博士論文

Efficient Exploitation of SIMD Instructions in Non-Numerical Applications

(SIMD 命令の効率的な活用による
非数値計算アプリケーションの高速化)

井上 拓

Efficient Exploitation of SIMD Instructions in Non-Numerical Applications

Doctoral Dissertation

Hiroshi Inoue

Submitted to Department of Information and Communication Engineering,
Graduate School of Information Science and Technology,
The University of Tokyo

in partial fulfillment of the requirements for the degree of
Doctor of Information Science and Technology

Supervisor: Prof. Kenjiro Taura

Abstract

To achieve high performance on today's systems, it is critically important to exploit different types of parallelisms available in the algorithms by mapping them onto hardware features. Multiple cores and multiple SMT threads in a core can accelerate applications by exploiting the thread-level parallelism. Another important processor feature to accelerate compute-intensive workloads is Single Instruction Multiple Data (SIMD) instructions, which can operate on multiple data in one instruction, to exploit data parallelism. Many high-performance processors support the SIMD instructions, such as the SSE and AVX instruction set of the x86 processors or the VMX and VSX instruction set of the PowerPC processors. To fully utilize the huge computing capability of today's processors, the programmers need to identify the thread-level and data parallelism available in algorithms and map the parallelism onto the hardware capabilities. Hence, there are many existing research projects to enhance important algorithms for parallelization with multiple threads or vectorization with SIMD instructions. So far, the SIMD instructions have been widely used in many scientific computing workloads (such as matrix computations), image processing workloads (such as movie encoding and decoding), and basic string operations since it is mostly straightforward to vectorize these algorithms. However, there are still many important algorithms and workloads we cannot efficiently exploit SIMD instructions.

In this dissertation, we present new high-performance algorithms for efficiently exploiting SIMD instructions on the following three key operations: sorting for integer values, sorting for structures, and sorted set intersection. These algorithms are important building blocks of many non-numerical applications, such as database management systems and search engines.

An obvious advantage of the SIMD instructions is the degree of data parallelism available in one instruction. In addition, they allow programmers to reduce the number of conditional branches in their programs. For example, a program can select the smaller or larger value from each element's pair of two vectors without conditional branches. For another example, a program can aggregate multiple conditional branches by using vector-comparison-based conditional branches, such as branch-if-all-equal instruction supported in most of SIMD instruction sets. On superscalar processors with long pipeline stages, conditional branches can potentially incur pipeline stalls and thus significantly limit

the performance. Today's processors are equipped with a branch prediction unit and speculatively execute one direction (taken or not-taken) of a conditional branch to maximize the utilization of processor resources. If the predicted direction of the branch does not match the actual outcome of the branch (branch misprediction), the hardware typically wastes more than ten CPU cycles because it needs to flush speculatively executed instructions and restart the execution from the fetch of the next instruction for the actual branch direction. The benefit of reduction in the number of conditional branches is potentially significant for many non-numerical workloads since the branch misprediction overhead is often larger for the non-numerical workloads compared to typical numerical applications such as matrix computations with regular control flows. For example, Zhou and Ross [2002] reported that SIMD instructions can accelerate many database operations, such as scan operations and nested-loop join operations, by removing branch misprediction overhead.

In this dissertation, we study efficient sorting algorithm since sorting has been one of the most important building blocks for many applications, such as database management systems. Hence many sequential and parallel sorting algorithms have been studied in the past. However, SIMD instructions in today's processors have limitations and popular sorting algorithms, such as quicksort, are not suitable to exploit SIMD instructions efficiently due to its scattered memory accesses. We propose a new high-performance sorting algorithm suitable for exploiting both the SIMD instructions and thread-level parallelism available on today's multicore processors. Our main contribution includes a new high-performance sorting algorithm that can effectively exploit SIMD instructions. It consists of two algorithms: a vectorized mergesort and a vectorized combsort. In our vectorized combsort, it is possible to eliminate all unaligned memory accesses from combsort. For the vectorized mergesort, we proposed a novel linear-time merge algorithm that can take advantage of the SIMD instructions. We combine these two algorithms. First we divide data into small blocks and sort them using the vectorized combsort. Then, we merge them using the vectorized (multiway) mergesort. We show that our algorithm using SIMD instructions outperformed other implementations of comparison-based sorting algorithm such as STL's std::sort, which implements a quicksort variant, and a SIMD implementation of the bitonic mergesort when sorting a large array of random 32-bit integers. Comparing against an optimized radix sort, our algorithm achieved almost comparable performance using 128-bit SIMD instructions and better performance using 256-bit SIMD instructions. Also, our new algorithm showed better scalability with increasing number of cores than the radix sort and the bitonic mergesort.

We also extend our new sorting algorithm for sorting an array of structures instead of an array of integers. In real workloads, sorting is mostly used to rearrange structures based on a sorting key included in each structure. Here, we call each structure to be sorted a record. 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. 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 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. We developed a new stable sorting algorithm that can take advantage of SIMD instructions while avoiding the frequent cache misses caused by the random memory accesses. The main contribution on this topic is a new approach in the multiway mergesort for sorting an array of structures, which 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. 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.

Set intersection, which selects common elements from two input sets, is another important workload we study in this dissertation. It is a fundamental operation in many applications, including multi-word queries in Web search engines and join operations in database management systems. For example, in Web search engines the set intersection is heavily used for multi-word queries to find documents containing two or more keywords by intersecting the sorted lists of matching document IDs from the individual query words. In such systems, the performance of the sorted set intersection often dominates the overall performance. We describe our new algorithm to improve the performance of the set intersection. Unlike most of the existing advanced techniques, we focus on improving the execution efficiency of the set intersection on the microarchitectures of today's processors by reducing the branch mispredictions. Moreover, we can effectively eliminate many of the comparisons by aggregating multiple comparisons and conditional branches with one branch

based on a SIMD comparison. Our algorithm roughly doubled the performance for set intersection for 32-bit and 64-bit integer datasets even without using SIMD instructions compared to the std::set_intersection implementation delivered with gcc. The use of SIMD instructions further doubled the performance on both processors.

In these three new algorithms, the key to achieve high performance is 1) to exploit data parallelism available in the algorithm and 2) to reduce the number of conditional branches 3) while avoiding non-contiguous memory accesses, which increases the memory access overhead. Although the data parallelism available in one instruction is an obvious and straightforward advantage of the SIMD instructions, the reduced branch misprediction overhead also gives non-negligible performance gain; and hence the advantage of the SIMD instructions can surpass the data parallelism of the SIMD instruction. For example, we demonstrated 7.3x to 12.8x performance improvement using 4-wide SIMD instruction (SSE for 32-bit integers) in various sorting algorithms that are suitable for SIMD.

To reduce the number of branch mispredictions, we take two different approaches for sorting and set intersection. For the set intersection, we aggregate multiple conditional branches into one since the direction of most of the conditional branches is same. For sorting, on the other hands, we replace conditional branches by SIMD minimum and maximum instructions. In sorting, especially for random numbers, the directions of conditional branches are mostly unpredictable and the branch directions are divergent; hence it is not effective to aggregate multiple branches. By replacing control flow of the unpredictable conditional branches into a data flow by arithmetic instructions avoid the huge overhead of branch mispredictions and hence very effective to improve the performance. Although we take different approach for handling conditional branches, some of the optimization techniques are common among our proposed algorithms. For example, using a smaller data type instead of a larger type to increase the data parallelism in one instruction is an important technique to get larger performance gain in sorting of structures and set intersection. Typically, using a small data type does not improve the computation performance with scalar processing, and hence it is unique to SIMD processing.

In addition to the superior performances with SIMD instructions, we have demonstrated that we can improve the energy efficiency (performance per watt) using SIMD instructions efficiently. The energy efficiency is critically important for computing systems today ranging from super computers to mobile devices. We observed only small increase in energy consumption in trade for huge performance boost for both sorting and set intersection.

Using SIMD increases energy consumption in vector ALUs, but it also reduces the execution time. In total, we observed significant improvement in the energy efficiency. Hence, our results show that our new algorithms can contribute wide range of applications and systems.

Acknowledgements

I wish to express my great appreciation to my supervisor, Professor Kenjiro Taura, for his guidance, suggestions and encouragement throughout this study. He has provided me a lot of opportunities to dive into new topics and to broaden my research landscape.

I would like to express my gratitude to the thesis committee members, Professor Masaru Kitsuregawa, Professor Masahiro Goshima (at National Institute of Informatics), Professor Hidetsugu Irie, and Professor Toshihiko Yamasaki, for their valuable discussions and insightful suggestions. I also thank Professor Hitoshi Iba, Professor Masaru Kitsuregawa and Professor Masahiro Goshima for their feedback on my research in the earlier years of my Ph.D. course study. I would like to thank members at Taura laboratory of these three years. It has been a very pleasant time for me to study in this great environment.

I thank all my collaborators at IBM who gave useful feedback on the research projects, which this dissertation based on. Especially, my sincere thanks also go to my former and current managers, Dr. Toshio Nakatani, Dr. Tamiya Onodera and Dr. Moriyoshi Ohara. I was not able to conduct research projects at IBM without their devoted supports and encouragements. I thank the former and current directors of IBM Research – Tokyo for allowing me to study in the university while I keep working in IBM and for providing the financial support. I thank Mr. Shannon Jacobs for his editorial assistance for most of my papers. I also thank my colleagues at IBM, especially the members of TRL Camera Club, for making my life at IBM fruitful.

At last, but of course not the least, I would like to express my special appreciation to my parents, Kan and Eiko Inoue.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Many modern high-performance processors provide multiple cores and Single Instruction Multiple Data (SIMD) instructions, such as the SSE and AVX instruction set of the x86 or the VMX and VSX instruction set of the PowerPC, to achieve higher performance in compute-intensive workloads. To benefit from the huge computing power of today's multicore processors, the programmers need to identify the thread-level and data parallelism available in algorithms and exploit hardware capabilities: thread-level parallelism by using multiple cores and data parallelism by SIMD instructions. Hence, there are many existing research projects to enhance important algorithms for *parallelization* with multiple threads or *vectorization* with SIMD instructions. For example, the SIMD instructions have been widely used in many scientific computing workloads, such as matrix computations, image processing workloads, and basic string operations since it is straightforward to vectorize these algorithms. However, there are still many algorithms and workloads we cannot efficiently exploit SIMD instructions.

An obvious advantage of the SIMD instructions is the degree of data parallelism available in one instruction. In addition, they allow programmers to reduce the number of

conditional branches in their programs. For example, a program can select the smaller or larger value from each element's pair of two vectors without conditional branches. For another example, a program can aggregate multiple conditional branches by using vector-comparison-based conditional branches, such as branch-if-all-equal instruction supported by most of SIMD instruction sets. On superscalar processors with long pipeline stages, conditional branches can potentially incur pipeline stalls and thus significantly limit the performance. Today's processors are equipped with a branch prediction unit and speculatively execute one direction (taken or not-taken) of a conditional branch to maximize the utilization of processor resources. If the predicted direction of the branch does not match the actual outcome of the branch (branch misprediction), the hardware typically wastes more than ten CPU cycles because it needs to flush speculatively executed instructions and restart the execution from the fetch of the next instruction for the actual branch direction. The benefit of reduction in the number of conditional branches is potentially significant for many non-numerical workloads since the branch misprediction overhead is often larger for the non-numerical workloads compared to typical numerical applications such as matrix computations. For example, Zhou and Ross [2002] reported that SIMD instructions can accelerate many database operations, such as scan operations and nested-loop join operations, by removing branch misprediction overhead.

1.2 Contribution

In this dissertation, we develop new algorithms for efficiently exploiting SIMD instructions on the following three key operations: sorting for integer values, sorting for structures, and sorted set intersection.

Sorting for integers [Inoue et al. 2012]: Sorting is one of the most important building blocks for many applications, such as database management systems [Graefe 2006]. Hence many sequential and parallel sorting algorithms have been studied in the past [Martin 1971, Knuth 1973]. However, SIMD instructions in today's processors have limitations and popular sorting algorithms, such as quicksort, are not suitable to exploit SIMD instructions efficiently due to their memory access patterns. We propose a new high-performance sorting algorithm suitable for exploiting both the SIMD instructions and thread-level parallelism available on today's multicore processors. The main contribution is a new high-performance sorting algorithm that can effectively exploit SIMD instructions. It consists of two algorithms: a vectorized combsort and a vectorized mergesort. In our vectorized combsort, it is possible to eliminate all unaligned memory accesses from combsort. For the vectorized

mergesort, we proposed a novel linear-time merge algorithm that can take advantage of the SIMD instructions.

Sorting for structures [Inoue and Taura 2015]: 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 [Satish et al. 2010]. 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 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. 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. The main contribution of our work is a new approach in the multiway mergesort for sorting an array of structures, which 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.

Set intersection [Inoue et al. 2014]: Set intersection, which selects common elements from two input sets, is a fundamental operation in many applications, including multi-word queries in Web search engines and join operations in database management systems. For example, in Web search engines the set intersection is heavily used for multi-word queries to find documents containing two or more keywords by intersecting the sorted lists of matching document IDs from the individual query words [Barroso et al. 2003]. In such systems, the performance of the sorted set intersection often dominates the overall performance. This paper describes our new algorithm to improve the performance of the set intersection. Unlike most of the existing advanced techniques, we focus on improving the execution efficiency of the set intersection on the microarchitectures of today's processors by reducing the branch mispredictions. Moreover, we can effectively eliminate many of the comparisons by

aggregating multiple comparisons and conditional branches with one branch based on a SIMD comparison.

1.3 Organization

Chapter 2 gives an overview of the SIMD instructions we use in this dissertation. Then we review the related work. Chapter 3 describes our new SIMD-based sorting algorithm for integer values and evaluates its performance. Chapter 4 extends the sorting algorithm for sorting structures, which consist of key and payload. Chapter 5 explains our algorithm for sorted set intersection. Finally, Chapter 6 sets out our conclusions.

Chapter 2

Background

2.1 SIMD overview

Single Instruction, Multiple Data (SIMD) is a category of parallel computing architecture defined by Flynn [1972]. In the SIMD architecture, a processor executes one operation for multiple data at once in parallel to achieve higher performance by exploiting data parallelism available in the executing algorithm. Figure 1 shows a schematic image of a SIMD instruction and a scalar (non-SIMD) instruction. Another important category defined by Flynn is *Mingle Instruction, Multiple Data* (MIMD). In MIMD architecture, multiple processors (or cores) having the different instruction pointers apply different operations for different data. Most of today's processors provide SIMD capability by vector registers and vector instructions, and also provide MIMD capability by multiple cores and simultaneous multi threading (SMT).

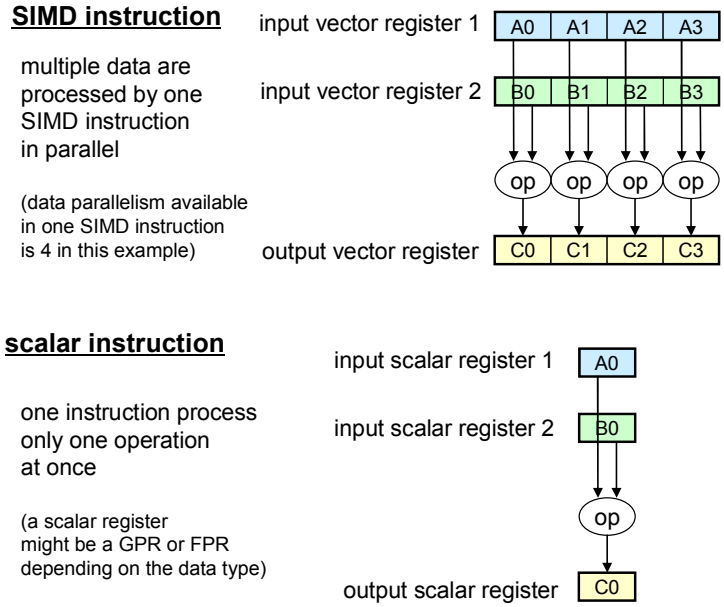


Figure 1. Schematic comparisons of a SIMD instruction and a scalar instruction.

Historically, SIMD was implemented as vector processors for supercomputers such as Cray-1. Then, after the successful introduction of Intel's MMX (MultiMedia eXtension) instruction set for its x86 processors in 1997, the SIMD instructions (and vector registers for them) became a popular way to accelerate compute-intensive workloads, such as video processing, among general-purpose processors. The SIMD instruction sets of general-purpose processors include, for example, MMX, SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) instruction sets of x86 processors [Intel 2015] or VMX (Vector Multimedia eXtension, also known as AltiVec) [IBM 1998, Freescale 1999] and VSX (Vector Scalar eXtension) [IBM 2010] instruction sets of PowerPC processors. In addition to general-purpose processors, programmable GPUs can be classified as advanced variants of the SIMD architecture. By supporting more flexible handling of divergent memory accesses and control flows, each slot of a GPU's SIMD instruction can be seen as a thread; hence it is called *Single Instruction, Multiple Thread* (SIMT) architecture [Nickolls et al. 2008].

Each SIMD extension consists of SIMD instructions and vector registers. In MMX, the vector registers share the same physical resource with floating point registers; hence the length of a vector register (*vector length*), which determines the parallelism in one SIMD instruction, is 64 bits. To achieve larger performance gain, the vector length has been getting

longer and longer; it is 128 bits in SSE using dedicated vector registers and 256 bits in AVX. The next generation AVX instruction set is going to use 512-bit vector registers. The functionalities of the SIMD instructions are also growing. MMX only supports integer data types since it focuses on media processing as its name shows. However, later instruction sets also support single- and double-precision floating point data types in addition to integers. Also today's processors support more and more special-purpose instructions such as those for string processing and cryptography.

2.2 Programming with SIMD

To generate binary code with SIMD instructions, the most straightforward approach is writing a program in assembly language or using inline assembly in high-level language code. To reduce the programmers' burden, many C or C++ compilers provide sets of built-in methods (*intrinsics*) that are directly translated into specific SIMD instructions. These compilers also support vector data types to represent vector registers. This intrinsic-based programming approach is platform dependent, i.e. the program is not portable among various processors supporting different SIMD instruction sets. Figure 2 shows example of program snippets written with two different intrinsics, one for AVX2 of x86 and another for VMX of PowerPC.

Original scalar code

```
int *a, *b, *c, i;
for (i = 0; i < 1024; i++) {
    c[i] = a[i] + b[i];
}
```

for AVX2 (process 8 elements at once using 256-bit vector registers)

```
for (i = 0; i < 1024; i+=8) {
    __m256i v1 = _mm256_loadu_si256((__m256i *)(a+i));
    __m256i v2 = _mm256_loadu_si256((__m256i *)(b+i));
    __m256i v3 = _mm256_add_epi32(v1, v2);
    _mm256_storeu_si256((__m256i *)(c+i), v3);
}
```

for VMX (process 4 elements at once using 128-bit vector registers)

```
for (i = 0; i < 1024; i+=4) {
    vector int v1 = vec_ld(0, a+i);
    vector int v2 = vec_ld(0, b+i);
    vector int v3 = vec_add(v1, v2);
    vec_st(v3, 0, c+i);
}
```

Figure 2. Comparisons of a simple loop manually vectorized using the intrinsics for AVX2 and VMX.

Another approach to generate SIMD instructions are automatic loop vectorization by optimizing compilers. In this approach, a programmer writes a loop without using SIMD intrinsics and the compiler automatically transform the loop into the SIMD code. Although there is a rich set of research projects on automatic loop vectorization in the history, the capability of the automatic loop vectorization is still limited to the vectorization of relatively simple loops [Maleki et al. 2011]. Hence, the intrinsic approach is preferable to achieve higher performance while the automatic vectorization yields much better programmability and portability. To help the analysis in the vectorizing compilers, there are a couple of ways to explicitly show the parallelism available in the program, for example using pragmas or DSLs. The pragmas for loop vectorization include standardized pragmas, such as OpenMP, and vendor-specific pragmas. OpenCL is one example of the DSLs for vectorization.

We use SIMD intrinsics provided by the C/C++ compilers to fully utilize the capability of the SIMD instructions. Unfortunately, our proposed algorithms are too complicated to automatically vectorize even for state-of-the-art vectorizing compilers.

2.3 SIMD instructions used in this dissertation

In this dissertation, we mainly use SSE instruction set of x86 processors and VSX instruction set of the PowerPC processors to present our new algorithms. Both instruction sets provide a set of 128-bit vector registers, each of which can be used as sixteen 8-bit values, eight 16-bit values, four 32-bit values, or two 64-bit values. In addition to the two 128-bit SIMD instructions, we also use the AVX2 instruction set of x86, which uses 256-bit vector registers, to study how the increased vector length improves the overall performance.

In this dissertation, we use the following SIMD instructions in addition to simple SIMD arithmetic instructions. These instructions are quite common and most of the existing SIMD instruction sets of general-purpose processors support them (with variations).

The vector compare instruction reads from two input vector registers and writes to one output register. It compares each value in the first input register to the corresponding value in the second input register and returns the result of comparisons as a mask in the output register. This mask can be used as input for other instructions, such as the vector select instruction. Also, it is possible to alter the control flow based on an aggregated result of a vector compare instruction like *branch if all equal*.

The vector select (or blend) instruction is an arithmetic operation to mix values from two vectors into one vector; it takes three vector registers as the inputs and one for the output. It selects a value for each bit from the first or second input registers by using the contents of the third input register as a mask for the selection. The vector minimum and the vector maximum instructions can be classified as variants of the vector select instruction, which combines a vector compare instruction and a vector select instruction.

The vector permutation (or shuffle) instruction reorders the elements of input vector register(s) arbitrarily into an output vector register. In the VSX instruction set, the permutation instruction takes three registers as the inputs and one for the output. The first two registers are treated as an array of 32 single-byte values, and the third register is used as an array of indexes to pick 16 arbitrary bytes from the input register. The SSE and AVX instruction sets support a wide variety of less flexible instructions for permutation.

The vector load and store instructions transfer data between a vector register and system memory. We use only the vector load and store instructions for contiguous data, which are most popular and performant. Some latest processors support load and store for non-contiguous data (so called *gather* and *scatter* instructions), however, the performance of the gather and scatter instructions are not comparable to the contiguous memory access instructions.

Chapter 3

Sorting for Integers

3.1 Introduction

Sorting is one of the most important building blocks for operating systems and many commercial and scientific applications, such as database management systems [Graefe 2006]. Hence many sequential and parallel sorting algorithms have been studied in the past [Martin 1971, Knuth 1973]. However popular sorting algorithms, such as quicksort, are not suitable for efficiently exploiting SIMD instructions due to inherently non-contiguous memory access patterns and loop-carried dependencies; many sorting algorithms require element-wise memory accesses, which incur additional overhead and attenuate the benefits of SIMD instructions.

In this chapter[†], we describe our new high-performance sorting algorithm suitable for exploiting both the SIMD instructions and thread-level parallelism available on today's multicore processors. Our sorting algorithm consists of two algorithms: a vectorized (multiway) mergesort as the main algorithm and a vectorized combsort for initial sorting of small blocks. Both phases can take advantage of the SIMD instructions and can also run in parallel with multiple threads.

Our vectorized mergesort employs our new vectorized merge algorithm, which combines SIMD-based sorting network into the usual merge operation. Its computational complexity is $O(N \cdot \log(N))$[‡] when sorting N elements even in the worst case. When the data to be sorted is small enough to fit within the cache memory of the processor, our vectorized combsort achieves better performance than our vectorized mergesort due to its simplicity and smaller misprediction overhead. The computational complexity for both the combsort and our vectorized combsort is $O(N \cdot \log(N))$ on average, and $O(N^2)$ in the worst case. 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 mergesort.

The complete sorting scheme first divides all of the data into blocks that fit in the L1D cache of each processor core. Next it sorts each block using the vectorized combsort. Finally, it merges the sorted blocks with our vectorized merge algorithm to complete the sorting. Both algorithms can be executed by multiple threads in parallel. The entire algorithm has the computational complexity of $O(N \cdot \log(N))$. Also it can be executed in parallel by multiple threads with the complexity of $O(N \cdot \log(N)/c)$ assuming the number of threads, c, is smaller than the number of blocks for the initial sorting.

We implemented and evaluated our algorithm on a system with 16 cores of the Xeon (SandyBridge-EP) processors using SSE instruction set. Additionally, we evaluated the performance on two systems with newer Haswell processors using AVX2 instruction sets to

[†] A part of the contents in this chapter was published in *Software: Practice and Experience,* Vol. 42, No. 6, pp. 753–777, 2012 [Inoue et al. 2012]. Also a preliminary version of this work was presented in *proceedings of the Sixteenth IEEE Parallel Architecture and Compilation Techniques* (PACT 2007) [Inoue et al. 2007].

[‡] log refers to logarithm with base 2 unless a different value is specified.

evaluate the performance and the energy efficiency with the longer SIMD architecture. In summary, our algorithm using SIMD instructions significantly outperformed std::sort sorting library of gcc's STL and the bitonic mergesort that uses SIMD instructions on one core of Xeon. Also, our algorithm can compete with an optimized radix sort implementation using 128-bit SIMD (SSE) instructions and it outperformed the radix sort using 256-bit SIMD (AVX) instructions. The performance of our algorithm did not depend on key distributions of the input data by reducing the number of data-dependent conditional branches executed.

On multiple cores, our algorithm demonstrated better performance scalability with increasing numbers of cores than STL's std::sort, the bitonic mergesort with SIMD and the radix sort. It achieved a speed up of 13.9x for 16 cores, while the bitonic mergesort achieved a speedup of 4.7x and the radix sort achieved a speed up of 11.5x.

The main contribution of this work is a new high-performance sorting algorithm that can effectively exploit SIMD instructions. It consists of two algorithms: a vectorized combsort and a vectorized mergesort. For the vectorized mergesort, we proposed a novel linear-time merge algorithm that can take advantage of the SIMD instructions. We show that our algorithm achieves higher performance and scalability with increasing numbers of processor cores than the best known algorithms. In our vectorized combsort, it is possible to eliminate all unaligned memory accesses from combsort. An obvious advantage of the SIMD instructions is the degree of data parallelism available in one instruction. In addition, they allow programmers to reduce the number of conditional branches in their programs. Our results with detailed analysis using the performance counters show that the reduced branch mispredictions are major source of performance gain by our algorithm.

3.2 Related work

Many sorting algorithms have been proposed in the past. Quicksort is one of the fastest algorithms used in practice, and hence there are many optimized implementations of quicksort available. However, there is no known technique to efficiently implement quicksort using existing SIMD instructions of the general-purpose processors.

Radix sort is another sorting algorithm widely used today. It has a smaller computational complexity than any comparison-based algorithms such as quicksort or our SIMD-based algorithms. However, its scattered memory accesses make it difficult for radix sort to exploit SIMD instructions. The scattered memory accesses also tend to increase the

required main memory bandwidth and thus the radix sort may suffer from a poor scalability on multicore processors because the system memory bandwidth tends to become a bottleneck in sorting on systems with multicore processors [Satish et al. 2010].

Sanders and Winkel [2004] pointed out that the performance of sorting on today's processors is often dominated by pipeline stalls caused by branch mispredictions. They proposed a new sorting algorithm, named super-scalar sample sort (sss-sort), to avoid such pipeline stalls by eliminating conditional branches. They implemented the sss-sort by using the predicated instructions of the processor and showed that the sss-sort achieves up to 2 times higher performance over the STL sorting function delivered with gcc. Our algorithm can also avoid pipeline stalls caused by branch miss predictions. Moreover, our algorithm makes it possible to take advantage of the data parallelism of SIMD instructions.

There are some sorting algorithms suitable for exploiting SIMD instructions [Purcell et al 2003, Govindaraju et al. 2005, Govindaraju et al. 2006]. They were originally proposed in the context of sorting on graphics processing units (GPUs), which were powerful programmable processors with SIMD instruction sets.

Govindaraju *et al.* [2006] presented a sorting algorithm called GPUTeraSort that improved on bitonic mergesort [Batcher 1968]. The bitonic mergesort has computational complexity of $O(N \cdot (\log(N))^2)$ and it can be executed by up to N processors in parallel. The GPUTeraSort improves this algorithm by altering the order of comparisons to improve the effectiveness of the SIMD comparisons and also by increasing the memory access locality. Comparing our vectorized mergesort to the GPUTeraSort, both algorithms can be effectively implemented with SIMD instructions and both can exploit thread-level parallelism. An advantage of our algorithm, which is based on the (branch-based) usual mergesort, is the computational complexity of $O(N \cdot \log(N))$, which is the optimal complexity for any comparison-based sorting algorithm, while the complexity for the GPUTeraSort (or other bitonic mergesort variants) is $O(N \cdot (\log(N))^2)$. Gedik *et al*. [2007] presented a sorting algorithm for Cell BE [Pham et al. 2005] called the CellSort. They also used the bitonic mergesort as their computing kernel to exploit the SIMD instruction set and thread-level parallelism of the processor. Thus the computational complexity of their algorithm was larger than ours.

Furtak *et al*. [2007] showed the benefits of exploiting SIMD instructions for sorting very small arrays. They demonstrated that replacing only the last few steps of quicksort by a

sorting network implemented with SIMD instructions improved the performance of the entire sort by up to 22%. They evaluated the performance benefits for the SSE instructions and the VMX instructions. Our algorithm can take advantage of SIMD instructions not only in the last part of the sorting, but also for entire stages.

Cederman and Tsigas [2008] demonstrated that their quicksort implementation on recent NVIDIA GPUs achieved much better performance than quicksort on general-purpose CPUs or the GPUTeraSort running on the same GPUs. Their quicksort for GPUs exploits the flexible memory access mechanisms of the recent GPUs. With these GPUs, each slot of a vector load or store instruction can access an arbitrary memory address, while the corresponding SIMD instruction sets suffers from accessing non-contiguous memory addresses. Our algorithm can run efficiently with SIMD instructions of general-purpose processors, which have more limitations than the GPUs. Also, avoiding non-contiguous memory accesses can reduce the memory access overhead even for GPUs supporting gather and scatter memory accesses by coalescing multiple memory accesses.

3.3 Our SIMD Sorting Algorithm

In this section, we present our new sorting algorithms: vectorized mergesort and vectorized combsort. We use 32-bit integers as the data type of the elements to be sorted and 128-bit vector length for the explanation; hence one 128-bit vector register contains four values. Note that our algorithm is not limited to this data type and degree of data parallelism as long as the SIMD instructions support them. Actually, we also evaluated the performance with 256-bit vector length using AVX2 instruction set. We assume the first element of the array to be sorted is aligned on a 128-bit boundary and the number of elements in the array, N, is a multiple of the degree of data parallelism of the SIMD instructions for ease of explanation. Figure 3 illustrates the layout of the array, $a[N]$. The array of integer values $a[N]$ is equivalent to an array of vector integers $va[N/4]$. A vector integer element $va[i]$ consists of the four integer values of $a[i*4]$ to $a[i*4+3]$.

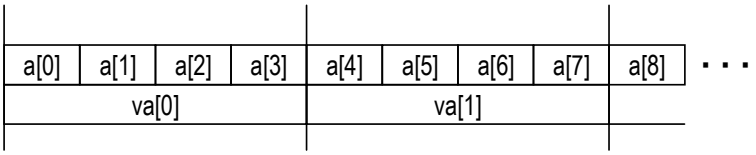


Figure 3. Data structure of the array to be sorted.

Our overall sorting scheme consists of the following phases using the two algorithms:

1) Divide all of the data into blocks that fit into the cache of the processor and sort each block with the vectorized combsort.
2) Merge the sorted blocks with the vectorized mergesort.

First we present these two vectorized sorting algorithms and then illustrate the overall sorting scheme.

3.3.1 Vectorized Mergesort

For the vectorized mergesort, we propose an innovative method to integrate a sorting network, such as bitonic merge or odd-even merge [Batcher 1968], implemented with SIMD instructions into a usual branch-based merge algorithm. Our method makes it possible for the merge operations to take advantage of SIMD instructions while still retaining the computational complexity of O(N). The bitonic merge and the odd-even merge, also suitable for implementing with SIMD instructions, have the computational complexity of O(N·log(N)) instead of the complexity of O(N) of our algorithm.

Figure 4 shows the data flow of the bitonic merge operation for eight values stored in the two vector registers, which contain four sorted values each. In the figure the boxes with inequality symbols signify comparison operations. Each of them reads two values from the two inputs (one each) and sends the smaller value to the left output and the larger one to the right. The odd-even merge operation requires log(P)+1 stages to merge two vector registers, each of which contain P elements. Here P=4 and log(P)+1=3. Each stage executes only one vector compare, two vector select, and one or two vector permutation instructions.

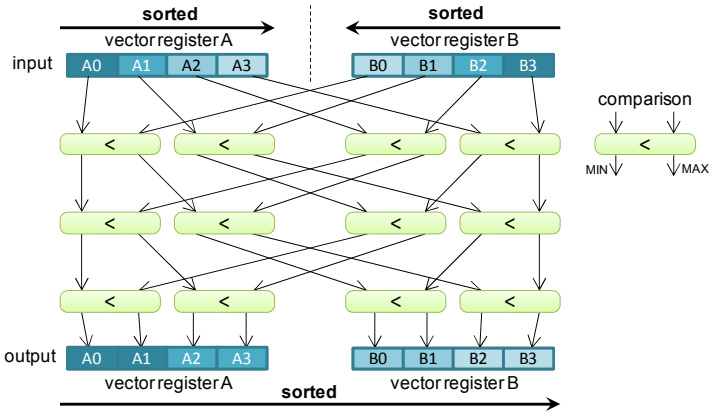


Figure 4. data flow of bitonic merge operation for two vector registers

The merge operation for two large arrays stored in memory can be implemented using this merge operation for the vector registers. Figure 5 shows the pseudocode for merging of two vector integer arrays *va* and *vb*. In this code, the *vector_merge* operation is the merge operation for the vector registers shown in Figure 4. In each iteration, this code

1) executes a merge operation of two vector registers, *vMin* and *vMax*,
2) stores the contents of *vMin*, the four smallest values, as output,
3) compares the next element of each input array, and
4) loads four values into *vMin* from the array whose next element is smallest and advances the pointer for the array.

Loading new elements from only one input array is sufficient, because at least one of the next elements of each input array must be larger than all of the data values in *vMax* and hence the larger of the two next elements must not be contained in the next four output values. There is only one conditional branch for the output of every *P* elements, while the naive merge operation requires one conditional branch for each output element.

```
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++];
}
```

Figure 5. Pseudocode of the (2-way) merge operation in memory.

The vectorized mergesort recursively repeats the merge operation described earlier. It does not require any unaligned memory accesses. However, it has lower performance than our vectorized combsort for the small amounts of data that can fit in the cache because the vectorized mergesort reduces the number of data-dependent conditional branches but still uses them while the vectorized combsort totally eliminates them as describe later. In contrast,

the vectorized mergesort achieves higher performance than the vectorized combsort when the data cannot fit in the cache. This is because the vectorized mergesort has much better memory access locality compared to the vectorized combsort, since it scans all of the values to be sorted in each iteration. Therefore, we switch to the vectorized combsort when the data becomes small enough. Switching to another algorithm for small block is a common technique for the mergesort (or quicksort) even implemented without SIMD.

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. After the first publication of our algorithm [Inoue et al. 2007], Chhugani *et al.* [2007] pointed out that 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. In our current implementation, we combine two vector registers and hence one vector merge operation merges two sequences of 8 elements into one 16-elements sequence with 128-bit SIMD (SSE) instructions. For AVX2 implementation, it reads two 16-elements sequences and merges them into one sequence of 32 elements.

In order to reduce the required memory bandwidth, which limits the performance of sorting when using many cores, we use a *multiway* merge technique. Multiway mergesort [Knuth 1973] 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 5 as a building block, we execute multiway merge operations consisting of multiple 2-way merge operations in a streaming manner using small memory buffers (16 KB each in our implementation) that can fit within the processor's cache memory. Figure 6 illustrates how we implement the 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.

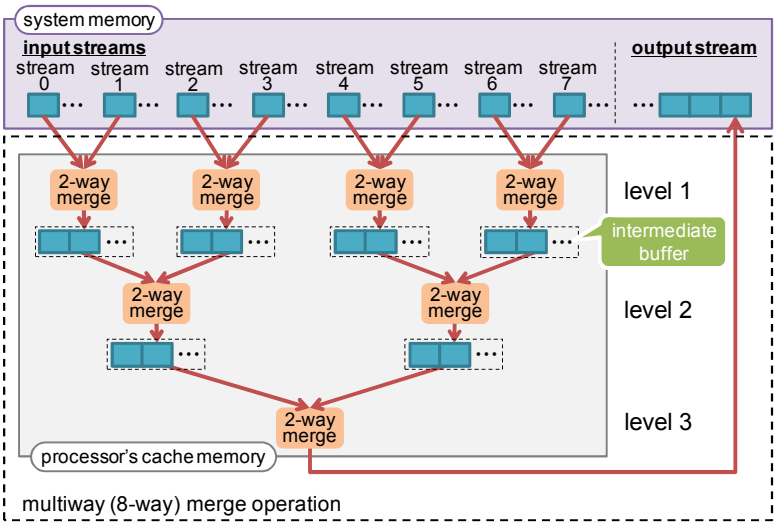


Figure 6. Overview of multiway merge operation with SIMD. Number of ways $k = 8$ in this example.

3.3.2 Vectorized Combsort

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.

Our vectorized combsort improves on combsort [Lacey and Box 1991], an extension to bubble sort. Bubble sort compares each element to the next element and swaps them if they are out of sorted order. Combsort compares and swaps two non-adjacent elements. Comparing two values with large separations improves the performance drastically, because each value moves toward its final position more quickly. Figure 7 shows the pseudocode of combsort. The separation (labeled *gap* in Figure 7) is divided by a number, the *shrink factor*, in each iteration until it becomes one. The authors used 1.3 for the shrink factor. Then the

final loop is repeated until all of the data is sorted. The computational complexity of combsort is almost proportional to $N \cdot \log(N)$ on average.

```
/* first, swap elements with larger gaps */
gap = N / SHRINK_FACTOR;
while (gap > 1) {
  if (gap == 9 || gap == 10) gap = 11;
  for (i = 0; i < N - gap; i++)
    if (a[i] > a[i+gap]) swap(a[i], a[i+gap]);
  gap /= SHRINK_FACTOR;
}

/* repeat bubble sort until fully sorted */
do {
  for (i = 0; i < N - 1; i++)
    if (a[i] > a[i+1]) swap(a[i], a[i+1]);
} while( not fully sorted );
```

Figure 7. Pseudocode of combsort.

The fundamental operation of many sorting algorithms including combsort and bitonic mergesort is to compare two values and swap them if they are out of order. Each conditional branch in this operation will be taken in arbitrary order with roughly 50% probability for random input data, and therefore it is very hard for branch prediction hardware to predict the branches. This operation can be implemented using vector compare and vector select instructions without conditional branches; this replace control flow by data flow and hence it reduces the branch misprediction overhead.

Combsort has two problems that reduce the effectiveness of SIMD instructions: 1) unaligned memory accesses and 2) loop-carried dependencies. Regarding the unaligned memory accesses, combsort requires unaligned memory accesses when the value of the gap is not a multiple of the degree of data parallelism of the SIMD instructions. A loop-carried dependency prevents exploitation of the data parallelism of the SIMD instructions when the value of the gap is smaller than the degree of data parallelism.

In our vectorized combsort, we resolved these problems with combsort. The key idea of our improvement is to first sort the values into the *transposed* order and reorder the sorted values into the original order after the sorting. Figure 8 shows the steps of our vectorization technique for combsort. It consists of the following 3 steps:

1) sort values within each vector,

2) execute combsort to sort the values into the transposed order, and then

3) reorder the values from the transposed order into the original order.

Figure 8 shows an example for an array with 16 integers (or four vector integers).

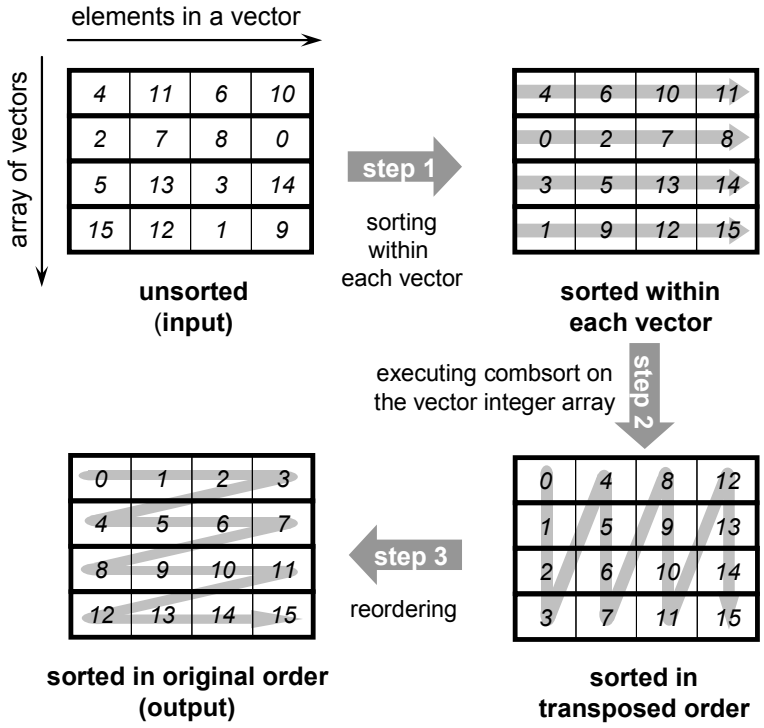


Figure 8. Steps of our vectorized combsort algorithm for sorting 16 values (4 vectors).

Step 1 sorts four values in each vector integer $va[i]$ ($0 \le i < N/4$, here N=16 in Figure 8). This step corresponds to the loops with the gaps of $N/4$, $N/4*2$, and $N/4*3$ in combsort, because the gap between consecutive elements in one vector register is $N/4$ in the transposed order. The sorting for values in a vector register can be implemented as a sorting network using vector comparison and vector permutation instructions [Furtak 2007].

Step 2 executes combsort on the vector integer array $va[N/4]$ in the transposed order. Figure 9 shows pseudocode for our vectorized combsort algorithm. In this code, *vector_cmpswap* is an operation that compares and swaps values in each element of the vector register A with the corresponding element of the vector register B as shown in Figure 10. This operation can be implemented using a pair of vector minimum and maximum instructions or one vector compare instruction and two vector select instructions. Similarly

vector_cmpswap_skew is an operation that compares and swaps the first to third elements of the vector register A with the second to fourth elements of the vector register B. It does not change the last element of the vector register A and the first element of the vector register B. Both operations can be implemented using SIMD instructions. Comparing the code of Figure 9 to the code of the original combsort in Figure 7, the innermost loop is divided into two loops with these two operations. With these two loops, all of the values are compared and swapped with the values for the distance of the *gap* in the transposed order. The original loop was divided into two because the pairs to be compared may reside in the same positions of the vector registers or in different positions.

The last do-while loop of step 2 executes bubble sort to assure the correct order of the output. In order to guarantee against the worst case performance of $O(N^2)$ caused by the bubble sort, we cancel the last loop in the Step 2 after executing a constant number of iterations. In our algorithm, we use 10 for the threshold and switch to the vectorized mergesort, whose complexity is $O(N \cdot \log(N))$ even for the worst case, when the execution of the combsort is canceled. In practice, however, we have never observed the cancellations of the vectorized combsort in our evaluations.

Step 3 reorders the sorted values into the correct order. This step does not require data-dependent conditional branches because it only moves each element in predefined orders, and hence the reordering does not incur troublesome overhead. Vector permutation instructions can efficiently execute this step when the number of vectors is a multiple of the number of elements in each vector, four in this example. For example, the four vectors in Figure 8 can be reordered by using only 8 vector permutation instructions. The first four permutations swap the upper-right 2x2 block (consists of 8, 9, 12, and 13) and the lower-left one (2, 3, 6, and 7). The next four permutations swap the upper-right value and lower-left value in each 2x2 block (such as 1 and 4). After applying this vectorized transposition technique, all of the vectors contains four sequential values, and thus the program can reorder the values into the original order by simply moving vectors with vector load and vector store instructions. For the final data moves, our implementation uses a temporary memory space with the same size as the data. This step also does not cause any unaligned memory accesses.

```
/* Step 1 */
for (i = 0; i < N/4; i++) {
  sort_within_a_vector(va[i]);
}

/* Step 2 */
gap = (N/4) / SHRINK_FACTOR;
while (gap > 1) {

  /* straight comparisons */
  for (i = 0; i < N/4 - gap; i++)
      vector_cmpswap(va[i], va[i+gap]);

  /* skewed comparisons when i+gap exceeds N/4   */
  for (i = N/4 - gap; i < N/4; i++)
      vector_cmpswap_skew(va[i], va[i+gap - N/4]);

  /* dividing gap by the shrink factor */
  gap /= SHRINK_FACTOR;
}

/* executing bubble sort */
loop_count = 0;
do {
  for (i = 0; i < N/4 - 1; i++)
      vector_cmpswap(va[i], va[i+1]);
  vector_cmpswap_skew(va[N/4-1], va[0]);
} while( not totally sorted && loop_count++ < THRESHOLD);

/* abort and switch another algorithm when loop_count reaches threshold */
if (not totally sorted) return false;

/* Step 3 */
for (i = 0; i < N/16; i++)
  transpose_4x4_block(va[i*4], va[i*4+1], va[i*4+2], va[i*4+3]);

for (i = 0; i < N/4; i++)
  move_vector_to_desired_location(va[i]);

return true; /* completed successfully */
```

Figure 9. Pseudocode of our vectorized combsort.

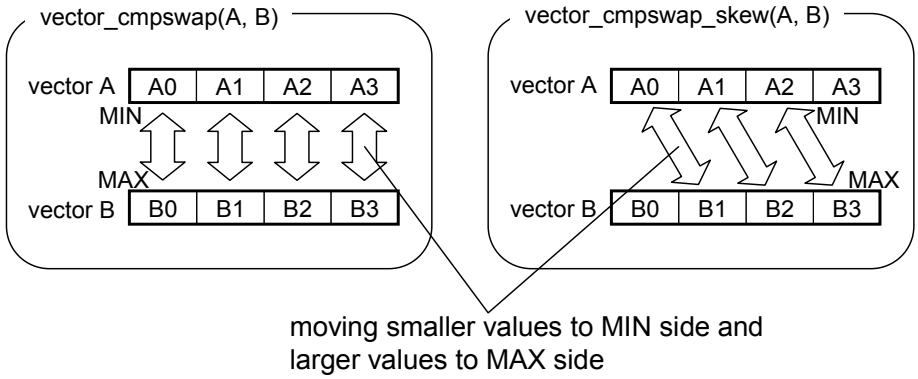


Figure 10. The vector_cmpswap and vector_cmpswap_skew operations.

In summary, our vectorized combsort consists of three steps. All three of the steps can be executed by SIMD instructions without unaligned memory accesses. Also, all of them can be implemented with a negligible number of data-dependent conditional branches.

Let N be the total number of elements to be sorted. The computational complexity of Step 1 and Step 3 is $O(N)$, and that of Step 2 is the same as that of combsort, $O(N \cdot \log(N))$ on average. Thus the computational complexity of the entire algorithm is dominated by Step 2. In Step 2, we cancel the execution of the vectorized combsort if the number of iterations exceeds a constant threshold and switch to the vectorized mergesort to guarantee the $O(N \cdot \log(N))$ complexity even for the worst case.

Our vectorized combsort suffers from poor memory access locality. Thus its performance may degrade if the data cannot fit into the cache of the processor. We have proposed another sorting algorithm, the vectorized mergesort, which takes that problem into account.

3.3.3 Overall Parallel Sorting Scheme

Our overall sorting algorithm executes the following phases using the two algorithms:

1) Divide all of the data to be sorted into blocks that fit in the cache of the processor and sort each block with the vectorized combsort in parallel using multiple threads, where each thread processes an independent block.

2) Merge the sorted blocks with the vectorized multiway mergesort using multiple threads.

Figure 11 shows the pseudocode for the entire sorting scheme and the Figure 12 depicts an example of the entire sorting execution with four parallel threads.

```
/* in-core sorting phase for small blocks*/
numBlocks = N / B;
blockSize = B;
blocksPerThread = numBlocks / numThreads;

for (i = blocksPerThread * myThreadID; i < blocksPerThread * (myThreadID+1); i++) {

  /* parameters are a pointer to the input data, a number of elements to sort,
and a threshold to cancel executing combsort */
  sorted = vectorized_combsort(data[blockSize*i], blockSize, 10);

  /* switch to vectorized mergesort if combsort did not completed
     within the predifined number of iterations */
  if (!sorted) vectorized_mergesort(data[blockSize*i], blockSize);
}

/* out-of-core merging phase */
while (numBlocks > 1) {
  blocksPerThread = numBlocks / numThreads;

  /* if there are enough blocks to execute 4-way merge by each thread */
  if (numBlocks >= numThreads * 4) {
    for (i = blocksPerThread * myThreadID; i < blocksPerThread * (myThreadID+1); i+=4)

      /* parameters are four pointers to the input data buffers, */
      /* a pointer for the output buffer,                        */
      /* and a number of elements to merge in each input data    */
      vectorized_4way_merge(data[blockSize*i  ], data[blockSize*i+1],
                            data[blockSize*i+2], data[blockSize*i+3],
                            tmp [blockSize*i  ], blockSize);
    numBlocks /= 4;  blockSize *= 4;
  }

  /* if there are enough blocks to execute 2-way merge by each thread */
  else if (numBlocks >= numThreads * 2) {
    for (i = blocksPerThread * myThreadID; i < blocksPerThread * (myThreadID+1); i+=2)
      vectorized_2way_merge(data[blockSize*i], data[blockSize*i+1],
                            tmp [blockSize*i], blockSize);
    numBlocks /= 2;  blockSize *= 2;
  }

  /* if there are not enough blocks to work all thread independently */
  else {
    barrier(); /* a barrier synchronization among threads */
    numThreadsToCooperate = numThreads / (numBlocks / 2);
    assignedBlock = myThreadID / numThreadToCooperate;
    vectorized_merge_with_multiple_threads(data[ 2*assignedBlock    * blockSize],
                                           data[(2*assignedBlock+1) * blockSize],
                                           tmp [ 2*assignedBlock    * blockSize], blockSize,
                                           blockSize, numThreadsToCooperate, myThreadID);
    numBlocks /= 2;  blockSize *= 2;
  }
  swap(data, tmp);  numMergeStages++;  /* swap pointers for the input and output buffers */
}

if (numMergeStages & 1) { memcpy(tmp, data, N * sizeof(element type)); }
```

Figure 11. Pseudocode of the entire algorithm

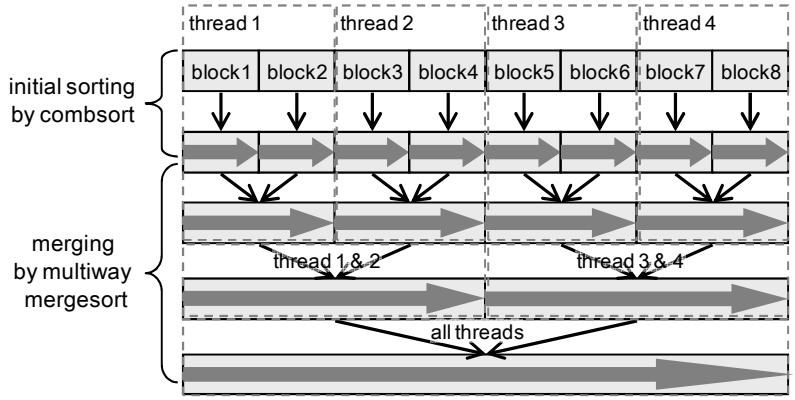


Figure 12. An example of the entire sorting process, where number of blocks (N/B) = 8 and the number of threads (c) = 4.

The block size for the initial sorting phase (B) is an important parameter. The selection of the block size depends on bandwidth and latency for each level of memory hierarchy of the target system. In our implementation for Xeon (SandyBridge-EP) processors, for example, we use the size of L1D cache (32 KB) based on the empirical evaluations.

If the total number of elements of data to sort is N and the number of elements in one block is B, then the number of blocks for the initial sorting is (N/B). The computational complexity of the initial sorting of each block is $O(B \cdot \log(B))$. We avoid the worst case computational time of $O(B^2)$ and guarantee the complexity for the initial sorting by switching from the vectorized combsort to vectorized mergesort. Hence the total computational complexity of the initial sorting phase is $(N/B) \cdot O(B \cdot \log(B)) = O(N \cdot \log(B))$. The sorting of each block is independent of the other blocks, so they can run in parallel on multiple threads up to the number of blocks. Thus the total complexity of the initial sorting phase with multiple threads is $O(N \cdot \log(B)/c)$ assuming the number of threads, c, is smaller than the number of blocks, (N/B).

In the merging phase, merging the sorted blocks involves $\log(N/B)$ stages and the computational complexity of each stage is $O(N)$, and thus the total computational complexity of this phase is $O(N \cdot \log(N/B))$, even in the worst case. Note that this complexity is not changed with the multiway merge technique. With the multiway (k-way) merge, the number of stages is reduced from $\log_2(N/B)$ to $\log_k(N/B)$, but the number of comparisons involved in one stage increases. It only reduced the required system memory bandwidth. The total complexity of the merging phase with c threads is $O(N \cdot \log(N/B)/c)$. For parallelizing the last few stages of the merging phase, the number of blocks becomes smaller than the number of

threads, and hence multiple threads must cooperate on one merge operation to fully exploit the thread-level parallelism [Francis 1988].

The entire algorithm has the computational complexity of $O(N \cdot \log(N))$, where $O(N \cdot \log(B))$ for the initial sorting phase and $O(N \cdot \log(N/B))$ for the merging phase, even for the worst case. Also it can be executed in parallel by multiple threads with complexity of $O(N \cdot \log(N)/c)$ assuming the number of threads is smaller than the number of blocks, (N/B).

3.4 Experimental Results

We implemented our SIMD-based sorting algorithms and the bitonic mergesort using 128-bit SIMD (SSE) instructions or 256-bit SIMD (AVX) instructions and evaluated it on Intel Xeon processors. We implemented the bitonic mergesort by following the GPUTeraSort [Govindaraju et al. 2006] for comparison because it is one of the best existing sorting algorithms for SIMD instructions. The CellSort [Gedik et al. 2007] uses the almost same algorithm for its sorting kernel. We also implemented a (non-SIMD) 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 [González et al. 1999]. We also applied the local-buffer-based optimization proposed by Satish *et al.* [2010] 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. 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). We also evaluated std::sort function of STL library delivered with gcc that implements a quicksort variant called introsort [Musser 1997]. We implemented the program in C++ using intrinsics provided by the compilers.

We used two systems for the evaluations. The first system 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. The system ran under Red hat Enterprise Linux 6.4. We compiled all the programs as 64-bit binaries using gcc-5.2.0 with the –O3 option. The second system, which we used to compare the performances with SSE and AVX instructions, was equipped with two 2.3-GHz Xeon E5-2699 v3 (Haswell-EP) processors with 768 GB of system memory. On this system, we compiled the programs using Intel C++ compiler 14.0.2 with the –O3 option.

On both systems, 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. We do not use additional hardware threads provided by the 2-way SMT (Hyper Threading) of the processor in the experiments.

For our vectorized mergesort, we use 128-way merge to reduce the system memory bandwidth requirement. For our vectorized combsort, we use 1.27 as the shrink factor. We use B = 8,192 elements (32 KB, the size of L1D cache) as the block size for the combsort. We empirically selected these parameter values. For the number of ways, the sorting performance was almost identical when we used 8-way to 256-way merge. For the shrink factor, we used slightly smaller value than the value used in the original paper, 1.3. When we applied combsort for very large arrays, the shrink factor slightly smaller than 1.3 gave much better performance; however, for small blocks, like those of 8,192 elements we used in our implementation, the shrink factor of 1.3 also worked well.

3.4.1 Performance of Vectorized Combsort and Vectorized Mergesort

This section focuses on the performance of each algorithm with one thread of SandyBridge with primary emphasis on the effects of SIMD instructions. Here, we only use 128-bit SIMD (SSE) instructions. In this section, we separately evaluate the two algorithms, our vectorized mergesort and our vectorized combsort, to illustrate the effect of SIMD instructions for each algorithm and determine the best block size for the initial sorting. In the later sections, we combine the two algorithms as described before.

Figure 13 compares the performance of the sorting algorithms for 8 K of random 32-bit integers using one thread. All of the data to be sorted can fit into the L1 cache of the processor. The performance of the combsort, the mergesort, and the bitonic mergesort with SIMD instructions were drastically improved compared to the implementations without using the SIMD instructions, and our vectorized combsort achieved the highest performance among all of the algorithms tested; our vectorized mergesort was the second best with about 35% performance gap against the vectorized combsort.

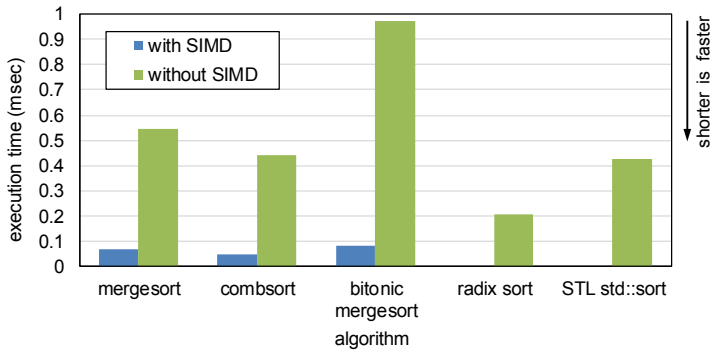


Figure 13. Acceleration by SIMD instructions for various algorithms when sorting 8 K random integers with one core.

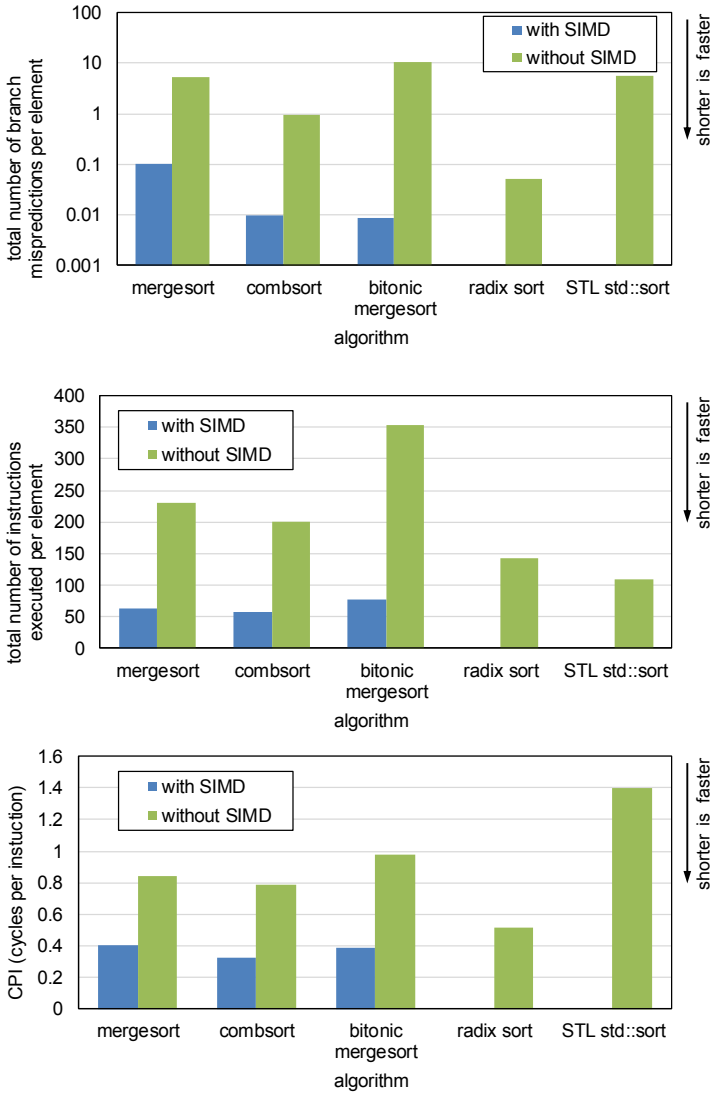


Figure 14. Improvements in the number branch mispredictions (shown in logarithmic scale), the number of executed instructions and the CPI by using SIMD instructions.

The degrees of acceleration with the SIMD instructions for mergesort, combsort and bitonic mergesort were 8.0x, 8.8x and 11.9x respectively. They were larger than the degree of parallelism available with the SIMD instructions (4x) due to the reduced branch misprediction overhead. Figure 14 shows the number of branch mispredictions, the number of executed instructions and the CPI (cycles per instruction) measured by using a performance counter of the processor. The branch misprediction rates were reduced by more than a factor of 10 for all algorithms. The change in misprediction rate was slightly smaller for the mergesort (about 50x reduction) because data-dependent conditional branches were reduced but not totally eliminated. The reductions in numbers of instructions were mainly due to the data parallelism of the SIMD instructions and the CPI improvements were due to the reduced branch overhead. For all three algorithms, the numbers of instructions were reduced almost in proportion to the degree of data parallelism available from the SIMD instructions (3.5x to 4.6x). The performance gain with SIMD instructions shown in Figure 13 related to these two factors: reductions in the numbers of instructions and improvements in cycles per instruction (CPI).

To determine the best value for the block size, we evaluated the performance of the vectorized combsort and the vectorized mergesort with different amounts of data. Figure 15 shows the relationship between the performances and the amounts of data for the vectorized mergesort and the vectorized combsort. The x-axis shows the number of elements to be sorted and the y-axis shows the sorting time. Both axes are displayed as logarithmic scales. The figure shows that the vectorized combsort was the fastest for all amounts of data equal or smaller than 32 K elements (128 KB, half of the size of the L2 cache). Especially, the performance differences were most significant when the data can fit into L1D cache; it was more than 40%. However, its performance degraded drastically when the amount of data exceeded the L2 cache size and it was slower for larger amounts of data. This was due to the high cache miss ratio caused by a poor access locality of the combsort. Based on this result, we selected 8 K elements, or 32 KB, as the block size for the initial sorting phase. This size corresponds to the size of the L1D cache of the processor.

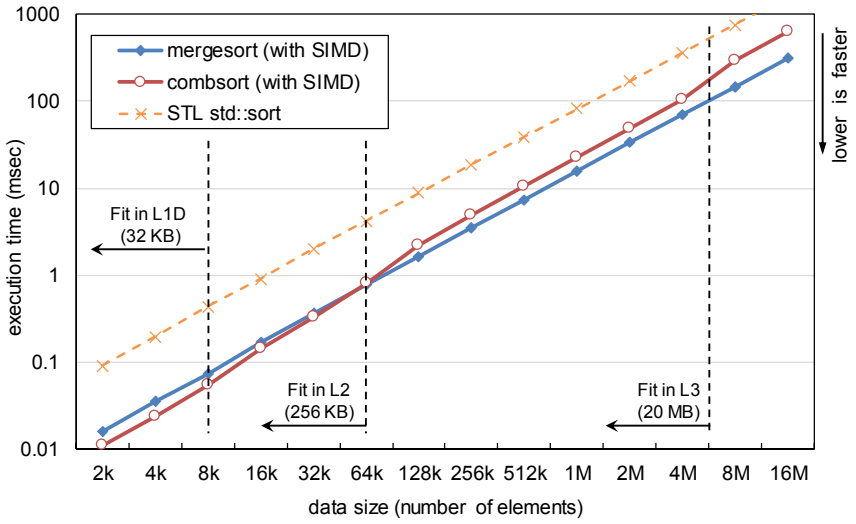


Figure 15. Performance comparisons of our vectorized combsort and vectorized mergesort for sorting random 32-bit integers with various amounts of data.

3.4.2 Performance of the Entire Sorting Algorithm for Large Arrays

In this section, we discuss the performance of sorting for large 32-bit integer arrays. Figure 16 compares the performance of four algorithms to sort random 32-bit integers on one core of SandyBridge. Our vectorized mergesort, which uses vectorized combsort for the initial sorting, and the bitonic mergesort were implemented with 128-bit SIMD instructions. The x-axis shows the number of elements up to 1-billion elements (4 GB) and the y-axis shows the execution time. Our vectorized mergesort achieved more than 5.2 to 6.1 times faster than the quicksort (STL) regardless of the data size. It also surpassed the performance of the bitonic mergesort implemented with SIMD instructions at least twice. The performance advantage of our vectorized mergesort over the bitonic mergesort became larger with larger amounts of data because of the larger computational complexity of the bitonic mergesort. Comparing to the radix sort, our vectorized mergesort was 40% faster to 4% slower depending on the data size. Although the computational complexity of the radix sort was better than the mergesort, the performances were not significantly different even for the largest data size we evaluated. We will show the results for larger input sizes, up to 32-billion integers, on Haswell later. In Figure 16, we used the vectorized mergesort with our combsort. When we use only the mergesort, the performances were slower by 12.0% to 21.5% depending on the size. Even for sorting a very large array, therefore, the algorithm used for the initial sorting mattered for the overall sorting performance.

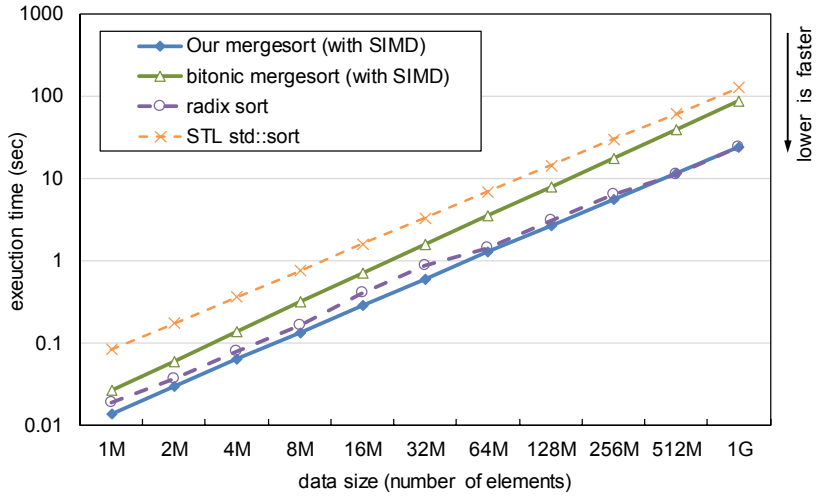


Figure 16. Performance of each algorithm on one core for sorting uniform random 32-bit integers with various data sizes.

Figure 17 shows the performance scalabilities of four algorithms with increasing number of cores for sorting 1-billion random integers using one core. The y-axis shows the relative (higher-is-better) performance over the STL's std::sort running on 1 core. The scalability of our vectorized mergesort was 7.8x using 8 cores and 13.9x using 16 cores. These speed ups were better than other algorithms; for example, the speed ups with 16 cores were 11.5x for the radix sort, 4.7x for the bitonic mergesort and 13.3x for STL. The better scalability of our vectorized mergesort was because the bitonic mergesort and the radix sort have higher communication/computation ratios and the memory bandwidth was a major bottleneck that limited the scalability.

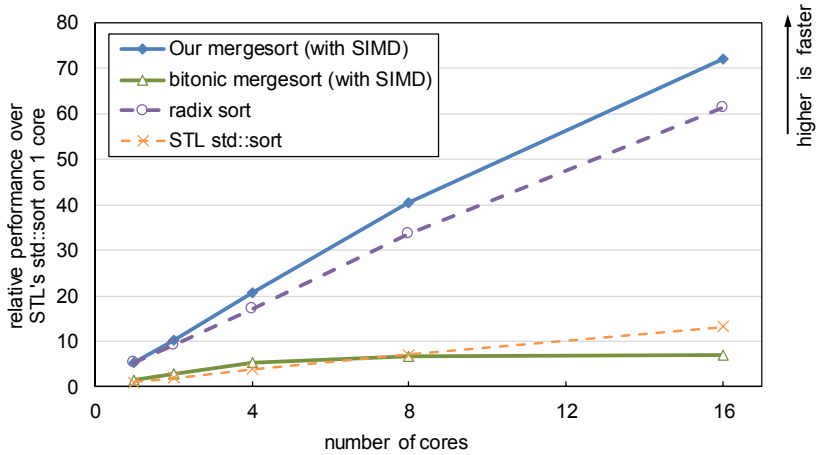


Figure 17. Performance scalabilities for sorting 1-billion random 32-bit integers with various data sizes.

To show the effect of the input data distribution, Figure 18 compares the performance of algorithms with different numbers of random key bits. For example, *key* is initialized by rand32() & 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 STL improved significantly because of the reduced branch misprediction overheads. The mergesort and the bitonic mergesort showed almost constant sorting times regardless of the data distribution because these algorithms replace many of the hard-to-predict conditional branches by the SIMD min and max instructions and hence the performance of the two algorithms were not significantly improved with the reduced entropy. The performance of the radix sort was also not significantly affected by the entropy of the input set because it does not use conditional branches to reorder elements. However, its performance was fluctuated depending on the input data due to cache behaviors.

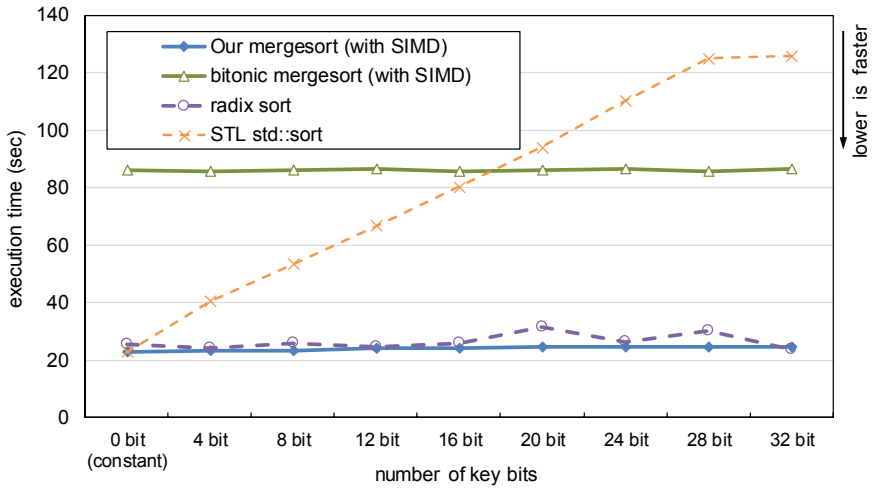


Figure 18. Execution times for sorting 1-billion 32-bit integers with different random key bits. For 8-bit case, keys were initialized with rand32() & 0xFF.

Here, we study how longer SIMD instructions can improve the performance of our vectorized mergesort. We evaluated the performance of our mergesort implemented with 128-bit SSE instructions and 256-bit AVX instructions on the Haswell system. Figure 19 illustrates performance improvements of our vectorized mergesort over std::sort with 128-bit or 256-bit SIMD instructions when sorting up to 32 billion random 32-bit integers (128 GB) on one core. By using the 256-bit SIMD instructions, we achieved 1.3x to 1.5x performance improvements over the 128-bit SIMD implementation. The gain was smaller than the increase in the data parallelism in one instruction (2x) because the vector merge kernel, implementing the bitonic mergesort, has a computational complexity larger than O(N) and

hence the number of instructions cannot be reduced by 2x. Also, the branch misprediction overhead was small enough even with 128-bit SIMD instructions and using the longer 256-bit SIMD instructions did not give additional performance benefits. Comparing our mergesort against the radix sort, the performance of radix sort was almost comparable against the SSE implementation of our mergesort. The AVX implementation of our mergesort consistently outperformed the radix sort regardless of the data size. Because the vector length of SIMD instructions is getting longer, these results show a promising future of our algorithm; our vectorized mergesort will be able benefit from longer SIMD instructions of future processors.

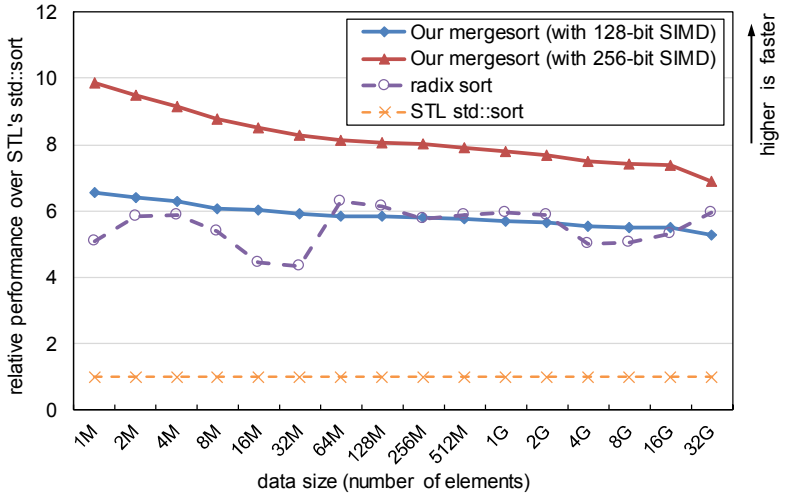


Figure 19. Performance improvements with our mergesort implemented by 128-bit SSE instructions and 256-bit AVX instructions over the STL's std::sort for sorting various number of random 32-bit integers.

At last, we investigated the energy efficiency of our algorithm. To evaluate the energy efficiency, we measured the energy consumption of a commodity PC that was connected to an external power meter (Yokogawa WT210[1]) while running each algorithm. This PC was equipped with one quad-core 3.4-GHz Core i7 4770 processor (Haswell) and 4 GB of system memory. All programs were compiled with gcc-5.2 and ran on Red hat Enterprise Linux 6.5. Figure 20(a) shows the measured energy consumption rate (Watt) by the whole system for our vectorized mergesort, the radix sort, and STL's std::sort when sorting 256-million random integers using one core or four cores. Because only our vectorized mergesort used the vector processing unit in the core and hence the energy consumption rate was slightly

[1] http://www.yokogawa.com/tm/wtpz/wt210/tm-wt210_01.htm (Accessed December 8, 2015)

higher for the vectorized mergesort compared to the radix sort and the STL. On this system, performance advantage of our mergesort over the radix sort was 3.8% on one core and 16.0% on four cores. When we calculate the total energy consumption for sorting by multiplying the measured energy consumption rate and the execution time, our vectorized mergesort consumed about 8.5% less energy than the radix sort on four cores while it was almost comparable for two algorithms on only one core. STL's std::sort was far less energy efficient compared to other two algorithms. Note that on this system with a client processor, the scalabilities with multiple cores were much poorer than those on the server processors shown in Figure 17 due to limited system memory bandwidth; 2.1x gain by 4 cores for our vectorized mergesort and 1.9x for the radix sort.

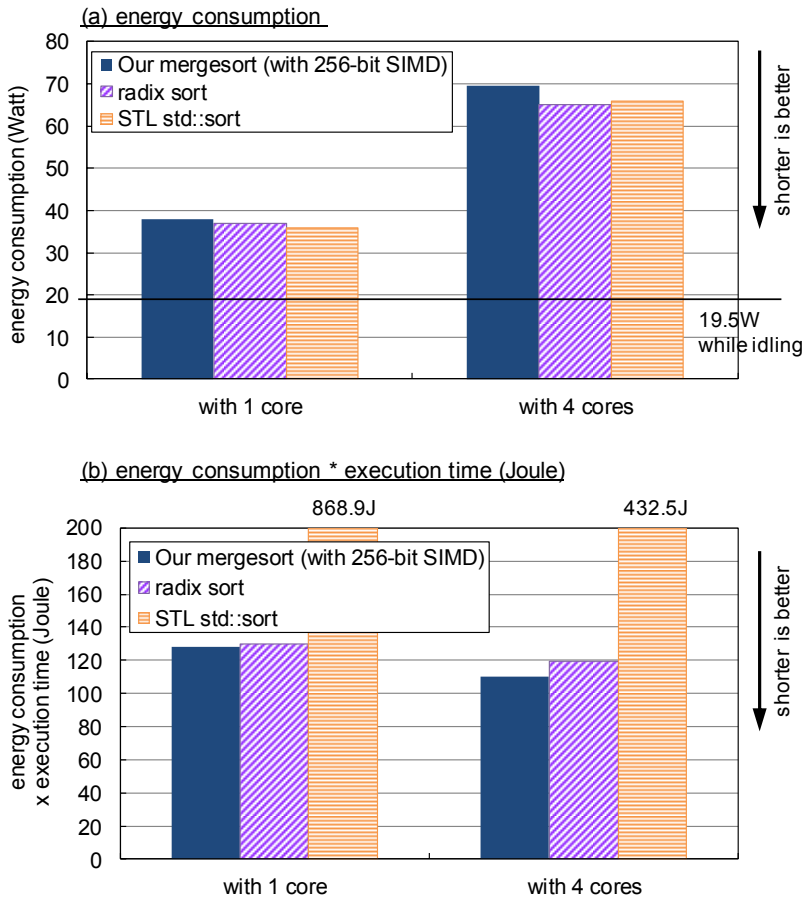


Figure 20. (a) Energy consumption rate (Watt) while sorting 256M random integers. (b) Total consumed energy (Joule) for sorting calculated from energy consumption rate and execution time.

3.5 Summary

This chapter describes a new high-performance sorting algorithm based on the multiway mergesort and the combsort, which is suitable for exploiting both the SIMD instructions and thread-level parallelism available on today's multicore processors. Our new algorithm does not involve any unaligned memory accesses that attenuate the benefit of SIMD instructions, and hence it can effectively exploit the SIMD instructions. We implemented and evaluated our algorithm using 128-bit SIMD (SSE) instructions and 256-bit SIMD (AVX) instructions on Xeon processors. In summary, our algorithm using SIMD instructions outperformed other implementations of comparison-based sorting algorithm such as STL's std::sort, which implements a quicksort variant, and a SIMD implementation of the bitonic mergesort when sorting a large array of random 32-bit integers. Comparing against an optimized radix sort, our algorithm achieved almost comparable performance using SSE and better performance using AVX instructions. Since the vector length of SIMD instructions are getting longer and longer on multicore processors today to achieve even higher peak performance, the performance gain from the longer SIMD instructions is an important characteristic of our algorithm. Also, our new algorithm showed better scalability with increasing number of cores than the radix sort and the bitonic mergesort.

Chapter 4

Sorting for Structures

4.1 Introduction

As discussed in the previous chapter, sorting is a fundamental operation for many software systems; hence, a large number of sorting algorithms have been devised. 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*. 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 [Satish et al. 2010]. 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 chapter[†], 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 *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 2.1x 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

[†] Most of the contents in this chapter was published in *PVLDB* Vol. 8, No. 11, pp 1274–1285 [Inoue et al. 2015].

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.

4.2 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 [Kim et al. 2009, Balkesen et al. 2013, Polychroniou and Ross 2014] or in distributed sorting systems running on large-scale supercomputers [Sunder et al. 2013] or clusters [Kim et al. 2012]. 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.* [2007] 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.* [2007] 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.* [2010] 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 [Batcher 1968], without combining them with the standard comparison-based mergesort. These sorting networks can be implemented efficiently using the SIMD instructions on CPUs [Gedik et al. 2007] or GPUs [Govindaraju et al. 2006]. 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.

In this chapter, we focus 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 [Satish et al. 2010] 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.* [2009] 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 4.3.2.

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 and Ross [2002] and Inoue *et al.* [2014] also used a similar idea to improve the performance of the nested-loop join and the set intersection (merge join).

4.3 Our Approach for Sorting Structures

As described in Section 4.2, the two existing approaches to sort an array of structures, direct approach and key-index approach, have different drawbacks; the key-index approach is not cache friendly and the direct approach is not SIMD friendly. 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.

4.3.1 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 6, in Chapter 3, 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 6 to make it more efficient with SIMD instructions.

The problem with the multiway merge in Figure 6 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 21 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 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.

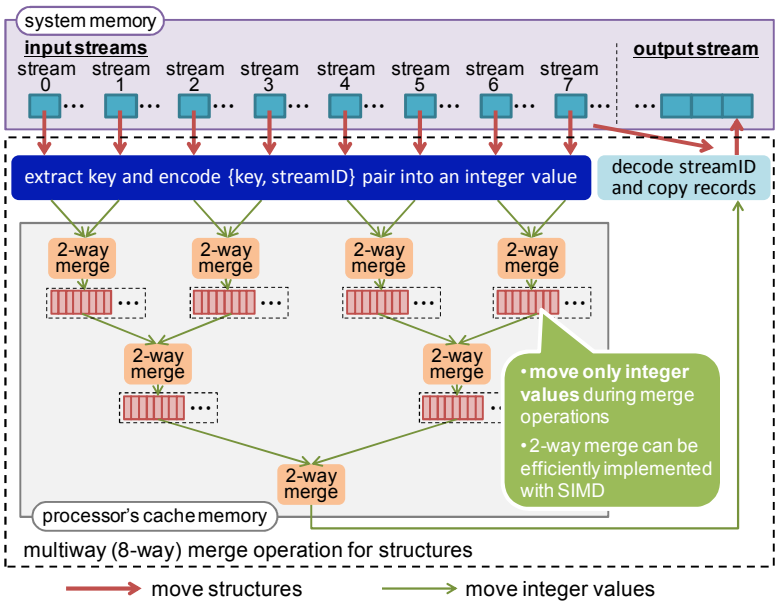


Figure 21. Overview of SIMD-and cache-friendly multiway merge operation of our approach. $k = 8$ in this example.

Figure 22 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.

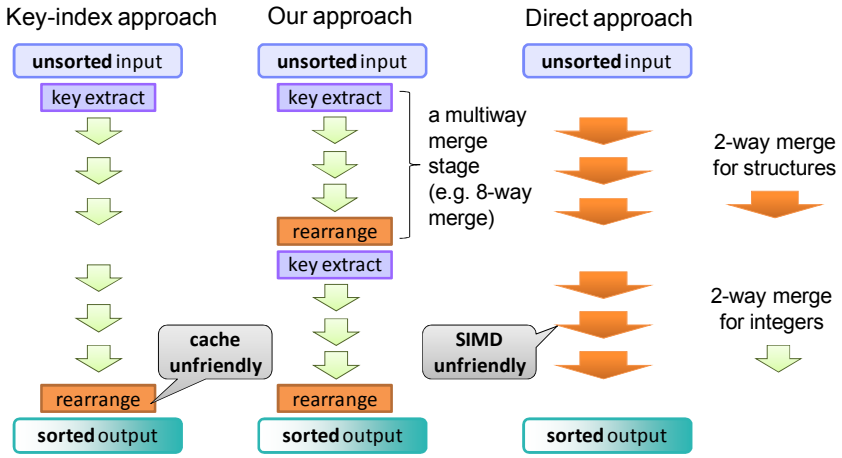


Figure 22. Overview of three approaches

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.

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

4.3.2 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 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.

4.3.3 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.

Increasing data parallelism

As discussed in Section 4.3.2, 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 21 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 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 4.4.

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.* [2007] used an odd-even merge on PowerPC and Cell BE processors. On Intel processors, Chhugani *et al.* [2007] 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.

(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);
}
```

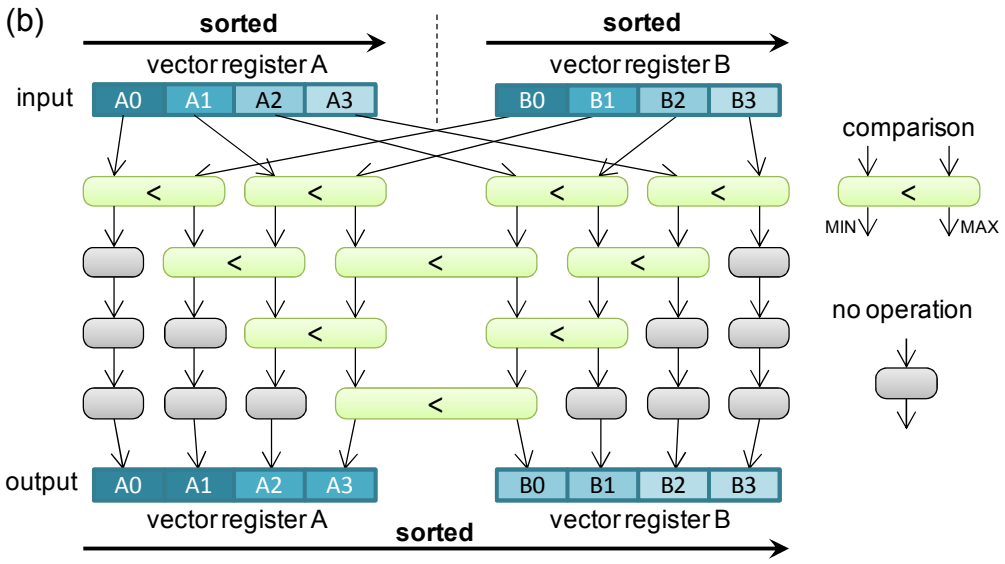


Figure 23 (a) Pseudo code of in-register vector merge for 32-bit integers (b) data flow of merge_4x4_32bit method.

In this chapter, 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 (*merge_4x4_32bit* in Figure 23). 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 23). As discussed before, 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. Although it uses more comparisons than the bitonic merge (shown in Figure 4) or the odd-even merge, it slightly outperformed the bitonic merge operation previously used on the Intel platform by avoiding the costly permute instructions when we used gcc-4.8. Note that, we observed the bitonic merge achieved slightly better performance with Intel compiler or newer versions of gcc but the performance gap was not significant.

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.

4.3.4 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 Chapter 3. 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 4.3.3). 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 4.3.3.

4.3.5 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.

4.4 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 Red hat 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 4.3 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 graphed the performance of the parallel versions of the STL sort functions and the radix sort. For the STL's sorting functions, we tested 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 <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 [Musser 1997], is an unstable sorting algorithm.

4.4.1 Performance Comparisons

Figure 24 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.

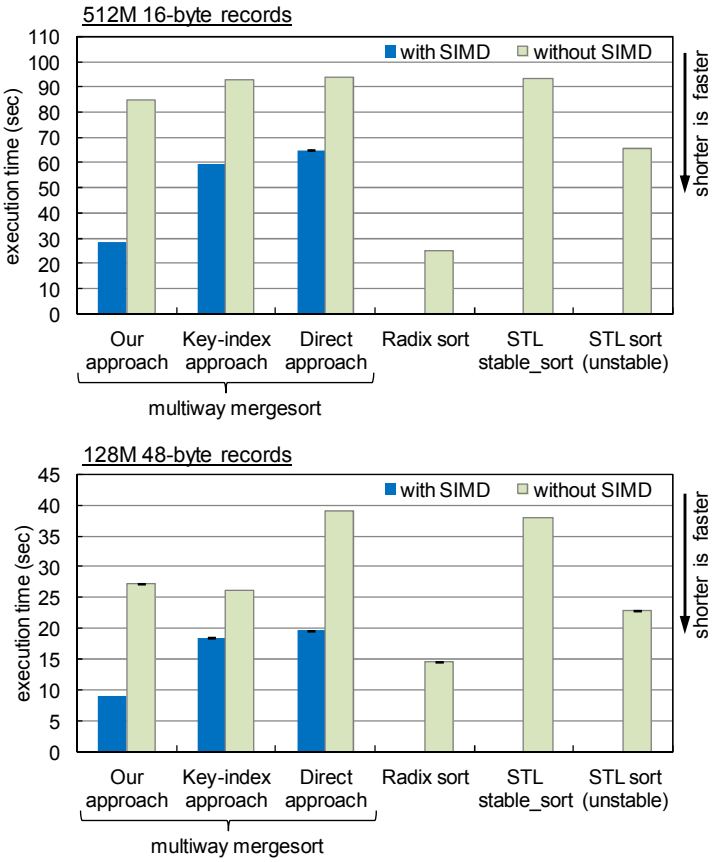


Figure 24. Execution time on 1 core for sorting 512M 16-byte records and 128M 48-byte records with 32-bit random integer keys using various algorithms implemented with and without SIMD instructions.

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 25 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 25, 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.

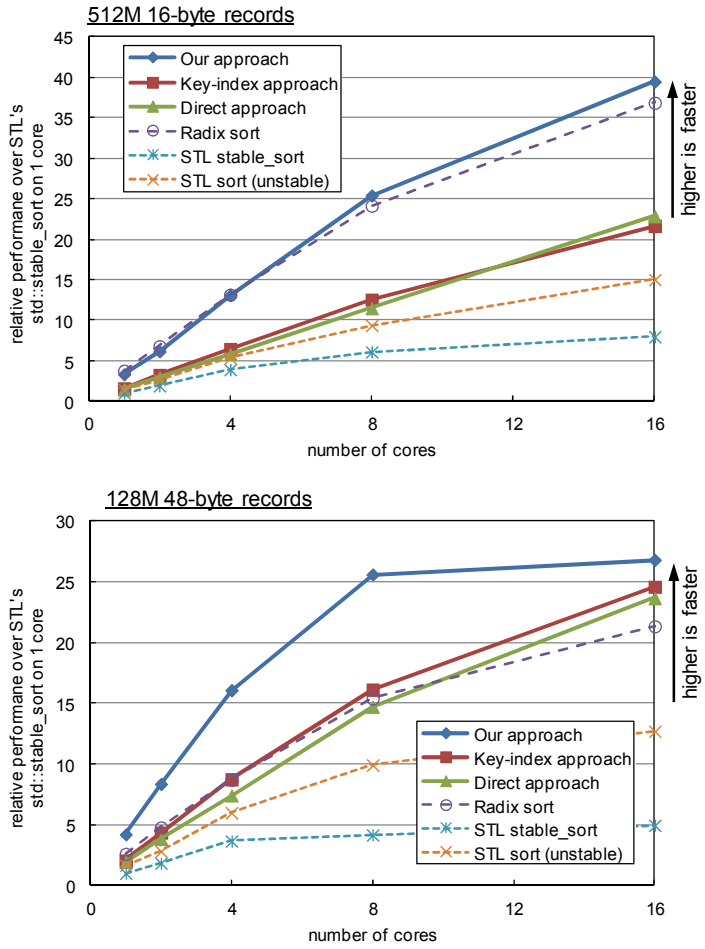


Figure 25. Performance scalability with increasing number of cores when sorting 512M 16-byte records and 128M 48-byte records.

Figure 26 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.

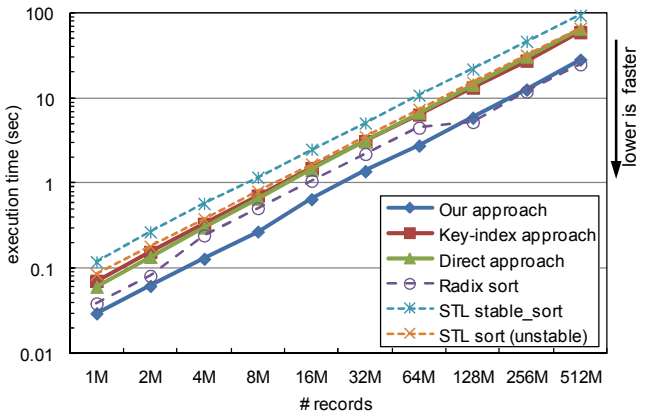


Figure 26. Performance scalability with increasing number of 16-byte records on 1 core.

Figure 27 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 [Satish et al. 2010].

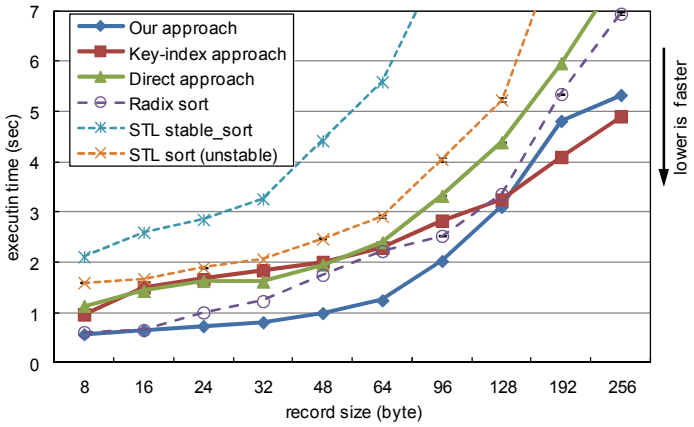


Figure 27. Execution times for sorting 16M records with various record sizes on 1 core.

To confirm that the rearranging really matters for the overall performance of the key-index approach, Figure 28 shows a breakdown of the execution times for the key-index approach in 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.

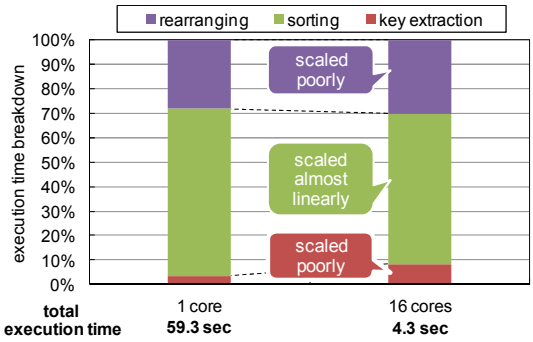


Figure 28. Execution time breakdowns for key-index approach (implemented with SIMD) into key extraction, sorting, and rearranging when sorting 512M 16-byte records on 1 and 16 cores.

To show the effect of the input data distribution, Figure 29 compares the performance of algorithms with different numbers of random key bits. For example, *key* is initialized by rand32() & 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.

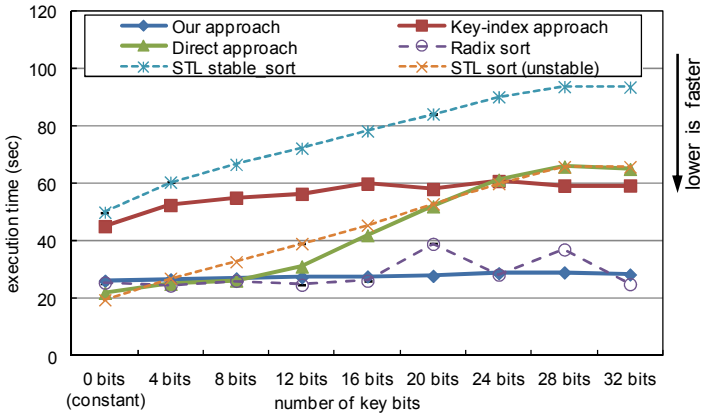


Figure 29. Execution times for sorting 512M 16-byte records with different random key bits. For 8-bit case, keys were initialized with rand32() & 0xFF.

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 30 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 [Govindaraju et al. 2006, Kim et al. 2012]), 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.

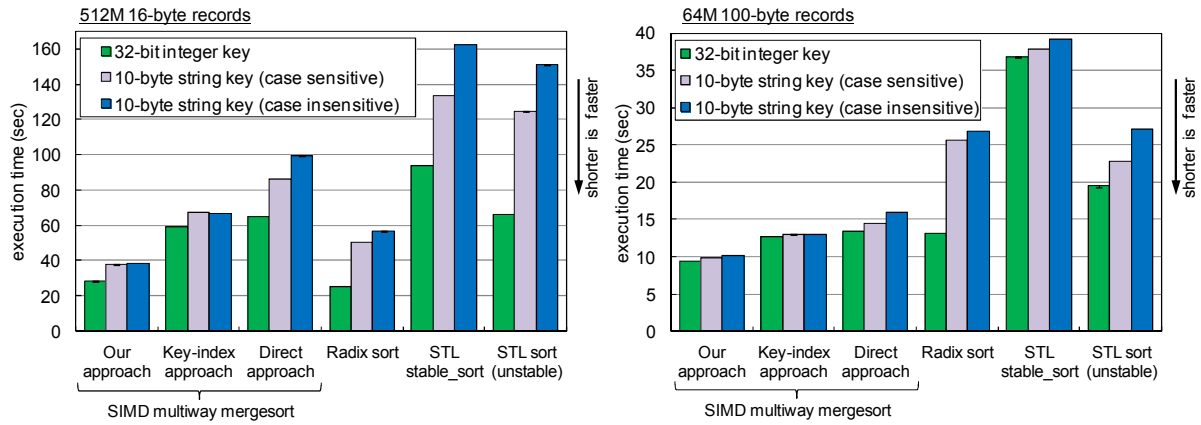


Figure 30. Execution time on 1 core for sorting 512M 16-byte records and 64M 100-byte records with 32-bit integer keys or 10-byte ASCII string keys.

Figure 31 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 (variable-sized) strings. For example, Burstsort [Sinha and Zobel 2004] 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.

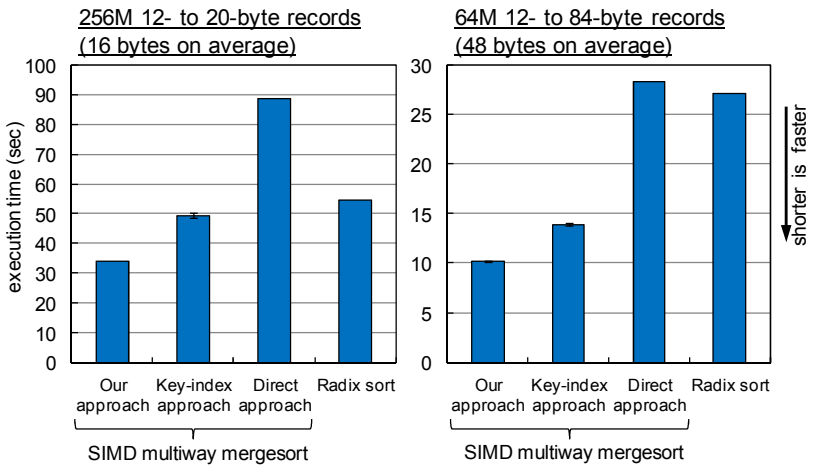


Figure 31. Execution times for case-insensitive sorting of variable-length string records with 256M records of 16-byte length on average and 64M records of 48-byte length on average.

4.4.2 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 32 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 2,048 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 21, 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.

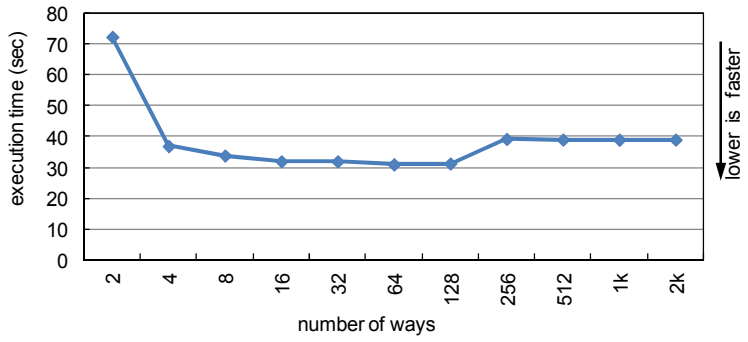


Figure 32. Execution times for sorting 512M 16-byte records with our algorithm using various k on 1 core.

Figure 33 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.

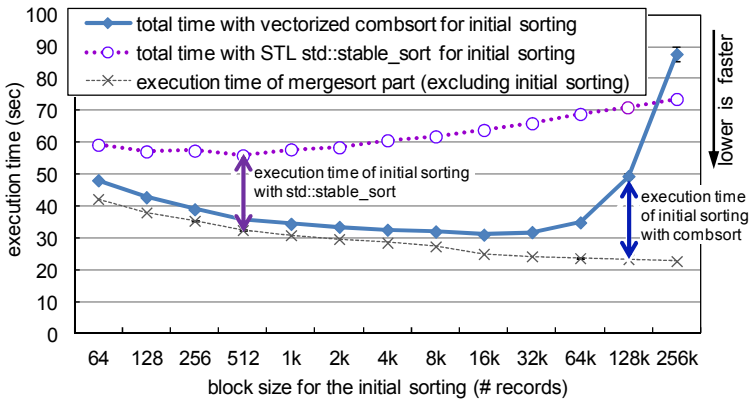


Figure 33. Execution times for sorting 512M 16-byte records with increasingly large blocks (B records) for the initial sorting using two different algorithms.

Figure 34 shows how the use of 4-wide SIMD in the multiway mergesort improved performance and partial conflict ratio. The partial key conflict ratio is ratio against number of total records merged using multiway merge operation (N * number of multiway merge stages). 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.

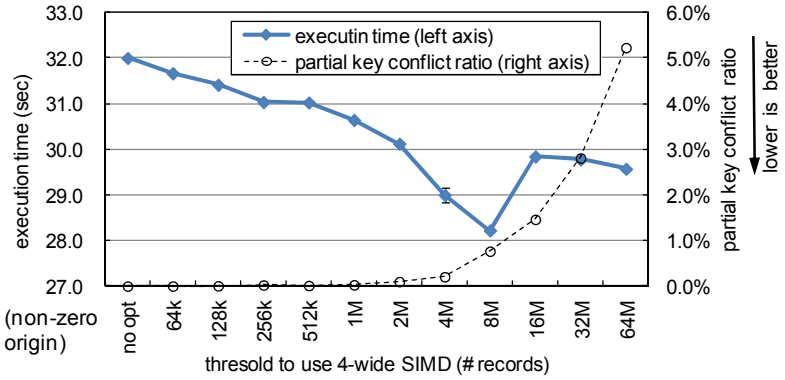


Figure 34. Execution times and partial conflict ratio for sorting 512M 16-byte records with various thresholds for using 4-wide SIMD comparisons.

4.5 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.

Chapter 5

Set Intersection

5.1 Introduction

Set intersection, which selects common elements from two input sets, is one of the most important operations in many applications, including multi-word queries in Web search engines and join operations in database management systems. For example, in Web search engines the set intersection is heavily used for multi-word queries to find documents containing two or more keywords by intersecting the sorted lists of matching document IDs from the individual query words [Barroso et al. 2003]. In such systems, the performance of the sorted set intersection often dominates the overall performance. Due to its importance, sorted set intersection has a rich history of research and many algorithms have been proposed to accelerate the operation. Many of these existing algorithms focus on reducing the number of comparisons to improve the performance. For example, special data structures, such as hash-based structures [Baeza-Yates and Salinger 2004, Ding and König 2011] or hierarchical data structures [Baeza-Yates 2004], can be used to boost performance by skipping redundant comparisons, especially when the sizes of the two input sets are significantly different. Unfortunately, such algorithms typically require preprocessing of the

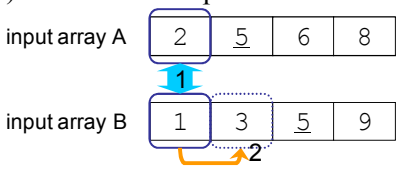
input data to represent it in a special form or to create additional data structures. Also, when the sizes of the two inputs are similar, it is much harder to achieve performance improvements because there are far fewer obviously redundant comparisons. As a result, simple merge-based implementations of set intersections are still widely used in the real world, such as the std::set_intersection implementation of STL in C++. Our goal is to improve the performance of set intersection even when the two input sets are of comparable size without preprocessing.

In this chapter[†], we describe our new algorithm to improve the performance of the set intersection. Unlike most of the existing advanced techniques, we focus on improving the execution efficiency of the set intersection on the microarchitectures of today's processors by reducing the branch mispredictions instead of reducing the number of comparisons. Today's processors are equipped with a branch prediction unit and speculatively execute one direction (*taken* or *not-taken*) of a conditional branch to maximize the utilization of processor resources. If the predicted direction of the branch does not match the actual outcome of the branch (*branch misprediction*), the hardware typically wastes more than ten CPU cycles because it needs to flush speculatively executed instructions and restart the execution from the fetch of the next instruction for the actual branch direction.

In our algorithm, we extend the naive merge-based set intersection algorithm by reading multiple elements, instead of just one element, from each of the two input arrays and compare all of the pairs of elements from the two arrays to find any elements with the same value. Figure 35 compares a naive algorithm and our algorithm using 2 as the number of elements read from each array at one time (*block size*). Surprisingly, this simple improvement gave significant performance gains on both Xeon and POWER7+.

[†] Most of the contents in this chapter was published in *PVLDB* Vol. 8, No. 3, pp 293–304 [Inoue and Taura 2014].

(a) Schematic and pseudocode for naive merge-based algorithm



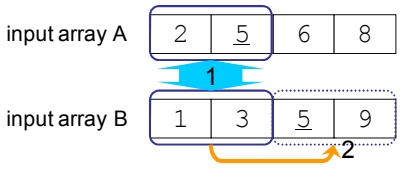
in each iteration

1. to check the equality of two elements
2. to advance a pointer by 1

```
1   while (Apos < Aend && Bpos < Bend) {
2      if (A[Apos] == B[Bpos]) {
3         C[Cpos++] = A[Apos];
4         Apos++; Bpos++;
5      }
6      else if (A[Apos] > B[Bpos]) {
7         Bpos++;
8      }
9      else {
10        Apos++;
11     }
12  }
```

hard-to-predict conditional branch

(b) Schematic and pseudocode for our algorithm (*block size* = 2)



in each iteration:

1. to find any matching pairs in blocks of *S* elements, here *S*=2 (*all-pairs comparison*)
2. to advance a pointer by *S*

```
1   while (1) {
2      Adat0 = A[Apos]; Adat1 = A[Apos + 1];
3      Bdat0 = B[Bpos]; Bdat1 = B[Bpos + 1];
4      if (Adat0 == Bdat0) {
5         C[Cpos++] = Adat0;
6      }
7      else if (Adat0 == Bdat1) {
8         C[Cpos++] = Adat0;
9         goto advanceB;
10     }
11     else if (Adat1 == Bdat0) {
12        C[Cpos++] = Adat1;
13        goto advanceA;
14     }
15     if (Adat1 == Bdat1) {
16        C[Cpos++] = Adat1;
17        goto advanceAB;
18     }
19     else if (Adat1 > Bdat1) goto advanceB;
20     else goto advanceA;
21  advanceA:
22     Apos+=2;
23     if (Apos >= Aend) { break; } else { continue; }
24  advanceB:
25     Bpos+=2;
26     if (Bpos >= Bend) { break; } else { continue; }
27  advanceAB:
28     Apos+=2;  Bpos+=2;
29     if (Apos >= Aend || Bpos >= Bend) { break; }
30  }
31  //  fall back to naive algorithm for remaining elements
```

all-pairs comparison (using S^2 comparisons)

hard-to-predict conditional branch

Figure 35. Overview of set intersection without and with our technique.

Our key insight for the performance improvement is that using larger block size yields a smaller number of hard-to-predict conditional branches in the set intersection. The comparison to select an input array for the next block, shown in bold in Figure 35, will be taken in arbitrary order, and therefore it is very hard for branch prediction hardware to predict the branches. However, most of the conditional branches in the all-pairs comparisons are not-taken and they do not cause frequent branch mispredictions in the most frequent cases of many applications, since the number of output elements is generally much smaller than the number of input elements for many applications. For example, Ding and König [2011] reported that 94% of the queries in a shopping portal site select less than 10% of the input elements as the output. Therefore, using the larger block size can reduce the number of hard-to-predict conditional branches in exchange for a larger number of total comparisons and easy-to-predict conditional branches. When the block size is S, one all-pairs comparison requires up to S^2 conditional branches, but the number of hard-to-predict conditional branches is reduced to only one per S input elements.

The larger number of comparisons from these all-pairs comparisons is an obvious drawback of our algorithm. However, we can effectively eliminate many of these additional comparisons by using SIMD instructions. By combining our algorithm with SIMD instructions, we can reduce the number of branch mispredictions without increasing the total number of executed instructions by executing multiple comparisons in parallel.

Our algorithm roughly doubled the performance for set intersection for 32-bit and 64-bit integer datasets even without using SIMD instructions compared to the std::set_intersection implementation delivered with gcc. The use of SIMD instructions further doubled the performance on both processors. We use SIMD instructions to filter out redundant scalar comparisons by using only a part of each element instead of finding matching pairs directly with SIMD comparisons. This approach increases the data parallelism within each SIMD instruction and leads to higher performance. It also allows us to use SIMD instructions if the data type is not natively supported by the SIMD instruction set, e.g. 64-bit integers on POWER7+.

Our new algorithm seeks to accelerate the set intersection operation when (1) the number of output elements is much smaller than the number of input elements and (2) the sizes of the two input sets are not significantly different. For datasets that do not satisfy these assumptions, other algorithms such as binary-search-based algorithms can outperform our algorithm. This is why we devised a practical technique to adaptively select the best

algorithm based on the ratio of the number of output elements over the number of input elements (selectivity) and the size ratio of two input sets. Although our algorithm is constrained by the characteristics of the input datasets, our results using the datasets generated from the Wikipedia database showed that our algorithm provides good overall performance for multi-word queries.

5.2 Related Work

Sorted set intersection has a rich history of research and many algorithms have been proposed to improve its performance. Previous studies, such as [Demaine et al. 2000], showed that simple algorithms based on a linear merge (e.g. Figure 35(a)) performed best when the sizes of the two input sets were similar. Another advantage of a merge-based algorithm is that it works well with input datasets that are compressed with various algorithms, such as delta encoding, due to its simplicity. Our algorithm extends the merge-based algorithm and improves its efficiency on today's processors. This means our algorithm inherits the advantages of the merge-based algorithms and performs well on input sets of similar size.

When one input set is much larger than another, the merge-based algorithm is less efficient and many techniques have been proposed to improve its performance. For example, algorithms based on binary search [Bentley and Yo 1976, Demaine et al. 2000, Bille et al. 2007, Sanders and Transier 2007] reduce the numbers of comparisons and memory accesses by picking values from the smaller set, and efficiently searching for the matching value in the larger set. Similarly, hash-based techniques [Baeza-Yates and Salinger 2005, Ding and König 2011] or techniques using hierarchical data representations [Baeza-Yates 2005] improve the performance by reducing the number of comparisons. However, most of these techniques are effective only when the sizes of the two input sets are significantly different (as by an order of magnitude or more). Our algorithm focuses on improving the performance of the input sets when the sets are roughly the same size. We can easily take advantages of both our algorithm and these existing techniques by selecting a suitable algorithm based on the size of the input sets before executing the operations as we describe in Section 5.4.

Recently, Ding and König [2011] proposed a hash-based technique that is effective even for input sets of similar size. They partitioned the input set into small partitions and encoded the values in each partition into a machine-word size bitmap in the preprocessing phase to efficiently calculate the intersection using bit-wise AND instructions at run time.

Though both their algorithm and ours are effective for arrays of similar sizes, an advantage of our algorithm is that we do not need preprocessing for additional data structures before an intersection operation.

Schlegel *et al.* [2011] exploited a special instruction called STTNI (STring and Text processing New Instruction) included in the SSE 4.2 instruction set of recent Intel processors for sorted set intersections. They compared multiple values read from each input array by using the special instruction to execute an all-pairs comparison in one step. They showed up to 5.3x and 4.8x accelerations for 8-bit and 16-bit data, respectively, on a Core i7 processor. Because the STTNI instruction does not support data types larger than 16 bits, for 32-bit integer data, they intersect the upper 16 bits and lower 16 bits separately. This technique achieved good speedups only when the value domain was limited, so that a sufficient number of elements shared the same value in their upper 16 bits, which limits the real-world use of this approach. Our algorithm does not have this limitation on the value domain even when we use the SIMD instructions because we use SIMD instructions as a filter to reduce scalar comparisons instead of finding matching pairs directly with SIMD comparisons. We evaluated our algorithm using 32-bit and 64-bit integer data. Also, our non-SIMD algorithm does not use any unique instructions, which makes it more portable among processors.

Lemire *et al.* [2014] accelerated decompression and set intersection used in index search systems by SIMD instructions. For the set intersection, they used three algorithms designed for SIMD instructions; so called V1 algorithms, V3 algorithm and galloping [Bentley and Yao 1976] with SIMD. They selected the algorithm based on the ratio of sizes of the two input arrays. When the sizes of the input sets are of similar size, the V1 algorithm performed best among the three algorithms. When the lengths of the two input arrays are significantly different, SIMD galloping was the best. Our algorithm focuses on the case when the sizes of the two inputs are similar. As showed in Section 4, our SIMD algorithm achieved about 2x better performance than their V1 SIMD algorithm. Also our SIMD algorithm can be used even if the data type is not natively supported by the SIMD instruction set, such as 64-bit integers on POWER7+, while their technique requires special handling for such case. When the gap between the two input sizes becomes large (by more than an order of magnitude), we switch to their SIMD galloping algorithm.

Schlegel *et al.* [2011] also showed that replacing the branch instruction in the merge-based set intersection with the predicated instructions of the Intel processor improved the performance over the branch-based implementation, though this was not the main focus

of their work. This optimization, which replaces control flow by data flow, is called *if-conversion* [Allen et al. 1983], and is a conventional technique to reduce branch misprediction overhead. Figure 36 shows an example of set intersection implementation without using a conditional branch to advance a pointer. As shown in the performance comparisons later, our technique yielded much better performance than a branchless implementation by the if-conversion approach on both of the tested processors. Although both techniques reduce the branch misprediction overhead, our algorithm achieved better performance even without SIMD instructions by having a shorter path length as shown in Section 5.4. Another benefit of our algorithm is that it can effectively exploit data parallelism using the SIMD instructions widely available in today's processors. Table 2 highlights these existing and our new algorithms.

```
1   while (Apos < Aend && Bpos < Bend) {
2      Adata = A[Apos];
3      Bdata = B[Bpos];
4      if (Adata == Bdata) {  // easy-to-predict branch
5         C[Cpos++] = Adata;
6         Apos++; Bpos++;
7      }
8      else { // advance pointers without conditional branches
9         Apos += (Adata < Bdata);
10        Bpos += (Bdata < Adata);
11     }
12  }
```

Figure 36. Pseudocode for set intersection without hard-to-predict conditional branches (the *branchless algorithm*).

Table 2. Summary of set intersection algorithms

the sizes of the two input sets (N_a, N_b)	without SIMD instructions	With SIMD instructions
Similar size	**Merge-based algorithms (e.g. STL)**: is simple and widely used, but suffers from branch mispredictions. **If-conversion**: eliminates the branch mispredictions in trade for longer path length. is hard to SIMDize. **Ding and König [2011]**: requires preprocessing. **Our block-based algorithm:** is an extention to the merge-based technique, reduces the branch mispredictions	**Schlegel [2011]**: is mainly targetting 8-bit or 16-bit integers supported by STTNI instructions **Lemire [2014] (V1 algorithm)**: can operate on 32-bit integers supported by using SIMD comparisons **Our block-based algorithm with SIMD:** yields larger gain by increased data parallelism. can support data types, even not natively supported by the SIMD instruction set.
Significantly different (more than 10x)	**Binary-search-based techniques (e.g. galloping [Bentley and Yao 1976])**: do not require preprocessing **Techniques using additional data structures (skip list, hash etc)**: require preprocessing	**Lemire [9] (SIMD galloping)**: is an extention to the gallop algorithm with SIMD instructions

We can improve the performance of the set intersection by reducing the branch mispredictions. Branch misprediction overhead is recognized as a major performance constraint in some workloads. For example, Zhou and Ross [2002] reported that the performance of many database operations, such as B+ tree search and nested loop joins for unsorted arrays, can be improved by reducing the branch misprediction overhead using SIMD instructions. Also, as already discussed in the previous chapters, some sorting algorithms [Sanders and Winkel 2004, Inoue et al. 2007, Chhungani 2008] improved the performance in sorting random input by reducing the branch misprediction overhead using the predicated instructions or SIMD select instructions.

Some set intersection algorithms exploit other characteristics of the modern hardware architectures beyond their branch performance, such as the large amount of cache memory and multiple cores [Tsirogiannis et al. 2009, Tatikonda et al. 2009, Ao et al. 2011]. For example, Tsirogiannis *et al.* [2009] improve the performance of a search-based intersection algorithm by using a small index called a *micro-index*, which can fit into the processor's cache memory.

Because our sorted set intersection algorithm is based on the merge operation, we can use the mergesort technique with SIMD instructions for the set intersection. A typical merge operation needs to decide on the order of all input elements, while the set intersection only handles the order of elements included in both sets. Hence our SIMD algorithm specialized for the set intersection can achieve higher performance in most cases.

5.3 Our Algorithm for Set Intersection

In this section, we first describe our block-based set intersection algorithm that reduces branch mispredictions with two sorted and unique-element arrays without using SIMD instructions. Then we show how we exploit SIMD instructions to execute multiple comparisons in parallel to further improve the performance. Although our algorithm is for intersecting two input sets, we can intersect multiple sets by using this algorithm as a building block, repeatedly executing our algorithm for the two smallest among all of the input data sets, which is a frequently used technique.

5.3.1 Key Observation

As shown in Figure 35(a), a naive merge-based algorithm for set intersection reads one value from each of the input arrays (*A* and *B* in the Figure) and compares the two values. If the two values are equal, the value is copied into the output array (*C*) and the pointers for both arrays are advanced to load the next values. If the two values are not equal, the pointer for the array whose element is smaller is advanced. This approach requires up to (N_a + N_b - 1) iterations to complete the operation. Here N_a is the number of elements in the array *A* and N_b is the number of elements in the array *B*. Each iteration includes one *if_equal* conditional branch (Line 2 in Figure 35(a)) and one *if_greater* conditional branch (Line 6 in Figure 35(a)).

Here, the *if_greater* conditional branches in the set intersection are hard to predict and incur significant branch misprediction overhead, while the *if_equal* conditional branches rarely cause mispredictions. Hence our algorithm focuses on reducing the number of costly *if_greater* conditional branches at the cost of using more *if_equal* conditional branches to optimize for the common case. The *if_greater* conditional branches cause frequent mispredictions because they will be taken in arbitrary order with roughly 50% probability when the sizes of the input sets are similar. This makes it very hard for branch prediction hardware to predict the branch directions correctly. Most of the conditional branches in the all-pairs comparisons are not-taken and they do not cause frequent branch mispredictions in typical cases for many applications, since it is known that the number of output elements is much smaller than the number of input elements in practice [Ding and König 2011].

To achieve its speedup, our algorithm assumes that:

- the number of output elements is much smaller than the number of input elements, and
- the sizes of input sets are not significantly different (as by an order of magnitude or more).

Our algorithm performs well for datasets that satisfy these assumptions. Otherwise, we adaptively switch to another algorithm to combine the advantages of our algorithm with the strengths of such algorithms as the binary-search-based algorithms. The first assumption is important to avoid frequent mispredictions in the *if_equal* conditional branches, which we assume are not costly. In Section 5.4, we describe an adaptive fallback technique to validate the first assumption at runtime. The second assumption ensures that the *if_greater*

conditional branches cause lots of mispredictions. We select the best algorithm and a parameter (block size) based on the sizes on the two input sets before executing the operations. If the sizes of the two input arrays are significantly different, we use a binary-search-based algorithm. Although many of the existing algorithms focus on the performance of the set intersection when the sizes of the two input sets are significantly different, our results show that the performances for two input sets with similar sizes are also important for the overall performance of multi-word queries when we used an optimized algorithm for the input sets with significantly different sizes.

5.3.2 Our Basic Approach without SIMD instructions

Our technique extends the naive merge-based algorithm shown in Figure 35(a) by reading multiple values from each input array and compares all of their pairs using *if_equal* conditional branches as shown in Figure 35(b). We call the number of elements compared at one time the *block size* (*S*). We repeat the following steps until we process all of the elements in the two input arrays (*A* and *B*):

(1) read *S* elements from each of two input arrays,

(2) compare all possible S^2 pairs of elements (e.g. four pairs in Figure 35(b), where $S = 2$) by using *if_equal* conditional branches to find any matching pairs,

(3) if there is one or more matching pairs, copy the value or values of the found pairs into the output array (*C*),

(4) compare the last elements of the two arrays used in Step 2,

(5) advance the pointer by *S* elements for the array whose last element is smaller in Step 4.

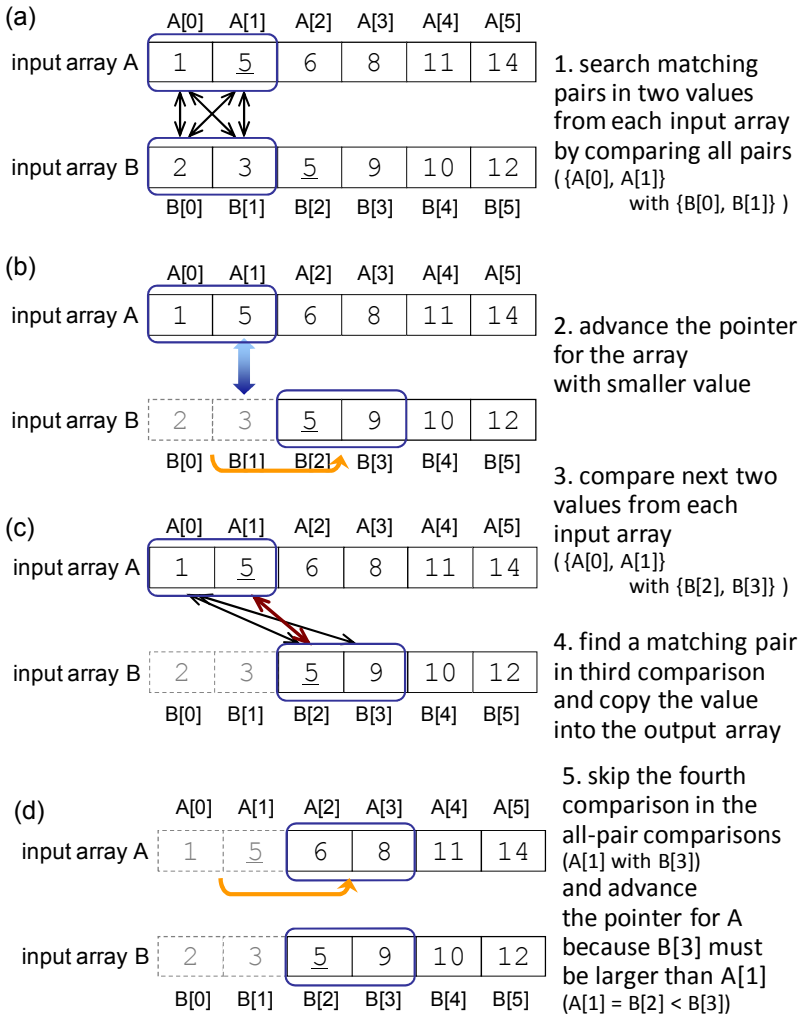


Figure 37. An example of set intersection of two sorted arrays with our technique using 2 as the block size. Parts (a) and (b) show the basic operation when no pairs match. Parts (c) and (d) show how a matching pair is output.

Figure 37 shows a step-by-step example of our algorithm with the block size of 2. Because the block size is 2, the all-pairs comparison shown in Figure 37 uses up to 4 *if_equal* conditional branches (Lines 4-18 in Figure 35(b)). Then, in Figure 37(b), we compare the second value from each array (A[1] = 5 and B[1] = 3) and advance the pointer for the array B because B[1] is smaller than A[1]. This step uses only one *if_greater* conditional branch (Line 19 in Figure 35(b)). Typically, this *if_greater* conditional branch is hard to predict and hence it causes frequent branch mispredictions. Our algorithm aims to reduce the number of the hard-to-predict conditional branches executed. If there is no

matching pair found in the all-pairs comparison (the most frequent case), then we repeat the steps shown in Figure 37(a) and (b).

If a matching pair is found in an all-pairs comparison (Figure 37(c)), then we copy the value (5 in the figure) into the output array and skip the following comparisons, which are no longer possible matches. In the Figure, *B*[3] (= 9) must be larger than B[2] and A[1] (= 5) because each array was sorted, and hence we can advance the pointer for the array A without comparing A[1] and B[3] (Figure 37(d)).

If the number of total elements in an input array is not a multiple of the block size *S*, we can just fall back to the naive approach for the remaining elements. If the number of elements in an array is large enough, this does not measurably affect the overall performance of the set intersection operation.

A naive approach for advancing a pointer by more than one element is by comparing {A[Apos], with B[Bpos + *gap*]} and {A[Apos + *gap*], with B[Bpos]} using *if_greater* conditional branches before comparing {A[Apos], with B[Bpos]}, i.e. Line 6 in Figure 35(a). However, in our tests we found this naive technique degraded the performance regardless of the *gap* size unless the sizes of the input sets were significantly different. This was because the *if_greater* conditional branches for the comparisons with the gaps caused many branch mispredictions and hence did not reduce the total misprediction overhead.

Identifying the best block size: With our scalar (non-SIMD) algorithm, the total number of *if_equal* conditional branches increases for larger block sizes. The number of *if_equal* conditional branches, which are typically easy to predict, involved in one all-pairs comparison is up to S^2. To complete the set intersection for the entire array, we need to execute the all-pairs comparisons up to $((N_a + N_b)/S - 1)$ times, so the total number of *if_equal* conditional branches is up to $S^2 \cdot ((N_a + N_b)/S - 1) = S \cdot (N_a + N_b) - S^2$. This is almost S times larger than the naive approach.

At the same time, the number of *if_greater* conditional branches, which are more costly than *if_equal* conditional branches due to their frequent branch mispredictions, decreases as the block size increases. We advance the pointer by S elements at a time instead of by just one element as in the naive approach. Hence, the number of *if_greater* conditional branches is $((N_a + N_b)/S - 1)$. This is almost S times smaller compared to the naive approach.

Table 3. Summary of the number of conditional branches without SIMD instructions using the same block size S for both input arrays

Approach	Number of hard-to-predict *if_greater* conditional branches (may cause frequent mispredictions)	Number of *if_equal* conditional branches (mispredictions infrequent)	Total number of conditional branches	Performance characteristics
Naive algorithm Figure 35(a)	up to N_a+N_b - 1	up to N_a+N_b - 1	up to 2 (N_a+N_b - 1)	- shorter path length - larger misprediction overhead
Our algorithm Figure 35(b)	up to $(N_a+N_b)/S$ - 1 **about S times less**	up to $S\cdot(N_a+N_b) - S^2$ **about S times more**	up to $(S^2+1)\cdot((N_a+N_b)/S - 1)$ **about $(S^2+1)/S$ times more**	- longer path length - smaller misprediction overhead

N_a, N_b: the number of elements in the two input arrays A and B. S: the block size.

We summarize the total numbers of comparisons in Table 3. Because our algorithm is a generalization of the naive algorithm, which is equivalent to our algorithm with a block size of 1, the number of comparisons in Table 3 is the same for both algorithms when $S = 1$.

The key parameter to find the best block size is the penalty of a branch misprediction compared to the number of CPU cycles to execute a conditional branch without a misprediction. To find the best block size, we calculate the cost of total branch instructions including the misprediction overhead. We assume that only the hard-to-predict conditional branches cause mispredictions for typical input. The best block size with this assumption is the S that minimizes this cost function $f(S)$:

$$f(S) = branch\_cycles \times (S^2+1) / S + mpred\_penalty \times mpred\_ratio / S, \quad (1)$$

Here, *branch_cycles* is the number of cycles to execute a conditional branch and *mpred_penalty* is the penalty of a branch misprediction (in terms of cycles). The *mpred_ratio* is the branch misprediction ratio for the hard-to-predict conditional branches. We assume the *mpred_ratio* is 50% when the sizes of the two input set are comparable because the hard-to-predict conditional branches are taken in arbitrary order and hence no branch predictor can predict them correctly. When the misprediction penalty is more than twice the cost of a successfully-predicted conditional branch, our technique improves the performance over the naive algorithm by using a block size of 2. When the relative cost of the misprediction is between 10 and 22, as is true for many of today's processors, the best block size is 3. The branch misprediction penalties for POWER7+ and Xeon were both about 16 cycles as measured with a micro-benchmark and the cost of a branch instruction is

expected to be 1 cycle. We predict that the block size of 3 yields the best performance and the block size of 4 is a close second best. We empirically confirmed this estimate in Section 5.4.

Because the best block size also depends on the input data and not just the processor, we used an adaptive control technique with a runtime check to detect pathological performance cases for our algorithm.

Using different block sizes for each input array: Up to now, we have been assuming that we use the same block size S for both input arrays. However, using different block size for each input array may give additional performance gains, especially when the sizes of the two input arrays are very different. When we use different block sizes S_a and S_b for each input array, the number of *if_greater* conditional branches is up to $N_a/S_a + N_b/S_b$, and the number of *if_equal* conditional branches is $(N_a/S_a + N_b/S_b) \times (S_a \cdot S_b)$. Hence the cost function becomes

$$f(S_a, S_b) = branch\_cycles \times (S_a \cdot S_b + 1) \times (q/S_a + (1-q)/S_b) +$$

$$mpred\_penalty \times mpred\_ratio \times (q/S_a + (1-q)/S_b). \quad (2)$$

Here, q shows how the sizes of two input arrays different, $q = N_a/(N_a+N_b)$. When $S_a = S_b$, equation (2) becomes equivalent with equation (1) regardless of q. When the sizes of two input arrays are significantly different, the misprediction rate at the hard-to-predict conditional branch is much difficult to estimate and depends on the branch predictor implementation. When we assume a simple predictor, which just predicts the more frequent side of the two branch directions, *taken* or *not-taken*, the misprediction rate is $min(q, 1-q)$. With this assumption and the misprediction penalty of 16 cycles, for example, the best block sizes for two arrays with sizes of N_a and $2N_a$, i.e. $q = 1/(1+2) = 1/3$, are $S_a = 2$ and $S_b = 4$, while $S_a = S_b = 3$ is the best if the two input arrays have the same size, i.e. $q = 1/2$, as already discussed.

Our scalar algorithm selects the block sizes before starting the operation based on the ratio of the sizes of the two input arrays. We use $S_a = 2$ and $S_b = 4$ if the size of the larger array N_b is more than twice the size of the smaller array N_a. Otherwise, we use $S_a = S_b = 3$. We show how the block size affects the performance in Section 5.4.

5.3.3 Exploiting SIMD Instructions

Our algorithm reduces the branch misprediction overhead but with an increased number of easy-to-predict conditional branches, as discussed in Section 3.2. To further improve the performance, we employ SIMD instructions to reduce the number of instructions by executing the all-pairs comparisons within each block in parallel. Unlike the previous SIMD-based set intersection algorithms, we use SIMD instructions to filter out unnecessary scalar comparisons by comparing only a part of each element. This approach increases the data parallelism within one SIMD instruction by using only a part of the elements. It also allows us to use SIMD instructions if the data type is not natively supported by the SIMD instruction set. For example, we can use processors without 64-bit integer SIMD comparisons to intersect 64-bit integer arrays, e.g. 64-bit integers on POWER7+.

For our SIMD algorithm, we used a multiple of four as the block size S (or S_a and S_b when using different block sizes for two arrays) so we could fully exploit the SIMD instructions of the processors, which can compare up to 16 or more data pairs of 1-byte elements in parallel with one instruction. Our SIMD algorithm selects $S_a = S_b = 4$, if the size of the larger array N_b is less than twice the size of the smaller array N_a. Otherwise, we use $S_a = 4$ and $S_b = 8$. As discussed in Section 3.2, the block size of 3 is best for our scalar (non-SIMD) algorithm. However, we can reduce the number of comparisons by using SIMD instructions, which justifies using a larger block size than used in the scalar algorithm.

The vector sizes of the SIMD instruction sets of today's processors, such as SSE/AVX of Xeon [Intel 2015] or VSX of POWER7+ [IBM 2010], are limited to 128 bits or 256 bits. This means we can execute only two or four comparisons of 64-bit elements in parallel. This parallelism is insufficient for an all-pairs comparison of large blocks in one step. To execute the all-pairs comparisons for larger blocks efficiently by increasing the parallelism available in one SIMD instruction, we use a parallel SIMD comparison, which compares only a part of each element, to filter out all of the values with no outputs before executing the all-pairs comparison using scalar comparisons. Unlike the previous SIMD approaches [Schlegel et al. 2011, Lemire et al. 2014], we did not fully replace the scalar comparisons with parallel SIMD comparisons. Because the number of matching pairs are typically much smaller than the number of input elements in practice, our filtering technique is effective to avoid most of the scalar comparisons and hence achieves higher overall performance. Zhou and Ross [2002] also used a similar idea of comparing only a part of each element to increase the data parallelism with one SIMD instruction for the nested loop join for unsorted arrays.

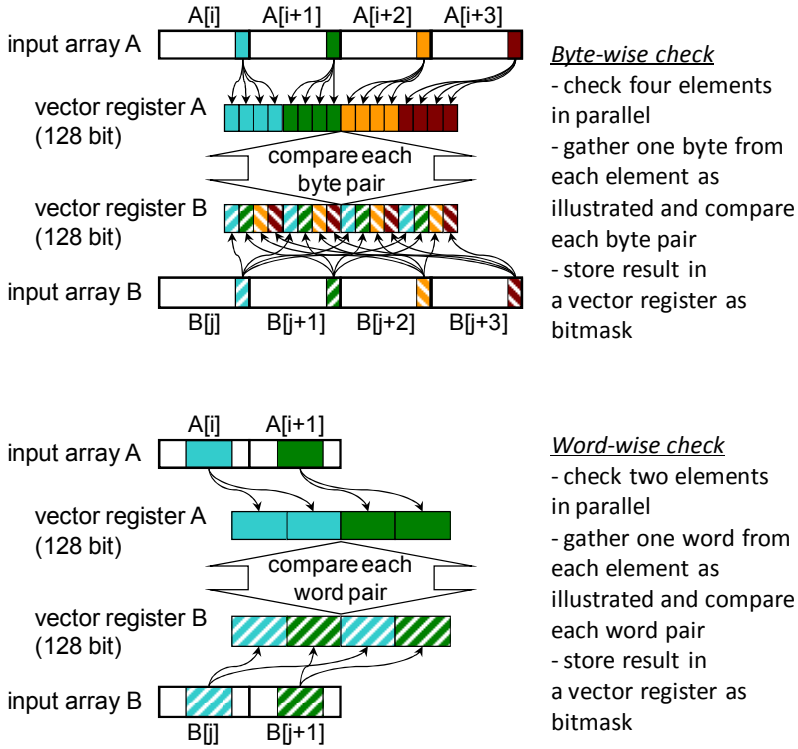


Figure 38. Overview of byte-wise check and word-wise check.

We use two different types of checks to find a matching pair in the all-pairs comparison hierarchically. Figure 38 shows an overview of our byte-wise check and word-wise check using SIMD parallel compare instructions. In this example, we assume 64-bit integers as the data type and an SIMD instruction set with 128-bit registers. These checks execute only a partial comparison of each element. It means that if the check does not find any matching byte or word pair, there cannot be any matching element pairs (no false negatives). However, if the check finds a matching byte or word pair, the matching pair may still be a false positive. To reduce the number of false-positive matches, we hierarchically do two different types of checks. If the data type of each element is a 64-bit integer and the block size S is 4, our hierarchical filtering uses these steps:

1) Do a byte-wise check for A[i .. i+3] and B[j .. j+3] using the least significant byte,

2) Do a byte-wise check for A[i .. i+3] and B[j .. j+3] using the second-least significant byte,

3) Do a bit-wise AND operation for the results of Steps 1 and 2,

4) If every bit is zero in Step 3, then skip further checks because there is no matching pair (most frequent case),

5) Do a word-wise check for A[i .. i+1] and B[j .. j+1] using the third to sixth bytes,

6) Do a scalar check for Step-5 matches, and

7) Repeat Steps 5 and 6 for {A[i .. i+1] and B[j+2 .. j+3]}, {A[i+2 .. i+3] and B[j .. j+1]}, and {A[i+2 .. i+3] and B[j+2 .. j+3]}.

Alternatively, we can replace Steps 5 to 7 with a count-leading-zero instruction to identify the location of the matching pairs found in Step 3. When we use a 32-bit integer data type, we use the first to fourth bytes in Step 5. Figure 39 shows the pseudocode for the set intersection algorithm with our hierarchical filtering with SIMD for the two block sizes we used.

Block size S_a = 4 and S_b = 4

```
1  while (1) {
2     // do byte-wise check: step (1)-(3) of the hierarchical check
3     compare_result = bytewise_check1(&A[Apos], &B[Bpos]) &
4                      bytewise_check2(&A[Apos], &B[Bpos]);
5     if (!is_all_bit_zero(compare_result)) { // step (4)
6        // found a potential metching value
7        // do word-level check and scalar check: step (5) - (7)
8        ...
9     }
10    else if (A[Apos+3] > B[Bpos+3]) goto advanceB;
11    else goto advanceA;
12 advanceA:
13    Apos+=4;
14    if (Apos >= Aend) { break; } else { continue; }
15 advanceB:
16    Bpos+=4;
17    if (Bpos >= Bend) { break; } else { continue; }
18 advanceAB:
19    Apos+=4;  Bpos+=4;
20    if (Apos >= Aend || Bpos >= Bend) { break; }
21 }
```

Block size S_a = 4 and S_b = 8

```
1  while (1) {
2     // do byte-wise check: step (1)-(3) of the hierarchical check
3     compare_result1 = bytewise_check1(&A[Apos], &B[Bpos  ]) &
4                       bytewise_check2(&A[Apos], &B[Bpos  ]);
5     compare_result2 = bytewise_check1(&A[Apos], &B[Bpos+4]) &
6                       bytewise_check2(&A[Apos], &B[Bpos+4]);
7     if (!is_all_bit_zero(compare_result1)) { // step (4)
8        // found a potential metching value
9        // do word-level check and scalar check: step (5) - (7)
10       ...
11    }
12    if (!is_all_bit_zero(compare_result2)) { // step (4)
13       // found a potential metching value
14       // do word-level check and scalar check: step (5) - (7)
15       ...
16    }
17    else if (A[Apos+3] > B[Bpos+7]) goto advanceB;
18    else goto advanceA;
19 advanceA:
20    Apos+=4;
21    if (Apos >= Aend) { break; } else { continue; }
22 advanceB:
23    Bpos+=8;
24    if (Bpos >= Bend) { break; } else { continue; }
25 advanceAB:
26    Apos+=4;  Bpos+=8;
27    if (Apos >= Aend || Bpos >= Bend) { break; }
28 }
```

Figure 39. Pseudocode of our SIMD algorithm for two block sizes.

On Xeon, we can use the STTNI (`pcmpestrm`) instruction, which is unique to the Xeon processor, to execute the all-pair comparison efficiently. This instruction can execute the all-pair comparisons of eight 2-byte characters in one vector register against eight characters in another vector register with only one instruction. Thus we can implement Steps 1 to 3 with a block size of 8 by using the STTNI instruction very efficiently. Unlike Schlegel's algorithm [2011], our algorithm uses the STTNI instruction to filter out redundant comparisons and thus we do not need to limit the data types to the 8- or 16-bit integer supported by this instruction. When we use the STTNI in our SIMD algorithm, we use the *popcnt* instruction to identify the position of matching pair efficiently because the processor does not support the count-leading-zero instruction. We can get the position of the least significant non-zero bit in the result of STTNI, x, by $popcnt((\sim x)\ \&\ (x\text{-}1))$.

When the SIMD instruction set supports a wider vector, such as a 256-bit vector in AVX, one way to exploit the wider vector is doing multiple byte-wise checks at in once step. For example, we could do Steps 1 and 2 with just one parallel comparison using 256-bit vector registers, with 16 pairs of 2-byte elements.

By using our hierarchical filtering with SIMD instructions, we can avoid increasing the number of instructions and still gain the benefits of the reduced branch mispredictions.

5.4 Experimental Results

We implemented and evaluated our algorithm on Intel Xeon and IBM POWER7+ processors with and without using SIMD instructions. On Xeon, we also evaluated our SIMD algorithm implemented using the Xeon-only STTNI instruction. We implemented the program in C++ using SSE intrinsics on Xeon and Altivec intrinsics on POWER7+, but the algorithm is the same for both platforms. The POWER7+ system used for our evaluation was equipped with a 4.1-GHz POWER7+ processor. Red hat Enterprise Linux 6.4 was running on the system. We compiled all of the programs using gcc-4.8.3 included in the IBM Advance Toolkit with the –O3 option. We also evaluated the performance of our algorithm on a system equipped with two 2.9-GHz Xeon E5-2690 (SandyBridge-EP), also with Red hat Enterprise Linux 6.4 as the OS, but the compiler on this system was gcc-4.8.2 (still with the –O3 option) on this Xeon system. We disabled the dynamic frequency scaling on both systems for more reproducible results.

In the evaluation, we used both artificial and more realistic datasets. With the artificial datasets generated using a random number generator, we assessed the characteristics of our algorithm for three key parameters, (1) the ratio of the number of output elements over the input elements (*selectivity*), (2) the difference in the sizes of the two input arrays and (3) the total sizes of the input arrays. We define the *selectivity* as the number of output elements over the number of elements in the smaller input array. To create input datasets with a specified selectivity, we first generate a long enough array of (unsorted) random numbers without duplicates. We then trim two input arrays from this long array with the specified number of elements included in both arrays. Each array is then sorted and the pair is used as an input for experiments. We executed the measurements 16 times using different seeds for the random number generator and averaged the results. For the realistic datasets, we used arrays generated from Wikipedia data. We generated a list of document IDs for the articles that include a specified word. Then we executed the set intersection operations for up to eight arrays to emulate the set intersection operation for the multi-word queries in a query serving system. We also averaged the results from 16 measurements for the real-world data.

We show the performance of our block-based algorithm with and without SIMD instructions and with various block sizes. The results shown as *naive* in the figures are the performance of the code shown in Figure 35(a). As already discussed, our algorithm is equivalent to the naive algorithm when the block size is 1. We also evaluated and compared the performances of the existing algorithms including the widely used std::set_intersection library method in the STL delivered with gcc, the branchless algorithm shown in Figure 36, a galloping algorithm [Bentley and Yao 1976] (as a popular binary-search-based algorithm), the two SIMD algorithms by Lemire *et al.* [2014] (which are the V1 SIMD algorithm and an SIMD galloping algorithm), and Schlegel's algorithm [2011] that uses the STTNI instruction of Xeon. We picked these algorithms for the comparisons because, like our algorithm, they need no preprocessing. Among these evaluated algorithms, the two galloping algorithms based on binary searches are tailored for paired arrays of very different sizes. The other algorithms, including ours, are merge-based algorithms, which are known to work better when the sizes of the two inputs are similar.

5.4.1 Performance Improvements from Our Algorithm

Figure 40 compares the performance of the set intersection algorithms for two datasets of 256k integers based on a random number generator. The selectivity was set to zero (the best case). We used 32-bit integer as the data type for both Xeon and POWER7+ and also tested 64-bit integers for POWER7+. The error bars show 95% confidence intervals.

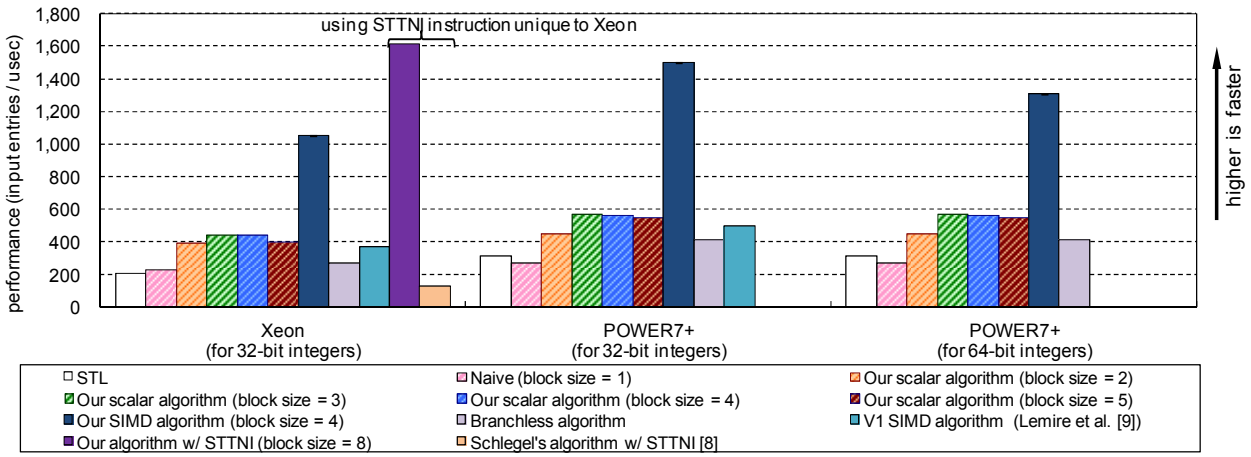


Figure 40. Performance for set intersection of 32-bit and 64-bit random integer arrays of 256k elements on Xeon and POWER7+.

The results show that our block-based algorithm improved the performance over the naive merge-based algorithms (STL and naive) on both platforms even without using the SIMD instructions. When comparing how the block size affected the performance of our scalar algorithm on these two platforms, the best performance was when the block size was set to 3. On both platforms, the block sizes of 3 and 4 gave almost comparable performance. Our prediction based on the simple model discussed in Section 5.3.2, which predicts the block size of 3 is the best and 4 is a close second best, seems reasonably accurate for both processors, although they have totally different instruction sets and implementations. The performance gains over the widely used STL were 2.1x on Xeon and 1.8x on POWER7+ with the block size of 3. Compared to our algorithm, the branchless algorithm did not yield large performance gains over STL, although it caused a smaller number of branch mispredictions than our algorithm (as shown later).

When we used the SIMD instructions, there were additional performance improvements of about 2.5x over our scalar algorithm on both platforms, where the total

improvement was 4.8x to 5.2x better than STL for 32-bit integers and 4.2x better for 64-bit integers. Although the V1 SIMD algorithm [Lemire et al. 2014] also achieved performance improvements over the STL using SIMD instructions, the performance of our SIMD algorithm was 2.9x and 3.0x better than V1 algorithm on Xeon and POWER7+ respectively. Also, the V1 algorithm cannot support 64-bit integer on POWER7+ because POWER7+ does not have SIMD comparisons for 64-bit integers, while our SIMD algorithm achieved good performance improvements even for 64-bit integers on POWER7+. This is because the V1 algorithm uses the SIMD comparison for the entire elements to find the matching pairs, but our algorithm uses the SIMD comparisons to filter out unnecessary scalar comparisons by comparing only a part of each element. Schlegel's algorithm [2011] did not achieve good performance for the artificial datasets generated by the random number generator. As the authors noted, Schlegel's algorithm is efficient only when the value domain is limited, so that a sufficient number of elements share matching values in their upper 16 bits, and this is not true for our artificial datasets.

5.4.2 Microarchitectural Statistics

For more insight into the improvements from our algorithm with and without SIMD instructions, Figure 41 displays some microarchitectural statistics of each algorithm for the artificial datasets in the 32-bit integer arrays as measured by the hardware performance monitors of the processors. We studied the branch misprediction rate (the number of branch mispredictions per input element), the CPI (cycles per instruction), and the path length (the number of instructions executed per input element).

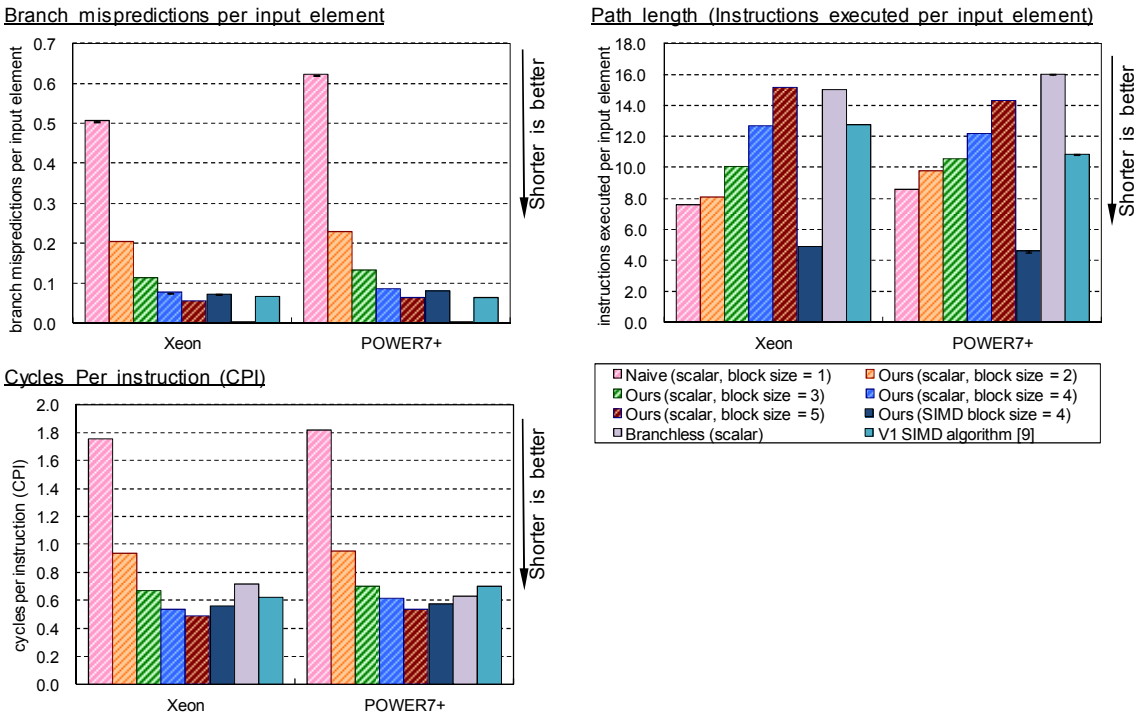


Figure 41. Branch misprediction rate, CPI, and path length.

We begin with the microarchitectural statistics of our algorithm when not using the SIMD instructions. When comparing the statistics for our scalar algorithm and the naive algorithm, which is equivalent to our algorithm with a block size of 1, the branch mispredictions are reduced as intended by using the larger block sizes. The reduction in the branch mispredictions directly affected the overall CPI, which was improved when we used the larger block sizes. The improvements in CPI were especially significant when we enlarged the block size from 1 to 2 and from 2 to 3. By using our scalar algorithm with the block size of 3, the branch mispredictions were reduced by more than 75% compared to the naive algorithm on both platforms, which was higher than the predicted reduction of 66%.

In contrast, the path lengths increased steadily with the increasing block sizes. As a result of the reduced CPI and the increased path length, our best performance without SIMD instructions was with the block sizes of 3 and 4. When the block size increased beyond 4, the benefits of reduced branch mispredictions were not significant enough to compensate for the increased path length. This supports our belief that the best block size might be larger on processors with larger branch misprediction overhead. Since most of today's high performance processors use pipelined execution units and typically have large branch

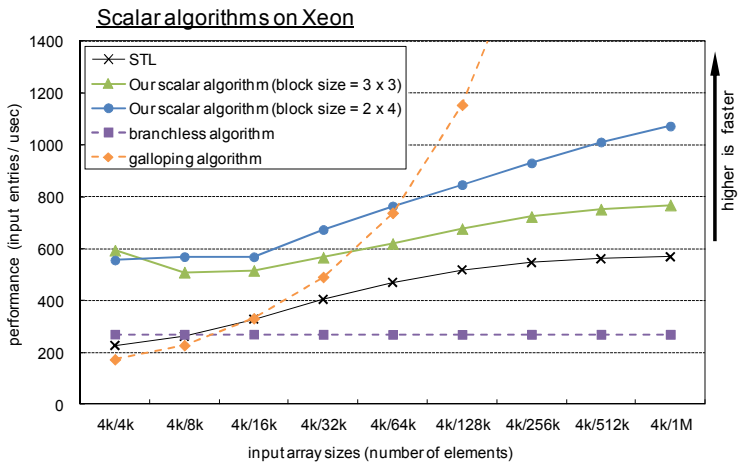
misprediction overhead, we expect that our algorithm would be generally effective for most of the modern processors, not just the two tested processors.

The branchless algorithm showed the smallest number of the branch mispredictions by totally replacing the hard-to-predict conditional branches with arithmetic operations. However, as shown in Figure 41, the path length of the branchless algorithm was larger than our algorithm. Due to this long path length, the branchless algorithm did not outperform our scalar algorithm in spite of its small number of branch mispredictions. Our algorithm achieved comparable or even better CPI than the branchless algorithm even with the larger numbers of branch mispredictions. This is due to our algorithm's higher instruction-level parallelism, since all of the comparisons in the all-pair comparisons can be done in parallel on the hyperscalar processors.

For our algorithm with the SIMD instructions, we observed significant improvements in the path lengths. Because the parallel comparisons of the SIMD instructions make the all-pairs comparisons of the costly scalar comparisons unnecessary in most cases, this greatly reduced the number of instructions executed, even with the large block size of 4. When we use STTNI on Xeon, our algorithm achieved further reductions in the path lengths. Due to the shorter path lengths, the SIMD instructions showed huge boosts in the overall performance.

5.4.3 Performance for Two Arrays of Various Sizes

In this section, we show how the differences in the sizes of the two input arrays and the total sizes of the input arrays affect the performance of each algorithm.



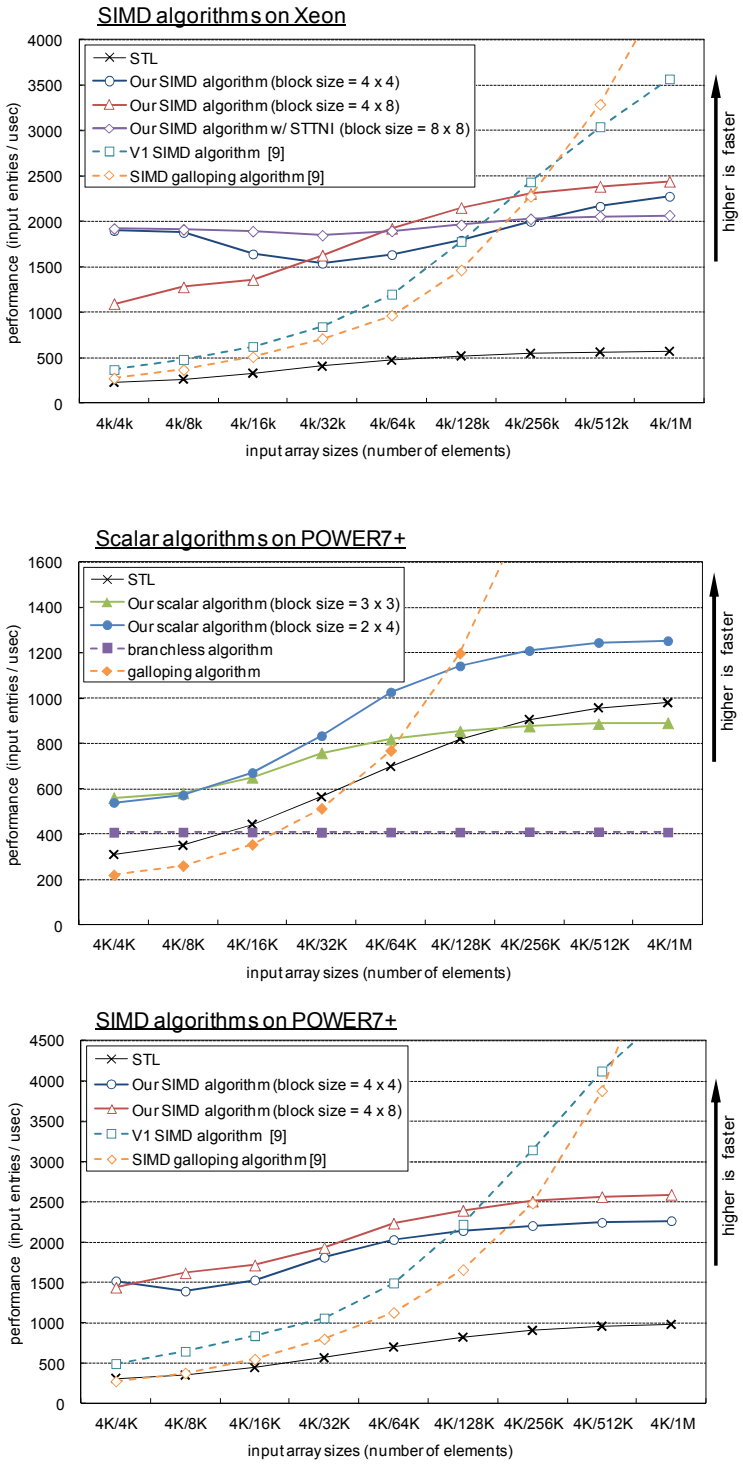


Figure 42. Performance of scalar and SIMD algorithms for intersecting 32-bit integer arrays on Xeon and POWER7+ when the sizes of the two input arrays are different.

Figure 42 compares the performances of scalar and SIMD algorithms for 32-bit integer arrays with changing ratios between the sizes of the two input arrays. When comparing our scalar algorithm with different block sizes, it worked best with the block size of $S_a = S_b = 3$ (we denote this block size as 3x3) when the sizes of two input arrays are the same (the leftmost point in the figure), while $S_a = 2$ and $S_b = 4$ (block size 2x4) worked better than the block size of 3x3 when the larger of the two input arrays was at least twice as large as the smaller array, as predicted in Section 3.2. For two input arrays with very different sizes, the numbers of branch mispredictions with merge-based algorithms, STL and ours, became much smaller than for the two arrays of the same size. When the sizes of the two input arrays are different, the hard-to-predict conditional branches to select which array's pointer to advance, e.g. the branches shown in bold in Figure 35, become relatively easier to predict because the frequency of one branch direction (taken or not-taken) becomes much higher than the other direction on average. This means there were fewer opportunities to improve the performance with our scalar algorithm. As a result, the absolute performances became higher for these algorithms and also the benefits of the reduced branch mispredictions with our algorithm became smaller with the larger gaps between the sizes of the two arrays. However, even for the largest differences between the sizes of the two arrays, our scalar algorithm with the block size of 2x4 achieved higher performance than STL. The branchless algorithm does not incur the branch misprediction overhead and hence its performance was not affected by the size ratio of the two arrays. As shown in many previous studies, when the ratio of the input sizes exceeds an order of magnitude, binary-search-based algorithms, such as the galloping algorithm in the figure, outperform the merge-based algorithms, including our algorithm.

For our SIMD algorithms, the block size of 4x8 yielded better performance than the block size of 4x4 (or the block size of 8x8 with STTNI on Xeon) when the sizes of the two arrays are significantly different, while the block size of 4x4 gave the best performance when the two arrays are of the same size (the leftmost point in the figure). When the ratio of the input array sizes is very large, the V1 SIMD algorithm and the SIMD galloping algorithm [Lemire et al. 2014] had better performances than our SIMD algorithm with the block size of 4x4 or 4x8. The V1 algorithm is a merge-based algorithm and is very similar to our algorithm with a block size of 1x8 implemented with SIMD instructions. Although this block size gave better performance for two arrays with very different sizes than 4x4 or 4x8, the binary-search-based galloping algorithm implemented with SIMD outperformed any merge-based algorithms we tested with such inputs.

Based on these observations, we combined our block-based algorithm with the galloping algorithm, so as to improve the performance for datasets with very different sizes. We select the best algorithm based on the ratio of the sizes of the two input arrays. When using SIMD, we use the SIMD galloping algorithm when the ratio of input sizes is larger than 32. Otherwise we use our SIMD algorithm. We use a block size setting of 4x8 when the ratio of input sizes is larger than 2.0, and a block size setting of 4x4 when the difference is smaller than this threshold. For the scalar algorithm, we used the (non-SIMD) galloping algorithm if the ratio is larger than 32. Otherwise, we use our block-based algorithm. The block size setting is 2x4 if the ratio is larger than 2.0 or otherwise the block size setting is 3x3.

Figure 43 shows how the total size of the two input arrays affects the performance. We used 32-bit integer arrays and the selectivity was zero. On both platforms, we observed small performance degradations when the total size exceeded the last-level (L3) cache of the processor because of the stall cycles to wait for data to be loaded from the main memory. The effects of the stall cycles due to the cache misses were not significant, because the memory accesses in the set intersection are almost sequential and this means the hardware prefetcher of the processors worked well to hide that latency by automatically reading the next data into the cache. The performance advantages of our scalar and SIMD algorithms over the other algorithms, the STL and V1 SIMD algorithms, were unchanged, even with the largest datasets we tested. On Xeon, the performance of four out of five tested algorithms was significantly improved when the input size was very small (left side of the figure). This was caused by very low branch misprediction overhead rather than the reduced cache miss stall cycles. The Xeon seems to employ a branch prediction mechanism that is very effective only when the input size is very small. POWER7+ did not exhibit this behavior.

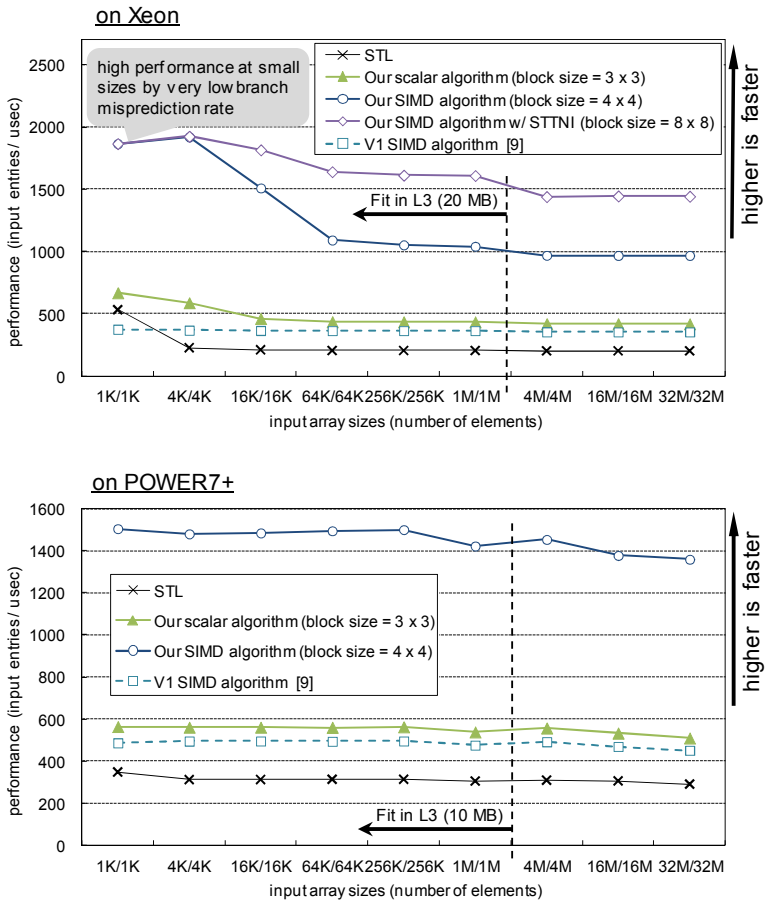
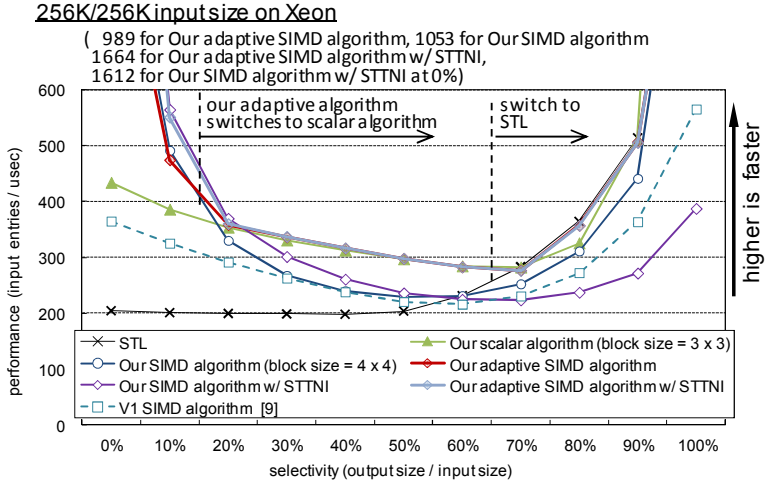


Figure 43. Performance for intersecting 32-bit integer arrays of various sizes.

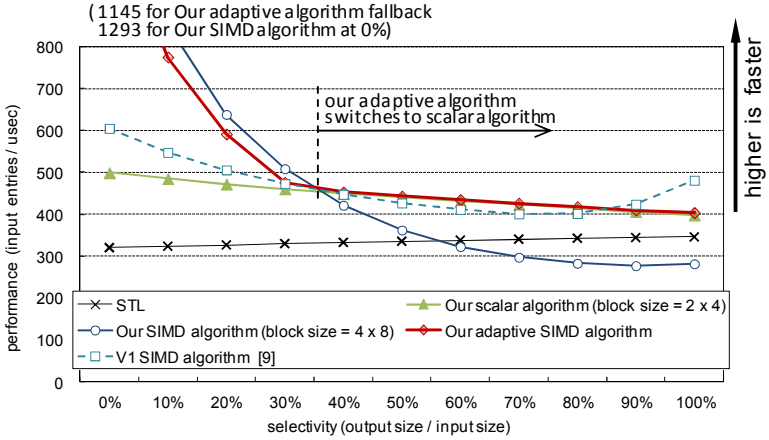
5.4.4 Adaptive Fall Back Based on Selectivity to Avoid Performance Degradations

Figure 44 shows how the selectivity affected the performance of our algorithm using 32-bit integers based on a random number generator with various selectivity values. Our algorithm worked best when the selectivity was small, which is true for many real-world applications. For example, Ding and König [2011] reported that the selectivity was less than 10% for 94% of the queries and less than 1% for 76% of the queries in the 10,000 most frequent queries in a shopping portal site. For this frequent situation, our algorithm worked well, especially with the SIMD instructions.

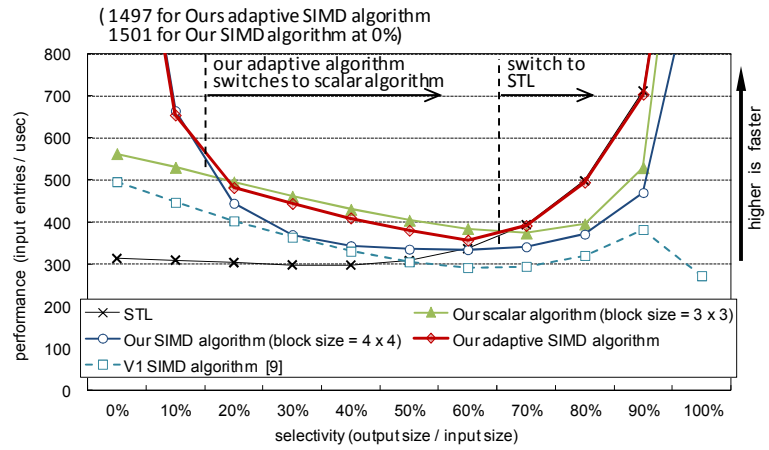
256K/256K input size on Xeon



256K/1M input size on Xeon



256K/256K input size on POWER7+



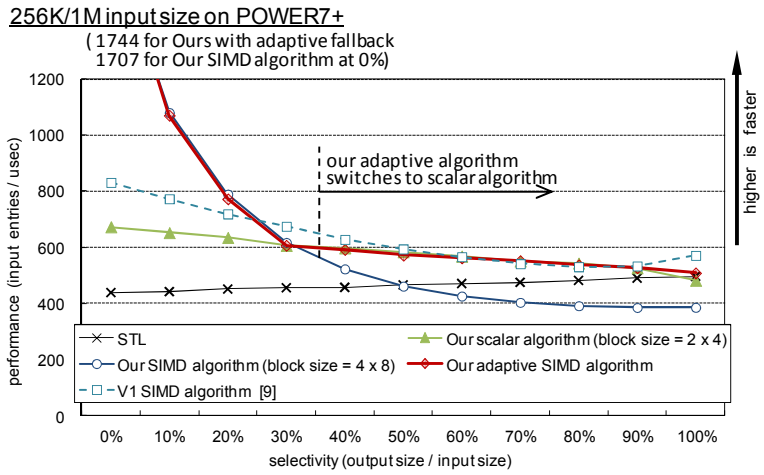


Figure 44. Performance of each algorithm for random 32-bit integers with various selectivity on Xeon and POWER7+.

However, the performance of our algorithm was worse than STL when the selectivity was high. To avoid these performance degradations, we added an adaptive fallback mechanism to our algorithm. We start execution with our SIMD algorithm, but with a periodic runtime check of the selectivity that may trigger the fallback mechanism. We calculate the selectivity after each 1,024 output elements by checking the numbers of input elements processed to generate these output elements. When the numbers of input elements is larger than the threshold in at least one array, we fall back to another algorithm. This insures the overhead caused by the runtime check is not significant when there are few output elements. An adaptive fallback using a runtime check is a standard heuristic technique to avoid worst case performance in many algorithms. For example, introsort [Musser 1997] used in the STL's std::sort library method uses quicksort with adaptive fallback to heapsort to avoid the $O(N^2)$ worst-case performance of quicksort.

From the results shown in Figure 44, for two input arrays with comparable sizes, we start execution with our SIMD algorithm using the block size setting of 4x4 (or 8x8 if we use STTNI on Xeon). We switch to the STL when the selectivity is higher than 65%. When the selectivity is higher than 15% but lower than 65%, we use our scalar algorithm with the block size setting of 3x3. We also execute a periodic check in our scalar algorithm that may fall back to the STL algorithm. If one of the input arrays is more than twice as large as the other, we start execution with our SIMD algorithm using the block size setting of 4x8 and fall back to our scalar algorithm with the block size setting of 2x4 if the selectivity becomes

higher than 35%. We do not switch to STL because out scalar algorithm consistently outperformed STL in Figure 44. We summarize how we select the algorithm and the block size based on the size of two input arrays and the selectivity with and without using SIMD instructions in Figure 45. We call these overall algorithms the *adaptive SIMD algorithm* and the *adaptive scalar algorithm*. Figure 44 shows that our fallback mechanisms selected the appropriate algorithm for each selectivity.

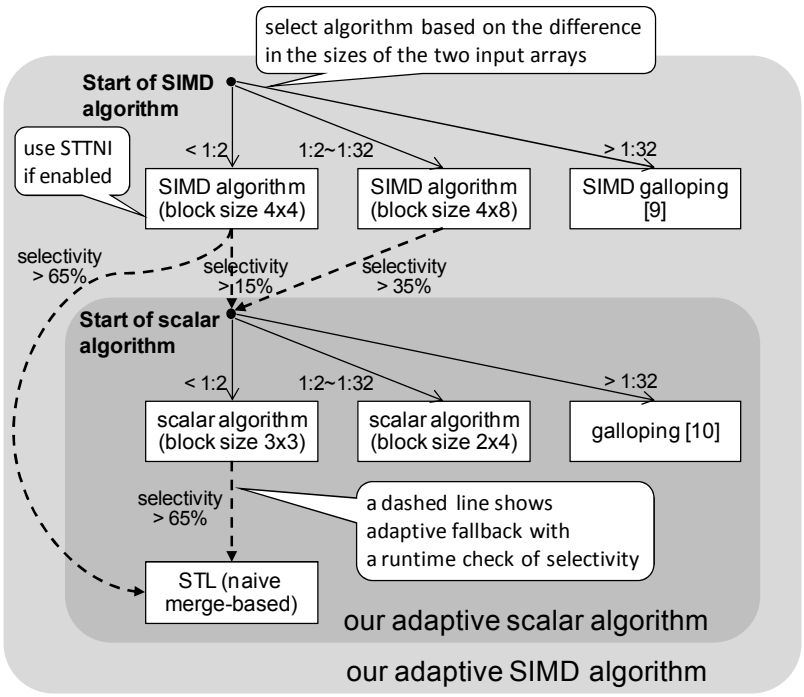


Figure 45. Overall scheme of our adaptive algorithm.

5.4.5 Performance with Realistic Datasets

Finally, we evaluated the performance of our algorithms for realistic datasets generated from a Wikipedia database dump to emulate the set intersection operation in a query serving system. We compare the performance of set intersections of multiple arrays to emulate multi-word queries. Here we compare our SIMD and scalar algorithm against a combination of existing SIMD algorithms, the V1 SIMD algorithm with SIMD galloping [Lemire et al. 2014]. We switched between these two algorithms based on the difference in the two input arrays and we used 1:50 as the selection threshold based on their results. We also compared the performance of our SIMD algorithm with the STTNI instruction against Schlegel's algorithm [2011], which also exploits the STTNI, combined with SIMD galloping on Xeon. As a baseline, we also measured a combination of STL (as a representative

merge-based algorithm) and a scalar galloping algorithm (as a binary-search-based algorithm). We prepared a list of document IDs for 16 search words and generated 2-word, 3-word, 6-word, and 8-word queries by randomly selecting the keywords from the 16 prepared words. The size of the list for each keyword ranged from 10,733 elements to 528,974 elements. For each class, we generated 100 queries and measured the total execution time of these queries. For intersecting multiple words, we repeatedly picked the two smallest sets and did set intersection for the two arrays, a technique that is frequently described in the literature.

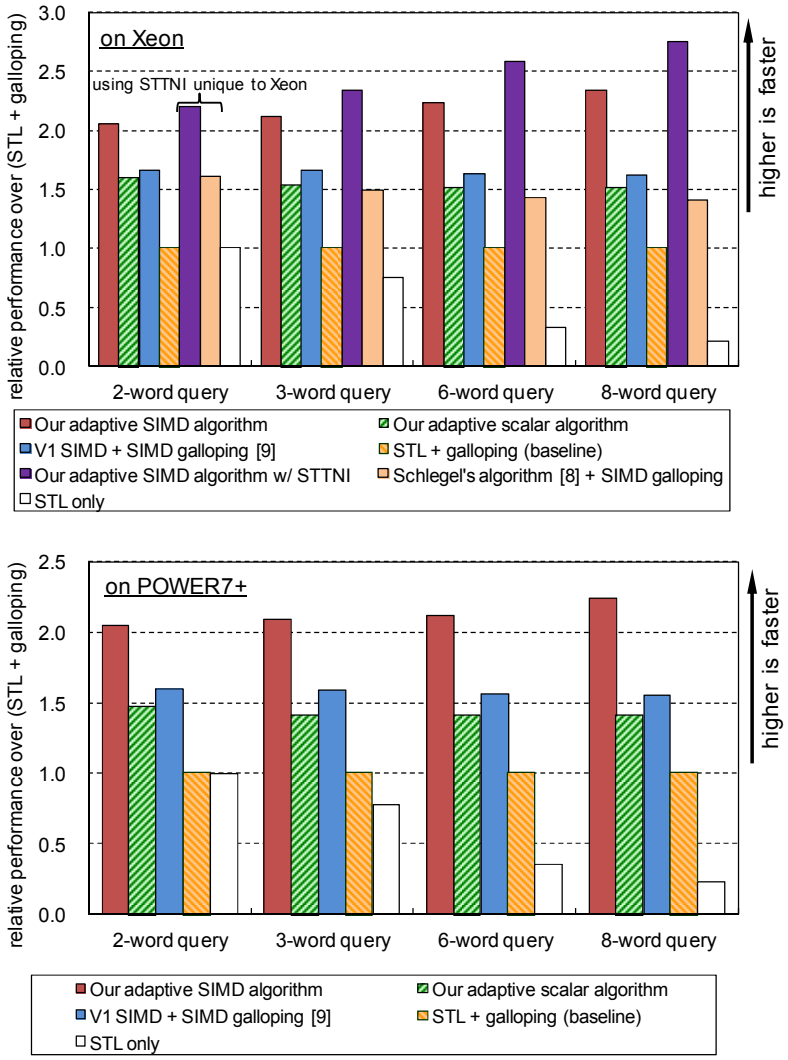


Figure 46. Performance of set intersection algorithms using the datasets generated from Wikipedia database.

Figure 46 shows the relative performance of each algorithm over the baseline (STL + galloping). On both platforms, our SIMD algorithm more than doubled the baseline performance. The V1 SIMD + SIMD galloping algorithm also accelerated the operation by exploiting SIMD instructions, but its gain was about 60% on both platforms and hence our SIMD algorithm outperformed V1 SIMD + SIMD galloping by from 24% (3-word queries on Xeon) to 44% (8-word queries on Xeon). Although V1 SIMD + SIMD galloping and our SIMD algorithm use the same SIMD galloping algorithm when intersecting two arrays with very different sizes, our algorithm achieved higher performance for arrays with similar sizes and this mattered for the overall performance. On Xeon, our SIMD algorithm can achieve even higher performance with STTNI. Schelegel's algorithm also accelerated the set intersection using the STTNI instruction, while the algorithm performed much worse than STL for the artificial dataset generated by the random number generator, as shown in Figure 40. This is because the value domain for the Wikipedia dataset was smaller than the artificial datasets and hence more elements shared the same values in their upper 16 bits. This is important for Schelegel's algorithm because they use STTNI to find matching pairs in the lower 16 bits within the elements that share the same value in the upper 16 bits. However, the performances of Schelegel's algorithm were not as good as our SIMD algorithm or the V1 SIMD algorithm.

Our scalar algorithm improved the performance by about 50% over the baseline in spite of not using the SIMD instructions. The performance of STL alone was significantly lower than the other algorithms because STL, or merge-based-algorithms in general, performed poorly when the sizes of two arrays are quite different and this configuration is known to be important for intersecting multiple sets.

5.4.6 Energy Efficiency

As we did in Chapter 3, we measured the total energy consumption of the system while running different set intersection algorithms. We used the same system of the external power meter and a Haswell-based commodity PC for this measurement.

Figure 47(a) shows the energy consumption rate as measured by the power meter while intersecting two 256k random integer arrays. The selectivity was set to 0. Since only our SIMD algorithm used the vector unit of the processor, it consumed up to 5.0% more energy per seconds than other two scalar algorithms. However, due to the huge differences in the performance as shown previously, our SIMD algorithm was much more energy efficient

to process the same number of input entries; our SIMD algorithm consumed only 19.9% energy per input entry compared to STL and our scalar algorithm consumed 43.5% of the STL.

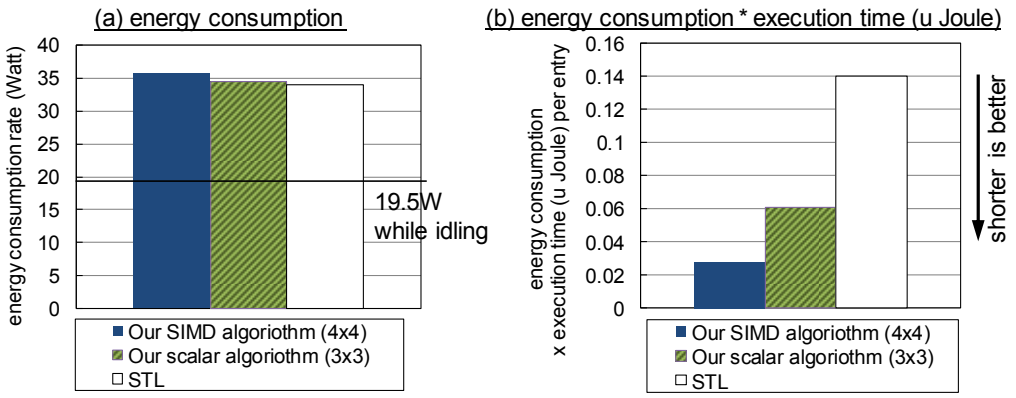


Figure 47. (a) energy consumption rate (Watt) while executing each set intersection algorithm for two 256k random 32-bit integer arrays. (b) Total consumed energy (μ Joule) per input entry.

5.5 Summary

This chapter described our new highly efficient algorithm for set intersections on sorted arrays on modern processors. Our approach drastically reduces the number of branch mispredictions and efficiently exploits the SIMD instructions. Our algorithm is not only efficient but also portable, easy to implement, and requires no preprocessing. Our results show that our simple and portable scalar algorithm improved the performance of the important set intersection operation by reducing the branch overhead. The use of the SIMD instructions in our algorithm gave additional performance improvements by reducing the path length significantly for many datasets. We believe our algorithm will be quite effective to improve the performance of the set intersection operations for many workloads.

Chapter 6

Conclusions

6.1 Conclusions

In this dissertation, we described our new algorithms for efficiently exploiting SIMD instructions in sorting and sorted set intersection, two of the most important workloads that were not traditional targets of acceleration with SIMD instructions. In all cases, we showed that SIMD instructions can give significant performance acceleration over the existing algorithms by enhancing the algorithms for the better exploitation of SIMD instructions.

The key to achieve high performance is 1) to exploit data parallelism available in the algorithm and 2) to reduce the number of branch mispredictions while avoiding non-contiguous memory accesses, which increases the memory access overhead. Although the data parallelism available in one instruction is an obvious and straightforward advantage of the SIMD instructions, the reduced branch misprediction overhead also gives non-negligible performance gain; and hence the advantage of the SIMD instructions can surpass the data parallelism of the SIMD instruction. For example, we demonstrated 7.3x to 12.8x performance improvement using 4-wide SIMD instruction (SSE for 32-bit integers) in various sorting algorithms that are suitable for SIMD.

To reduce the number of branch mispredictions, we take two different approaches for sorting and set intersection. For set intersection, we aggregate multiple conditional branches

into one since the directions of most of the conditional branches are same, as we discussed in Chapter 5. For sorting, on the other hands, we replace conditional branches by SIMD minimum and maximum instructions. In sorting, especially for random numbers, the directions of conditional branches are unpredictable; hence it is not effective to aggregate multiple branches. By replacing control flow of the unpredictable conditional branches into a data flow by arithmetic instructions avoid the huge overhead of branch mispredictions and hence very effective to improve the performance.

Although we take different approaches for handling conditional branches, some of the optimization techniques involved in our algorithms is common among our target workloads. For example, using a smaller data type instead of a larger type to increase the data parallelism in one instruction is an important technique to get larger performance gain in sorting of structures and set intersection. Typically, using a small data type does not improve the computation performance with scalar processing on today's processors, and hence it is unique to SIMD processing.

In addition to the superior performance with SIMD instructions, we have demonstrated that we can improve the energy efficiency (performance per Watt) using SIMD instructions efficiently. The energy efficiency is critically important for computing systems today ranging from super computers to mobile devices. We observed only small increase in energy consumption in trade for huge performance boost for both sorting and set intersection. Using SIMD increases energy consumption in vector ALUs, but it reduces the execution time. It also reduces the wasted energy due to the speculatively-executed instructions that are not committed. In total, we observed significant improvement in the energy efficiency. Hence, our results show that our new algorithms can contribute wide range of applications and systems.

6.2 Future direction

We developed new SIMD algorithms for sorting and set intersection. However, there is still wide variety of applications that we cannot efficiently execute with SIMD instructions despite of their importance. Such applications include, for example, data conversion, data compression/decompression, or graph data processing. Developing new algorithms suitable for SIMD instructions for such applications are important future work to achieve higher performance in real systems. In some applications, the branch mispredictions and the divergent control flow are important problems we need to tackle. In other applications, such as graph-based applications, divergence in memory accesses are also a quite important

problem to solve. The gather and scatter instructions to handle non-contiguous data flow are becoming popular among new processors, including both GPUs and general-purpose processors. However, gather and scatter are more costly than contiguous memory accesses even with advanced processor architectures and hence how to efficiently handle divergence in data accesses will matter for future processors.

In addition to the gather and scatter instructions, new processors are supporting richer sets of SIMD instructions; they include the predicated instructions with vector masks, and special-purpose instructions for string processing or cryptography. Also the vector length is becoming longer and longer. However, in trade for the higher peak performance, longer vector ISAs often have more limitation in permutation (shuffle) instructions; which is key instructions to enable non-trivial algorithms with SIMD instructions. To identify ways to use these advanced features efficiently will a key contribution of future research projects. However, we need to take care about the tradeoff between portability among processor architectures and the performance gain, since the new features are often unique to a specific processor. Hence, developing a new way of programming with SIMD instructions to balance various metrics, including performance, programmability and portability, is another direction for future work for the wider use of the SIMD instructions in the real world.

Bibliography

J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the Symposium on Principles of Programming Languages* (POPL), 1983.

N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. In *Proceedings of VLDB Endow.* (PVLDB), 4(8), pp. 470–481, 2011.

R. Baeza-Yates and A. Salinger. Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences. In *Proceedings of the International Conference on String Processing and Information Retrieval* (SPIRE), pp. 13–24, 2005.

R. Baeza-Yates. A Fast Set Intersection Algorithm for Sorted Sequences. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching* (CPM), pp. 400–408, 2004.

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

L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro* 23(2), pp. 22–28. 2003.

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

J. L. Bentley and A. C. Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3), pp. 82-87, 1976.

P. Bille, A Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In *Proceedings of the International Conference on Algorithms and Computation* (AICoB), pp. 739–750, 2007.

D. Cederman and P. Tsigas, A Practical Quicksort Algorithm for Graphics Processors. In *Proceedings of the European Symposium on Algorithms* (ESA), pp. 246–258, 2008.

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.* (PVLDB), Vol. 1 (2), pp. 1313–1324, 2007.

E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Annual Symposium on Discrete Algorithms* (SODA), pp. 743–752, 2000.

B. Ding and A. C. König. Fast set intersection in memory. In *Proceedings of VLDB Endow.* (PVLDB), Vol. 4 (4), pp. 255–266, 2011.

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transaction on Computers*. Vol. 21 (9), pp. 948–960, 1972.

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

Freescale Semiconductor Inc. AltiVec Technology Programming Interface Manual. 1999.

T. Furtak, J. N. Amaral, and R. Niewiadomski, Using SIMD Registers and Instructions to Enable Instruction-Level Parallelism in Sorting Algorithms. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA), pp. 348–357, 2007.

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* (VLDB), pp. 1286–1297, 2007.

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

N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD), pp. 611–622, 2005.

N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD), pp. 325–336, 2006.

G. Graefe, Implementing sorting in database systems, *ACM Computing Surveys*, Vol. 38 (3), Article No. 10, 2006.

IBM Corp. Power ISA Version 2.06 Revision B. 2010.

IBM Corp. PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual. 1998.

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 (PACT), pp. 189–198, 2007.

H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. A high-performance sorting algorithm for multicore single-instruction multiple-data processors, *Software: Practice and Experience*, Vol. 42 (6), pp. 753–777, 2012.

H. Inoue, M. Ohara, and K. Taura. Faster Set Intersection with SIMD instructions by Reducing Branch Mispredictions, In *Proceedings of VLDB Endow.* (PVLDB), Vol. 8 (3), pp. 293–304, 2014.

H. Inoue and K. Taura. SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures, In *Proceedings of VLDB Endow.* (PVLDB), Vol. 8 (11), pp. 1274–1285, 2015.

Intel Corp. *Intel 64 and IA-32 Architectures Software Developer Manuals*, version 056, 2015 (accessed December 7, 2016)

J. Keller and C. W. Kessler, Optimized Pipelined Parallel Merge Sort on the Cell BE, In *Euro-Par 2008 Workshops - Parallel Processing*. 2009.

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.* (PVLDB), Vol. 2 (2), pp. 1378–1389, 2009.

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* (SIGMOD), pp. 841–850, 2012.

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

S. Lacey and R. Box, A Fast, Easy Sort. *Byte Magazine*, April, pp. 315–320, 1991.

D. Lemire, L. Boytsov and N. Kurz. SIMD Compression and the Intersection of Sorted Integers. arXiv:1401.6399, 2014. (to be published in *Software: Practice and Experience*; DOI: 10.1002/spe.2326)

S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An Evaluation of Vectorizing Compilers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (PACT), pp. 372–382, 2011.

W. A. Martin, Sorting, *ACM Computing Surveys*, Vol. 3 (4), pp. 147–174, 1971.

D. R. Musser. Introspective Sorting and Selection Algorithms, *Software Practice and Experience*, Vol. 27 (8), pp. 983-993, 1997.

J. Nickolls, I. Buck, and M. Garland. Scalable Parallel Programming with CUDA. *ACM Queue*, Vol. 6 (2), pp. 40–53. 2008.

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* (SIGMOD), pp. 755–766, 2014.

D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, The Design and Implementation of a First–Generation CELL Processor. In *Proceedings of the IEEE International Solid–State Circuits Conference* (ISSCC), pp. 184–185. 2005.

T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan, Photon mapping on programmable graphics hardware. In *Proceedings of ACM SIGGRAPH/ EUROGRAPHICS Workshop On Graphics Hardware*, pp. 41–50, 2003.

P. Sanders and S. Winkel. Super Scalar Sample Sort. In *Proceedings of the European Symposium on Algorithms* (ESA), volume 3221 of LNCS, pp. 784–796, 2004.

P. Sanders, F. Transier. Intersection in Integer Inverted Indices. In *Proceedings of the Workshop on Algorithm Engineering and Experiments* (ALENEX), pp. 71–83, 2007.

N. Satish N, M. Harris, and M. Garland, Designing efficient sorting algorithms for manycore GPUs. In *Proceeding of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (IPDPS). pp. 1–10. 2009.

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* (SIGMOD), pp. 351–362, 2010.

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.

B. Schlegel, T. Willhalm, and W. Lehner. Fast Sorted-Set Intersection using SIMD Instructions. In *Proceedings of the Second International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures* (ADMS), 2011.

R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *ACM Journal of Experimental Algorithmics,* Vol. 9, Article 1.5, 2004.

E. Sintorn and U. Assarsson, Fast parallel GPU-sorting using a hybrid algorithm, *Journal of Parallel and Distributed Computing* Vol. 68 (10), pp. 1381-1388. 2008.

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* (ICS), pp. 293–302, 2013.

S. Tatikonda, F. Junqueira, B. B. Cambazoglu, and V. Plachouras. On efficient posting list intersection with multicore processors. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval* (SIGIR), 2009.

D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. In *Proceedings of VLDB Endow.* (PVLDB), 2(1), pp. 838–849, 2009.

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

Appendix

In appendix, we show additional pseudocode of our algorithms to show more detail of our algorithm and implementation used in the evaluations.

Appendix 1:
Pseudocode of odd-even merge algorithm for 64-bit integers with SSE using floating-point min and max instructions

```
merge_4x4_64bit(__m128i &vA0, __m128i &vA1,        // input 1
                __m128i &vB0, __m128i &vB1,        // input 2
                __m128i &vMin0, __m128i &vMin1,    // output
                __m128i &vMax0, __m128i &vMax1) {  // output
    __m128i vTmp0, vTmp1, vTmp2, vTmp3, vTmp4, vTmp5, vTmp6;
    vMin0 = (__m128i)_mm_min_pd((__m128d)vA0, (__m128d)vB0);
    vMin1 = (__m128i)_mm_min_pd((__m128d)vA1, (__m128d)vB1);
    vMax0 = (__m128i)_mm_max_pd((__m128d)vA0, (__m128d)vB0);
    vMax1 = (__m128i)_mm_max_pd((__m128d)vA1, (__m128d)vB1);
    vTmp0 = (__m128i)_mm_min_pd((__m128d)vMin1, (__m128d)vMax0);
    vTmp1 = (__m128i)_mm_max_pd((__m128d)vMin1, (__m128d)vMax0);
    vTmp2 = _mm_shuffle_epi64(vMax1, vTmp0, 1);
    vTmp6 = _mm_shuffle_epi64(vTmp0, vTmp1, 3);
    vTmp4 = _mm_shuffle_epi64(vTmp1, vMax1, 0);
    vTmp3 = (__m128i)_mm_max_pd((__m128d)vTmp2, (__m128d)vMin0);
    vMin0 = (__m128i)_mm_min_pd((__m128d)vTmp2, (__m128d)vMin0);
    vTmp5 = (__m128i)_mm_max_pd((__m128d)vTmp4, (__m128d)vTmp6);
    vTmp1 = (__m128i)_mm_min_pd((__m128d)vTmp4, (__m128d)vTmp6);
    vMax1 = _mm_shuffle_epi64(vTmp5, vTmp3, 1);
    vMax0 = _mm_shuffle_epi64(vTmp5, vTmp1, 2);
    vMin1 = _mm_shuffle_epi64(vTmp3, vTmp1, 1);
}
```

Appendix 2:
Pseudocode of bitonic merge algorithm for 32-bit integers with SSE

```
merge_4x4_32bit(__m128i &vA, __m128i &vB,        // input
                __m128i &vMin, __m128i &vMax) {  // output
    __m128i vTmp, vTmp2, vTmp3, vTmp4;
    vTmp  = _mm_shuffle_epi32(vB, _MM_SHUFFLE_BE(3,2,1,0)); // reverse
    vMin  = _mm_min_epu32(vA, vTmp);
    vMax  = _mm_max_epu32(vA, vTmp);
    vTmp  = (__m128i)_mm_shuffle_ps((__m128)vMin, (__m128)vMax, _MM_SHUFFLE(1,0,1,0));
    vTmp2 = (__m128i)_mm_shuffle_ps((__m128)vMin, (__m128)vMax, _MM_SHUFFLE(3,2,3,2));
    vMin  = _mm_min_epu32(vTmp, vTmp2);
    vMax  = _mm_max_epu32(vTmp, vTmp2);
    vTmp  = (__m128i)_mm_blend_ps((__m128)vMin, (__m128)vMax, 0xA);
    vTmp2 = (__m128i)_mm_blend_ps((__m128)vMax, (__m128)vMin, 0xA);
    vTmp2 = _mm_shuffle_epi32(vTmp2, _MM_SHUFFLE_BE(1,0,3,2));
    vTmp3 = _mm_min_epu32(vTmp, vTmp2);
    vTmp4 = _mm_max_epu32(vTmp, vTmp2);
    vMin  = _mm_unpacklo_epi32(vTmp3, vTmp4);
    vMax  = _mm_unpackhi_epi32(vTmp3, vTmp4);
}

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
    __m128i vTmp0, vTmp1, vTmp2, vTmp3;
    vTmp2 = _mm_shuffle_epi32(vB0, _MM_SHUFFLE(0,1,2,3)); // reverse
    vTmp3 = _mm_shuffle_epi32(vB1, _MM_SHUFFLE(0,1,2,3)); // reverse
    vTmp1 = _mm_min_epu32(vA1, vTmp2);
    vTmp0 = _mm_min_epu32(vA0, vTmp3);
    vB0   = _mm_max_epu32(vA1, vTmp2);
    vB1   = _mm_max_epu32(vA0, vTmp3);
    bmerge_vreg_32(vTmp0,vTmp1,vMin0,vMin1, true);
    bmerge_vreg_32(vB1  ,vB0  ,vMax0,vMax1, true);
}
```

Appendix 3:
Pseudocode of our scalar set intersection algorithm with various block sizes

Our scalar algorithm with block size of 3x3

```
while (1) {
   Adat0 = A[Apos]; Adat1 = A[Apos + 1]; Adat2 = A[Apos + 2];
   Bdat0 = B[Bpos]; Bdat1 = B[Bpos + 1]; Bdat2 = B[Bpos + 2];
   if (Adat0 == Bdat2) {
      Cvec[Cpos++] = Adat0;
      goto advanceB; // no more pair
   }
   else if (Adat2 == Bdat0) {
      Cvec[Cpos++] = Adat2;
      goto advanceA; // no more pair
   }
   else if (Adat0 == Bdat0) {
      Cvec[Cpos++] = Adat0;
   }
   else if (Adat0 == Bdat1) {
      Cvec[Cpos++] = Adat0;
   }
   else if (Adat1 == Bdat0) {
      Cvec[Cpos++] = Adat1;
   }
   if (Adat1 == Bdat1) {
      Cvec[Cpos++] = Adat1;
   }
   else if (Adat1 == Bdat2) {
      Cvec[Cpos++] = Adat1;
      goto advanceB;
   }
   else if (Adat2 == Bdat1) {
      Cvec[Cpos++] = Adat2;
      goto advanceA;
   }
   if (Adat2 == Bdat2) {
      Cvec[Cpos++] = Adat2;
      goto advanceAB;
   }
   else if (Adat2 > Bdat2) goto advanceB;
   else goto advanceA;
advanceA:
   Apos+=3;
   if (Apos >= Aend) { break; } else { continue; }
advanceB:
   Bpos+=3;
   if (Bpos >= Bend) { break; } else { continue; }
advanceAB:
   Apos+=3;  Bpos+=3;
   if (Apos >= Aend || Bpos >= Bend) { break; }
}
//  fall back to naive algorithm for remaining elements
```

Our scalar algorithm with block size of 2x4

```
if (input A is longer) { swap input A and B; }
while (1) {
   Adat0 = A[Apos]; Adat1 = A[Apos + 1];
   Bdat0 = B[Bpos]; Bdat1 = B[Bpos + 1]; Bdat2 = B[Bpos + 2]; Bdat3 = B[Bpos + 3];
   if (Adat0 == Bdat0) {
      Cvec[Cpos++] = Adat0;
   }
   else if (Adat0 == Bdat1) {
      Cvec[Cpos++] = Adat0;
   }
   else if (Adat0 == Bdat2) {
      Cvec[Cpos++] = Adat0;
   }
   else if (Adat0 == Bdat3) {
      Cvec[Cpos++] = Adat0;
   }
   if (Adat1 == Bdat0) {
      Cvec[Cpos++] = Adat1;
      goto advanceA;
   }
   else if (Adat1 == Bdat1) {
      Cvec[Cpos++] = Adat1;
      goto advanceA;
   }
   else if (Adat1 == Bdat2) {
      Cvec[Cpos++] = Adat1;
      goto advanceA;
   }
   else if (Adat1 == Bdat3) {
      Cvec[Cpos++] = Adat1;
      goto advanceAB;
   }
   if (Adat1 < Bdat3) goto advanceA;
   else goto advanceB;
advanceB:
   Bpos+=4;
   if (Bpos >= Bend) { break; } else { continue; }
advanceA:
   Apos+=2;
   if (Apos >= Aend) { break; } else { continue; }
advanceAB:
   Apos+=3;  Bpos+=3;
   if (Apos >= Aend || Bpos >= Bend) { break; }
}
//  fall back to naive algorithm for remaining elements
```

List of Publications

Publications included in this dissertation

H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. A high-performance sorting algorithm for multicore single-instruction multiple-data processors, *Software: Practice and Experience*, Vol. 42 (6), pp. 753–777, 2012.

H. Inoue, M. Ohara, and K. Taura. Faster Set Intersection with SIMD instructions by Reducing Branch Mispredictions, In *Proceedings of VLDB Endow.* (PVLDB), Vol. 8 (3), pp. 293–304, 2014.

H. Inoue and K. Taura. SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures, In *Proceedings of VLDB Endow.* (PVLDB), Vol. 8 (11), pp. 1274–1285, 2015.

Other publications

H. Inoue and T. Nakatani, "Adaptive SMT Control for More Responsive Web Applications", *2014 IEEE International Symposium on Workload Characterization* (IISWC 2014). pp 41-50, 2014.