SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures

Hiroshi Inoue[†‡] Kenjiro Taura[‡]

[†]IBM Research – Tokyo, NBF Toyosu Canal Front, 5-6-52, Toyosu, Tokyo, 135-8511, Japan

[‡]University of Tokyo, 7-3-1 Hongo, Tokyo, 113-0033, Japan

inouehrs@jp.ibm.com, tau@eidos.ic.i.u-tokyo.ac.jp

ABSTRACT

This paper describes our new algorithm for sorting an array of structures by efficiently exploiting the SIMD instructions and cache memory of today's processors. Recently, multiway mergesort implemented with SIMD instructions has been used as a high-performance in-memory sorting algorithm for sorting integer values. For sorting an array of structures with SIMD instructions, a frequently used approach is to first pack the key and index for each record into an integer value, sort the key-index pairs using SIMD instructions, then rearrange the records based on the sorted key-index pairs. This approach can efficiently exploit SIMD instructions because it sorts the key-index pairs while packed into integer values; hence, it can use existing high-performance sorting implementations of the SIMD-based multiway mergesort for integers. However, this approach has frequent cache misses in the final rearranging phase due to its random and scattered memory accesses so that this phase limits both single-thread performance and scalability with multiple cores. Our approach is also based on multiway mergesort, but it can avoid costly random accesses for rearranging the records while still efficiently exploiting the SIMD instructions. Our results showed that our approach exhibited up to 2.1x better single-thread performance than the key-index approach implemented with SIMD instructions when sorting 512M 16-byte records on one core. Our approach also yielded better performance when we used multiple cores. Compared to an optimized radix sort, our vectorized multiway mergesort achieved better performance when the each record is large. Our vectorized multiway mergesort also yielded higher scalability with multiple cores than the radix sort.

1. INTRODUCTION

Sorting is a fundamental operation for many software systems; hence, a large number of sorting algorithms have been devised. Recently, multiway mergesort implemented with SIMD instructions has been used as a high performance in-memory sorting algorithm for sorting 32-bit or 64-bit integer values [1-9]. By using the SIMD instructions efficiently in the merge operation, multiway mergesort outperforms other comparison-based sorting algorithms, such as quicksort, that are not suitable for exploiting the SIMD instructions. Multiway mergesort extends the standard (2-way) mergesort by reading input data from more than two streams and writing the output into one output stream. This reduces the number of merge stages from $\log_2(N)$ to $\log_k(N)$, where k is the number of ways and N is the total number of records to sort. Hence, multiway mergesort can reduce the required main memory bandwidth compared to the standard 2-way mergesort because each stage of the mergesort accesses all the data.

In real workloads, sorting is mostly used to rearrange structures based on a sorting key included in each structure. We call each structure to be sorted a *record* in this paper. For sorting large records using SIMD instructions, a common approach is to first pack the key and index for each record into an integer value, such as combining each 32-bit integer key and a 32-bit index into one 64-bit integer value. The key-index pairs are then sorted using SIMD instructions, and the records are finally rearranged based on the sorted key-index pairs [3]. This *key-index approach* can efficiently exploit SIMD instructions because it sorts the key-index pairs while packed into integer values, allowing it to use existing high-performance sorting implementations of SIMD-based multiway mergesort for integers. However, the key-index approach causes frequent cache misses in the final rearranging phase due to its random memory accesses, and this phase limits both single-thread performance and scalability with multiple cores, especially when the size of each record is smaller than the cache line size of the processor. When the record size is small, only a part of the transferred data is actually used; thus, a large amount of unused data wastes the memory bandwidth.

A more straightforward approach is directly sorting the records without generating the key-index pairs by moving all of the records for each comparison. Unlike the key-index approach, this *direct approach* does not require random memory accesses to rearrange the records. However, it is difficult to efficiently use SIMD instructions for the direct approach because reading keys from multiple records into a vector register scatters the memory accesses, which incurs additional overhead and offsets the benefits of the SIMD instructions.

In this paper, we report on a new stable sorting algorithm that can take advantage of SIMD instructions while avoiding the frequent cache misses caused by the random memory accesses. Our new algorithm, based on multiway mergesort, does the key encoding and record rearranging for each multiway merge stage, while the key-index approach does the encoding only at the beginning of the entire sorting operation and record rearrangement at the end. In each multiway merge stage, which reads input data from k input streams (k = 32 in our implementation) and writes the merged results into one output stream, we read the key from each record and pack the key and streamID that the records came from into an integer value (an

Proceedings of the VLDB Endowment, Vol. 8, No. 11
Copyright 2015 VLDB Endowment 2150-8097/15/07

intermediate integer), merge the intermediate integers using SIMD instructions, and finally rearrange the records based on the streamIDs encoded in the intermediate integers. Unlike the key-index approach, if the number of ways *k* is not too large compared to the numbers of cache and TLB entries, our approach will not cause excessive cache misses. We also describe our techniques to increase data parallelism within one SIMD instruction by using 32-bit integers as the intermediate integers instead of using 64-bit integers.

Our results on Xeon (SandyBridge-EP) showed that our approach implemented with the SSE instructions outperformed both the key-index approach and the direct approach also implemented with the SSE instructions by 2.1 and 2.3x, respectively, when sorting 512 million 16-byte records (4-byte key and 12-byte payload) using one core. Our approach exhibited better performance scalability with an increasing number of cores than the key-index approach unless main memory bandwidth was saturated. It also outperformed by 3.3x the std::stable_sort function of the STL delivered with gcc, which uses multiway mergesort without SIMD. Comparing our vectorized multiway mergesort of our approach against an optimized radix sort, our algorithm exhibited better performance when the size of a record was larger than 16 bytes on 1 core while the radix sort was almost comparable to our algorithm when each record was small (e.g. 8 bytes). With the key-index approach, the vectorized mergesort outperformed the radix sort only when the record was larger than 128 bytes. Hence, our new approach makes the vectorized mergesort a better choice than the radix sort in many workloads. Also, our algorithm yielded higher performance scalability with increasing numbers of cores compared to the radix sort due to the better memory access locality.

The main contribution of our work is a new approach in the multiway mergesort for sorting an array of structures. We can effectively exploit the SIMD instructions while avoiding the random memory accesses. Avoiding the waste of memory bandwidth due to random memory accesses is quite important with multicore processors because the total computing capability of the cores in a processor has been growing faster than the memory bandwidth to the system memory.

The rest of the paper is organized as follows. Section 2 discusses related work and reviews existing SIMD-based multiway mergesort for sorting integer values and structures. Section 3 describes our approach for sorting the structures. Section 4 gives a summary of our results. Finally, Section 5 summarizes the paper.

2. BACKGROUND

This section discusses related work and gives details about the vectorized multiway mergesort [1], the basis of our new algorithm.

2.1 Related work

Sorting is one of the most important operations in many workloads, and many sorting algorithms have been proposed. To efficiently exploit SIMD instructions and the multiple cores of today's processors, multiway mergesort has gained popularity as a high-performance in-memory sorting algorithm for sorting 32-bit or 64-bit integer values in database systems [7-9] or in distributed sorting systems running on large-scale supercomputers [5] or clusters [6]. Because many widely used sorting algorithms, such as quicksort, are not suitable for exploiting the SIMD instructions, multiway mergesort outperforms them by exploiting the SIMD instructions. In the vectorized mergesort, we can reduce the overhead of branch mispredictions by integrating a branchless sorting network implemented on vector registers into the standard comparison-based merge operation.

Inoue *et al.* [1] introduced a vectorized multiway mergesort algorithm, and their implementation on Cell BE and PowerPC 970 outperformed the bitonic mergesort implemented with SIMD instructions, the vendor's optimized library, and STL's sort function by more than 3 times when sorting 32-bit integers. Chhungani *et al.* [2] improved that vectorized mergesort by using a sorting network larger than the width of vector registers to increase instruction-level parallelism. Their implementation for 32-bit integers on a quad-core Core2 processor using the 4-wide SSE instruction set exhibited better performance than other algorithms. Satish *et al.* [3] compared the vectorized mergesort against the radix sort and found that the radix sort outperformed the vectorized mergesort unless the key size was larger than 8 bytes on both the latest CPUs and GPUs. Using a new analytic model, they also showed that the vectorized mergesort may outperform the radix sort on future processors due to its efficiency with SIMD instructions and its lower memory bandwidth requirements.

It is possible to sort a large number of records using a sorting network, such as a bitonic mergesort or an odd-even mergesort [10], without combining them with the standard comparison-based mergesort. These sorting networks can be implemented efficiently using the SIMD instructions on CPUs [11] or GPUs [12]. However, due to the larger computational complexity of these algorithms, they cannot compete with the performance of the vectorized mergesort for large amounts of data.

This paper focuses on the performance of the vectorized multiway mergesort for sorting an array of structures instead of an integer array. To sort a large array of structures, there are two approaches. One approach [3] is to first pack the key and index of each record into a 64-bit integer value (e.g. 32-bit key in the higher bits and a 32-bit index in the lower bits), sort the key-index pairs as integer values, then rearrange the records based on the sorted key-index pairs. This key-index approach can efficiently exploit SIMD instructions because sorting is done for key-index pairs that are packed into integer values. However, the final rearranging phase may cause frequent cache and TLB misses because it accesses main memory at random. Especially when the record size is much smaller than the size of a cache line of the processor, the random accesses during the rearrangement phase use the memory bandwidth inefficiently because only a part of the data in each cache line are actually used. Also, the hardware prefetcher of the processor does not work for the random memory accesses. Another approach is to directly sort the records without generating the key-index pairs by including each entire record during the sorting. This direct approach is not negatively affected by the overhead of random memory accesses. However, it is difficult to efficiently implement a direct approach using SIMD instructions because the keys are not stored contiguously in memory; hence, we need to use costly gather operations to load the keys into the vector registers. Because we cannot read the entire records into a vector register to fully exploit the data parallelism of the SIMD instructions, we load the keys into a vector register using the gather operation and associate the keys with the record locations. After merging them in the vector registers, we copy the entire records based on the merged results. The overhead due to the gather operations offsets the benefits we can gain from the SIMD instructions: data parallelism and reduced branch mispredictions. Also, the direct approach does more memory copying because it moves entire records during the sorting, while the key-index approach only moves integers. Our SIMD- and cache-friendly approach in the vectorized multiway mergesort resolves the problems these two approaches have in sorting structures.

Kim *et al.* [7] used the key-index approach with the vectorized mergesort to efficiently execute the sort-merge join in a DBMS. They concluded that the sort-merge join will be better than the hash join on future processors with wider SIMD and limited memory bandwidth. Our algorithm can potentially improve the performance of their sort-merge join by avoiding the overhead of rearranging after the join operation if the join operation makes a large number of outputs. They also reported that the performance of the sort-merge join will decrease if a key and index pair cannot fit into a 64-bit value. Our approach can avoid this problem even for a large number of inputs because it does not encode the record ID directly, as we discuss in Section 3.2.

We improved the performance of sorting by using only a part of the key of each record to increase the data parallelism within each SIMD instruction. Zhou *et al.* [13] and Inoue *et al.* [14] also used a similar idea to improve the performance of the nested-loop join and the set intersection (merge join).

2.2 Vectorized Multiway Mergesort for Sorting Integers

2.2.1 Mergesort with SIMD instructions

To efficiently exploit SIMD instructions in a 2-way merge of integer values, a standard technique involves combining an SIMD sorting network with a comparison-based merge operation [1]. Figure 1(a) shows the vectorized merge algorithm for two vector integer arrays `va` and `vb`, assuming that the size of each vector register is four, i.e., 32-bit integers in a 128-bit SIMD architecture. In each iteration, this code executes a merge operation of two vector registers, `vMin` and `vMax`, by using a sorting network, such as an odd-even merge or a bitonic merge, implemented with the SIMD minimum and maximum instructions to avoid conditional branches (`vector_merge` method). Then the contents of `vMin`, the four smallest values, are stored in memory as output. The pointer is advanced for the input stream whose next value is smallest. Here, `a[aPos*4]` (`b[bPos*4]`) is the first element of the next vector integer to read from `va` (`vb`). Figure 1(b) shows the data flow of the bitonic merge operation as an example of an implementation of the `vector_merge` method. When one vector register can hold fewer values, i.e., 64-bit integers in a 128-bit SIMD architecture, we can combine multiple vector registers to emulate a longer vector register. Implementing a sorting network whose input size is larger than the actual vector register size is better for higher performance because it hides the latency of the instructions by overlapping multiple comparisons to improve instruction-level parallelism [2].

There are two benefits from SIMD instructions in mergesort. The obvious benefit of SIMD instructions is the data parallelism available in each SIMD instruction. Second, the use of SIMD instructions reduces the number of conditional branches to select an array for the next data, which are difficult to predict in the branch predictor of the processors for random input data; therefore, we can reduce the overhead of branch mispredictions.

2.2.2 Multiway Mergesort

Multiway mergesort [15] enhances the standard (2-way) mergesort. It repeats the multiway merge operation, which reads input records from more than two data streams and outputs the merged records into one output stream, to sort all the input records. Multiway mergesort reduces the number of merging stages from $\log_2(N)$ to $\log_k(N)$, where N is the number of records and k is the number of ways. Mergesort scans all the elements in each merging stage; thus, using larger k reduces both the number of stages and amount of required memory bandwidth. This lower memory bandwidth is the key for both higher single-thread performance and more scalability with an increasing number of cores.

To reduce the required memory bandwidth to the system memory in multiway mergesort using the 2-way vectorized merge operation shown in Figure 1 as a building block, we execute multiway merge operations consisting of multiple 2-way merge operations in a streaming manner using small memory buffers (4 KB each in our implementation) that can fit within the processor's cache memory. Figure 2 illustrates how we implement the

(a)
```
aPos = bPos = outPos = 0;
vMin = va[aPos++];
vMax = vb[bPos++];
while (aPos < aEnd && bPos < bEnd) {
  /*  merge vMin and vMax */
  vector_merge(vMin, vMax);

  /*  store the smaller vector as output*/
  vMergedArray[outPos++] = vMin;

  /*  load next vector and advance pointer    */
  /*  a[aPos*4] is first element of va[aPos] */
  /*  and b[bPos*4] is that of vb[bPos]       */
  if (a[aPos*4] < b[bPos*4])
      vMin = va[aPos++];
  else
      vMin = vb[bPos++];
}
```

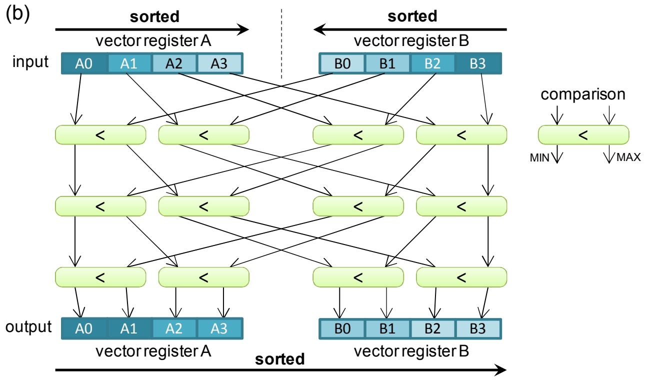
(b)


Figure 1. (a) Pseudocode of vectorized merge operation and (b) data flow of bitonic merge operation for two vector registers

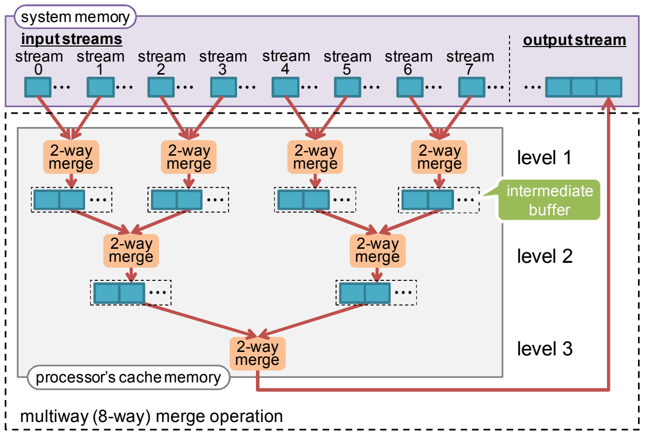


Figure 2. Overview of multiway merge operation with SIMD. Number of ways $k = 8$ in this example. It is efficient with SIMD for integers, but not for large structures.

multiway merge operation in the cache memory. We use $k = 8$ (8-way merge) as an example. One 8-way merge operation includes three levels of 2-way merge operations, as shown in the figure. We execute this merge operation in a single thread. The intermediate results are stored in small memory buffers that can fit in the cache memory; hence, we need to access the system memory only at the first and last levels. We first fill all the intermediate buffers. Then we execute last-level 2-way merge operations until one of the two input buffers for the last-level merge operation becomes empty. When an intermediate buffer becomes empty, we refill the intermediate buffer by going back to the previous level of the 2-way merging. After filling the buffer, we restart the merge operation with the new records. We repeat these operations until we reach the ends of all of the input streams.

2.2.3 Combsort with SIMD for small blocks

For the vectorized mergesort to operate efficiently, each input stream must have a sufficient number of records. Hence, we first use another sorting algorithm for sorting small blocks then execute the multiway merge to merge these sorted blocks.

For the initial sorting of the small blocks, we can use a vectorized combsort [1]. The combsort [16] extends bubble sort by comparing non-adjacent elements, in contrast to the bubble sort, which compares only adjacent elements. Comparing values with larger separations drastically improves performance because each value moves toward its final position more quickly. The separation is divided by a number so called *shrink factor* (1.3 in our implementation) in each iteration until it becomes one. Then the final loop with the separation of one is repeated until all of the data is sorted. The average computational complexity of combsort approximates $N\cdot\log(N)$ [16]. The vectorized combsort can eliminate almost all the data-dependent conditional branches and hence does not suffer from the branch misprediction overhead. Due to its simplicity and smaller misprediction overhead, the vectorized combsort can exhibit higher performance than the vectorized multiway mergesort for sorting a small block. However, because of its poor memory-access locality the performance of the vectorized combsort degrades drastically for data that are too large for the processor's cache memory. That is why the vectorized combsort is most suitable for sorting small blocks before executing the vectorized multiway mergesort.

2.2.4 Parallelization with multiple threads

By using the two algorithms, vectorized multiway (k-way) mergesort and vectorized combsort, we can sort all the data using p threads, where p is the number of threads, in two phases:

(1) Divide all the data to be sorted into p blocks and assign one thread for each block to sort in parallel using multiple threads. Each thread independently executes the vectorized mergesort and switches to the vectorized combsort when each input sequence becomes smaller than the predefined size (b records).

(2) Merge the p sorted blocks with the vectorized k-way mergesort using multiple threads.

In this first phase, each thread can run without synchronizing with the other threads. For parallelizing the second phase, multiple threads must cooperate on one multiway merge operation to fully exploit thread-level parallelism because the number of blocks becomes smaller than the number of threads. We divide each input stream into p sub-streams by using binary search to parallelize one merge operation. Then the i-th thread merges with the i-th sub-streams from each of the k input streams. Additionally, it rebalances the data among threads if the data size for each thread is not balanced [18]. After dividing the input streams into sub-streams, threads run without synchronizing with each other. Because the block size b is a constant independent from N, the entire computational complexity is determined by the complexity of the vectorized mergesort, which is $O(N\cdot\log(N))$, even in the worst case.

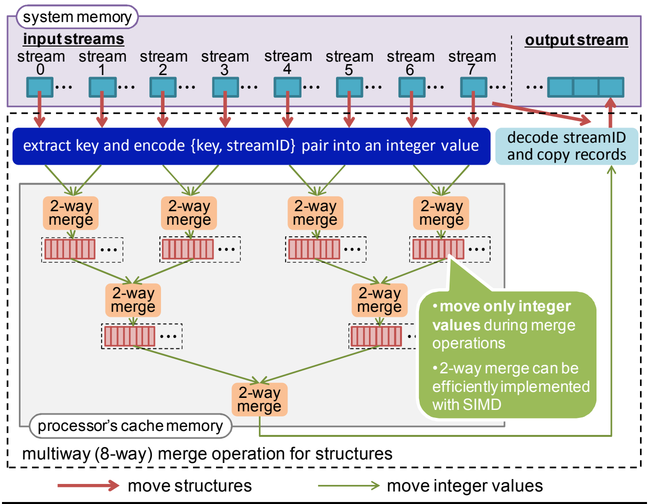

system memory
input streams
stream 0
stream 1
stream 2
stream 3
stream 4
stream 5
stream 6
stream 7
output stream
extract key and encode {key, streamID} pair into an integer value
decode streamID and copy records
2-way merge
•move only integer values during merge operations
•2-way merge can be efficiently implemented with SIMD
processor's cache memory
multiway (8-way) merge operation for structures
move structures
move integer values

Figure 3. Overview of SIMD- and cache- friendly multiway merge operation of our approach. $k = 8$ in this example.

3. OUR APPROACH FOR SORTING STRUCTURES

As described in Section 2, the two existing approaches to sort an array of structures have different drawbacks; the key-index approach is not cache friendly and the direct approach is not SIMD friendly. In this paper, we propose an approach with a vectorized multiway mergesort that is both SIMD and cache friendly. In this section, we assume that the size of each record is fixed. We initially assume that the size of each key is 32 bits before we consider how to handle larger keys.

3.1 SIMD- and Cache-Friendly Multiway Merge Operation

To avoid costly rearrangements of the records, we use a hybrid approach combining the direct and key-index approaches. Figure 2 shows an overview of the multiway merge operation in the direct approach when a record to be sorted is a structure. We extend this multiway merge in Figure 2 to make it more efficient with SIMD instructions.

The problem with the multiway merge in Figure 2 for merging structures is that it requires gather operations to read the keys from multiple records into vector registers because each record includes a payload, a part that is not used for sorting, in addition to the sorting key. The SIMD instructions of a modern processor perform best when the data are loaded from and stored to contiguous memory.

Figure 3 shows an overview of the SIMD- and cache-friendly multiway merge operation of our approach. To make the overhead of the gather operation as small as possible, we encode the key and streamID into integer values (*intermediate integers*) at the beginning of the multiway merge operation, i.e. when the 2-way merge operations in the first level read the records from main memory, and merge them by using the SIMD instructions. At the end of the multiway merge, we rearrange the records based on the merged intermediate integers. In this step, we sequentially read

Table 1. Summary of three approaches with multiway mergesort

	data type to sort	number of memory copies for each record	memory access pattern
Key-index approach	integer that encodes {key and index} pair	**only once** – to move each record at end of sorting	**random access; not cache friendly**, especially when each record is smaller than cache line
Our approach	integer that encodes {key and streamID} pair	**$\log_k(N)$ times** – to move each record at end of each k-way multiway merge stage	sequential access if k is not too large
Direct approach	**entire record (structure); not SIMD friendly**	**$\log_2(N)$ times** – to move each record at each 2-way merge stage	sequential access

- N: number of records to sort, k: number of ways used in multiway merge operation
- one additional copy per record may be necessary to copy back the records from an temporary array to the destination

records from k input streams and sequentially write them into one merged output stream. This avoids excessive cache and TLB misses due to random accesses to the system memory if k is not too large. Because the merge operation is executed for intermediate integers, we do not need gather operations to read the keys in all the levels except for the first level, which reads the records from the system memory and encodes them into the intermediate integers. Figure 4 compares the three approaches. Comparing our approach against the key-index approach, we rearrange records multiple times instead of only once to avoid excess cache misses in the rearranging phase in trade for larger memory copy overheads. When the record size is smaller than the cache line size, the cost of random memory accesses exceeds the cost of additional (sequential) memory copy of records with our approach.

With our approach, the number of ways (k) is an important parameter to achieve high performance. Because we need to execute the key extraction at the beginning of the multiway merge and copy the entire records from an input stream to the output stream at the end of multiway merge operation, a larger k reduces the overhead for extracting keys and copying records. When k becomes too large, however, copying the records at the end of the multiway merge operation may potentially cause many cache and TLB misses. Hence k must be smaller than the number of cache or TLB entries to avoid this problem. Also a larger k makes the total size of the intermediate buffers used for the multiway merge operation larger. Thus larger k may also increase the cache misses during the merge operation.

The key-index approach is almost equivalent to our approach with $k = N$. In another extreme case, our approach becomes almost identical to the direct approach when $k = 2$, i.e., rearranging records in every 2-way merge operation. Hence, our approach is a generalization of the two existing approaches parameterized by parameter k, but we found that the best value for k is between the two extreme values corresponding to these two approaches. In our implementation, we used $k = 32 = 2^5$. This means each multiway merge operation includes five levels, and each level executes a 2-way vectorized merge operation. We can reduce the overhead for extracting keys (including the cost of the gather operations) and copying the objects to only 1/5 compared to the direct approach. Using the large k reduces these costs further, but the increased cache misses result in a net performance decline, as shown in the performance evaluations. Table 1 summarizes the three approaches.

3.2 Key and StreamID Encoding

With our approach, we encode each pair consisting of a key and its streamID (in the range of 0 to k-1) into intermediate integer values instead of a key and index pair (in the range of 0 to N-1) used in the key-index approach. The k is typically much smaller than N. We only need to encode the streamID instead of the index

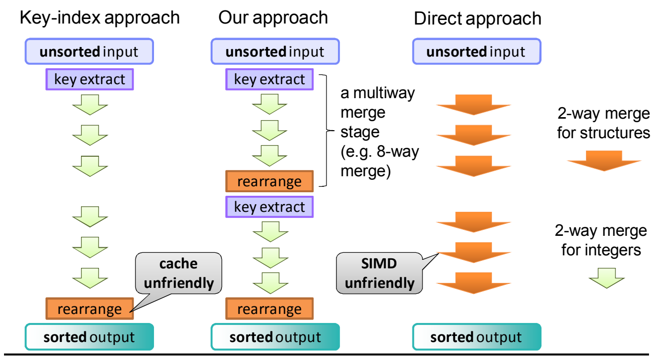


Figure 4. Overview of three approaches

into the intermediate integer because the records within each input stream are already sorted; therefore, they cannot be reversed in the final output stream. We can guarantee the merged records are sorted by maintaining a pointer to the next records to copy for each input stream and incrementing a stream's pointer when we copy a record from that stream to the output. We use higher bits to encode the keys to sort the integers based on the encoded keys and encode the streamIDs in the lower bits.

An important advantage of our approach over the existing key-index approach beyond performance is that our approach can sort larger arrays than the key-index approach. With the current key-index approach, only 2^{32} (= 4 G) records can be sorted because we only have 32 bits to encode the index when the key is 32 bits and the intermediate integer is 64 bits. With our approach, there is no limit on the total number of records because we only encode the streamID, which is a configurable parameter independent of the total number of records, into the intermediate integer values. We are limited by the number of ways k instead of the number of records N. However, this is not a problem because k must be smaller than the number of cache and TLB entries to achieve good performance and hence k cannot be too large. We used $k = 32$ in our evaluation and only 5 bits were needed for encoding the streamID. This means we can encode a larger key, up to 59 (= 64 – 5) bits, and the streamID into each 64-bit integer.

3.3 Optimization Techniques in Vectorized Merge Operation

In this section we describe some optimization techniques to improve the efficiency of the SIMD instructions and the overall execution performance of our approach from Section 3.2.

3.3.1 Increasing data parallelism

As discussed in Section 3.2, our approach has an advantage over the existing key-index approach in the size of the ID (streamID or index) encoded in the intermediate integers. By exploiting this advantage, we can increase the data parallelism within each SIMD instruction by encoding a key-streamID pair into a 32-bit integer instead of a 64-bit integer. When we use 128-bit vector registers (such as the SSE instruction set we used in our tests), we can execute 4 operations at one time for 32-bit integers but only 2 operations for 64-bit integers. Hence, using 32-bit integers during the vectorized merge operation shown in Figure 3 boosts the performance over the same vectorized merge operation using 64-bit integers by processing a larger number of elements at one time, yielding shorter path lengths and reduced branch misprediction overhead.

When encoding the key-streamID pair into a 32-bit integer, we can only include a part of the 32-bit key. In the current implementation with $k = 32$, we can encode 27 bits out of the 32-bit key. If the partial keys of multiple records have the same value while the entire keys are not the same (*partial-key conflict*), the order of the records with the conflicting partial key may be incorrect in the merged output stream. Hence, we need to confirm that the records in the merged output stream are correct by using the entire key. We merge the records based on the partial keys using SIMD instructions then check and possibly adjust the order of the records using the entire keys as we rearrange the records in the final phase of the multiway merge. We execute this check as an insertion sort using scalar (non-SIMD) comparisons.

In this partial-key technique, an important question is how best to select the partial key from the entire key. The most naive way to select the bits to use in sorting from the key is to extract the most significant bits of the key. However, this naive selection might cause significant performance degradation. One obvious pathological example would be if all the partial keys from all the records have the same value, though the other bits differ, i.e. the keys are actually 5-bit integers stored in the 32-bit key field of the records. To avoid such cases and maximize performance, we encode the key in the following way:

1) identify the minimum and maximum key values (*min* and *max*) in the records to merge by checking the first and last records of each input stream,
2) calculate *key_pos* = *count_leading_zeros*(*max* - *min*), and
3) encode the key (*key*) and streamID (*sid*) of each record into an intermediate integer *ii* as *ii* = ((((*key* - *min*) << *key_pos*) & *key_mask*) | *sid*).

For example, if *max* = 0x1000000F and *min* = 0x10000000, then the naive partial key selection, which just selects the most significant bits, does not work well for sorting because all the keys share the same value in the most significant part. However, we can obtain good performance by selecting better partial keys; *key_pos* = *count_leading_zeros*(0xF) = 28; hence, for example, key 0x10000008 is encoded as *ii* = ((0x8 << 28) | *sid*).

Our encoding scheme can reduce the frequency of the partial-key conflicts, but it cannot totally prevent them. Therefore, when we use 32-bit integers during the vectorized merge operation to increase data parallelism, we still need to confirm that the complete records in the merged output stream are correct by using the entire key. We do this check when we copy the records from an input stream to the output stream with a scalar comparison. When we output a record, we compare the entire key of that record against the key of the previous record. If the two records are not in the correct order, we swap the two elements and repeat the comparison; thus, finding the correct position as an insertion sort. Our implementation ensures that the sorted results are not only sorted but also stable, i.e., the order of records having the same key is not altered by sorting. Because we do this scalar check only at the end of each multiway merge operation, the overhead due to this check is quite small compared to the benefits

(a)
```
merge_4x4_32bit(__m128i &vA,    __m128i &vB,        // input 1 & 2
                __m128i &vMin,  __m128i &vMax) {    // output
    __m128i vTmp; // temporary register
    vTmp = _mm_min_epu32(vA, vB);
    vMax = _mm_max_epu32(vA, vB);
    vTmp = _mm_alignr_epi8(vTmp, vTmp, 4);
    vMin = _mm_min_epu32(vTmp, vMax);
    vMax = _mm_max_epu32(vTmp, vMax);
    vTmp = _mm_alignr_epi8(vMin, vMin, 4);
    vMin = _mm_min_epu32(vTmp, vMax);
    vMax = _mm_max_epu32(vTmp, vMax);
    vTmp = _mm_alignr_epi8(vMin, vMin, 4);
    vMin = _mm_min_epu32(vTmp, vMax);
    vMax = _mm_max_epu32(vTmp, vMax);
    vMin = _mm_alignr_epi8(vMin, vMin, 4);
}

merge_8x8_32bit(__m128i &vA0,   __m128i &vA1,      // input 1
                __m128i &vB0,   __m128i &vB1,      // input 2
                __m128i &vMin0, __m128i &vMin1,    // output
                __m128i &vMax0, __m128i &vMax1) {  // output
// 1st step
    merge_4x4_32bit(vA1,vB1,vMin1,vMax1);
    merge_4x4_32bit(vA0,vB0,vMin0,vMax0);
// 2nd step
    merge_4x4_32bit(vMax0,vMin1,vMin1,vMax0);
}
```

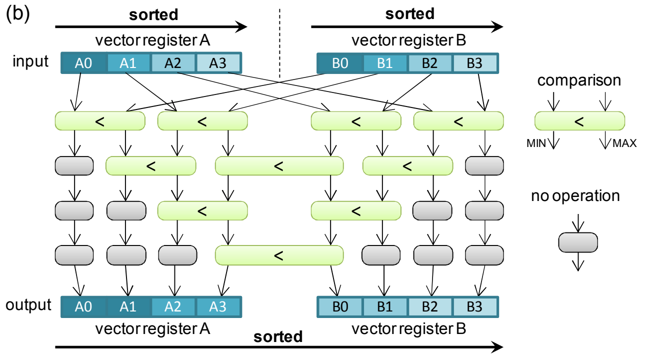


Figure 5. (a) Pseudo code of in-register vector merge for 32-bit integers (b) data flow of merge_4x4_32bit method and (c)

of using the 4-wide SIMD instructions during the merge operation, unless the partial-key conflicts are quite frequent. When the number of records to merge in one multiway merge operation becomes too large, the frequency of the partial-key conflicts could potentially increase too much. To avoid such a case, we use 4-wide SIMD only when the total number of records to merge (i.e. the total number of records included in all the input streams) is small enough to justify the overhead of the check. We empirically determine the threshold for 4-wide SIMD in Section 4.

3.3.2 Sorting network in vector registers

The vector merge operation in vector registers is a key to the vectorized mergesort. The merge operations in the vector registers can be implemented without conditional branches by using the vector min/max instructions and vector permute instructions.

For merging the 32-bit integer values in two vector registers with 128-bit SIMD instructions, Inoue *et al.* [1] used an odd-even merge on PowerPC and Cell BE processors. On Intel processors, Chhugani *et al.* [2] implemented a bitonic merge operation because the permute instruction of the Intel SSE SIMD instruction is not flexible enough to implement odd-even merge efficiently. Other implementations for Intel also use the bitonic merge.

In this work, we used a much simpler data flow suitable for the instruction sets with limited permutation capabilities, such as the SSE. Our data flow can be implemented with only the min, max, and rotate instructions to merge 32-bit integer values in two vector registers, as shown in Figure 5 (*merge_4x4_32bit* function). Although it uses more comparisons than the bitonic merge (shown in Figure 1) or the odd-even merge, it slightly outperformed the bitonic merge operation previously used on the Intel platform by avoiding the costly permute instructions. To increase data parallelism, we use a larger sorting network. We build this merging kernel by executing the odd-even merge at the vector register level using the smaller 4x4 merging kernel as the building block (*merge_8x8_32bit* in Figure 5). As discussed in Section 2.2.1, using inputs larger than the hardware vector register size is important to improve instruction-level parallelism, which leads to higher throughput. Our data flow is much easier to implement compared to the complicated data flow of the bitonic merge, but it exhibited better performances on Xeon with SSE.

For merging 64-bit integer values, we used an odd-even merge operation because the permutation instructions of the SSE are sufficiently flexible to efficiently implement the odd-even merge for 64-bit integer values, and the odd-even merge requires fewer comparisons compared to the bitonic merge. Our in-register merge operation for the 64-bit integers also takes two input data streams, each of which consists of four integers stored in two vector registers. There is some overhead for the in-register merge operations for 64-bit integers using SSE because the SSE does not support min and max instructions for 64-bit integers, therefore, we need to use one vector compare instruction and two vector blend instructions instead of a pair of vector min and max instructions. To reduce this overhead, we use the min and max instructions for double precision floating point values by encoding each integer value into the fraction part (52 bits) of the IEEE floating point format. Because our approach encodes a small streamID instead of the index of each record, the fraction part is large enough to encode the 32-bit key and streamID.

3.4 Vectorized Combsort for Structures

The overall algorithm to sort structures with our vectorized mergesort is similar to the existing sorting scheme for sorting integer values described in Section 2.2. We use the vectorized combsort when the size of one sorted sequence is small enough to fit within the processor's cache memory.

To exploit SIMD instructions efficiently in the combsort, we use a key-index pair approach. Because we use the vectorized combsort only for small blocks (of *b* records) that can fit within the processor's cache memory, the random accesses to reorder records after sorting are not costly. We also use the 4-wide SIMD instructions by encoding the key and index (within each block to be sorted) into 32-bit integers instead of using 2-wide SIMD instructions for 64-bit integer values. The overall technique to encode key-index pairs into 32-bit integer values is almost the same as that used for the vectorized mergesort (described in Section 3.3.1). For example, if the size of a block is 1,024 records, we use 10 bits for the index and 22 bits for the (partial) key.

We first extract and encode the key-index pairs from the records to sort into a temporary array. We implemented this part with scalar instructions because the SIMD instructions did not exhibit any performance improvement over the scalar implementation. Then we sort the 32-bit integers with the vectorized combsort implemented with SSE SIMD instructions. Finally, we rearrange the records based on the sorted key-index pairs. Because the sorting uses only a part of the key to increase data parallelism, we must check that the records are in the correct order by using the entire key, as described in Section 3.3.1.

3.5 Sorting Records with Larger Keys

Up to now, we have been assuming that the size of a key is 32 bits. Even when the size of a key is larger than 32 bits, such as a 64-bit integer, we can apply almost the same technique described for 32-bit keys. We have already described the techniques to use only a part of the 32-bit keys to exploit the 4-wide SIMD instructions for 32-bit data. The same technique can be used to sort records using 2-wide or 4-wide SIMD instructions by using only a part of the 64-bit keys. When sorting records with 64-bit or larger keys, we confirm that the sorted results from the partial keys are correct by using the entire key at the end of each multiway merge operation, even when we use 64-bit intermediate integers in the multiway merge operation. If the partial-key conflicts are frequent even when we use 64-bit intermediate integers, we can do the sorting hierarchically as in the MSB-radix sort. When we find too many records with the same value in the partial keys, we can execute our sorting algorithm for the records having the same partial key using the next few bytes of the key as the partial key for sorting.

4. EVALUATIONS

We implemented our new algorithm using SSE instructions and evaluated it on an Intel Xeon processor. We implemented the program in C++ using SSE intrinsics. The system used for our evaluation was equipped with two 2.9-GHz Xeon E5-2690 (SandyBridge-EP) processors with 96 GB of system memory. Thus, the system had 16 cores. We do not use additional hardware threads provided by the 2-way SMT (Hyper Threading) of the processor in the experiments. The system ran under Redhat Enterprise Linux 6.4. We compiled all the programs as 64-bit binaries using gcc-4.8.2 with the –O3 option. We disabled dynamic frequency scaling (speed step and turbo boost) for more stable results. To fully utilize the main memory bandwidth available in the system, we executed all the programs with the interleave policy for NUMA memory allocation by using numactl --interleave=all command. Using the local allocation policy resulted in better performance with a small number of cores, where the main memory bandwidth did not limit the performance, but the interleave policy resulted in a higher peak performance with a larger number of cores when the performance was limited by the system memory bandwidth. We did not use the large pages in any of the experiments. The multiway mergesort (with all three approaches) and the radix sort use a temporary memory area of the same size as the data. The current implementation assumes a power of two in *N* and *p*.

In the evaluations, we used our implementations of the vectorized multiway mergesort in our approach described in Section 3 and also the two existing approaches, the key-index and direct approaches. We implemented these three approaches with and without SIMD instructions. As already discussed, the direct approach is not SIMD-friendly; hence, we did not use SIMD instructions for the multiway mergesort, but we used the vectorized combsort for the initial sorting of the small blocks for fair comparisons. We also implemented a cache-conscious radix sort, which combines the MSB-radix sort and LSB-radix sort to efficiently exploit the cache memory of the processor by improving the memory-access locality [17]. We also applied the local-buffer-based optimization proposed by Satish *et al.* [3] to reduce the cache misses. These two optimization techniques exhibited more than 3x performance improvement over the naive implementation of the radix sort, and we believe that this implementation is reasonably fast to represent the performance of the state-of-the-art radix sort implementations. We used 8 bits as the digit size in the radix sort unless we explicitly show another digit size.

We also graphed the performance of the parallel versions of the std::stable_sort function and std::sort function in the STL delivered with gcc. To enable the parallel version of the STL sort functions, we defined _GLIBCXX_PARALLEL and included the

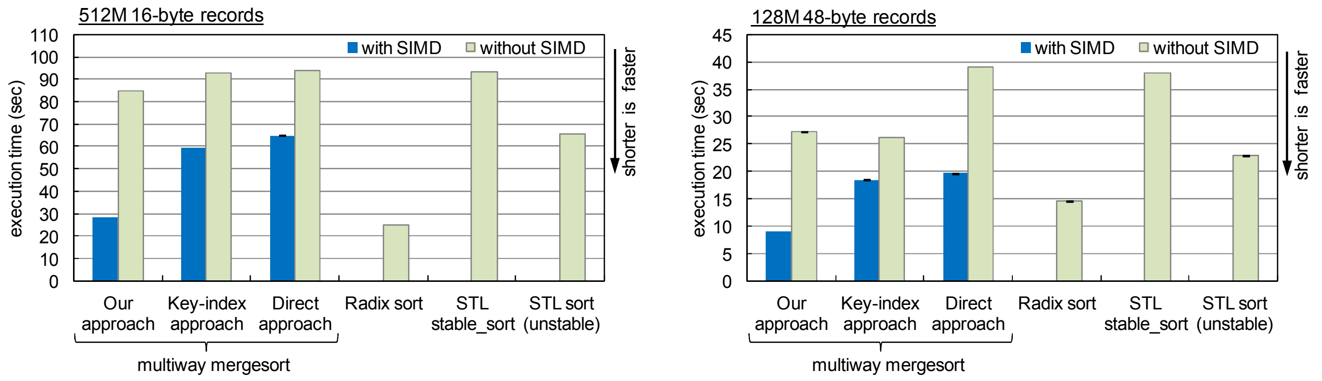


Figure 6. Execution time on 1 core for sorting 512M 16-byte records and 128M 48-byte records with 32-bit random integer keys using various algorithms implemented with and without SIMD instructions.

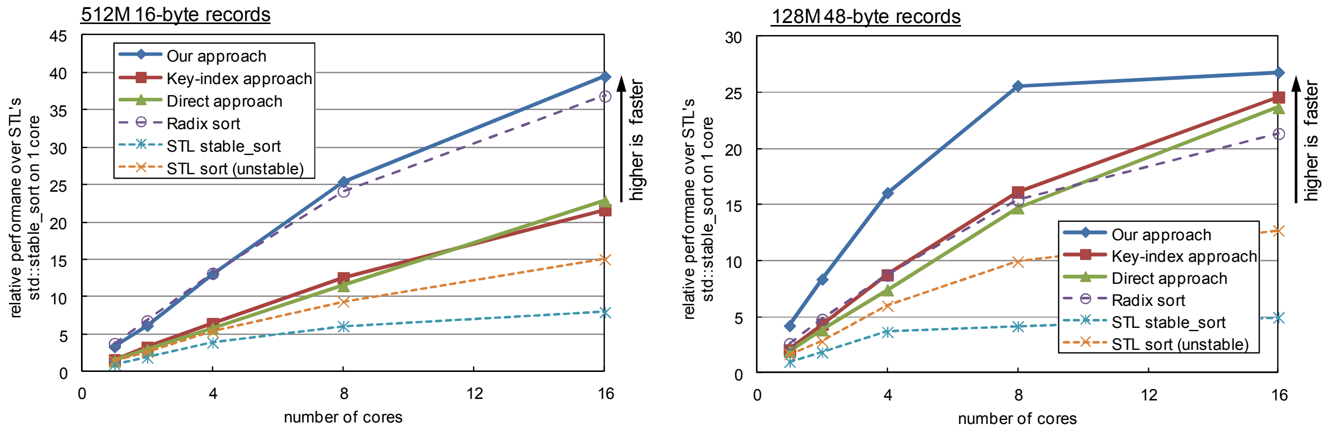


Figure 7. Performance scalability with increasing number of cores when sorting 512M 16-byte records and 128M 48-byte records. Three approaches with multiway mergesort were implemented with SIMD

<parallel/algorithm> header file instead of the standard <algorithm> header file in the source code. Among these tested algorithms, only the std::sort, which implements a variant of quicksort [20], is an unstable sorting algorithm.

4.1 Performance Comparisons

Figure 6 compares the performance of our approach with the multiway mergesort against two existing approaches, the key-index and direct approaches, with and without using SIMD instructions for sorting 512M 16-byte records (8 GB total) or 128M 48-byte records (6 GB total) with a 32-bit random integer key using only one thread. The figure also shows the performance of the radix sort and STL's sort functions.

Our approach implemented with SIMD showed the highest performance among the three approaches with the multiway mergesort. The performances of the three approaches for 16-byte records were almost comparable when implemented without SIMD, but our approach had the largest performance improvement of 3.0x from the use of SIMD instructions. For the other two approaches, the gains from the SIMD were 1.6x for the key-index approach and 1.4x for the direct approach. As a result of efficient SIMD exploitation, our approach outperformed the key-index approach by 2.1x and the direct approach by 2.3x when we used SIMD. As already discussed, the sorting part of the key-index approach can benefit from the SIMD instructions, but the final rearranging phase did not benefit from SIMD because it only moves records within system memory, so its performance is limited by the memory system performance rather than computational performance. The use of SIMD also did not help the performance of the direct approach in the mergesort. We actually implemented the SIMD version of the mergesort with a direct approach, but the performance of the SIMD version was slower than the non-SIMD version due to the overhead of the noncontiguous memory accesses, so we used the non-SIMD version in our evaluations. The performance gain in the direct approach shown in the figure came from the use of vectorized combsort for the initial sorting for small blocks. We used std::stable_sort when we disabled SIMD, instead of using our vectorized combsort. Our approach also outperformed the other two approaches for 48-byte records.

Comparing the performance of the vectorized multiway mergesort with our approach against the other algorithms, our approach achieved 3.3x higher performance than the standard stable_sort function included in STL. The performance of the cache-conscious radix sort was slightly better than our algorithm for 16-byte records (by 11.9%) and slower for 48-byte records (by 61.7%) in this configuration. The radix sort achieved comparable or sometimes better performance than our algorithm for sorting 16-byte or smaller records. However, its performance degraded more compared to the multiway mergesort when the records were larger.

Figure 7 shows the performance scalability with an increasing number of cores for each algorithm when sorting 512M of 16-byte records or 128M of 48-byte records. Our approach yielded the best performance among the three approaches with the multiway mergesort regardless of the number of cores used. The performance scalability was limited when using 16 cores, mostly due to the limited system memory bandwidth. As can be observed in Figure 7, using larger record sizes resulted in lower scalability because sorting an array of larger records requires more system memory bandwidth to copy the records in the system memory. Our algorithm achieved slightly better scaling than the radix sort when sorting 16-byte records; hence, it achieved better performance when using 16 cores. From these results, the vectorized mergesort with our approach can compete with optimized radix sort implementations even when sorting small records and can outperform the radix sort when sorting large records or sorting on multiple cores.

Figure 8 shows performance with an increasing numbers of 16-byte records (N) sorted on 1 core. Of the tested algorithms, only radix sort had O(N) computational complexity while the other algorithms had an average computational complexity of O(N·log(N)). However, we did not observe significant differences in the scalability among the algorithms. For the radix sort, the number of cache misses increased with increasing number of records to be sorted, and the memory performance limited the scalability of the radix sort. The performance of the radix sort was also sensitive to digit-size tuning. The current digit size of 8 bits was selected to achieve best performance for a large amount of data, but the digit size of 9 bits resulted in better performance for sorting small datasets.

Figure 9 compares the execution time for sorting 16M records of various record sizes on 1 core. For the radix sort, we selected the better number for each data point from performances with two different digit size configurations (8 bits or 9 bits) because neither configuration resulted in reasonable performance for all data points. The performance advantage of our approach over the key-index approach became smaller with larger record sizes. This is because, as shown in Table 1, our approach moves all of the records at the end of each multiway merge operation and hence multiple times during the sorting process, while the key-index approach moves the records only once in the rearranging phase at the end of sorting. When the record size was large, the cost of moving the records offset the performance advantage of our approach, especially when the record size exceeded the size of a cache line, 64 bytes on Xeon. Rearranging records smaller than the cache line size wastes memory bandwidth because only a small portion of the transferred data was actually used, and the unused data wasted the memory bandwidth. We believe the processors with larger cache line sizes are affected by larger overhead due to the rearranging; hence, our approach may have a larger performance advantage. For sorting 8-byte records, our approach outperformed that for simply sorting 64-bit integers using 2-wide SIMD instructions by efficiently exploiting 4-wide SIMD instructions instead of 2-wide SIMD instructions.

Comparing the vectorized multiway mergesort with our approach against the radix sort, the vectorized multiway mergesort outperformed the radix sort when the record size was larger than 16 bytes, while the two algorithms achieved almost comparable performances for smaller records. The radix sort required more memory bandwidth than the vectorized multiway mergesort due to its random memory accesses for reorder records. Hence, the radix sort did not perform well with larger records because a larger record required more memory bandwidth. These performance improvements with the vectorized multiway mergesort and larger records are consistent with previous studies [3].

To confirm that the rearranging really matters for the overall performance of the key-index approach, Figure 10 shows a breakdown of the execution times for the key-index approach in

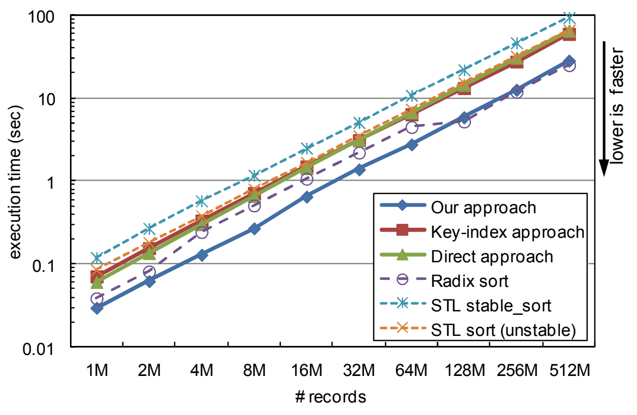


Figure 8. Performance scalability with increasing number of 16-byte records on 1 core. Three approaches with multiway mergesort are implemented with SIMD.

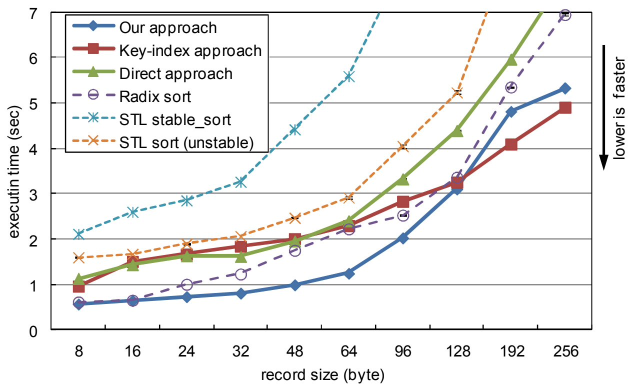


Figure 9. Execution times for sorting 16M records with various record sizes on 1 core. For radix sort, we selected better number for each data point from two different digit size configurations (8 or 9 bits). Three approaches in multiway mergesort are implemented with SIMD.

three phases: key extraction, sorting, and rearranging, when sorting 512M 16-byte records on 1 and 16 cores. On one core, about 32% of the total execution time was spent for extracting keys and rearranging records. As the number of cores increased, the sorting phase scaled very well, increasing to 15.2x with 16 cores. However, the rearranging and key extraction phases scaled rather poorly, with these two phases only reaching 11.3x with 16 cores. This is because these two phases are memory intensive, and the performance bottleneck was the system memory performance rather than the core computing capabilities. As a result, the key extraction and rearranging consumed about 39% of the execution time on 16 cores. This means that the key-index approach is negatively affected by the overhead of key extraction and record rearranging, especially when using many cores. We observed that the rearranging phase of the key-index approach alone caused more L2 cache misses than the total cache misses for our approach or the direct approach because the rearranging phase accesses the records randomly based on the sorted results. Due to the memory bus contentions of the frequent cache misses, the rearranging phase did not scale well with an increasing number of cores.

To show the effect of the input data distribution, Figure 11 compares the performance of algorithms with different numbers of random key bits. For example, *key* is initialized by rand32() &

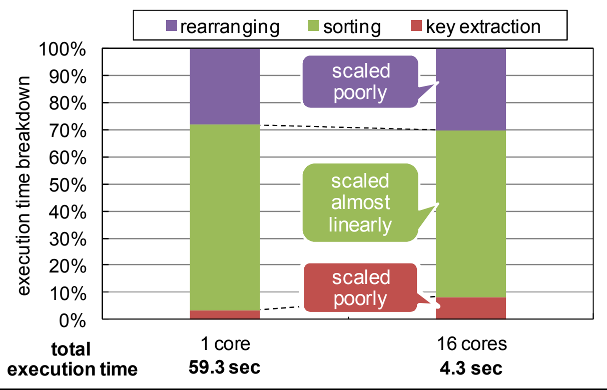


Figure 10. Execution time breakdowns for key-index approach (implemented with SIMD) into key extraction, sorting, and rearranging when sorting 512M 16-byte records on 1 and 16 cores.

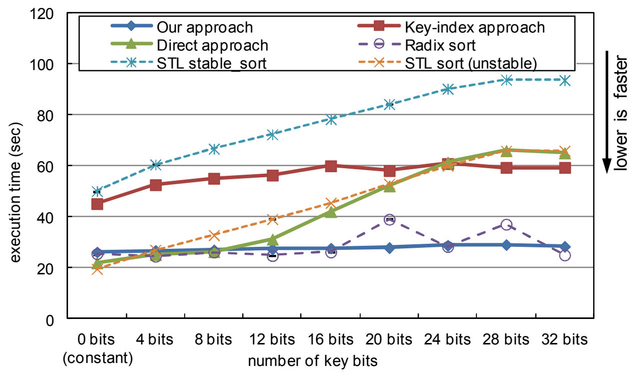
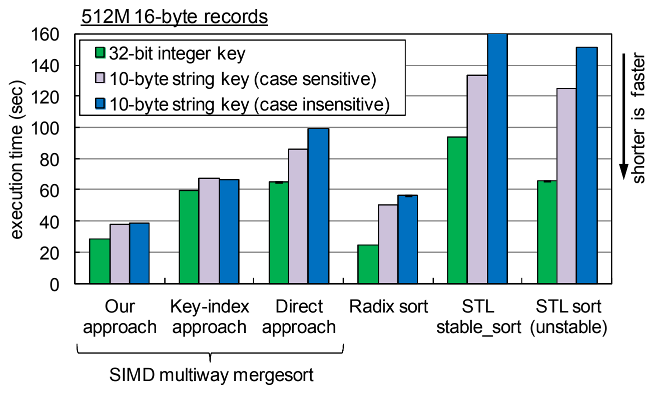


Figure 11. Execution times for sorting 512M 16-byte records with different random key bits. For 8-bit case, keys were initialized with rand32() & 0xFF. Three approaches withmultiway mergesort were implemented with SIMD.



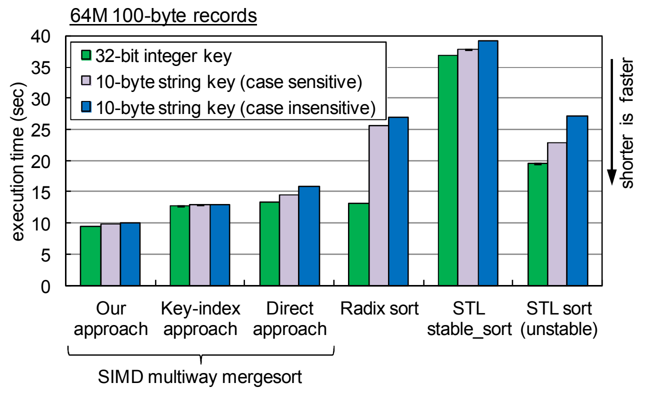


Figure 12. Execution time on 1 core for sorting 512M 16-byte records and 64M 100-byte records with 32-bit integer keys or 10-byte ASCII string keys. We evaluated both case-sensitive sorting and case-insensitive sorting with string keys.

0xFF when the number of bits was 8 bits. In the figure, 0 bits (leftmost) means that all the input records had the same key. When the number of key bits reduced (lower entropy), the performances of the direct approach and STL sort functions improved significantly because of the reduced branch misprediction overheads. The vectorized mergesort replaces many of the hard-to-predict conditional branches by the SIMD min and max instructions and hence the performance of the two algorithms based on the vectorized mergesort, our approach and key-index approach, were not significantly improved with the reduced entropy.

To show that our approach can improve sorting performance in a wider range of applications, we evaluated its performance for sorting records with string keys. Figure 12 shows the performance for sorting fixed-size (16 bytes or 100 bytes) records with string keys. By following the dataset configuration of the Sort Benchmark (http://sortbenchmark.org/), which is widely used in database research projects (such as [6, 12]), we used 10-byte random ASCII string key and sorted the records into the order of the memcmp function (case-sensitive sorting) or strcasecmp function (case-insensitive sorting). To initialize the keys, we used the method from the input generator of the Sort Benchmark. For both record sizes, our approach exhibited the best performance among the tested algorithms for sorting with the string keys. Because comparing the string keys is more costly than comparing the simple integer keys, there were slight degradations in the performance for all algorithms. However, the degradations were smaller for our approach and the key-index approach compared to the direct approach or STL algorithms because most of the comparisons were done in an encoded form (intermediate integers) for our approach and key-index approach. The performance of the radix sort also degraded due to the larger key size. As already discussed for integer key sorting, the performance advantage of our algorithm was smaller for larger record size. However, there was about a 30% performance advantage over the key-index approach for sorting 100-byte records, which is the default record size for the Sort Benchmark.

Figure 13 compares the performance to sort variable-sized records. In each record, the first two bytes show the length of the record and the other bytes are random ASCII string keys. The sizes of records are randomly distributed within the range of 12 (i.e. 10-byte string) to 20 bytes or 12 to 84 bytes. Hence, the average sizes are 16 and 48 bytes respectively. We sort the records in case-insensitive order. Although additional overhead to access variable-sized records attenuated the benefit of our approach, it achieved the best performance among the four tested algorithms. The basic idea of our approach works for sorting variable-sized records without changes. Parallelizing the vectorized multiway mergesort for variable-sized records using multiple threads is less efficient compared to sorting for the fixed-sized records because we cannot depend on the binary search for variable-sized records without preprocessing. The current implementations for variable-sized records do not support parallel sorting. Also, there are many algorithms specialized for sorting

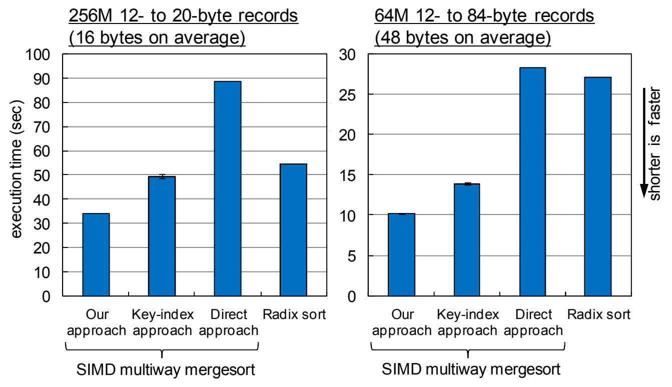


Figure 13. Execution times for case-insensitive sorting of variable-length string records with 256M records of 16-byte length on average and 64M records of 48-byte length on average.

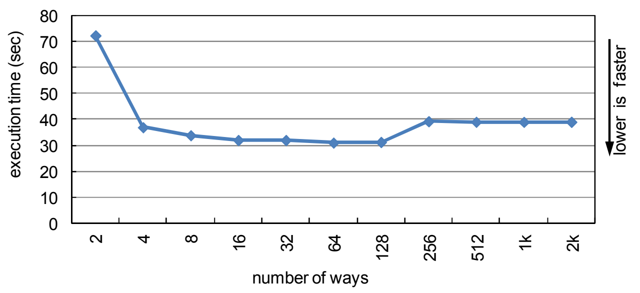


Figure 14. Execution times for sorting 512M 16-byte records with our algorithm using various *k* on 1 core.

(variable-sized) strings. For example, Burstsort [19] uses a trie-based data structure to represent string records for efficient comparisons and better memory-access locality. We did not use such advanced optimizations specialized for string sorting. How to integrate such techniques into our algorithm is an interesting topic for further performance improvements with sorting of the variable-length strings.

4.2 Effect of Parameters

In this section, we study in detail the effects of the three most important parameters in our approach implemented with SIMD instructions: the number of ways (*k*) in the vectorized multiway mergesort, the block size for the initial sorting (*b*) with the combsort, and the threshold to use the 4-wide SIMD comparison.

Figure 14 shows how *k* affects the performance of sorting 512M 16-byte records using 1 thread. We used 64 records as *b* and did not use the 4-wide SIMD comparisons. The x-axis is the *k* from 2 ways (standard binary mergesort) to 2048 ways. We found that *k* = 16 (16-way merge) to 128 (128-way merge) resulted in the best performance. In this range of *k*, *k* = 64 resulted in the best single-thread performance and *k* = 16 resulted in the best performance with 16 cores, but the performance differences were not significant. As already discussed, using a larger *k* reduced the overhead of copying records because our algorithm copies all of the records for each multiway merge operation. However, using a larger *k* requires more intermediate memory buffers, as shown in Figure 3, and this may result in more cache misses. Due to the net benefit of the reduced overhead of memory copies and the cost of the increased L1 cache misses, using *k* larger than 128 caused performance degradation. From these results, we used *k* = 32.

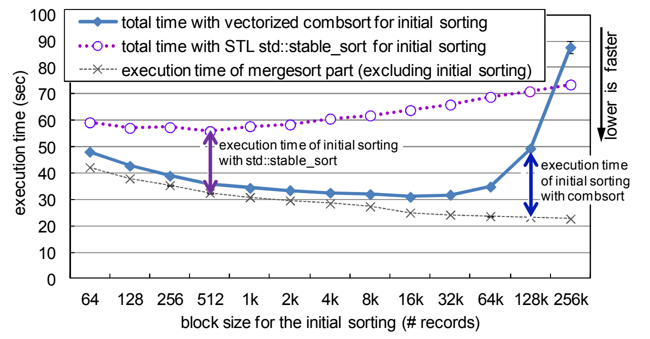


Figure 15. Execution times for sorting 512M 16-byte records with increasingly large blocks (*b* records) for the initial sorting using two different algorithms.

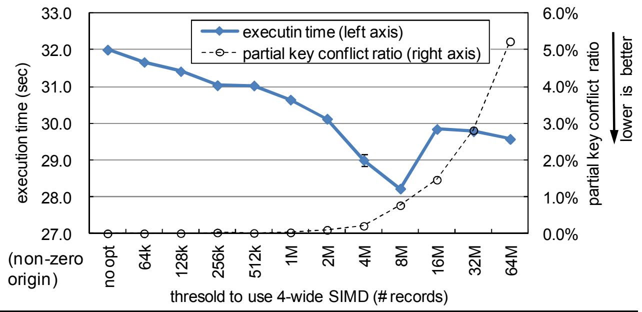


Figure 16. Execution times and partial conflict rate for sorting 512M 16-byte records with various thresholds for using 4-wide SIMD comparisons. Partial key conflict ratio is ratio against number of total records merged using multiway merge operation (*N* * number of multiway merge stages).

Figure 15 shows the single-thread performances with various block sizes for the initial sorting. We used *k* = 32 and did not use the 4-wide SIMD comparisons. To confirm that using the vectorized combsort for the initial sorting matters for the overall performance of sorting large arrays, we also show the performance when we use the STL's std::stable_sort function for the initial sorting. When using the vectorized combsort for the initial sorting, the overall performance was best with *b* = 16,384 records. When *b* became larger than 65,536 records, performance significantly degraded. This is because the combsort has poor memory-access locality and we need to keep all the data within the processor's cache memory (the 256-KB L2 cache in this case). When *b* was larger than 65,536 records, the intermediate 32-bit integers (256 KB) could not fit within the L2 cache memory. Another reason of poor performance with the vectorized combsort with excessively large *b* is the frequent partial-key conflicts. The frequency of the partial-key conflicts remained less than 1% of the total number of records sorted with the combsort for *b* = 8,192 while it was more than 10% for *b* larger than 32,768. From these results, we used *b* = 8,192 for all of the evaluations because the processor supports two hardware threads that share the L2 cache on one core by using Hyper Threading and the effective size of the L2 cache per thread is halved when we use both hardware threads. The performance with the standard STL function for the initial sorting, which implements non-SIMD multiway mergesort, was best when *b* = 512 records. However, the best performance with the STL was about 1.8x slower than when we used the vectorized combsort for the initial sorting. This means that the algorithm for the initial sorting is quite important for overall performance even when sorting large arrays.

Figure 16 shows how the use of 4-wide SIMD in the multiway mergesort improved performance. We used $k = 32$ and $b = 8{,}192$. The x-axis shows the threshold to switch from 2-wide SIMD to 4-wide SIMD. The leftmost point is performance when we used 2-wide SIMD for all the mergesort stages and did not use the 4-wide SIMD. We observed that performance was best with 8M records as the threshold. The best performance was about 11.8% better than that without optimization (the leftmost point). In the evaluations discussed in Section 4.1, we used 8M records as the threshold for this optimization; hence, the first two stages of the multiway merge were executed using 4-wide SIMD instructions. We observed a significant increase in the frequency of the partial-key conflicts, which may result in excessive overhead when we use a threshold larger than 8M records.

5. SUMMARY

We described our new sorting algorithm for sorting an array of structures by efficiently exploiting the SIMD instructions and cache memory. We showed that the key-index approach, which sorts only the key-index pairs using SIMD instructions then rearranges the records based on the sorted key-index pairs, caused significant overhead when rearranging the records due to random and scattered memory accesses. Our approach can prevent costly random accesses for rearranging the records while still efficiently exploiting the SIMD instructions.

Our results showed that our new approach achieved up to 2.1x better single-thread performance than the key-index approach implemented with SIMD instructions when sorting 16-byte records. Our approach also yielded better performance when we used multiple cores. In real-world workloads, sorting is mostly used to reorder data structures according to their keys and hence our new algorithm can contribute to a wide range of applications by accelerating this important sorting operation.

6. REFERENCES

[1] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 189–198, 2007.

[2] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *Proceedings of VLDB Endow.,* 1 (2), pp. 1313–1324, 2007.

[3] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 351–362, 2010.

[4] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs, GPUs and Intel MIC architectures. *Intel Technical report*, 2010.

[5] H. Sundar, D. Malhotra, and G. Biros. HykSort: a new variant of hypercube quicksort on distributed memory architectures. In *Proceedings of the 27th ACM International conference on supercomputing*, pp. 293–302, 2013.

[6] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 841–850, 2012.

[7] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. D. Blas, and P. Dubey. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. In *Proceedings of VLDB Endow.,* 2(2), pp. 1378–1389, 2009.

[8] C. Balkesen, G. Alonso, and M. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. In *Proceedings of the VLDB Endow.*, 7(1), pp. 85–96, 2013.

[9] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 755–766, 2014.

[10] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference* 32. AFIPS, pp. 307–314, 1968.

[11] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: high performance sorting on the cell processor. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pp. 1286–1297, 2007.

[12] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 325–336, 2006.

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

[14] H. Inoue, M. Ohara, and K. Taura. Faster Set Intersection with SIMD instructions by Reducing Branch Mispredictions, In *Proceedings of VLDB Endow.,* 8(3), 2014.

[15] D. E. Knuth. *The Art of Computer Programming*. Vol. 3: Sorting and Searching. 1973.

[16] S. Lacey, R. Box. A Fast, Easy Sort. In *Byte Magazine* (April), pp. 315–320, 1991.

[17] D. J. González, J.-L. Larriba-Pey, and J. J. Navarro. Communication conscious radix sort. In *Proceedings of the 13th International Conference on Supercomputing,* pp. 76–82, 1999.

[18] R. Francis, I. Mathieson. A Benchmark Parallel Sort for Shared memory Multiprocessors. *IEEE Transactions on Computers* 37(12), pp. 1619–1626. 1988.

[19] R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics* 9, Article 1.5, 2004.

[20] D. R. Musser. Introspective Sorting and Selection Algorithms. *Software Practice and Experience* 27(8), pp. 983–993, 1997.