IBM Research – Tokyo

THE UNIVERSITY OF TOKYO

Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions

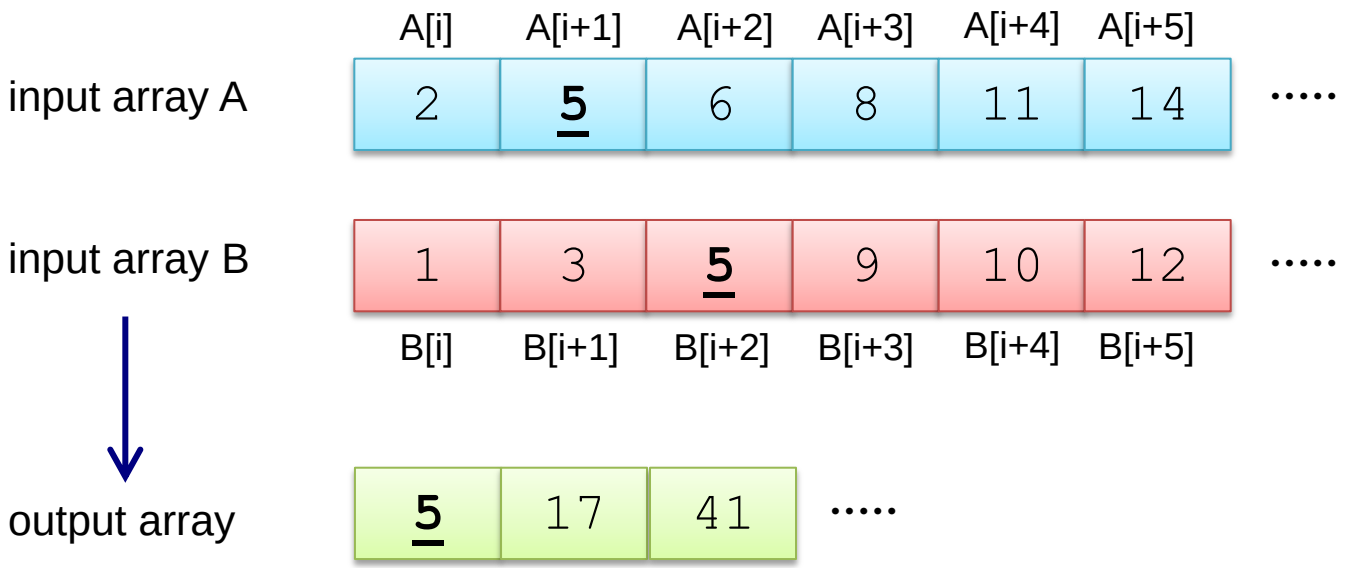
Hiroshi Inoue[†‡], Moriyoshi Ohara[†], Kenjiro Taura[‡]

[†] IBM Research – Tokyo [‡] University of Tokyo

September 2, 2015

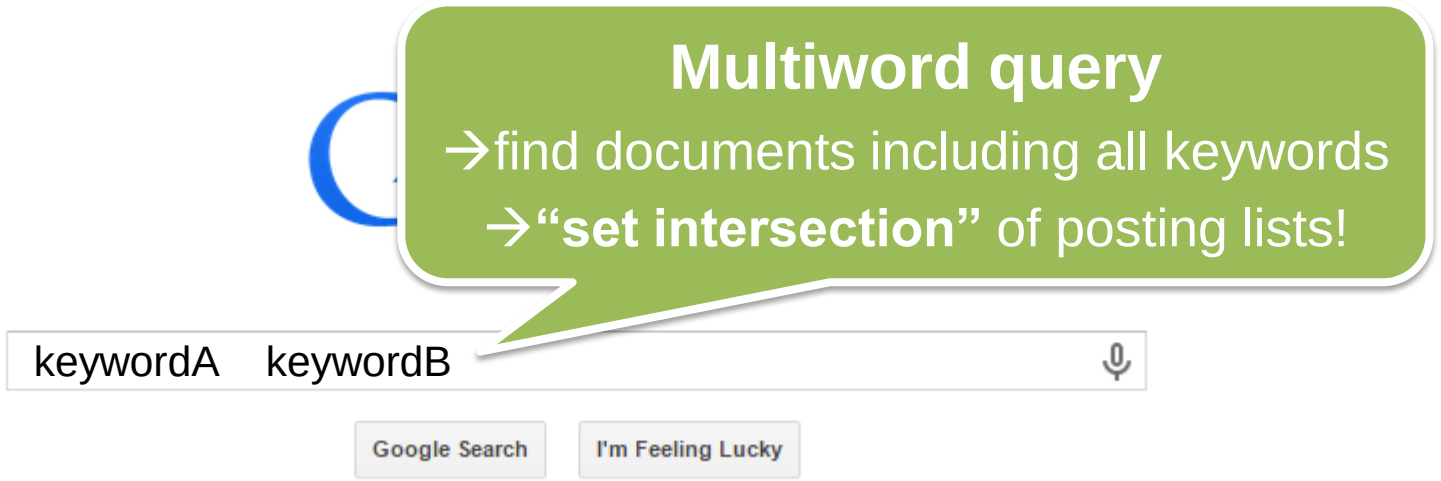
What is Set Intersection?

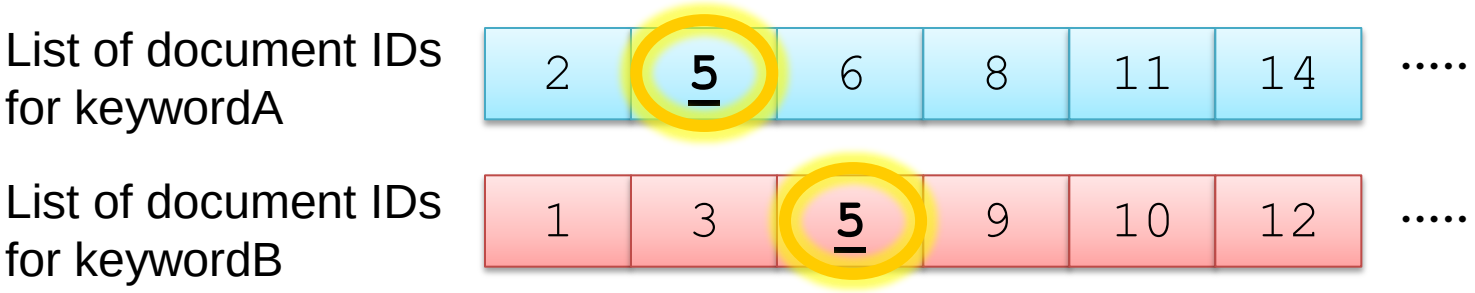
- The operation to find common elements from two sets
- We think intersecting two sorted integer arrays (e.g. std::set_intersection in STL of C++)



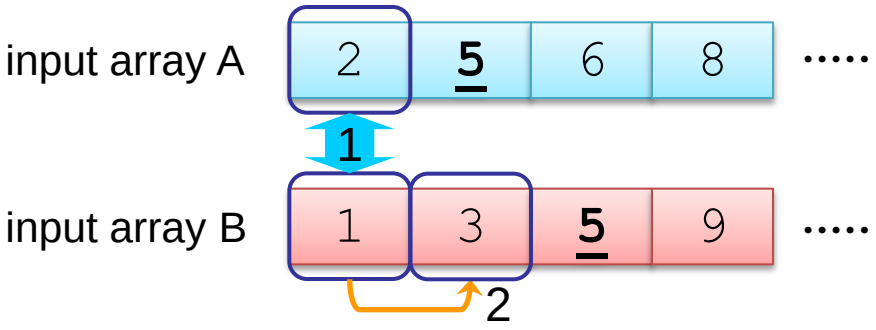
Does it matter?

- Heavily used in DBMS (join operator) and information retrieval systems (multiword AND query)





How can we implement this?



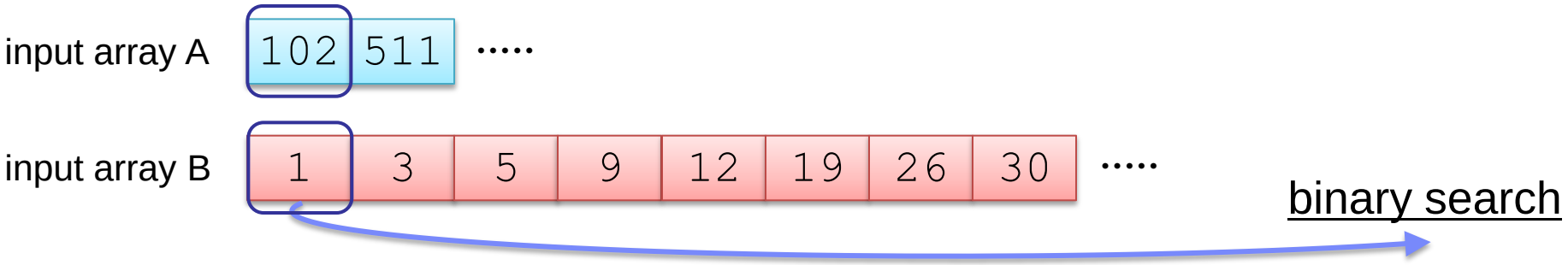
in each iteration

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

Merge-based approach

```
while (pA < pAend && pB < pBend) {
      if      (*pA == *pB) { *pOut++ = *pA++; pB++; }
      else if (*pA  < *pB) { pA++; }
      else                 { pB++; }
}
```

Existing intersection algorithms



- Many techniques have been proposed for intersecting two arrays of very different sizes (10x ~)
  - based on binary search (e.g. galloping)
  - based on additional data structures (e.g. skip list, hash etc)
- They focus on reducing the number of comparisons
- For arrays with similar sizes, the merge-based algorithm is faster than these advanced algorithms ➔ our focus

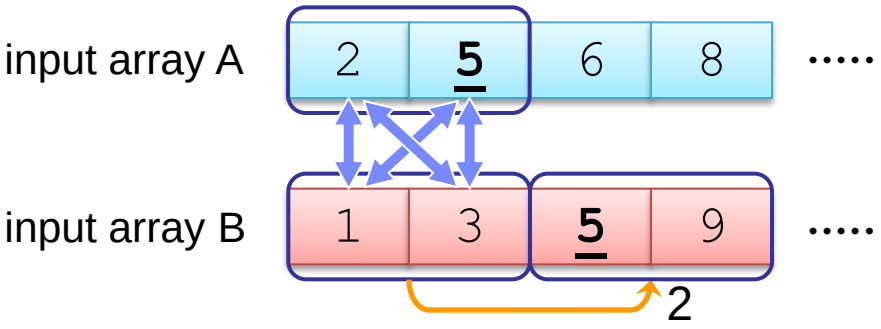
Key observation

```
while (pA < pAend && pB < pBend) {
      if      (*pA == *pB) { *pOut++ = *pA++; pB++; }
      else if (*pA  < *pB) { pA++; }
      else                 { pB++; }
}
```

- The comparison to select an input array for the next block is hard to predict for branch prediction hardware
  - It will be taken in arbitrary order
- The comparison to check equality is much easier to predict
  - It is not taken frequently for many applications

➔We reduce the hard-to-predict conditional branches

Our approach for reducing branch mispredictions



in each iteration:

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

- ☺ reduce the number of the hard-to-predict conditional branches to 1/S
- ☹ increase other (easy-to-predict) conditional branches by S times

➔Based on a simple cost model, the block size of 3 is the best when misprediction penalty is 10~22 cycles

Pseudo code of our approach (with block size $S = 2$)

S^2 easy-to-predict branches per S elements ➔ S times more

```
while (pA < pAend-1 &&  < pBend-1) {
  A0 = *pA; A1 = *(pA+1); B0 = *pB; B1 = *(pB+1);
  if      (A0 == B0) { *pOut++ = A0; }
  else if (A0 == B1) { *pOut++ = A0; Bpos+=2; continue; }
  else if (A1 == B0) { *pOut++ = A1; Apos+=2; continue; }
  if      (A1 == B1) { *pOut++ = A1; Apos+=2; Bpos+=2; }
  else if (A1  < B1) { Apos+=2; }
  else               { Bpos+=2; }
}
```

increment a pointer by S

only one while processing S elements ➔ reduced to $1/S$

Determining the best block size

- A simple cost model of branches for block size *S*

	execution per element	misprediction rate	total cost
if_equal branches	*S*	0%	$S * cost_{exec}$
if_greater branches	1/*S*	50%	$(cost_{exec} + cost_{misp} * 0.5) / S$

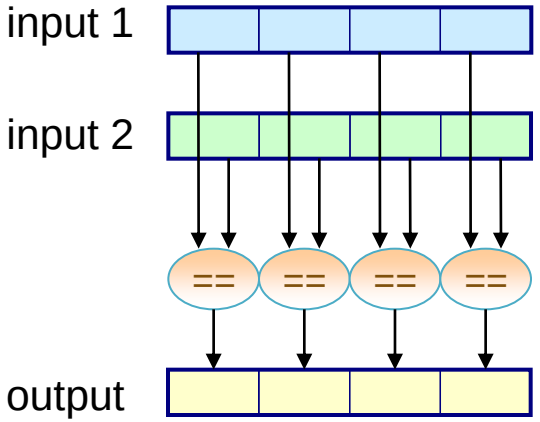
- Best block size is determined by $r = cost_{misp} / cost_{exec}$
  - *S* = 1 when $r \leq 2$
  - S = 2 when $2 \leq r \leq 10$
  - S = 3 when $10 \leq r \leq 22$
  - S = 4 when $22 \leq r \leq 38$

the case for many of recent processors

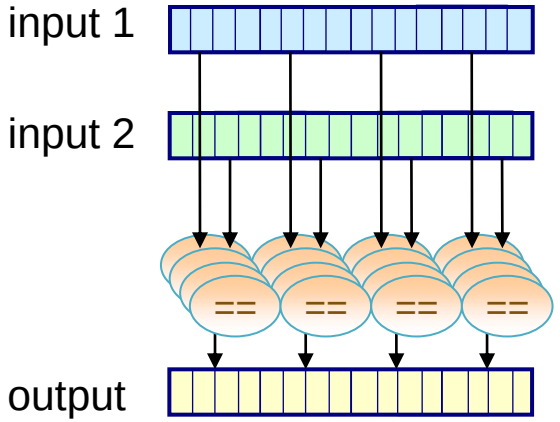
with SIMD, we use *S* = 4 to fully exploit vector register size

Our approach for exploiting SIMD instructions

- Existing approach: full comparison by SIMD to find matching pairs [Lemire *et al.* 2015, Schlegel *et al.* 2011]
  - limited data parallelism
  - limited element size

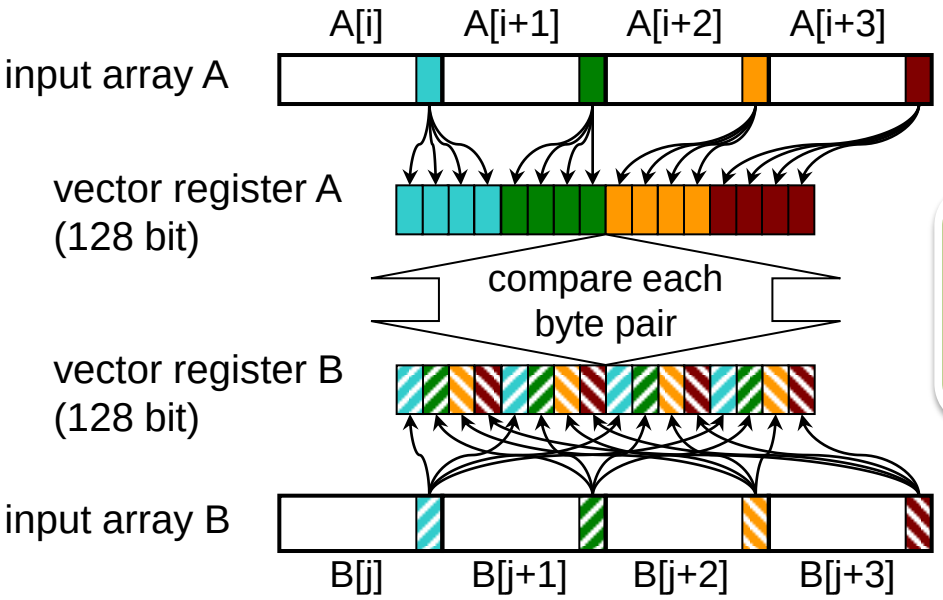


- Our approach: partial comparison by SIMD to filter out redundant comparisons
  - We can enjoy higher data parallelism
  - We can support larger elements (e.g. 32-bit or 64-bit integers)
  - Optimized for the common case



Partial comparison by SIMD

- We introduce partial comparison by SIMD before the scalar comparison to reduce redundant comparisons



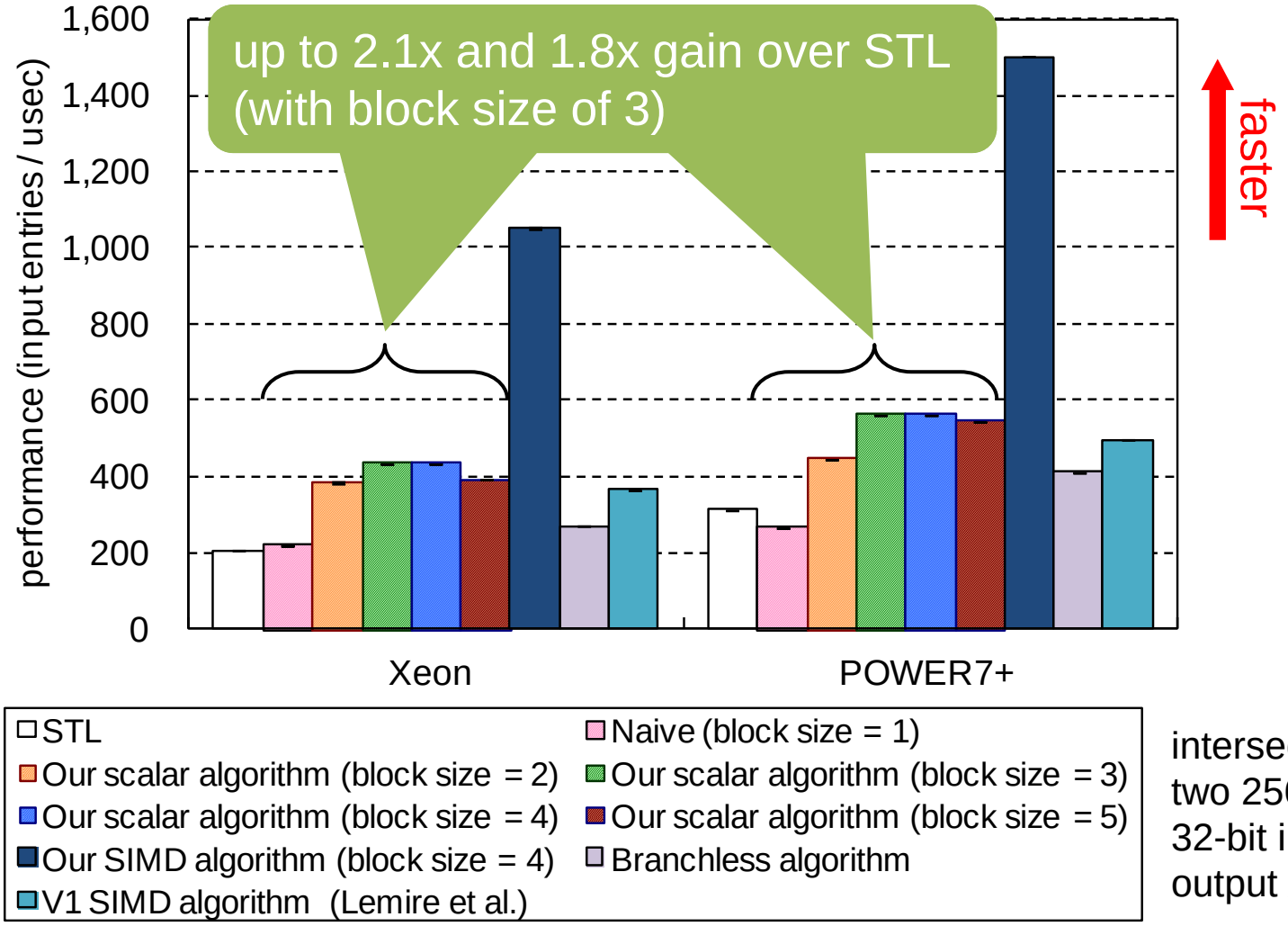
compare only a part of each element to increase parallelism

➔We can skip the all-pairs comparison by scalar if the no matching pair found in the partial comparison by SIMD

Performance Evaluations

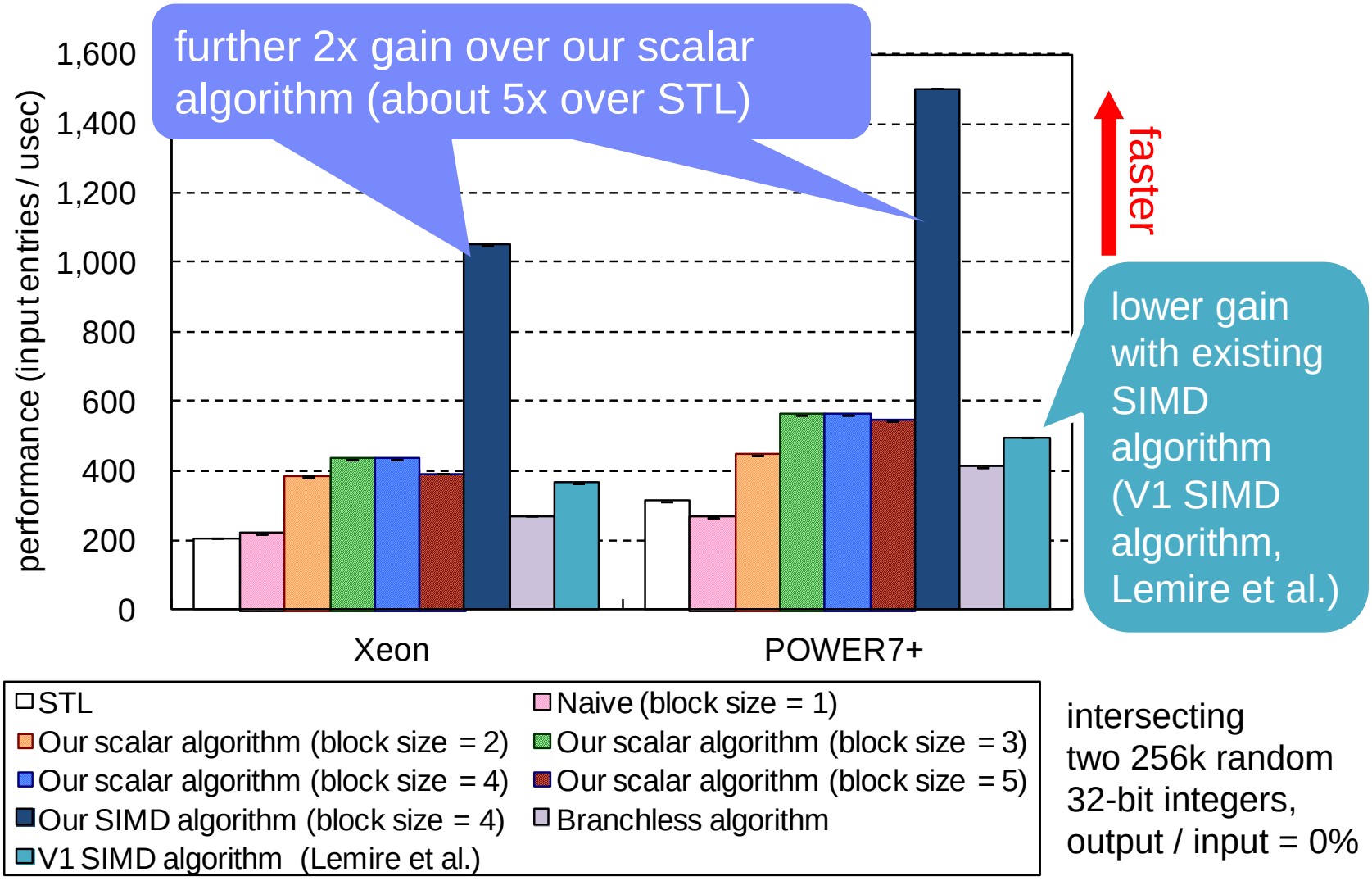
- Systems
  - 2.9-GHz Xeon E5-2690 (SandyBridge-EP) processors
    - using SSE instructions (128-bit SIMD)
    - Redhat Enterprise Linux 6.4, gcc-4.8.2
  - 4.1-GHz POWER7+ processors
    - using VSX instructions (128-bit SIMD)
    - Redhat Enterprise Linux 6.4, gcc-4.8.3

Performance improvements by our scalar algorithm



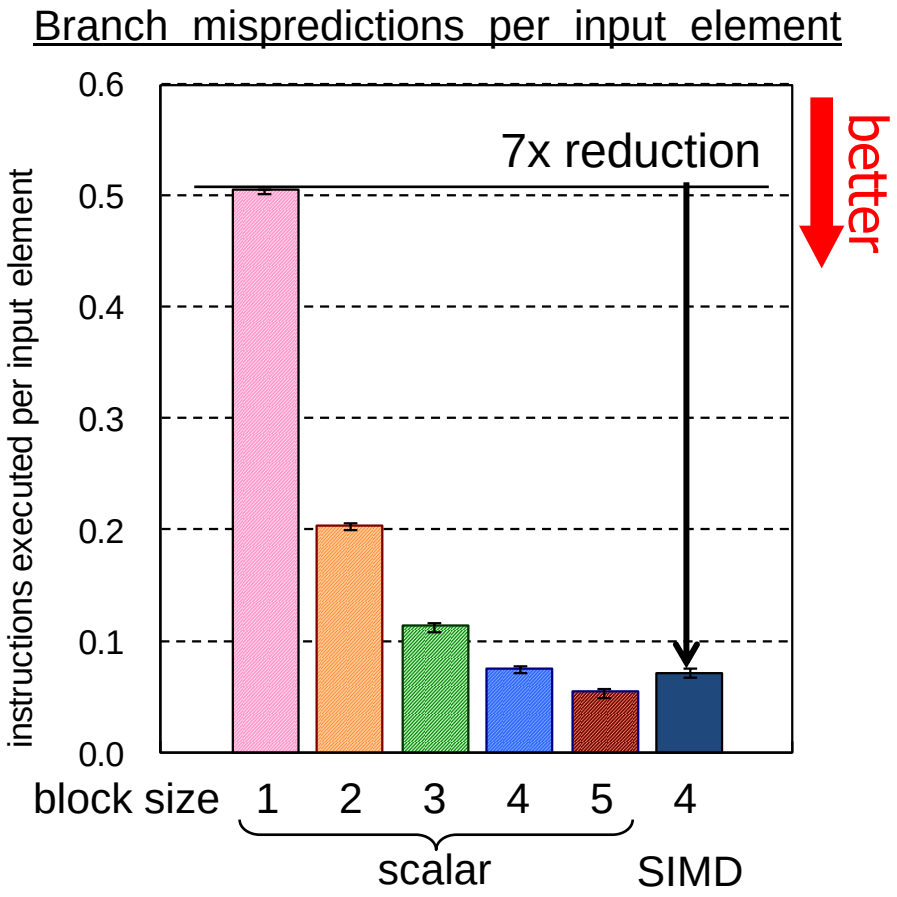
intersecting
two 256k random
32-bit integers,
output / input = 0%

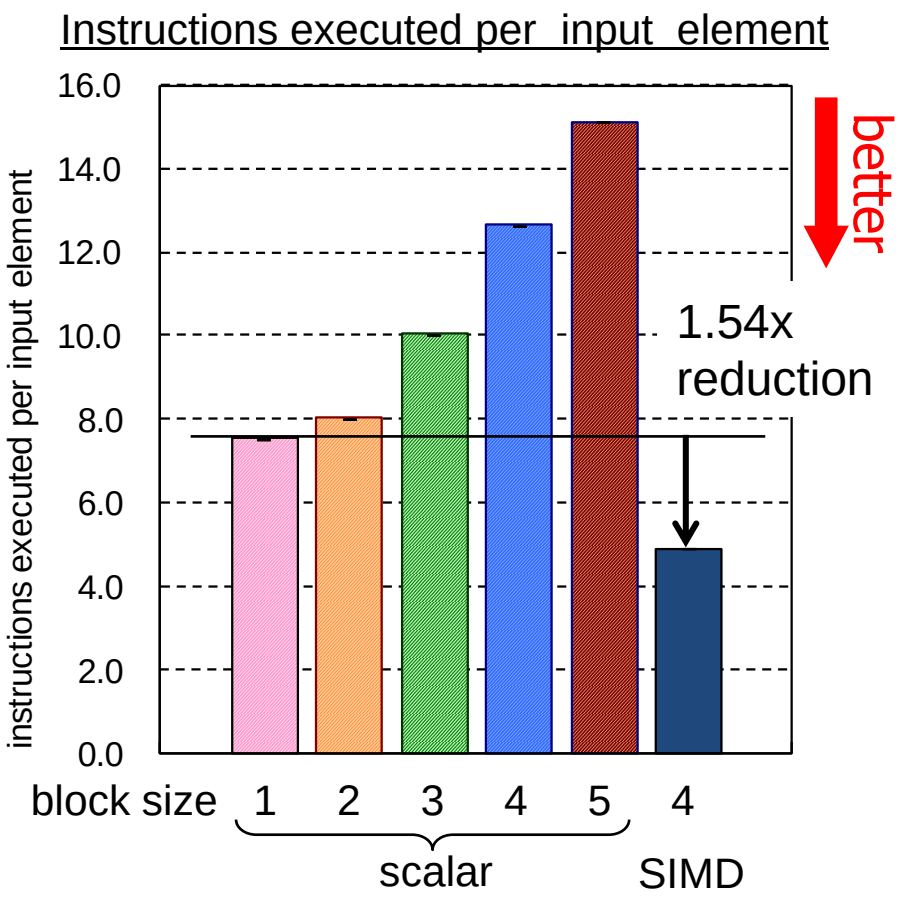
Performance improvements with SIMD instructions



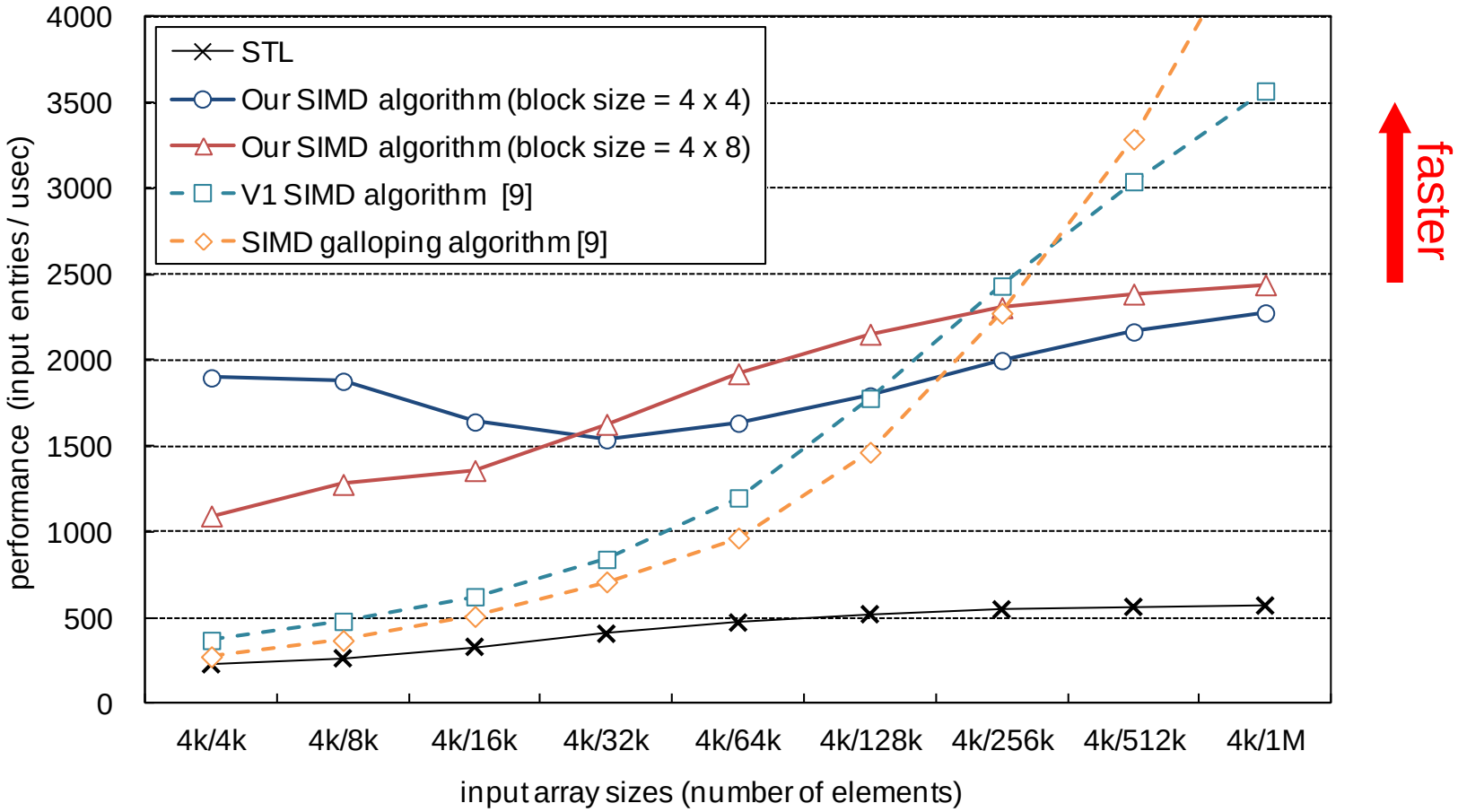
intersecting two 256k random 32-bit integers, output / input = 0%

Numbers of branch mispredictions and instructions



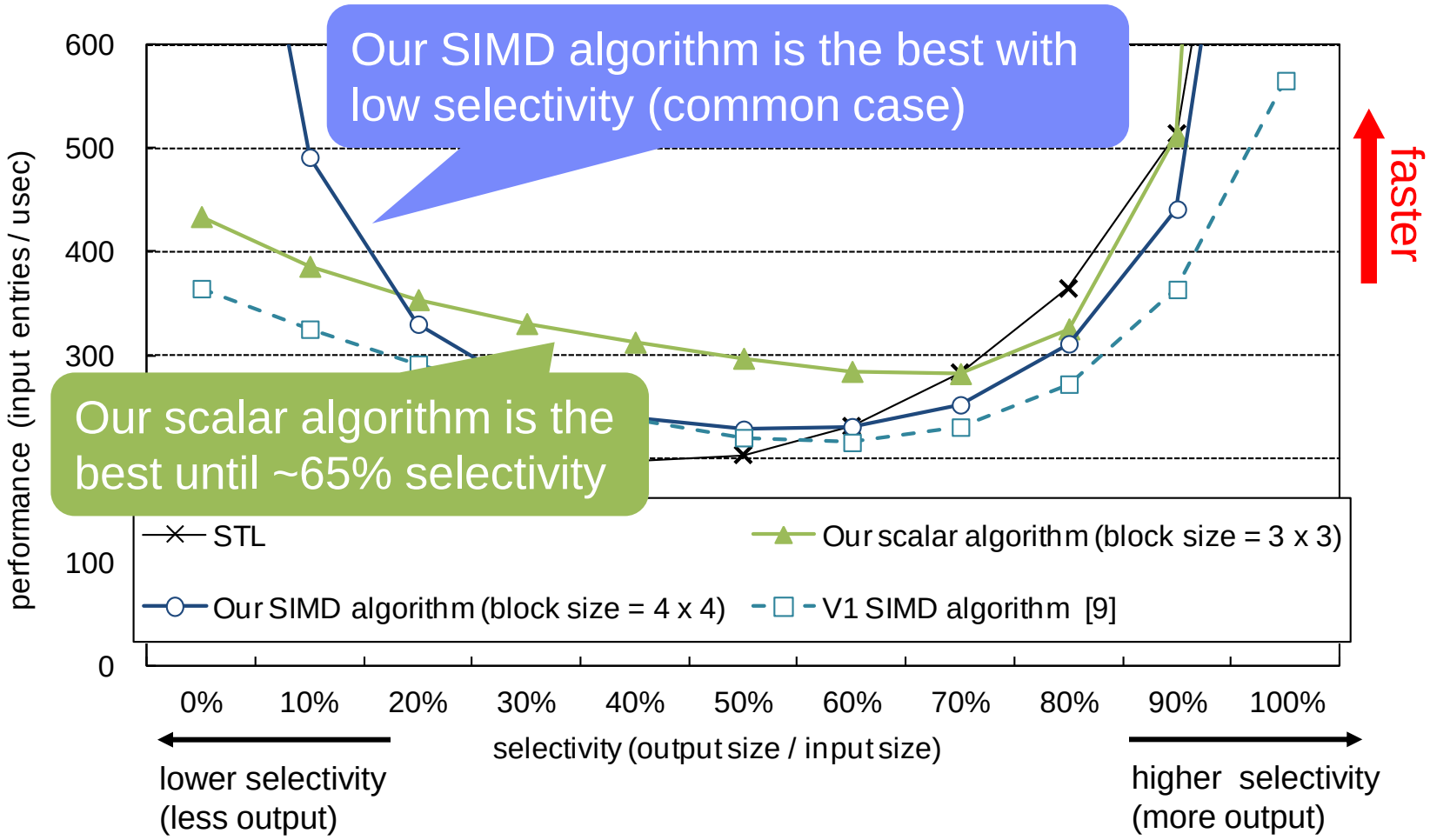


Performance for arrays with different sizes



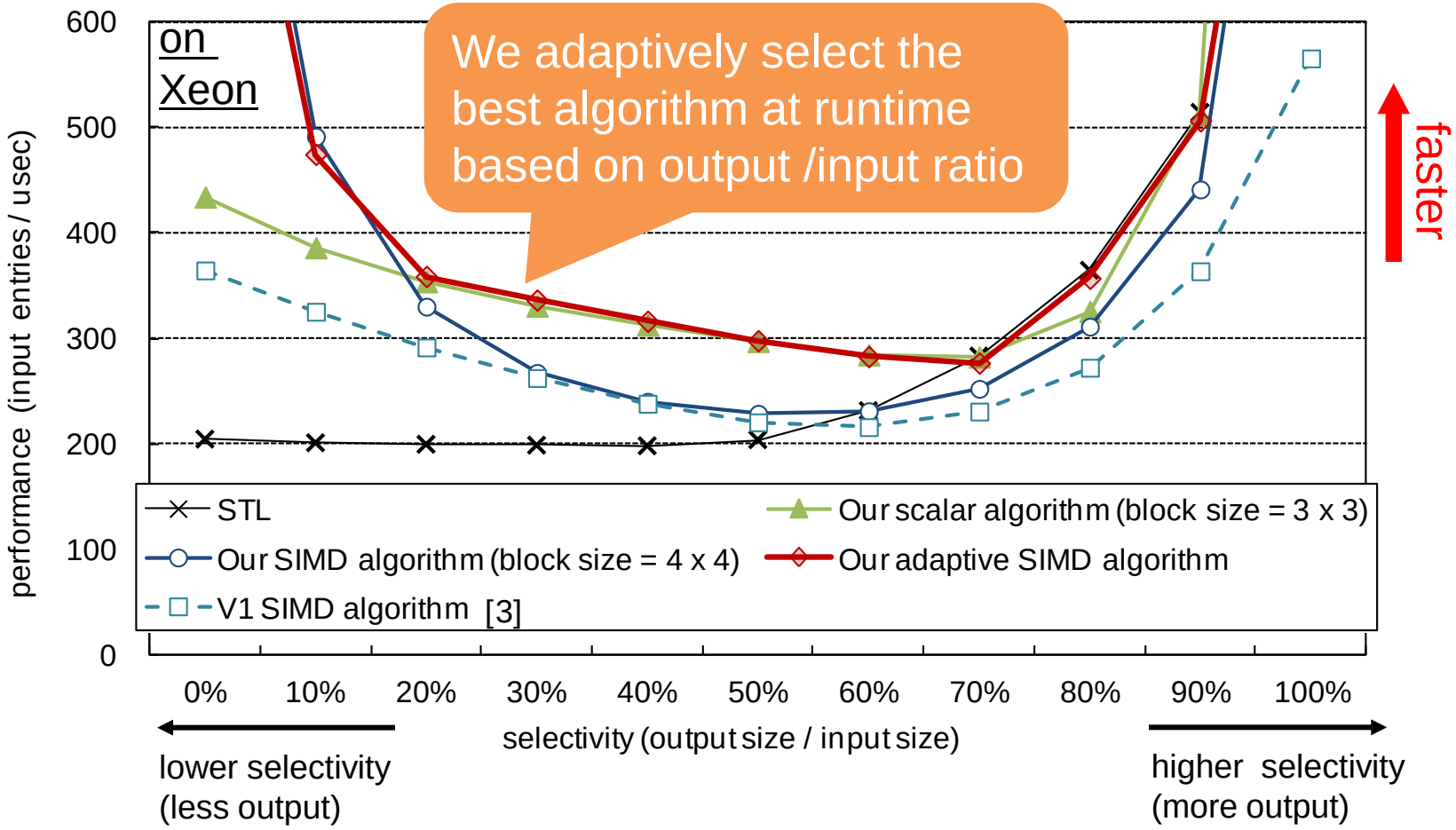
intersecting two random 32-bit integer arrays

Adaptive fallback to avoid pathological degradations



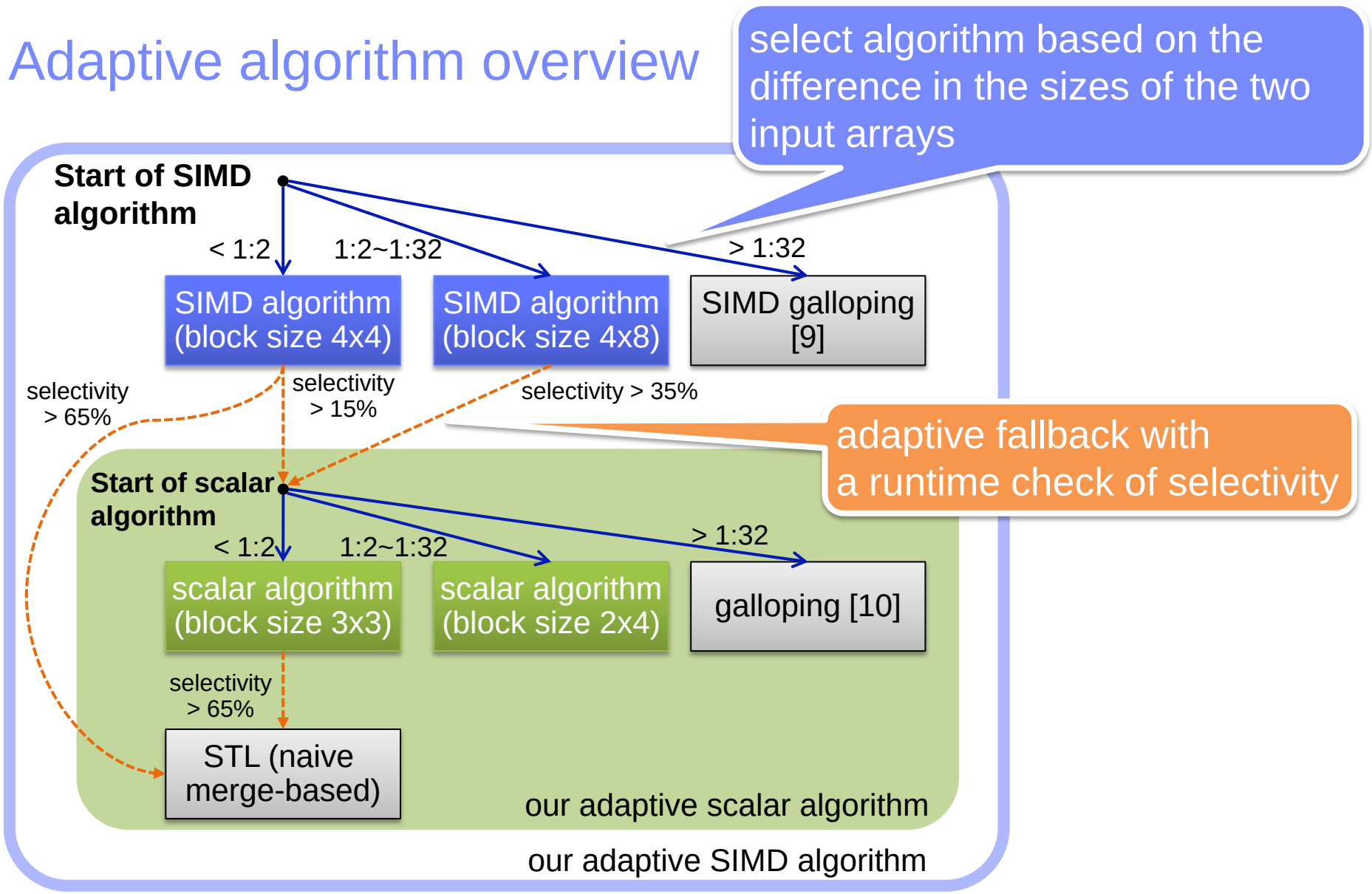
intersecting two 256k random 32-bit integers

Adaptive fallback to avoid pathological degradations

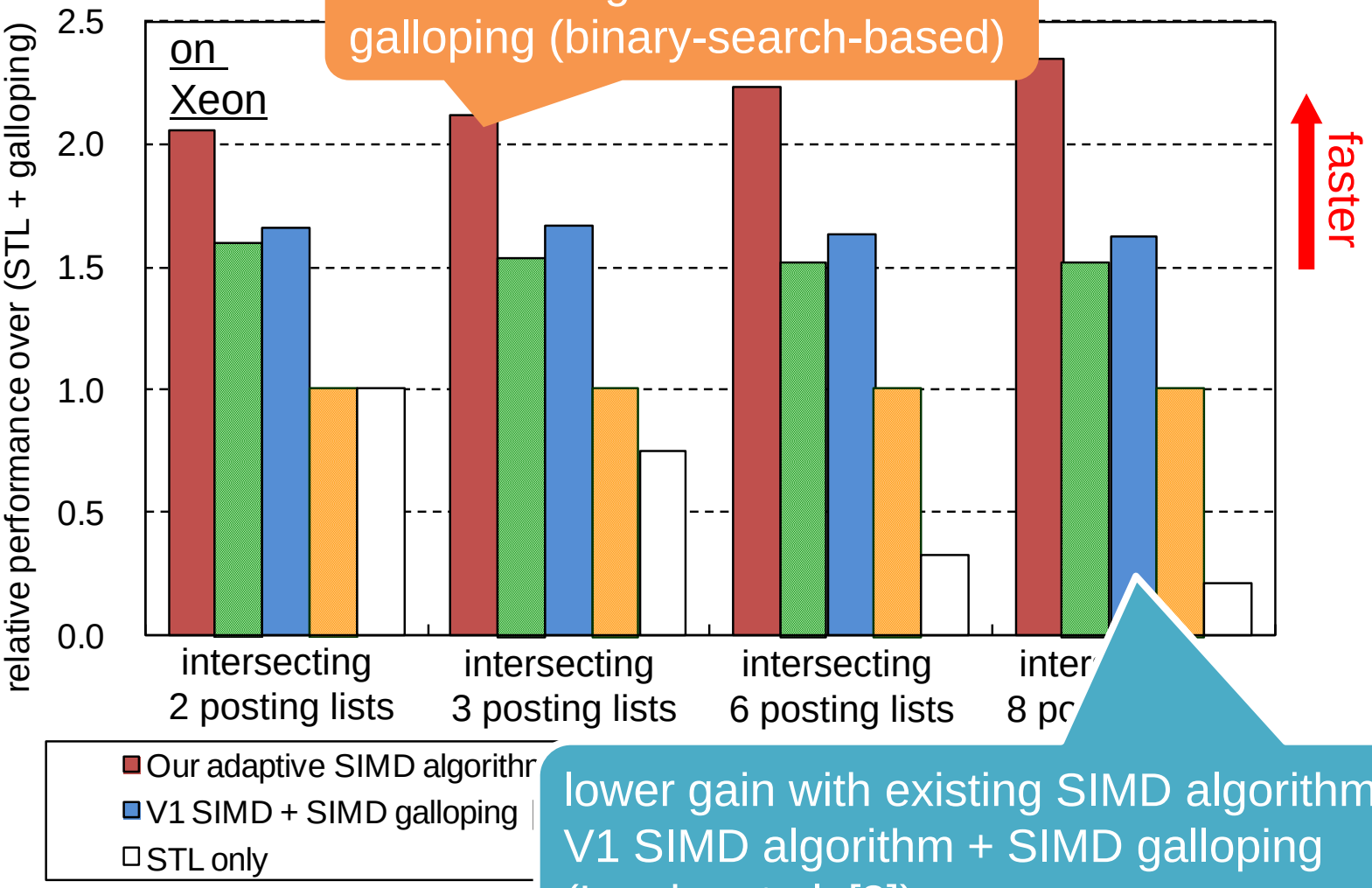


Our SIMD algorithm ➔ Our non-SIMD algorithm ➔ Naive algorithm

Adaptive algorithm overview



Performance with realistic dataset (multiword queries in Wikipedia)



Summary

- We proposed a new set intersection algorithm which is efficient on today's processors
  - by reducing branch mispredictions
  - by avoiding redundant comparisons using SIMD
- Our new algorithm accelerates set intersection for artificial dataset compared to STL
  - by up to 2x without SIMD
  - by up to 5x using SIMD
- It also achieves better performance in an emulated query serving system
  - by up to 2.3x with SIMD over STL
  - by up to 1.5x over existing SIMD algorithms [Lemire et al. '15]