uniform k-ary search. The diagram shows the run-times on arrays with the usual 32-bit keys (int32), but also with 16-bit (int16) and 64-bit keys (int64). The k of the scalar search is chosen to match the vectorized search with 128-bit SSE registers. Note that the vectorized search is limited to arrays of less than 2^{15} elements when 16-bit

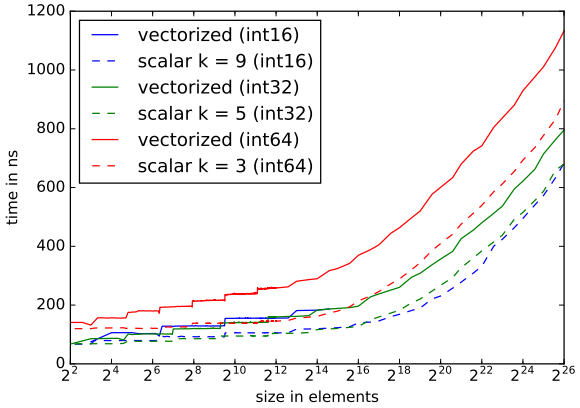

Figure 5.5: Comparison of the vectorized (SSE4.1) k-ary search and the scalar branchless uniform k-ary search

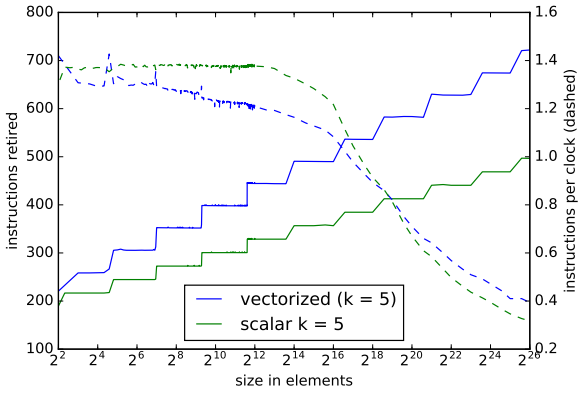

Figure 5.6: Instructions retired and instructions per clock of the vectorized (SSE4.1) and scalar k-ary search with 32-bit keys

keys are used. As we can see, the SIMD search is slower than the scalar implementation. Figure 5.6 shows the average number of retired instructions and the average number of instructions per clock for the scalar and vectorized k-ary search with 32-bit search keys. The vectorized search executes 30 to 230 more instructions per search run and can not achieve a sufficiently higher instruction throughput than the scalar implementation, explaining its longer run-time.

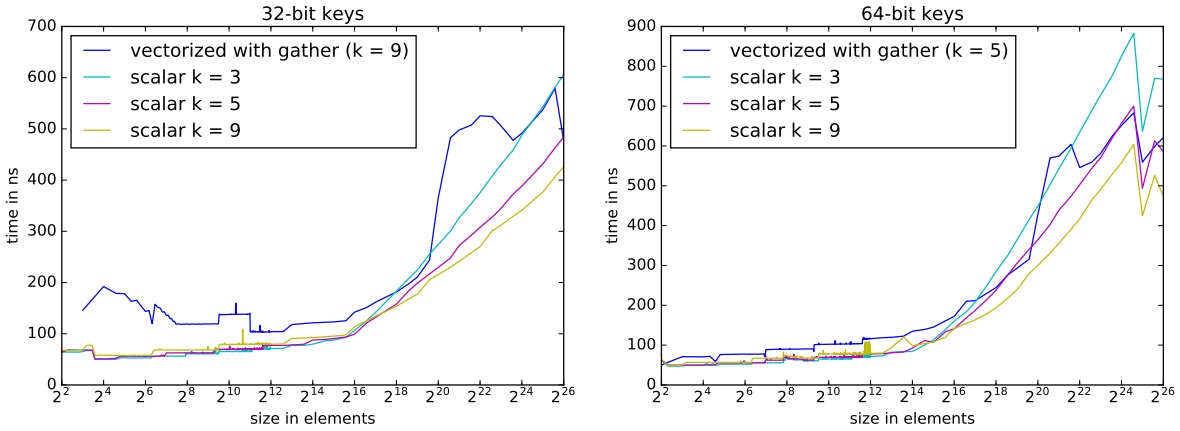

Figure 5.7: Comparison of the vectorized (AVX2) k-ary search and the scalar branchless uniform k-ary search

AVX2 Gather Instructions

Remember, that the SSE implementation of k-ary searching uses scalar loads to gather the separator elements, writes them to a temporary location and then loads them into an SSE register. AVX2 contains instructions to directly load values from non-continuous memory locations. Since our usual evaluation system does not support AVX2, we evaluated the AVX2 implementation on an Intel Xeon E5-2630v3 processor (32 KiB

L1D and 256 KiB L2 cache per core; 20 MiB shared L3 cache). The source code was compiled with the GNU C++ compiler version 5.1.1. The AVX2 implementation performs better relative to the scalar k-ary search algorithms, especially when 64-bit keys are used (Figure 5.7). Nevertheless, the scalar 9-ary search is always faster than the AVX2 implementation, except for an array with just 4 64-bit keys.

5.3.5 Exact Match

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 4.3.6. For this evaluation, we tested $k = 3$ and 5, because these give the best performance. The search keys were generated by Algorithm 2 (see Section 4.3.1), so that all searches are successful.

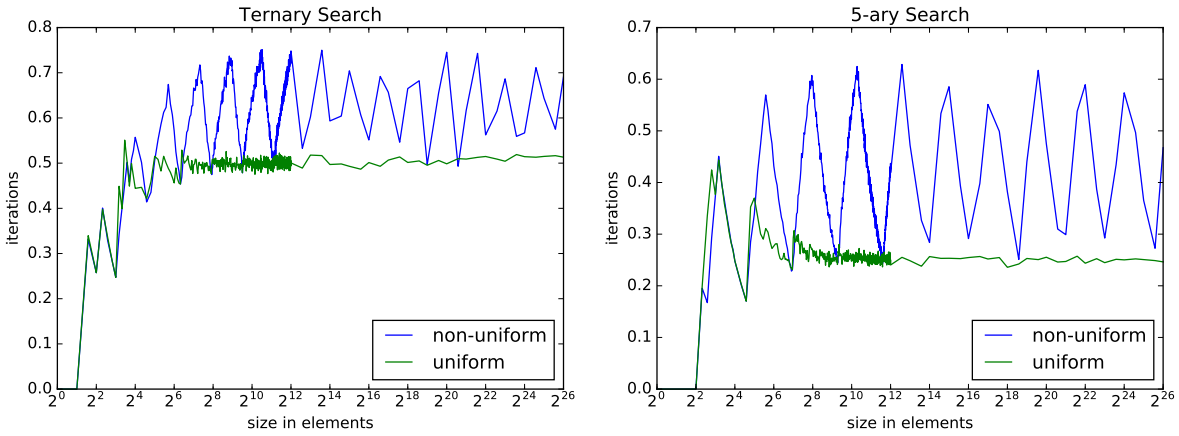

Figure 5.8: Average number of additional iterations needed by the lower bound based exact match k-ary search

Remember from Section 4.3.6, 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 created by a k-ary search has up to k times more leafs 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 is 3 and 0.25 iterations difference if k is 5 (see Figure 5.8). The differences for the non-uniform k-ary search are slightly larger with about 0.6 and 0.4, respectively.

Despite the smaller difference in iterations, the direct exact match k-ary search is more useful to improve the run-time than in the binary search. The branching direct exact match implementation is on average about 4% to 7% faster than the branching lower bound based implementation (see Figure 5.9). The direct exact match implementation is also faster for the branchless non-uniform k-ary search. However, the lower bound based variant is on average about 12% to 15% faster for the branchless uniform search on

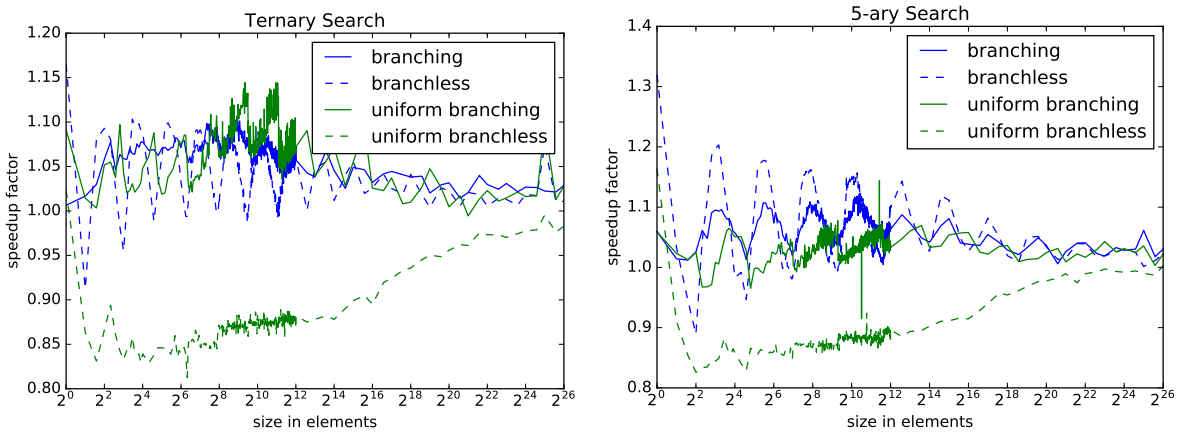

Figure 5.9: Relative speedup of the direct exact match k-ary search compared to the lower bound based search

arrays with less then about 2^{17} keys. For larger arrays the difference starts to approach zero.

5.3.6 Linearized k-ary Search Trees

The left side of Figure 5.10 shows the run-time of the lower bound k-ary search on a linearized search tree. Since we were using SSE and 32-bit keys, k is 5. For comparison, a search on linearized binary trees, the scalar uniform branchless k-ary search and an optimized binary search (uniform, branchless and using prefetching) are plotted. As we can see, the 5-ary and binary search algorithms are faster than searching on a linearized tree for arrays with up to about 2^{20} keys. For larger arrays, directly searching on the sorted keys is slower. Clearly, the most often referenced nodes of the linearized tree are kept in the processor cache. This combined with the separator elements being tightly packed in the nodes avoids stalls due to memory accesses. In contrast to the vectorized k-ary search, the (scalar) binary search on linearized binary trees fails to outperform the corresponding binary search operating directly on a sorted array. The vectorized k-ary search makes better use of the linearized tree layout, since the cost of a SIMD load is similar to a scalar load, while the scalar binary search needs more iterations and has more spread out memory accesses.

The higher search time in a linearized tree for smaller key counts is explained by the overhead introduced by the need to construct an iterator, that is used to efficiently loop over the keys in sorted order. The right side of Figure 5.10 shows the exact match search on the same linearized tree as before. The exact match search is 10% to 30% faster than the lower bound algorithm and is always faster than the 5-ary search on the sorted array. However, it only reliably beats an optimized binary search on arrays of more than 2^{15} keys.

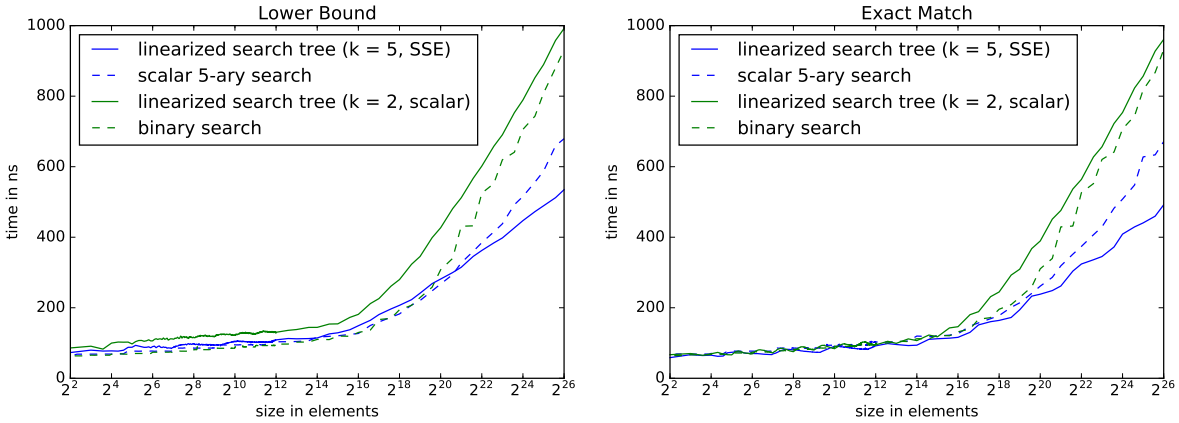

Figure 5.10: Lower bound and exact match search on a linearized tree compared with searching on a sorted array

5.4 Summary

We have implemented the scalar k-ary search algorithm and introduced a uniform implementation variant using precalculated tables to reduce the cost of index computations. Eliminating the conditional branches out of the loops of these algorithms yielded additional variants. Our experimental evaluation showed, that k should not be a power of two to avoid the cache aliasing we have already observed in the binary search. Setting k to 3 or 5 offers the best performance in most cases. We found, that on average the uniform branchless 5-ary search is the best scalar k-ary search for $k > 2$ on the evaluation system.

To evaluate the viability of SIMD k-ary searching, we implemented the SIMD k-ary search discussed in the background chapter. The vectorized k-ary search proved to be inferior to the scalar k-ary search.

Comparing direct and lower bound based exact match search implementations, we found that testing for equality with the search key in the search loop has more potential to speed up the k-ary search than the binary search. However, our fastest k-ary search function, the uniform branchless 5-ary search was slowed down by these additional conditional jumps.

Finally, we implemented and evaluated vectorized k-ary search functions operating on linearized trees. These functions are faster if the sorted array is too large to be effectively cached, but the lower bound variant adds additional overhead for iteration over a range of keys in sorted order. This means, that searching directly on the sorted list is actually faster for less than 2^{20} keys in our test. In case of an exact match search using linearized search trees becomes worthwhile for more than 2^{15} keys. We conclude, that the k-ary search on linearized trees is only useful for very large arrays, where to cost of LLC misses becomes dominant. Of course the higher search speed also needs to be balanced with the cost of constructing the linearized tree.

6. Comparison of Sequential, Binary and k-ary Searching

In this chapter, we compare the best sequential, binary and k-ary search algorithms discussed in the previous chapters and analyze their advantages and disadvantages on arrays of different sizes. In particular, we look at three array size ranges we found to favor different search algorithms. For each size range, we point out the fastest lower bound search algorithms. The relative order of the best performing algorithms is the same for the corresponding exact match searches. Note that we do not include the k-ary search on linearized trees in this comparison, since it is a special case using a different array layout.

Arrays of less than 32 keys

For very small arrays of up to about 32 keys the vectorized branchless sequential search is the fastest algorithm (Figure 6.1). Loop unrolling does not improve the run-time on such small arrays, it only becomes effective for more than about 50 keys. Binary searching in form of the uniform (branchless) binary search becomes faster than vectorized sequential searching for more than about 32 keys. If we restrict our view to scalar implementations, the point where binary searching becomes faster then sequential searching is already reached at 8 keys.

Arrays sized between 32 and 2^{16} keys

The branchless uniform binary search stays the fastest search function for up to 2^{13} keys (Figure 6.2), with the branchless uniform binary search employing software prefetching only 1 ns to 2 ns behind. For arrays larger than 2^{13} elements, the prefetching uniform search becomes faster than the non-prefetching version. k-ary searching does not reach the performance of binary searching on these array sizes. Offset binary searching is tens of nanoseconds slower than both the uniform binary search and the k-ary search and thus can not be recommended in this case.

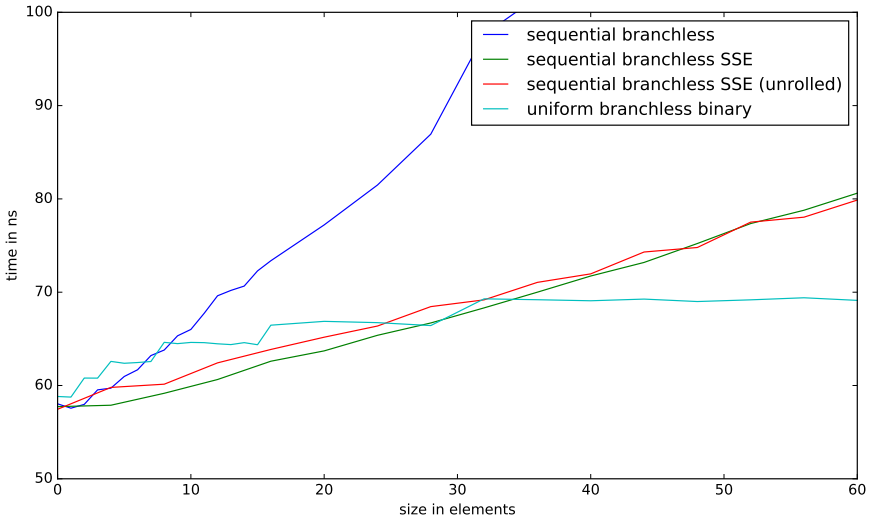

Figure 6.1: Fastest lower bound search algorithms for less than 32 keys, showing up to 64 keys

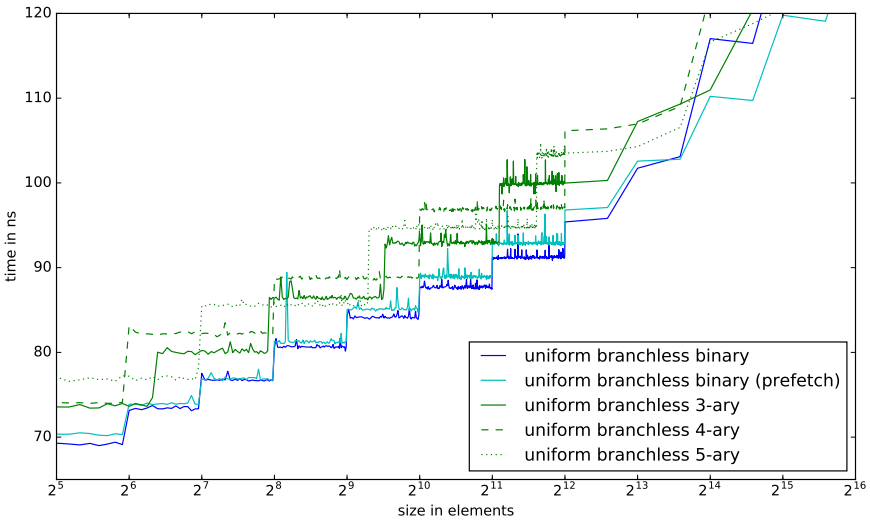

Figure 6.2: Fastest lower bound search algorithms for arrays of 32 to 2^{16} keys

Arrays of more than 2^{16} keys

For arrays of more than 2^{16} keys (Figure 6.3), the number of LLC misses rises and due to the cache aliasing problems discussed in Section 4.3.8, regular binary searching becomes inefficient for large arrays, so the k-ary search functions are better for sizes above 2^{18} keys. For arrays of this size, the differences between uniform and non-uniform, branching and branchless k-ary search implementations become much less significant, since memory latency is the bottleneck.

For more than 2^{24} keys the branching k-ary searches, sometimes avoiding to load all separators in a search step, become faster than their branchless counterparts, since

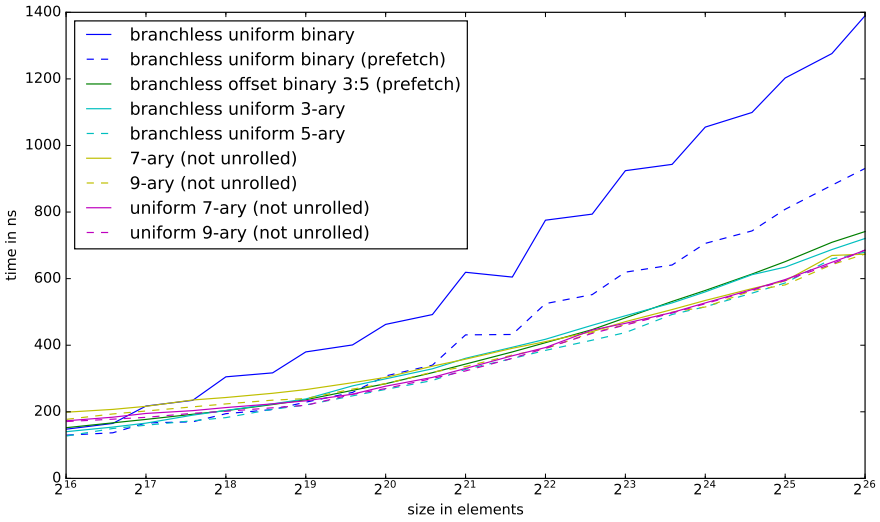

Figure 6.3: Fastest lower bound search algorithms for more than 2^{16} keys

very costly LLC misses become even more frequent. Additionally, functions using $k = 7$ and 9 become faster relative to the other search algorithms, probably because they can profit from the hardware prefetcher recognizing the loads of the evenly spaced separator elements. Like mentioned in Section 5.3, not unrolling the inner loop of the k-ary search can be faster than unrolling it, especially for larger k and array sizes. This leads to the not unrolled 7 and 9-ary search to be among the fastest algorithms on arrays of 2^{19} and more keys.

On large arrays, the offset binary and branchless uniform ternary search perform almost as good as the faster k-ary search functions, but require less main memory bandwidth (see Figure 6.4). In fact, the ternary search uses the least bandwidth of the fastest search functions, thereby offering a good latency–bandwidth compromise.

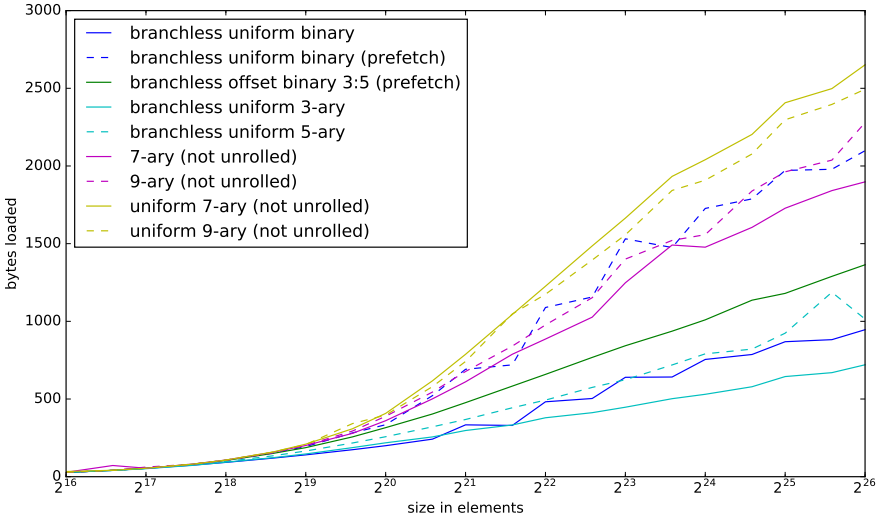

Figure 6.4: Average number of bytes loaded from main memory per search

If the search keys are generated by Algorithm 3 instead of Algorithm 1 or 2 (Figure 6.5), the branchless uniform 3 and 5-ary search functions stay among the fastest even for large arrays, since there are less LLC misses. The branching 3, 6 and 9-ary search are also very fast. Implementations with an unrolled inner loop are often a few percent faster than the not unrolled ones.

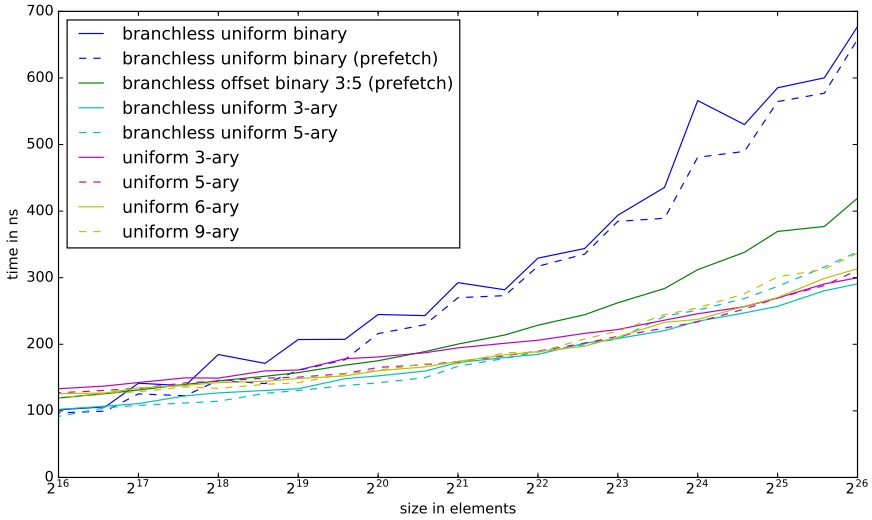

Figure 6.5: Fastest lower bound search algorithms with search keys are generated by Algorithm 3

Comparison to `std::lower_bound`

Finally, we have plotted the speedup obtained by using optimized search algorithms compared to the C++ standard library function `std::lower_bound` (as implemented by Microsoft Visual C++ 2015) in Figure 6.6. Improvements of up to 60% are possible, and even for large arrays the speedup is still at least 5% and can be as high as 40%.

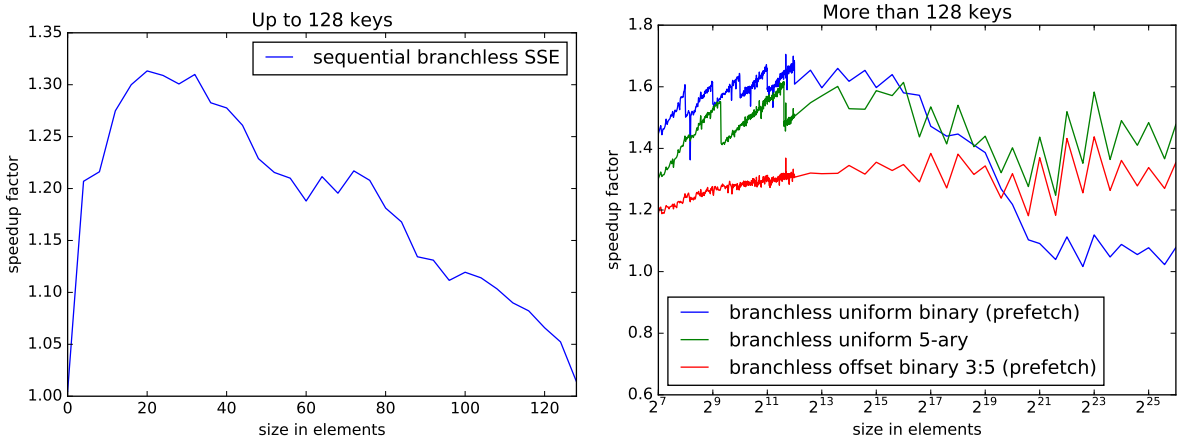

Figure 6.6: Relative performance of optimized search algorithms compared to the general purpose `std::lower_bound`

7. Related Work

We have compared branching, branchless and branchless with explicit prefetching implementations of binary searching. Khuong and Morin [KM17] have done the same with similar results, although they state that their explicit prefetching implementation consumes more memory bandwidth than the branching one. In our evaluation, explicit prefetching needed less memory bandwidth. They also have analyzed the cache usage of the binary search and point out the cache aliasing issues we discuss in Section 4.3.8. There is also an online article by Khuong examining the caching behavior of binary, offset binary, ternary and quaternary searching in detail [Khu12].

Zeuch et al. employ hardware performance counters to characterize to processor utilization reached by the relational selection operator concerning branch prediction, cache misses and superscalar execution [ZF15], similar to how we used them to analyze search algorithms.

Optimizing Sequential Searching with Sentinel Keys

A well-known optimization of sequential searching, we have not considered in Chapter 3, is to place a sentinel key at the end of the array to avoid the conditional jump exiting the search loop when the array is exhausted [Knu98]. For an exact match search the sentinel key should be equal to the search key and for a lower bound search it should be larger than all other keys. This way termination is ensured, even if the search key is not in the array or in case of a lower bound search, if the lower bound is beyond the end of the array.

Other Binary Search Variants

Another variant of the uniform binary search is the bit set binary search [Pul11][Can17]. It uses only bit manipulation operations to calculate array indices and an efficient branchless implementation is also possible. Cannizzo also proposes to vectorize the

binary search to search for multiple keys in parallel instead of loading clusters of adjacent separators, searching for a single key [Can17]. Since the vectorized binary search we have implemented in Section 4.1.3 was inferior to an optimized scalar binary search, exploring this direction of parallelization might be interesting.

Other Array Layouts

This works is focused on searching in sorted arrays, but as we have seen by implementing the array layout proposed by Schlegel et al. to speed up k-ary searching [SGL09], other array layouts can offer large benefits. The linearized k-ary tree layout implemented in this work is also known as Btree layout [KM17]. There are many more array layouts to improve cache locality in search algorithms based on different tree traversals and blocking schemes in the literature. Examples include the Eytzinger and van Emde Boas layout, as well as layouts based on preorder and inorder tree traversals [BFJ02][KM17]. In Section 5.3.6 we have seen that packing more keys into one node of a linearized tree can considerably improve performance, but we have not tried to increase the size of a node to fill a complete cache line. FAST is an optimized tree layout organized to exploit the width of SIMD loads, the cache line size and page size [KCS+10].

Interpolation Search

A completely different way to improve search performance on sorted data is to make assumptions about the distribution of said data. The search algorithm can then make more informed decisions on which elements are selected as separators. Interpolation search is a dichotomic search algorithm using this idea. Classically it assumes a uniform distribution. If the data is indeed uniformly distributed, the expected number of iterations is only $\log_2(\log_2(size)) + \mathcal{O}(1)$ with a variance of $\mathcal{O}(\log_2(\log_2(size)))$, making it asymptotically better than binary searching [GRG80]. Interpolation search can be adapted to other distributions by using a suitable approximation to the distribution [Pri71].

8. Conclusion and Future Work

The next paragraphs summarize the most important results of this work. Then we give an outlook to possible future work. With all our results, it is important to keep in mind that we mostly used only a single evaluation system. While our results should be applicable to most modern CPUs, details may change from machine to machine, especially concerning prefetching and SIMD.

Sequential searching has a worse asymptotic time complexity than binary searching, but due to different constants factors, one could assume the very simple sequential search to be faster on small arrays. We found, that this is only true for arrays of very few keys, since an optimized binary search is executed very efficiently on modern processors. In our evaluation the sequential search only beat the binary search by a few nanoseconds on an array of less then 8 32-bit keys, thereby occupying less than half of a 64-byte cache line. Nevertheless sequential searching on sorted data has its usage, if SIMD processing is used. SIMD sequential searching is the fastest way we found to search in small arrays. Note however, that we only considered cases where the array was aligned to the SIMD word size and a multiple of the SIMD word size in length. If these constraints are not met, special handling of the 'overhanging' elements and/or unaligned loads are needed, potentially degrading performance.

Binary searching, the most obvious and theoretically optimal choice for searching in randomly accessible sorted data, performed well in our evaluation, but we found huge optimization potential in the classical textbook implementation. Reducing the overhead of index computations and branching, by using a uniform implementation variant and employing branch predication, worked especially well. A problem of binary searching is its interaction with the power of two based cache addressing scheme found in most current processors, that can lead to a very inefficient utilization of caches. We found both binary searching with an offset split location and k-ary searching with k not a power of two effective in avoiding this situation.

Binary search is struggling, if an arrays becomes too large to fit in the processor caches, because of cache aliasing issues and random, irregular spaced memory accesses, that make prefetching difficult even with software assistance. On such large arrays k-ary searching has the better performance, although it is difficult to pinpoint the optimal choice of k and optimization techniques. Testing on the specific hardware with the specific arrays sizes to be used is probably necessary. In addition to the k-ary search operating on sorted arrays, we implemented a variant using linearized search trees to improve the locality of memory references. Like expected, an optimized array layout further improves performance on large arrays.

The vectorization techniques we tried on the binary and k-ary search algorithms yielded inferior implementations in all cases. The only functions that came close to scalar searching used AVX2 gather load instructions. SIMD processing only improved the search algorithms, if SIMD loads from continuous memory locations could be used. This was the case for the sequential search and on linearized k-ary search trees. Nevertheless, SIMD processing in binary and k-ary searching might prove useful, if other vectorization techniques, like searching multiple keys in parallel, are used, or on other processors implementing SIMD differently.

On multiple occasions, we found a tradeoff between search speed and the memory bandwidth used. For example, a branchless binary search, thus using less branch prediction, consumes much less memory bandwidth than a regular branching one, but also waits for memory much longer. This latency–bandwidth tradeoff might be important to keep in mind if a search algorithm is selected as part of a greater function.

Future Work

There are a number of points we have not addressed in this work, that might be interesting in the future:

Array alignment and non-continuous keys: This thesis is restricted to SIMD word aligned arrays of tightly packed keys. It would be interesting to lift these restrictions, for example by considering keys interleaved with other data.

Specialization to specific array sizes: The search functions in this thesis work on arrays of arbitrary length. Further optimizations are possible, if we create functions restricted to one specific array length.

Offset k-ary search: The position of the separator elements in a k-ary search could be offset like in the offset binary search to avoid cache aliasing if k is a power of two.

Combining search algorithms: Combining different search algorithms, for example by first utilizing binary and, after the search interval is narrowed down, SIMD sequential searching, might further improve performance. Especially interesting in this regard is the interpolation search, since its expected number of iterations is

better than in the binary search, but with higher constant overhead per iteration and only if its assumptions on the distribution of the keys are met. Otherwise it can degenerate to an inefficient sequential search. Therefore it is commonly combined with other search algorithms like sequential or binary search [GR77][SS85].

Other vectorization techniques: The SIMD binary and k-ary search algorithms we evaluated were not very successful. By using different vectorization techniques, like parallelizing over multiple search runs, it might be possible to obtain better performance.

Compare different processor architectures: We have mainly used an Intel processor of the Ivy Bridge microarchitecture in the evaluation. Other microarchitectures, processors from different vendors and non-x86 processors will probably respond differently to certain optimizations, especially to software controlled prefetching.

Linearized k-ary search trees: A logical extension of the SIMD search on k-ary search trees would be to increase the size of the nodes to fill entire cache lines. This way no bytes loaded on a last level cache miss would be wasted, because cache lines are always loaded as complete units from main memory. Since the main costs of searching on large arrays comes from last level cache misses, this could further improve performance on arrays not entirely fitting in the processor cache. For example, on a machine with 64-byte cache lines and 128-bit SIMD registers we could fit 16 keys into one node, yielding a 17-ary search tree. Since searching through one node would then no longer be possible with a single SIMD comparison we would have to select a secondary search algorithm for this task.

Evaluating the search algorithms in the context of a larger system: Searching is often part of a larger algorithm. Therefore it would be interesting to incorporate the search optimization techniques we have found useful into larger algorithms and data structures. An example of such a data structure is Elf, where the search in its sorted dimension lists could be optimized [BKSS17].

A. Helper Functions

This appendix contains the definitions of the helper functions used throughout this work. The preprocessor constants `USE_SSE`, `USE_VEX`, `USE_AVX` and `USE_AVX2` are used to configure the code. Only one of `USE_SSE`, `USE_AVX` and `USE_AVX2` may be defined, to select the the SSE4.1, AVX or AVX2 instruction set. If `USE_SSE` is defined, `USE_VEX` can additionally be defined to use the new SSE instruction encoding introduced with AVX.

A.1 Bit Scans

Listing A.1: Bit scans

```
1  inline unsigned char bitScanForward(
2    unsigned long *index, unsigned long mask) {
3  #ifdef MSVC
4    return _BitScanForward(index, mask);
5  #elif (defined GNU)
6    if (mask == 0) return 0;
7    *index = (unsigned long)__builtin_ctz(mask);
8    return 1;
9  #endif
10 }
11
12 inline unsigned char bitScanReverse(
13   unsigned long *index, unsigned long mask) {
14 #ifdef MSVC
15   return _BitScanReverse(index, mask);
16 #elif (defined GNU)
17   if (mask == 0) return 0;
18   *index = (unsigned long)(8*sizeof(int) - __builtin_clz(mask) - 1);
19   return 1;
20 #endif
21 }
```

A.2 Mathematical Functions

The uniform binary search needs to efficiently calculate the function $\lceil \log_2(x) \rceil$ for integer x. The function `ceilLog` does this using bit scans for $0 < x \leq 2^{31}$, assuming `unsigned long` is 32 bits wide.

Listing A.2: Definition of `ceilLog2`

```
1  inline unsigned long ceilLog2(unsigned long x) {
2    unsigned long i;
3    bitScanReverse(&i, x); // i = floor(log2(x))
4    x += (1 << i) - 1;     // round up
5    bitScanReverse(&i, x);
6    return i;
7  }
```

Both, the scalar and vectorized uniform k-ary search need to calculate the expression k^x for integer x. Since k is a constant, a precomputed table accessed via the template `pow_const_base` is used for this. $k = 2, 4$ and 8 are treated as special cases, where cheap bit shifts are used instead of table look-ups.

Listing A.3: Definition of `pow_const_base`

```
1  template <typename T, unsigned int base> struct pow_const_base {
2    const static T table[];
3  };
4  template <typename T, unsigned int base> constexpr T powConstBase(T x)
5    { return pow_const_base<T, base>::table[x]; }
6  template <>
7  inline constexpr uint8_t powConstBase<uint8_t, 2>(uint8_t x)
8    { return 1 << x; }
9  template <>
10 inline constexpr uint16_t powConstBase<uint16_t, 2>(uint16_t x)
11   { return 1 << x; }
12 template <>
13 inline constexpr uint32_t powConstBase<uint32_t, 2>(uint32_t x)
14   { return 1 << x; }
15 template <>
16 inline constexpr uint64_t powConstBase<uint64_t, 2>(uint64_t x)
17   { return 1ull << x; }
18 template <>
19 inline constexpr uint8_t powConstBase<uint8_t, 4>(uint8_t x)
20   { return 1 << (2 * x); }
21 template <>
22 inline constexpr uint16_t powConstBase<uint16_t, 4>(uint16_t x)
23   { return 1 << (2 * x); }
24 template <>
25 inline constexpr uint32_t powConstBase<uint32_t, 4>(uint32_t x)
26   { return 1 << (2 * x); }
27 template <>
28 inline constexpr uint64_t powConstBase<uint64_t, 4>(uint64_t x)
29   { return 1ull << (2 * x); }
30 template <>
```

```
31 inline constexpr uint8_t powConstBase<uint8_t, 8>(uint8_t x)
32   { return 1 << (3 * x); }
33 template <>
34 inline constexpr uint16_t powConstBase<uint16_t, 8>(uint16_t x)
35   { return 1 << (3 * x); }
36 template <>
37 inline constexpr uint32_t powConstBase<uint32_t, 8>(uint32_t x)
38   { return 1 << (3 * x); }
39 template <>
40 inline constexpr uint64_t powConstBase<uint64_t, 8>(uint64_t x)
41   { return 1ull << (3 * x); }
```

A.3 SSE/AVX Intrinsic Wrappers

Our code uses the following wrapper templates to select suitable SSE/AVX intrinsic functions for the given key and index data types. These functions are used throughout all our SIMD implementations. Note that out vector type is always one of the integer types `__m128i` and `__m256i`. For floating point keys casts are used. The distinction between integer and floating point SIMD types is only meaningful to the compiler, thus the cast are purely syntactical and incur no run-time overhead.

Listing A.4: Definition of `Vector`

```
1 #ifdef USE_SSE
2 using Vector = __m128i;
3 #else
4 using Vector = __m256i;
5 #endif
```

Listing A.5: Definition of `keys_per_simd_word`

```
1  template <typename VectorType, typename T>
2    constexpr unsigned int keys_per_simd_word();
3  template <> inline constexpr unsigned int
4    keys_per_simd_word<__m128i, int8_t>() { return 16; }
5  template <> inline constexpr unsigned int
6    keys_per_simd_word<__m128i, int16_t>() { return 8; }
7  template <> inline constexpr unsigned int
8    keys_per_simd_word<__m128i, int32_t>() { return 4; }
9  template <> inline constexpr unsigned int
10   keys_per_simd_word<__m128i, int64_t>() { return 2; }
11 template <> inline constexpr unsigned int
12   keys_per_simd_word<__m128i, uint8_t>() { return 16; }
13 template <> inline constexpr unsigned int
14   keys_per_simd_word<__m128i, uint16_t>() { return 8; }
15 template <> inline constexpr unsigned int
16   keys_per_simd_word<__m128i, uint32_t>() { return 4; }
17 template <> inline constexpr unsigned int
18   keys_per_simd_word<__m128i, uint64_t>() { return 2; }
19 template <> inline constexpr unsigned int
20   keys_per_simd_word<__m128i, float>() { return 4; }
```

```
21 template <> inline constexpr unsigned int
22   keys_per_simd_word<__m128i, double>() { return 2; }
23 #if (defined USE_AVX) || (defined USE_AVX2)
24 template <typename VectorType, typename T>
25   constexpr unsigned int keys_per_simd_word();
26 template <> inline constexpr unsigned int
27   keys_per_simd_word<__m256i, int8_t>() { return 32; }
28 template <> inline constexpr unsigned int
29   keys_per_simd_word<__m256i, int16_t>() { return 16; }
30 template <> inline constexpr unsigned int
31   keys_per_simd_word<__m256i, int32_t>() { return 8; }
32 template <> inline constexpr unsigned int
33   keys_per_simd_word<__m256i, int64_t>() { return 4; }
34 template <> inline constexpr unsigned int
35   keys_per_simd_word<__m256i, uint8_t>() { return 32; }
36 template <> inline constexpr unsigned int
37   keys_per_simd_word<__m256i, uint16_t>() { return 16; }
38 template <> inline constexpr unsigned int
39   keys_per_simd_word<__m256i, uint32_t>() { return 8; }
40 template <> inline constexpr unsigned int
41   keys_per_simd_word<__m256i, uint64_t>() { return 4; }
42 template <> inline constexpr unsigned int
43   keys_per_simd_word<__m256i, float>() { return 8; }
44 template <> inline constexpr unsigned int
45   keys_per_simd_word<__m256i, double>() { return 4; }
46 #endif
```

Listing A.6: Loding, storing and setting SSE/AVX registers

```
1  inline __m128i loadVector(const __m128i *mem_addr)
2    { return _mm_load_si128(mem_addr); }
3  inline void storeVector(__m128i *mem_addr, __m128i a)
4    { _mm_store_si128(mem_addr, a); }
5
6  template <typename VectorType> inline VectorType getAllOnesVector();
7  template <> inline __m128i getAllOnesVector<__m128i>() {
8    __m128i a = _mm_undefined_si128();
9    return _mm_cmpeq_epi32(a, a);
10 }
11
12 template <typename VectorType, typename T> VectorType _mm_set1(T a);
13 template <> inline __m128i _mm_set1(int8_t a)
14   { return _mm_set1_epi8(a); }
15 template <> inline __m128i _mm_set1(int16_t a)
16   { return _mm_set1_epi16(a); }
17 template <> inline __m128i _mm_set1(int32_t a)
18   { return _mm_set1_epi32(a); }
19 template <> inline __m128i _mm_set1(int64_t a)
20   { return _mm_set1_epi64x(a); }
21 template <> inline __m128i _mm_set1(uint8_t a)
22   { return _mm_set1_epi8(a); }
23 template <> inline __m128i _mm_set1(uint16_t a)
24   { return _mm_set1_epi16(a); }
```

```
25 template <> inline __m128i _mm_set1(uint32_t a)
26   { return _mm_set1_epi32(a); }
27 template <> inline __m128i _mm_set1(uint64_t a)
28   { return _mm_set1_epi64x(a); }
29 template <> inline __m128i _mm_set1(float a)
30   { return _mm_castps_si128(_mm_set1_ps(a)); }
31 template <> inline __m128i _mm_set1(double a)
32   { return _mm_castpd_si128(_mm_set1_pd(a)); }
33
34 #if (defined USE_AVX) || (defined USE_AVX2)
35 inline __m256i loadVector(const __m256i *mem_addr)
36   { return _mm256_load_si256(mem_addr); }
37 inline void storeVector(__m256i *mem_addr, __m256i a)
38   { _mm256_store_si256(mem_addr, a); }
39
40 template <> inline __m256i getAllOnesVector<__m256i>() {
41 #ifdef USE_AVX2
42   __m256i a = _mm256_undefined_si256();
43   return _mm256_cmpeq_epi32(a, a);
44 #else
45   __m256 a = _mm256_setzero_ps();
46   return _mm256_castps_si256(_mm256_cmp_ps(a, a, _CMP_EQ_OQ));
47 #endif
48 }
49
50 template <> inline __m256i _mm_set1(int8_t a)
51   { return _mm256_set1_epi8(a); }
52 template <> inline __m256i _mm_set1(int16_t a)
53   { return _mm256_set1_epi16(a); }
54 template <> inline __m256i _mm_set1(int32_t a)
55   { return _mm256_set1_epi32(a); }
56 template <> inline __m256i _mm_set1(int64_t a)
57   { return _mm256_set1_epi64x(a); }
58 template <> inline __m256i _mm_set1(uint8_t a)
59   { return _mm256_set1_epi8(a); }
60 template <> inline __m256i _mm_set1(uint16_t a)
61   { return _mm256_set1_epi16(a); }
62 template <> inline __m256i _mm_set1(uint32_t a)
63   { return _mm256_set1_epi32(a); }
64 template <> inline __m256i _mm_set1(uint64_t a)
65   { return _mm256_set1_epi64x(a); }
66 template <> inline __m256i _mm_set1(float a)
67   { return _mm256_castps_si256(_mm256_set1_ps(a)); }
68 template <> inline __m256i _mm_set1(double a)
69   { return _mm256_castpd_si256(_mm256_set1_pd(a)); }
70 #endif
```

A.3.1 SSE/AVX Comparisons

The following template functions provide signed SIMD comparisons including the function `adjustForSignedComparison` needed to preprocess unsigned operands.

Listing A.7: SIMD comparison

```
template <typename T> __m128i _mm_cmpeq(__m128i a, __m128i b);
template <> inline __m128i _mm_cmpeq<int8_t>(__m128i a, __m128i b)
  { return _mm_cmpeq_epi8(a, b); }
template <> inline __m128i _mm_cmpeq<int16_t>(__m128i a, __m128i b)
  { return _mm_cmpeq_epi16(a, b); }
template <> inline __m128i _mm_cmpeq<int32_t>(__m128i a, __m128i b)
  { return _mm_cmpeq_epi32(a, b); }
template <> inline __m128i _mm_cmpeq<int64_t>(__m128i a, __m128i b)
  { return _mm_cmpeq_epi64(a, b); }
template <> inline __m128i _mm_cmpeq<uint8_t>(__m128i a, __m128i b)
  { return _mm_cmpeq_epi8(a, b); }
template <> inline __m128i _mm_cmpeq<uint16_t>(__m128i a, __m128i b)
  { return _mm_cmpeq_epi16(a, b); }
template <> inline __m128i _mm_cmpeq<uint32_t>(__m128i a, __m128i b)
  { return _mm_cmpeq_epi32(a, b); }
template <> inline __m128i _mm_cmpeq<uint64_t>(__m128i a, __m128i b)
  { return _mm_cmpeq_epi64(a, b); }
template <> inline __m128i _mm_cmpeq<float>(__m128i a, __m128i b) {
  return _mm_castps_si128(
    _mm_cmpeq_ps(_mm_castsi128_ps(a), _mm_castsi128_ps(b)));
}
template <> inline __m128i _mm_cmpeq<double>(__m128i a, __m128i b) {
  return _mm_castpd_si128(
    _mm_cmpeq_pd(_mm_castsi128_pd(a), _mm_castsi128_pd(b)));
}

template <typename T> __m128i _mm_cmpgt(__m128i a, __m128i b);
template <> inline __m128i _mm_cmpgt<int8_t>(__m128i a, __m128i b)
  { return _mm_cmpgt_epi8(a, b); }
template <> inline __m128i _mm_cmpgt<int16_t>(__m128i a, __m128i b)
  { return _mm_cmpgt_epi16(a, b); }
template <> inline __m128i _mm_cmpgt<int32_t>(__m128i a, __m128i b)
  { return _mm_cmpgt_epi32(a, b); }
template <> inline __m128i _mm_cmpgt<int64_t>(__m128i a, __m128i b)
  { return _mm_cmpgt_epi64(a, b); }
template <> inline __m128i _mm_cmpgt<uint8_t>(__m128i a, __m128i b)
  { return _mm_cmpgt_epi8(a, b); }
template <> inline __m128i _mm_cmpgt<uint16_t>(__m128i a, __m128i b)
  { return _mm_cmpgt_epi16(a, b); }
template <> inline __m128i _mm_cmpgt<uint32_t>(__m128i a, __m128i b)
  { return _mm_cmpgt_epi32(a, b); }
template <> inline __m128i _mm_cmpgt<uint64_t>(__m128i a, __m128i b)
  { return _mm_cmpgt_epi64(a, b); }
template <> inline __m128i _mm_cmpgt<float>(__m128i a, __m128i b) {
  return _mm_castps_si128(
    _mm_cmpgt_ps(_mm_castsi128_ps(a), _mm_castsi128_ps(b)));
}
template <> inline __m128i _mm_cmpgt<double>(__m128i a, __m128i b) {
  return _mm_castpd_si128(
    _mm_cmpgt_pd(_mm_castsi128_pd(a), _mm_castsi128_pd(b)));
}
```

```
52
53 template <typename T> __m128i _mm_cmplt(__m128i a, __m128i b)
54 { return _mm_cmpgt<T>(b, a); }
55
56 #if (defined USE_AVX) || (defined USE_AVX2)
57 template <typename T> __m256i _mm_cmpeq(__m256i a, __m256i b);
58 template <> inline __m256i _mm_cmpeq<float>(__m256i a, __m256i b) {
59   return _mm256_castps_si256(_mm256_cmp_ps(
60     _mm256_castsi256_ps(a), _mm256_castsi256_ps(b), _CMP_EQ_OQ));
61 }
62 template <> inline __m256i _mm_cmpeq<double>(__m256i a, __m256i b) {
63   return _mm256_castpd_si256(_mm256_cmp_pd(
64     _mm256_castsi256_pd(a), _mm256_castsi256_pd(b), _CMP_EQ_OQ));
65 }
66
67 template <typename T> __m256i _mm_cmpgt(__m256i a, __m256i b);
68 template <> inline __m256i _mm_cmpgt<float>(__m256i a, __m256i b) {
69   return _mm256_castps_si256(_mm256_cmp_ps(
70     _mm256_castsi256_ps(a), _mm256_castsi256_ps(b), _CMP_GT_OQ));
71 }
72 template <> inline __m256i _mm_cmpgt<double>(__m256i a, __m256i b) {
73   return _mm256_castpd_si256(_mm256_cmp_pd(
74     _mm256_castsi256_pd(a), _mm256_castsi256_pd(b), _CMP_GT_OQ));
75 }
76
77 template <typename T> __m256i _mm_cmplt(__m256i a, __m256i b)
78   { return _mm_cmpgt<T>(b, a); }
79
80 #ifdef USE_AVX2
81 template <> inline __m256i _mm_cmpeq<int8_t>(__m256i a, __m256i b)
82   { return _mm256_cmpeq_epi8(a, b); }
83 template <> inline __m256i _mm_cmpeq<int16_t>(__m256i a, __m256i b)
84   { return _mm256_cmpeq_epi16(a, b); }
85 template <> inline __m256i _mm_cmpeq<int32_t>(__m256i a, __m256i b)
86   { return _mm256_cmpeq_epi32(a, b); }
87 template <> inline __m256i _mm_cmpeq<int64_t>(__m256i a, __m256i b)
88   { return _mm256_cmpeq_epi64(a, b); }
89 template <> inline __m256i _mm_cmpeq<uint8_t>(__m256i a, __m256i b)
90   { return _mm256_cmpeq_epi8(a, b); }
91 template <> inline __m256i _mm_cmpeq<uint16_t>(__m256i a, __m256i b)
92   { return _mm256_cmpeq_epi16(a, b); }
93 template <> inline __m256i _mm_cmpeq<uint32_t>(__m256i a, __m256i b)
94   { return _mm256_cmpeq_epi32(a, b); }
95 template <> inline __m256i _mm_cmpeq<uint64_t>(__m256i a, __m256i b)
96   { return _mm256_cmpeq_epi64(a, b); }
97
98 template <> inline __m256i _mm_cmpgt<int8_t>(__m256i a, __m256i b)
99   { return _mm256_cmpgt_epi8(a, b); }
100 template <> inline __m256i _mm_cmpgt<int16_t>(__m256i a, __m256i b)
101   { return _mm256_cmpgt_epi16(a, b); }
102 template <> inline __m256i _mm_cmpgt<int32_t>(__m256i a, __m256i b)
103   { return _mm256_cmpgt_epi32(a, b); }
104 template <> inline __m256i _mm_cmpgt<int64_t>(__m256i a, __m256i b)
```

```
105   { return _mm256_cmpgt_epi64(a, b); }
106 template <> inline __m256i _mm_cmpgt<uint8_t>(__m256i a, __m256i b)
107   { return _mm256_cmpgt_epi8(a, b); }
108 template <> inline __m256i _mm_cmpgt<uint16_t>(__m256i a, __m256i b)
109   { return _mm256_cmpgt_epi16(a, b); }
110 template <> inline __m256i _mm_cmpgt<uint32_t>(__m256i a, __m256i b)
111   { return _mm256_cmpgt_epi32(a, b); }
112 template <> inline __m256i _mm_cmpgt<uint64_t>(__m256i a, __m256i b)
113   { return _mm256_cmpgt_epi64(a, b); }
114 #endif
115 #endif
```

Listing A.8: Definition of `adjustForSignedComparison`

```
1  extern const uint32_t VEC16_INT8_MIN[4] alignas(16);
2    // = { 0x80808080, 0x80808080, 0x80808080, 0x80808080 };
3  extern const uint32_t VEC16_INT16_MIN[4] alignas(16);
4    // = { 0x80008000, 0x80008000, 0x80008000, 0x80008000 };
5  extern const uint32_t VEC16_INT32_MIN[4] alignas(16);
6    // = { 0x80000000, 0x80000000, 0x80000000, 0x80000000 };
7  extern const uint32_t VEC16_INT64_MIN[4] alignas(16);
8    // = { 0x80000000, 0x00000000, 0x80000000, 0x00000000 };
9
10 template <typename T> __m128i
11   adjustForSignedComparison(__m128i);
12 template <> inline __m128i
13   adjustForSignedComparison<int8_t>(__m128i a) { return a; }
14 template <> inline __m128i
15   adjustForSignedComparison<int16_t>(__m128i a) { return a; }
16 template <> inline __m128i
17   adjustForSignedComparison<int32_t>(__m128i a) { return a; }
18 template <> inline __m128i
19   adjustForSignedComparison<int64_t>(__m128i a) { return a; }
20 template <> inline __m128i
21   adjustForSignedComparison<float>(__m128i a) { return a; }
22 template <> inline __m128i
23   adjustForSignedComparison<double>(__m128i a) { return a; }
24 template <> inline __m128i
25   adjustForSignedComparison<uint8_t>(__m128i a) {
26     return _mm_add_epi8(a, _mm_load_si128(
27       reinterpret_cast<const __m128i*>(VEC16_INT8_MIN))); }
28 template <> inline __m128i
29   adjustForSignedComparison<uint16_t>(__m128i a) {
30     return _mm_add_epi16(a, _mm_load_si128(
31       reinterpret_cast<const __m128i*>(VEC16_INT16_MIN))); }
32 template <> inline __m128i
33   adjustForSignedComparison<uint32_t>(__m128i a) {
34     return _mm_add_epi32(a, _mm_load_si128(
35       reinterpret_cast<const __m128i*>(VEC16_INT32_MIN))); }
36 template <> inline __m128i
37   adjustForSignedComparison<uint64_t>(__m128i a) {
38     return _mm_add_epi64(a, _mm_load_si128(
39       reinterpret_cast<const __m128i*>(VEC16_INT64_MIN))); }
```

```
40
41 #if (defined USE_AVX) || (defined USE_AVX2)
42 extern const uint32_t VEC32_INT8_MIN[8] alignas(32);
43   // = { 0x80808080, 0x80808080, 0x80808080, 0x80808080,
44   //     0x80808080, 0x80808080, 0x80808080, 0x80808080 };
45 extern const uint32_t VEC32_INT16_MIN[8] alignas(32);
46   // = { 0x80008000, 0x80008000, 0x80008000, 0x80008000,
47   //     0x80008000, 0x80008000, 0x80008000, 0x80008000 };
48 extern const uint32_t VEC32_INT32_MIN[8] alignas(32);
49   // = { 0x80000000, 0x80000000, 0x80000000, 0x80000000,
50   //     0x80000000, 0x80000000, 0x80000000, 0x80000000 };
51 extern const uint32_t VEC32_INT64_MIN[8] alignas(32);
52   // = { 0x80000000, 0x00000000, 0x80000000, 0x00000000,
53   //     0x80000000, 0x00000000, 0x80000000, 0x00000000 };
54
55 template <typename T> __m256i
56   adjustForSignedComparison(__m256i);
57 template <> inline __m256i
58   adjustForSignedComparison<int8_t>(__m256i a) { return a; }
59 template <> inline __m256i
60   adjustForSignedComparison<int16_t>(__m256i a) { return a; }
61 template <> inline __m256i
62   adjustForSignedComparison<int32_t>(__m256i a) { return a; }
63 template <> inline __m256i
64   adjustForSignedComparison<int64_t>(__m256i a) { return a; }
65 template <> inline __m256i
66   adjustForSignedComparison<float>(__m256i a) { return a; }
67 template <> inline __m256i
68   adjustForSignedComparison<double>(__m256i a) { return a; }
69
70 #ifdef USE_AVX2
71 template <> inline __m256i
72   adjustForSignedComparison<uint8_t>(__m256i a) {
73     return _mm256_add_epi8(a, _mm256_load_si256(
74       reinterpret_cast<const __m256i*>(VEC32_INT8_MIN))); }
75 template <> inline __m256i
76   adjustForSignedComparison<uint16_t>(__m256i a) {
77     return _mm256_add_epi16(a, _mm256_load_si256(
78       reinterpret_cast<const __m256i*>(VEC32_INT16_MIN))); }
79 template <> inline __m256i
80   adjustForSignedComparison<uint32_t>(__m256i a) {
81     return _mm256_add_epi32(a, _mm256_load_si256(
82       reinterpret_cast<const __m256i*>(VEC32_INT32_MIN))); }
83 template <> inline __m256i
84   adjustForSignedComparison<uint64_t>(__m256i a) {
85     return _mm256_add_epi64(a, _mm256_load_si256(
86       reinterpret_cast<const __m256i*>(VEC32_INT64_MIN))); }
87 #endif
88 #endif
```

A.3.2 Mask Evaluation

The following functions are used to evaluate the result masks generated by SSE/AVX comparisons.

Listing A.9: Mask Evaluation

```
1  template <typename T> int createMask(__m128i a)
2    { return _mm_movemask_epi8(a); }
3  template <> inline int createMask<float>(__m128i a)
4    { return _mm_movemask_ps(_mm_castsi128_ps(a)); }
5  template <> inline int createMask<double>(__m128i a)
6    { return _mm_movemask_pd(_mm_castsi128_pd(a)); }
7
8  template <typename VectorType, typename T> struct MASK { };
9
10 template <typename T> struct MASK<__m128i, T> {
11   static constexpr int NONE = 0;
12   static constexpr int ALL = 0xFFFF;
13 };
14
15 template <> struct MASK<__m128i, float> {
16   static constexpr int NONE = 0;
17   static constexpr int ALL = 0x0F;
18 };
19
20 template <> struct MASK<__m128i, double> {
21   static constexpr int NONE = 0;
22   static constexpr int ALL = 0x03;
23 };
24
25 #if (defined USE_AVX) || (defined USE_AVX2)
26 template <typename T> int createMask(__m256i a)
27   { return _mm256_movemask_epi8(a); }
28 template <> inline int createMask<float>(__m256i a)
29   { return _mm256_movemask_ps(_mm256_castsi256_ps(a)); }
30 template <> inline int createMask<double>(__m256i a)
31   { return _mm256_movemask_pd(_mm256_castsi256_pd(a)); }
32
33 template <typename T> struct MASK<__m256i, T> {
34   static constexpr int NONE = 0;
35   static constexpr int ALL = 0xFFFFFFFF;
36 };
37
38 template <> struct MASK<__m256i, float> {
39   static constexpr int NONE = 0;
40   static constexpr int ALL = 0xFF;
41 };
42
43 template <> struct MASK<__m256i, double> {
44   static constexpr int NONE = 0;
45   static constexpr int ALL = 0x0F;
46 };
47 #endif
```

```
48
49 template <typename T> unsigned int countPositiveResults(int mask) {
50   return (unsigned int)(_mm_popcnt_u32((unsigned int)mask) / sizeof(T));
51 }
52 template <> inline unsigned int countPositiveResults<float>(int mask) {
53   return (unsigned int)(_mm_popcnt_u32((unsigned int)mask));
54 }
55 template <> inline unsigned int countPositiveResults<double>(int mask) {
56   return (unsigned int)(_mm_popcnt_u32((unsigned int)mask));
57 }
58
59 template <typename T> unsigned char getFirstPositiveResult(
60   unsigned long *i, int mask) {
61   if (bitScanForward(i, (unsigned int)mask)) {
62     *i /= sizeof(T);
63     return 1;
64   }
65   else return 0;
66 }
67 template <> inline unsigned char getFirstPositiveResult<float>(
68   unsigned long *i, int mask)
69     { return bitScanForward(i, (unsigned int)mask); }
70 template <> inline unsigned chargetFirstPositiveResult<double>(
71   unsigned long *i, int mask)
72     { return bitScanForward(i, (unsigned int)mask); }
```

Listing A.10: Definition of `_mm_testz` and `_mm_testc`

```
1  template <typename T> int _mm_testz(__m128i a, __m128i b)
2    { return _mm_testz_si128(a, b); }
3  #ifdef USE_VEX
4  template <> inline int _mm_testz<float>(__m128i a, __m128i b)
5    { return _mm_testz_ps(_mm_castsi128_ps(a), _mm_castsi128_ps(b)); }
6  template <> inline int _mm_testz<double>(__m128i a, __m128i b)
7    { return _mm_testz_pd(_mm_castsi128_pd(a), _mm_castsi128_pd(b)); }
8  #endif
9  template <typename T> int _mm_testc(__m128i a, __m128i b)
10   { return _mm_testc_si128(a, b); }
11 #ifdef USE_VEX
12 template <> inline int _mm_testc<float>(__m128i a, __m128i b)
13   { return _mm_testc_ps(_mm_castsi128_ps(a), _mm_castsi128_ps(b)); }
14 template <> inline int _mm_testc<double>(__m128i a, __m128i b)
15   { return _mm_testc_pd(_mm_castsi128_pd(a), _mm_castsi128_pd(b)); }
16 #endif
17
18 #if (defined USE_AVX) || (defined USE_AVX2)
19 template <typename T> int _mm_testz(__m256i a, __m256i b)
20   { return _mm256_testz_si256(a, b); }
21 template <> inline int _mm_testz<float>(__m256i a, __m256i b) {
22   return _mm256_testz_ps(_mm256_castsi256_ps(a), _mm256_castsi256_ps(b));
23 }
24 template <> inline int _mm_testz<double>(__m256i a, __m256i b) {
25   return _mm256_testz_pd(_mm256_castsi256_pd(a), _mm256_castsi256_pd(b));
```

```
26 }
27 template <typename T> int _mm_testc(__m256i a, __m256i b)
28   { return _mm256_testc_si256(a, b); }
29 template <> inline int _mm_testc<float>(__m256i a, __m256i b) {
30   return _mm256_testc_ps(_mm256_castsi256_ps(a), _mm256_castsi256_ps(b));
31 }
32 template <> inline int _mm_testc<double>(__m256i a, __m256i b) {
33   return _mm256_testc_pd(_mm256_castsi256_pd(a), _mm256_castsi256_pd(b));
34 }
35 #endif
```

A.3.3 SSE/AVX Arithmetic

The vectorized k-ary search performs index computations with SSE/AVX. The following functions provide addition, subtraction and multiplication.

Listing A.11: SSE/AVX Arithmetic

```
1  template <typename T> __m128i _mm_add_epi(__m128i a, __m128i b);
2  template <> inline __m128i _mm_add_epi<int8_t>(__m128i a, __m128i b)
3    { return _mm_add_epi8(a, b); }
4  template <> inline __m128i _mm_add_epi<int16_t>(__m128i a, __m128i b)
5    { return _mm_add_epi16(a, b); }
6  template <> inline __m128i _mm_add_epi<int32_t>(__m128i a, __m128i b)
7    { return _mm_add_epi32(a, b); }
8  template <> inline __m128i _mm_add_epi<int64_t>(__m128i a, __m128i b)
9    { return _mm_add_epi64(a, b); }
10
11 template <typename T> __m128i _mm_sub_epi(__m128i a, __m128i b);
12 template <> inline __m128i _mm_sub_epi<int8_t>(__m128i a, __m128i b)
13   { return _mm_sub_epi8(a, b); }
14 template <> inline __m128i _mm_sub_epi<int16_t>(__m128i a, __m128i b)
15   { return _mm_sub_epi16(a, b); }
16 template <> inline __m128i _mm_sub_epi<int32_t>(__m128i a, __m128i b)
17   { return _mm_sub_epi32(a, b); }
18 template <> inline __m128i _mm_sub_epi<int64_t>(__m128i a, __m128i b)
19   { return _mm_sub_epi64(a, b); }
20
21 template <typename T> __m128i _mm_mullo_epi(__m128i a, __m128i b);
22 template <> inline __m128i _mm_mullo_epi<int16_t>(__m128i a, __m128i b)
23   { return _mm_mullo_epi16(a, b); }
24 template <> inline __m128i _mm_mullo_epi<int32_t>(__m128i a, __m128i b)
25   { return _mm_mullo_epi32(a, b); }
26 template <> inline __m128i _mm_mullo_epi<int64_t>(__m128i a, __m128i b)
27   { return _mm_mul_epi32(a, b); }
28
29 #ifdef USE_AVX2
30 template <typename T> __m256i _mm_add_epi(__m256i a, __m256i b);
31 template <> inline __m256i _mm_add_epi<int8_t>(__m256i a, __m256i b)
32   { return _mm256_add_epi8(a, b); }
33 template <> inline __m256i _mm_add_epi<int16_t>(__m256i a, __m256i b)
34   { return _mm256_add_epi16(a, b); }
```

```
35 template <> inline __m256i _mm_add_epi<int32_t>(__m256i a, __m256i b)
36   { return _mm256_add_epi32(a, b); }
37 template <> inline __m256i _mm_add_epi<int64_t>(__m256i a, __m256i b)
38   { return _mm256_add_epi64(a, b); }
39
40 template <typename T> __m256i _mm_sub_epi(__m256i a, __m256i b);
41 template <> inline __m256i _mm_sub_epi<int8_t>(__m256i a, __m256i b)
42   { return _mm256_sub_epi8(a, b); }
43 template <> inline __m256i _mm_sub_epi<int16_t>(__m256i a, __m256i b)
44   { return _mm256_sub_epi16(a, b); }
45 template <> inline __m256i _mm_sub_epi<int32_t>(__m256i a, __m256i b)
46   { return _mm256_sub_epi32(a, b); }
47 template <> inline __m256i _mm_sub_epi<int64_t>(__m256i a, __m256i b)
48   { return _mm256_sub_epi64(a, b); }
49
50 template <typename T> __m256i _mm_mullo_epi(__m256i a, __m256i b);
51 template <> inline __m256i _mm_mullo_epi<int16_t>(__m256i a, __m256i b)
52   { return _mm256_mullo_epi16(a, b); }
53 template <> inline __m256i _mm_mullo_epi<int32_t>(__m256i a, __m256i b)
54   { return _mm256_mullo_epi32(a, b); }
55 template <> inline __m256i _mm_mullo_epi<int64_t>(__m256i a, __m256i b)
56   { return _mm256_mul_epi32(a, b); }
57 #endif
```

A.3.4 AVX2 Gather Loads

AVX2 supports loads from non-continuous memory locations. The vectorized k-ary search can use this to speed up loads.

Listing A.12: AVX2 gather

```
1  #ifdef USE_AVX2
2  template <typename KeyVector, typename IndexVector, typename T>
3    KeyVector _mm_gather(const T *base_addr, IndexVector vindex);
4
5  template <> inline __m256i _mm_gather(
6    const int32_t *base_addr, __m256i vindex) {
7    return _mm256_i32gather_epi32(base_addr, vindex, 4);
8  }
9  template <> inline __m256i _mm_gather(
10   const uint32_t *base_addr, __m256i vindex) {
11   return _mm256_i32gather_epi32(
12     reinterpret_cast<const int32_t*>(base_addr), vindex, 4);
13 }
14 template <> inline __m256i _mm_gather(
15   const int64_t *base_addr, __m128i vindex) {
16   return _mm256_i32gather_epi64(
17     reinterpret_cast<const long long*>(base_addr), vindex, 8);
18 }
19 template <> inline __m256i _mm_gather(
20   const uint64_t *base_addr, __m128i vindex) {
21   return _mm256_i32gather_epi64(
```

```
22      reinterpret_cast<const long long*>(base_addr), vindex, 8);
23  }
24  template <> inline __m256i _mm_gather(
25    const float *base_addr, __m256i vindex) {
26    return _mm256_castps_si256(_mm256_i32gather_ps(base_addr, vindex, 4));
27  }
28  template <> inline __m256i _mm_gather(
29    const double *base_addr, __m128i vindex) {
30    return _mm256_castpd_si256(_mm256_i32gather_pd(base_addr, vindex, 8));
31  }
32
33  template <> inline __m128i _mm_gather(
34    const int32_t *base_addr, __m256i vindex) {
35    return _mm256_i64gather_epi32(base_addr, vindex, 4);
36  }
37  template <> inline __m128i _mm_gather(
38    const uint32_t *base_addr, __m256i vindex) {
39    return _mm256_i64gather_epi32(
40      reinterpret_cast<const int32_t*>(base_addr), vindex, 4);
41  }
42  template <> inline __m256i _mm_gather(
43    const int64_t *base_addr, __m256i vindex) {
44    return _mm256_i64gather_epi64(
45      reinterpret_cast<const long long*>(base_addr), vindex, 8);
46  }
47  template <> inline __m256i _mm_gather(
48    const uint64_t *base_addr, __m256i vindex) {
49    return _mm256_i64gather_epi64(
50      reinterpret_cast<const long long*>(base_addr), vindex, 8);
51  }
52  template <> inline __m128i _mm_gather(
53    const float *base_addr, __m256i vindex) {
54    return _mm_castps_si128(_mm256_i64gather_ps(base_addr, vindex, 4));
55  }
56  template <> inline __m256i _mm_gather(
57    const double *base_addr, __m256i vindex) {
58    return _mm256_castpd_si256(_mm256_i64gather_pd(base_addr, vindex, 8));
59  }
60  #endif
```

A.3.5 Constants

The `generateLaneFactors` functions load constants needed by the vectorized k-ary search.

Listing A.13: Definition of `generateLaneFactors`

```
1  template <typename IndexVector, typename IndexType>
2    IndexVector generateLaneFactors();
3  template <> inline __m128i generateLaneFactors<__m128i, int16_t>()
4    { return _mm_set_epi16(8, 7, 6, 5, 4, 3, 2, 1); }
5  template <> inline __m128i generateLaneFactors<__m128i, int32_t>()
```

```
6   { return _mm_set_epi32(4, 3, 2, 1); }
7  template <> inline __m128i generateLaneFactors<__m128i, int64_t>()
8   { return _mm_set_epi64x(2, 1); }
9  #ifdef USE_AVX2
10 template <> inline __m256i generateLaneFactors<__m256i, int16_t>() {
11   return _mm256_set_epi16(16, 15, 14, 13, 12, 11, 10, 9,
12     8, 7, 6, 5, 4, 3, 2, 1);
13 }
14 template <> inline __m256i generateLaneFactors<__m256i, int32_t>()
15   { return _mm256_set_epi32(8, 7, 6, 5, 4, 3, 2, 1); }
16 template <> inline __m256i generateLaneFactors<__m256i, int64_t>()
17   { return _mm256_set_epi64x(4, 3, 2, 1); }
18 #endif
```

Bibliography

[BBHS14] David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward hardware-sensitive database operations. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 229–234, 2014. (cited on Page 2)

[BBS15] David Broneske, Sebastian Breß, and Gunter Saake. Database scan variants on modern CPUs: A performance study. In *In-Memory Data Management and Analytics (IMDM)*, pages 97–111. Springer, 2015. (cited on Page 7)

[BFJ02] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 39–48. Society for Industrial and Applied Mathematics (SIAM), 2002. (cited on Page 92)

[BKSS17] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Accelerating multi-column selection predicates in main-memory–the elf approach. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 647–658. IEEE, 2017. (cited on Page 95)

[Bot62] Hermann Bottenbruch. Structure and use of ALGOL 60. *Journal of the ACM (JACM)*, 9:161–221, 1962. (cited on Page 14)

[Can17] Fabio Cannizzo. Fast and vectorizable alternative to binary search in O(1) applicable to a wide domain of sorted arrays of floating point numbers. https://arxiv.org/abs/1506.08620v2, March 2017. (cited on Page 91 and 92)

[CD97] Michel Cekleov and Michel Dubois. Virtual-address caches part 1: problems and solutions in uniprocessors. volume 17, pages 64–71. IEEE, 1997. (cited on Page 5)

[Duf88] Tom Duff. Website, August 1988. Available online at http://www.lysator.liu.se/c/duffs-device.html; visited on May 9th, 2017. (cited on Page 29)

[Gie16] Fabian Giesen. SSE: mind the gap! Website, April 2016. Available online at https://fgiesen.wordpress.com/2016/04/03/sse-mind-the-gap/; visited on April 18th, 2017. (cited on Page 9)

[GR77] Gaston H. Gonnet and Lawrence D. Rogers. The interpolation-sequential search algorithm. *Information Processing Letters*, 6(4):136–139, 1977. (cited on Page 95)

[GRG80] Gaston H. Gonnet, Lawrance D. Rogers, and J. Alan George. An algorithmic and complexity analysis of interpolation search. *Acta Informatica*, 13(1):39–52, 1980. (cited on Page 92)

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth edition, 2011. (cited on Page 4)

[Int16a] Intel 64 and IA-32 Architectures Optimization Reference Manual, November 2016. (cited on Page 3, 4, 10, and 82)

[Int16b] Intel 64 and IA-32 Architectures Software Developer's Manual, December 2016. (cited on Page 7, 9, 56, and 59)

[JBB+10] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 151–162. IEEE, 2010. (cited on Page 4)

[KCS+10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 339–350. ACM, 2010. (cited on Page 92)

[Khu12] Paul-Virak Khuong. Binary search is a pathological case for caches. Website, July 2012. Available online at https://www.pvk.ca/Blog/2012/07/30/binary-search-is-a-pathological-case-for-caches/; visited on June 30h, 2017. (cited on Page 91)

[Kie53] Jack Kiefer. Sequential minmax search for a maximum. *Proceedings of the American Mathematical Society (AMS)*, 4(3):502–506, 1953. (cited on Page 16)

[KM17] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching. *Journal of Experimental Algorithms (JEA)*, 22(1):1.3:1–1.3:39, 2017. (cited on Page 91 and 92)

[Knu98] Donald Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, second edition, 1998. (cited on Page 14, 15, 16, 17, and 91)

[Les83] R. Lesuisse. Some lessons drawn from the history of the binary search algorithm. *The Computer Journal*, 26(2):154–163, 1983. (cited on Page 15)

[Pri71] C.E. Price. Table lookup techniques. *ACM Computer Surveys (CSUR)*, 3(2):49–64, June 1971. (cited on Page 92)

[Pul11] Matt Pulver. Binary search revisited. Website, September 2011. Available online at http://eigenjoy.com/2011/09/09/binary-search-revisited/; visited on June 30h, 2017. (cited on Page 91)

[SGL09] Benjamin Schlegel, Rainer Gemulla, and Wolfgang 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. (cited on Page 10, 15, 16, 18, 20, 21, 23, and 92)

[SS85] Nicola Santoro and Jeffrey B. Sidney. Interpolation-binary search. *Information Processing Letters*, 20:179–181, 1985. (cited on Page 95)

[SSH11] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken: Implementierungstechniken*. MITP, third edition, 2011. (cited on Page 4)

[Tan05] Andrew S. Tanenbaum. *Computerarchitektur. Strukturen - Konzepte - Grundlagen*. Pearson Studium, fifth edition, 2005. (cited on Page 7)

[ZAF07] Mohamed Zahran, Kursad Albayraktaroglu, and Manoj Franklin. Non-inclusion property in multi-level caches revisited. volume 14, page 99. ISCA, 2007. (cited on Page 4)

[ZDJ04] Ying Zehng, Brian T. Davis, and Metthew Jordan. Performance evaluation of exclusive cache hierarchies. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–96. IEEE, 2004. (cited on Page 4)

[ZF15] Steffen Zeuch and Johann-Christoph Freytag. Selection on modern CPUs. In *In-Memory Data Management and Analytics (IMDM)*, pages 5:1–5:8. ACM, 2015. (cited on Page 91)

[ZHF14] Steffen Zeuch, Frank Huber, and Johann-Christoph Freytag. Adapting tree structures for processing with SIMD instructions. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2014. (cited on Page 10)

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 145–156. ACM, 2002. (cited on Page 15)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 26. Juli 2017