Faster Set Intersection with SIMD instructions by Reducing Branch Mispredictions

Hiroshi Inoue[†‡], Moriyoshi Ohara[†], and Kenjiro Taura[‡]

†IBM Research – Tokyo, ‡University of Tokyo

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

ABSTRACT

Set intersection is one of the most important operations for many applications such as Web search engines or database management systems. This paper describes our new algorithm to efficiently find set intersections with sorted arrays on modern processors with SIMD instructions and high branch misprediction penalties. Our algorithm efficiently exploits SIMD instructions and can drastically reduce branch mispredictions. Our algorithm extends a merge-based algorithm by reading multiple elements, instead of just one element, from each of two input arrays and compares all of the pairs of elements from the two arrays to find the elements with the same values. The key insight for our improvement is that we can reduce the number of costly hard-to-predict conditional branches by advancing a pointer by more than one element at a time. Although this algorithm increases the total number of comparisons, we can execute these comparisons more efficiently using the SIMD instructions and gain the benefits of the reduced branch misprediction overhead. Our algorithm is suitable to replace existing standard library functions, such as std::set_intersection in C++, thus accelerating many applications, because the algorithm is simple and requires no preprocessing to generate additional data structures. We implemented our algorithm on Xeon and POWER7+. The experimental results show our algorithm outperforms the std::set_intersection implementation delivered with gcc by up to 5.2x using SIMD instructions and by up to 2.1x even without using SIMD instructions for 32-bit and 64-bit integer datasets. Our SIMD algorithm also outperformed an existing algorithm that can leverage SIMD instructions.

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 [1]. 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 [2, 3] or hierarchical data structures [4], 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.

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 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 1 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+.

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 1, 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 [2] 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

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st – September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 3
Copyright 2014 VLDB Endowment 2150-8097/14/11

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.

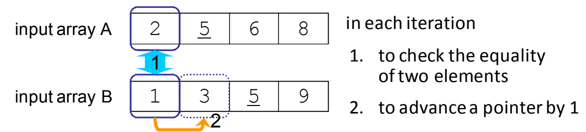
The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes our technique to reduce branch mispredictions. Section 4 gives a summary of our results. Finally, Section 5 summarizes our work.

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 [5], showed that simple algorithms based on a linear merge (e.g. Figure 1(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.

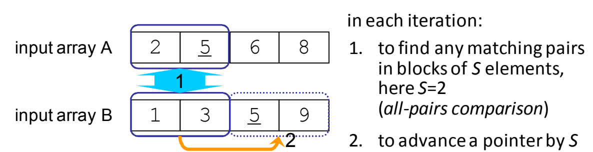
When one input set is much larger than another, the merge-based algorithm is less efficient and many techniques have been proposed to improve its performance. For example, algorithms based on binary search [5-7] reduce the numbers of comparisons and memory accesses by picking values from the smaller set, and efficiently searching for the matching value in the larger set. Similarly, hash-based techniques [2, 3] or techniques using hierarchical data representations [3] 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

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




```
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  }
```

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




```
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
```

(lines 4–18: all-pairs comparison (using S^2 comparisons))

Figure 1. Overview of set intersection without and with our technique. **Red bold** (line 6 in (a) and line 19 in (b)) shows a hard-to-predict conditional branch used to select the pointer to advance.

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 4.

Recently, Ding and König [2] 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.

Table 1. 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 suffer from branch mispredictions. **If-conversion**: eliminates the branch mispredictions in trade for longer path length. is hard to SIMDize. **Ding and König [2]**: requires preprocessing. **Our block-based algorithm:** is an extention to the merge-based technique, reduces the branch mispredictions	**Schlegel [8]**: is mainly targetting 8-bit or 16-bit integers supported by STTNI instructions **Lemire [9] (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 [10])**: 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 [10] with SIMD instructions

Schlegel *et al.* [8] 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. Also, our non-SIMD algorithm does not use any unique instructions, which makes it more portable among processors.

Lemire *et al.* [9] 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 [10] 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.* [8] 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* [11], and is a conventional technique to reduce branch misprediction overhead. Figure 2 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. Table 1 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 2. Pseudocode for set intersection without hard-to-predict conditional branches (the *branchless algorithm*).

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 *et al.* [12] 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, some sorting algorithms [13-15] improved the performance in sorting random input by reducing the branch misprediction overhead using the predicated instructions or SIMD 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 [16-18]. For example, Tsirogiannis *et al.* [16] 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.

3. OUR ALGORITHM

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.

3.1 Key Observation and Assumptions for Input Data

As shown in Figure 1(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 1(a)) and one *if_greater* conditional branch (Line 6 in Figure 1(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 [2].

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 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 switch to a binary-search-based algorithm.

3.2 Our Basic Approach without SIMD instructions

Our technique extends the naive merge-based algorithm shown in Figure 1(a) by reading multiple values from each input array and compares all of their pairs using *if_equal* conditional branches as shown in Figure 1(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 1(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.

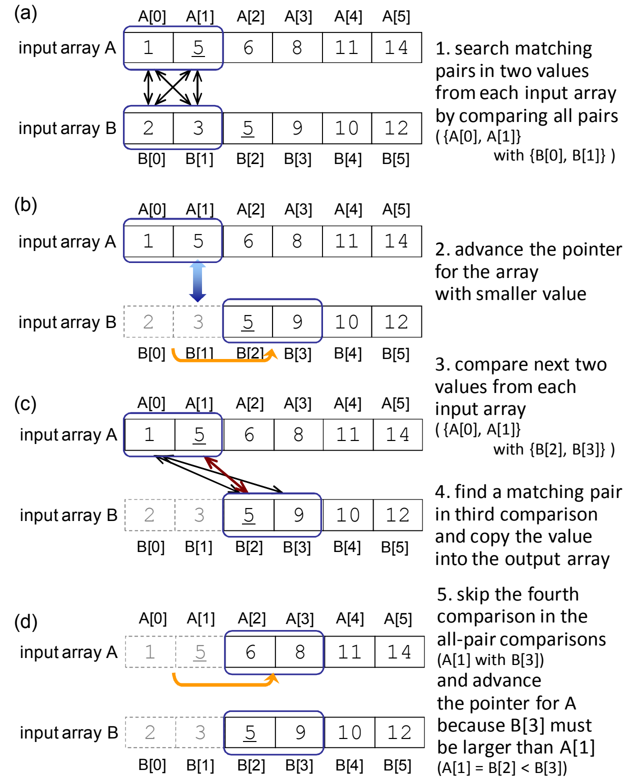


Figure 3. An example of set intersection of two sorted arrays with our technique using 2 as the block size.

Figure 3 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 3 uses up to 4 *if_equal* conditional branches (Lines 4-18 in Figure 1(b)). Then, in Figure 3(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 1(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 Figures 3(a) and 3(b). If a matching pair is found in an all-pairs comparison (Figure 3(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 3(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.

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

Table 2. 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 1(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 1(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.

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. We summarize the total numbers of comparisons in Table 2. 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 1 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 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 4.

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 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 comparisons in their SIMD instructions to intersect 64-bit integer arrays, e.g. 64-bit integers on POWER7+. Also it increases the data parallelism within one SIMD instruction by using only a part of the elements.

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

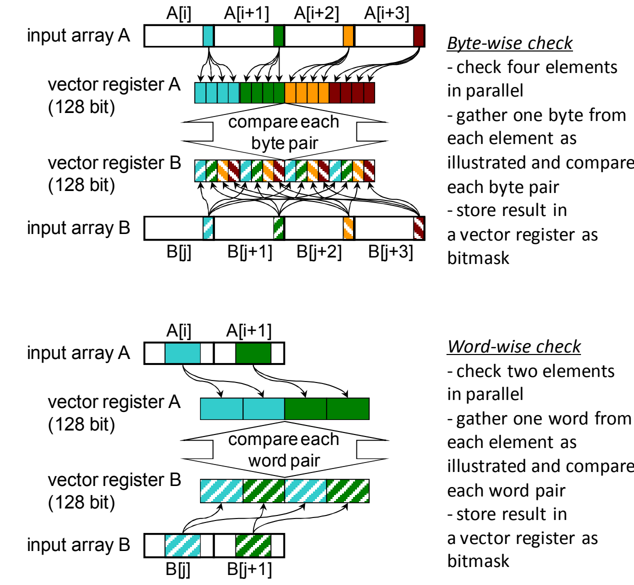


Figure 4. Overview of byte-wise check and word-wise check.

$S_a = 4$ 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 or VSX of POWER7+, 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 [8, 9], 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 *et al.* [12] 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.

We use two different types of checks to find a matching pair in the all-pairs comparison hierarchically. Figure 4 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  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 matching value
7        // do word-wise 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 }
```

Figure 5. Pseudocode of our SIMD algorithm for block size of 4x4.

(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 5 shows the pseudocode for the set intersection algorithm with our hierarchical filtering with SIMD.

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 [8], 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 $popcnt((\sim x) \,\&\, (x-1))$.

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.

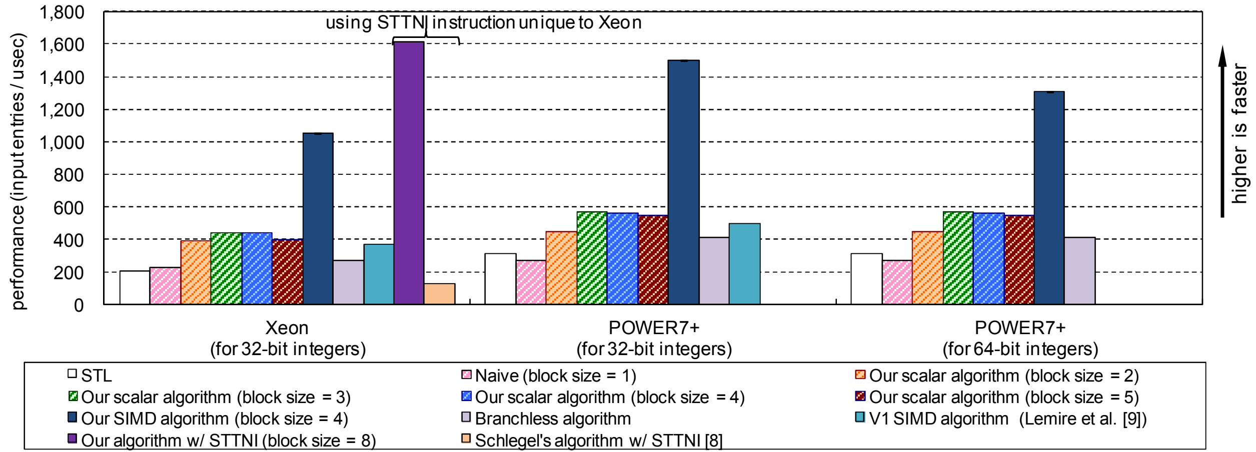


Figure 6. Performance for set intersection of 32-bit and 64-bit random integer arrays of 256k elements on Xeon and POWER7+. The selectivity was set to 0 (as the best case). The error bars show 95% confidence intervals.

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. Redhat 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 Redhat 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 1(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 2, a galloping algorithm [10] (as a popular binary-search-based algorithm), the two SIMD algorithms by Lemire *et al.* [9] (which are the V1 SIMD algorithm and an SIMD galloping algorithm), and Schlegel's algorithm that uses the STTNI instruction of Xeon [8]. 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.

4.1 Performance Improvements from Our Algorithm

Figure 6 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. We used 32-bit integer as the data type for both Xeon and POWER7+ and also tested 64-bit integers for 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 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 [9] 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 [8] 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.

4.2 Microarchitectural Statistics

For more insight into the improvements from our algorithm with and without SIMD instructions, Figure 7 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).

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 7, 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.

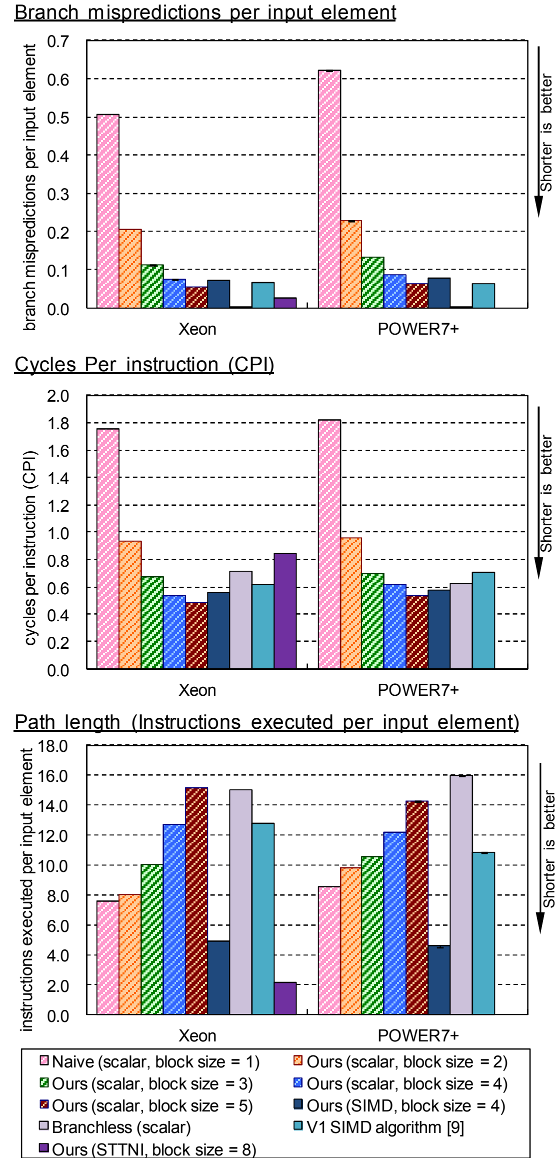


Figure 7. Branch misprediction rate, CPI, and path length.

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.

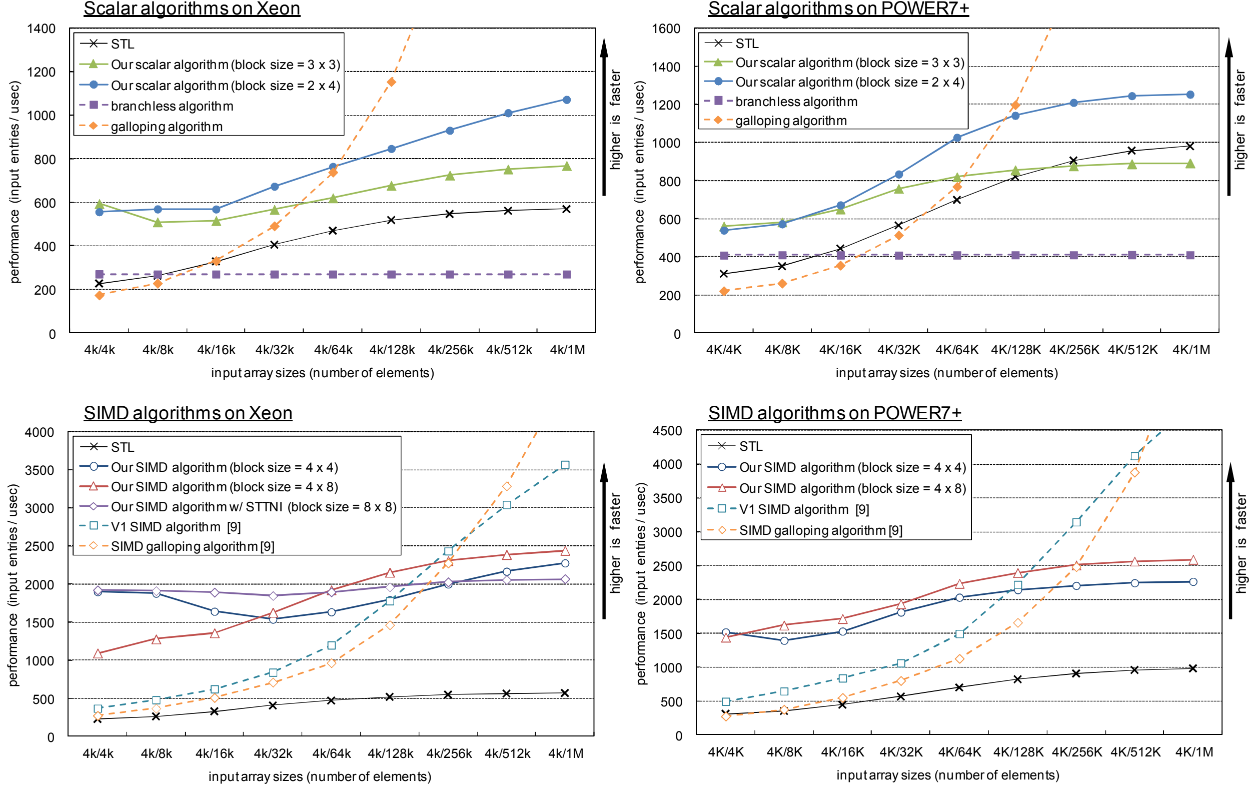


Figure 8. Performance of scalar and SIMD algorithms for intersecting 32-bit integer arrays on Xeon and POWER7+ when the sizes of the two input arrays are different. The selectivity was set to 0.

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 8 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 1, 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 [9] 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 new 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.

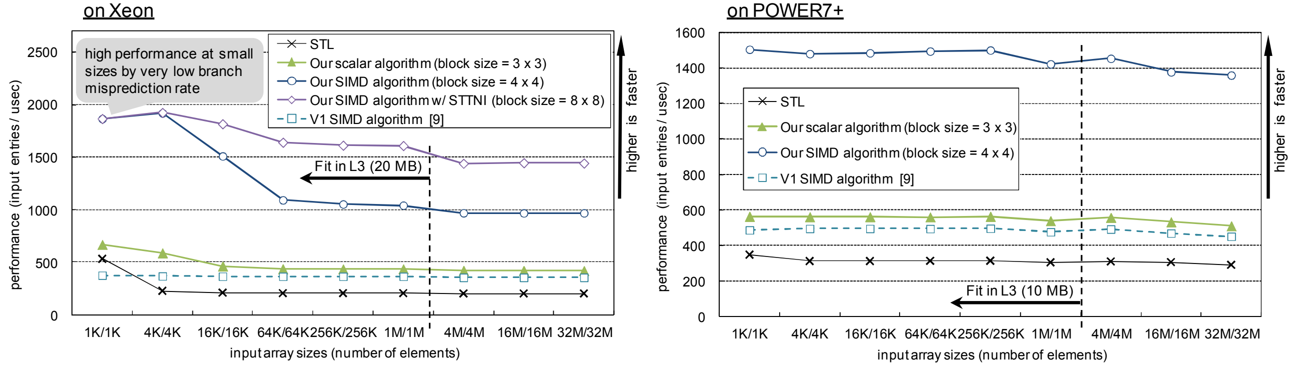


Figure 9. Performance for intersecting 32-bit integer arrays of various sizes. The selectivity was set to 0.

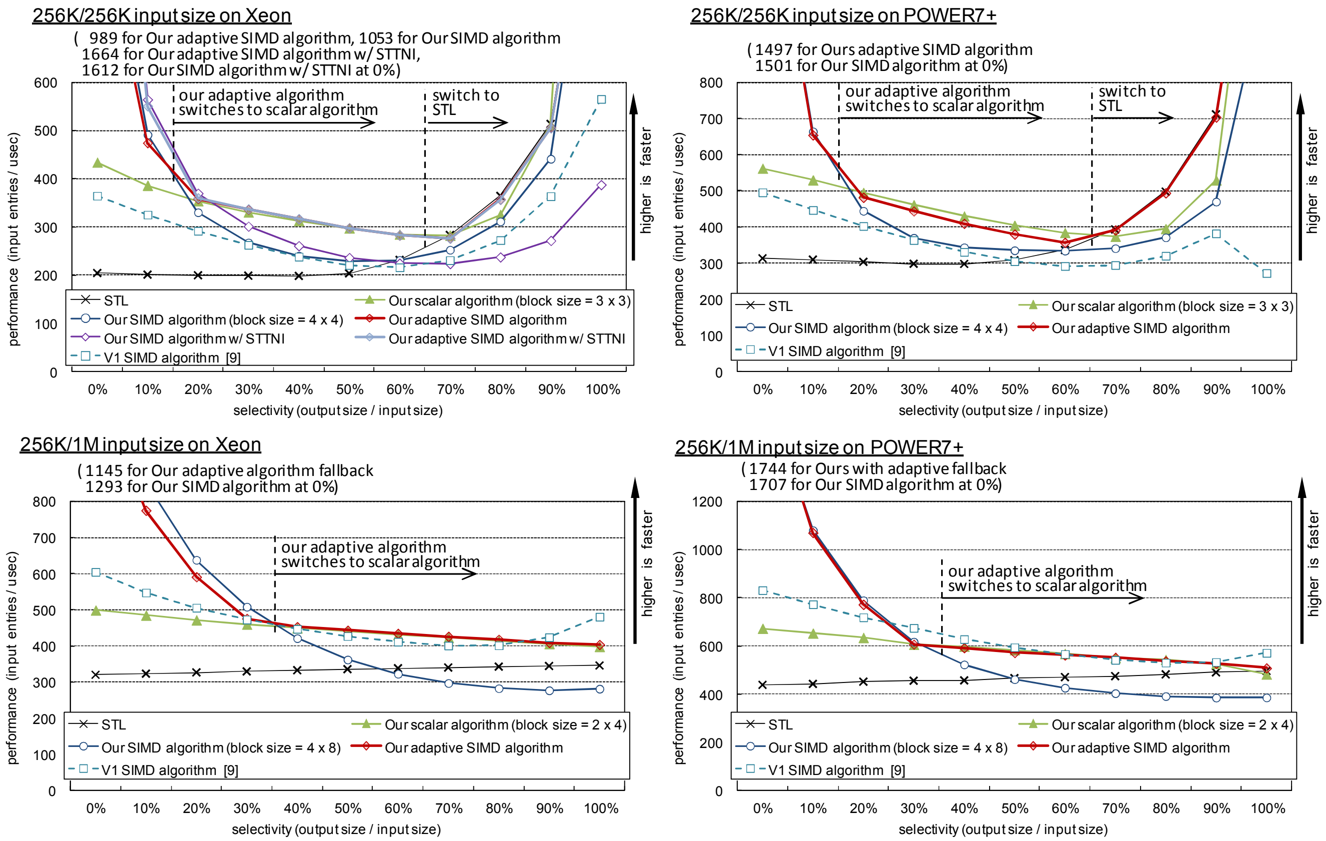


Figure 10. Performance of each algorithm for random 32-bit integers with various selectivity on Xeon and POWER7+.

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 9 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.

4.4 Adaptive Fall Back Based on Selectivity to Avoid Performance Degradations

Figure 10 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 [2] 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.

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 [19] 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 10, 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 10. 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 11. We call these overall algorithms the *adaptive SIMD algorithm* and the *adaptive scalar algorithm*. Figure 10 shows that our fallback mechanisms selected the appropriate algorithm for each selectivity.

4.5 Performance of our algorithm 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 [9]. 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 [8], 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

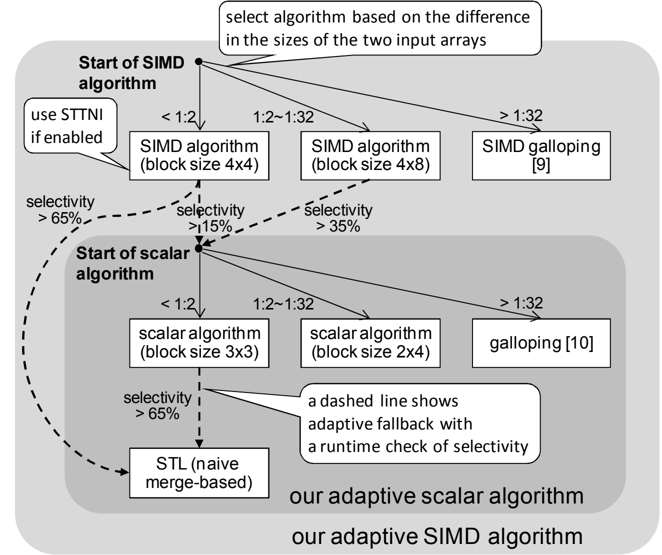


Figure 11. Overall scheme of our adaptive algorithm.

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 12 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 6. 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.

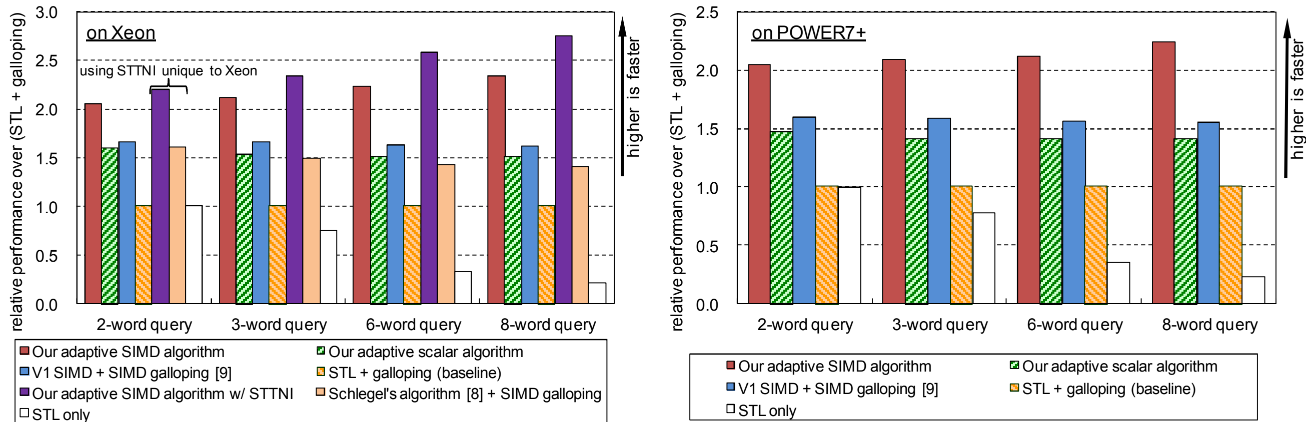


Figure 12. Performance of set intersection algorithms using the datasets generated from Wikipedia database.

5. SUMMARY

This paper 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.

6. ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their valuable comments and suggestions. We thank Toshio Nakatani, Tamiya Onodera, and Takanori Ueda for their useful feedback on earlier drafts on this work.

7. REFERENCES

[1] 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.

[2] B. Ding and A. C. König. Fast set intersection in memory. In Proceedings of VLDB Endow., 4, pp. 255–266, 2011.

[3] 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, pp. 13–24, 2005.

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

[5] 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, pp. 743–752, 2000.

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

[7] P. Sanders, F. Transier. Intersection in Integer Inverted Indices. In Proceedings of the Workshop on Algorithm Engineering and Experiments, pp. 71–83, 2007.

[8] 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, 2011.

[9] D. Lemire, L. Boytsov and N. Kurz. SIMD Compression and the Intersection of Sorted Integers. arXiv:1401.6399, 2014

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

[11] 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, 1983.

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

[13] P. Sanders and S. Winkel. Super Scalar Sample Sort. In Proceedings of the European Symposium on Algorithms, Volume 3221 of LNCS, pp. 784–796, 2004.

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

[15] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. In Proceedings. VLDB Endow., 1(2), pp. 1313–1324, 2008.

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

[17] 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, 2009.

[18] 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., 4(8), pp. 470–481, 2011.

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