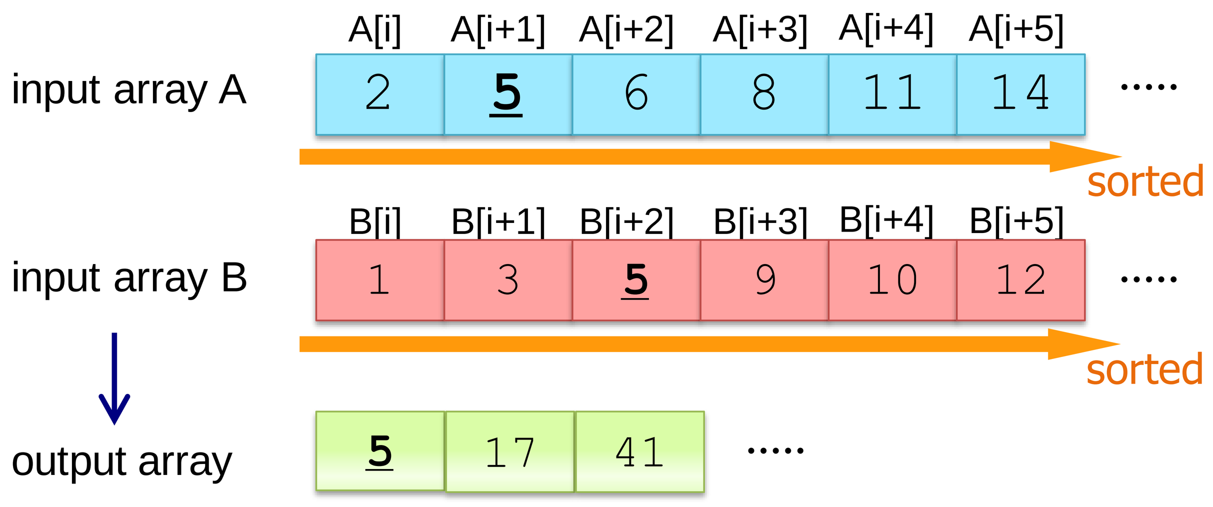
Faster Set Intersection with SIMD Instructions by Reducing Branch Mispredictions

Hiroshi Inoue (inouehrs@jp.ibm.com)† ‡   Moriyoshi Ohara†   Kenjiro Taura‡
†IBM Research - Tokyo   ‡University of Tokyo

Introduction

- Set intersection is the operation to find common elements from two sets; we cover intersecting two sorted integer arrays in this work



- Heavily used in DBMS (merge join) or in search engines (multi-word AND query)

Key observation

not so costly; easy to predict (mostly not taken)

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

costly due to frequent branch mispredictions
➔ we focus on reducing this branch

In a merge-based implementation above,

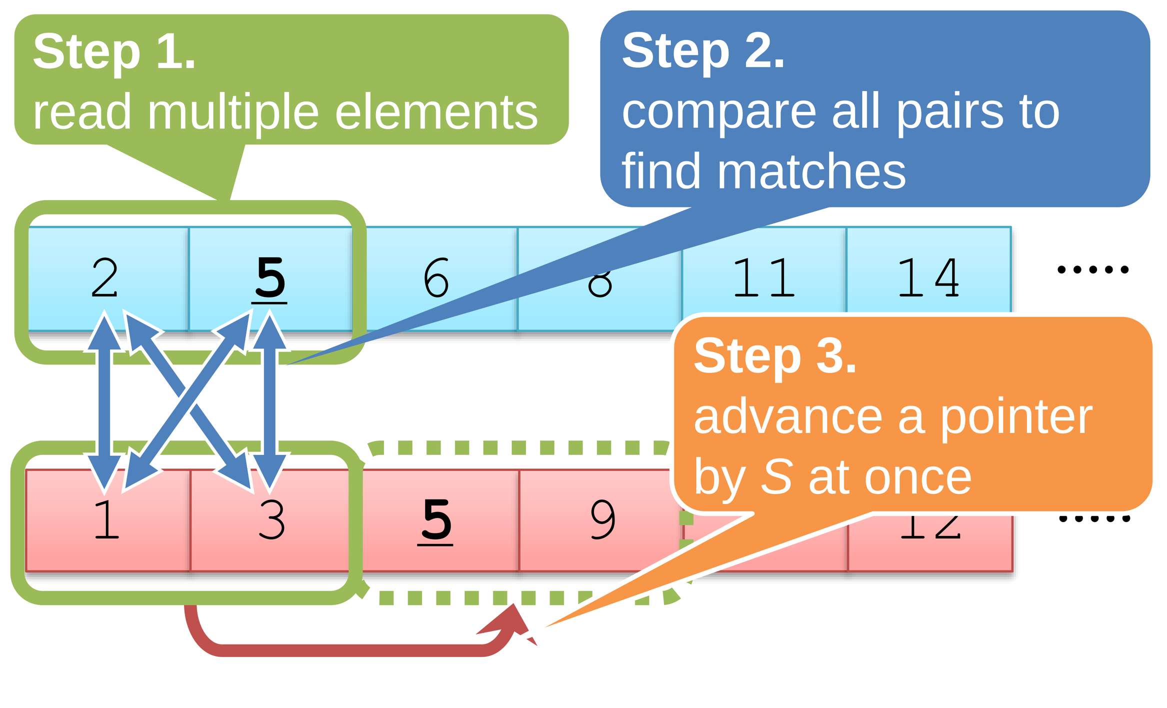
✓The comparison to select an input array for the next block is hard to predict for branch predictor and costly due to misprediction overhead

✓The comparison to check equality is much easier to predict and not so costly (assuming the number of output is much smaller than the input)

➔We focus on reducing the hard-to-predict conditional branches

Our block-based approach

1. We read multiple elements (block size S, here S=2), instead of just one element, from each of the two input arrays,
2. compare all of the pairs of elements from the two arrays to find any matching pairs,
3. then increment a pointer by S, instead of one



☺ reduce hard-to-predict branches to only 1/S (one comparison for each S elements in step 3)

☹ increase easy-to-predict branches by S times (S^2 comparisons in step 2)

➔ We observed about 2x gain with S = 3 or 4 even without using SIMD instructions

Our block-based approach

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

```
while (pA < pAend-1 && pB < 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

- A simple cost model to determine the best 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 can be determined based on $r = cost_{misp} / cost_{exec}$

S_{best} = 1 when $r \le 2$
S_{best} = 2 when $2 \le r \le 10$
S_{best} = 3 when $10 \le r \le 22$
S_{best} = 4 when $22 \le r \le 38$

S_{best} = 3 for many of today's processors

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

Exploiting SIMD instructions

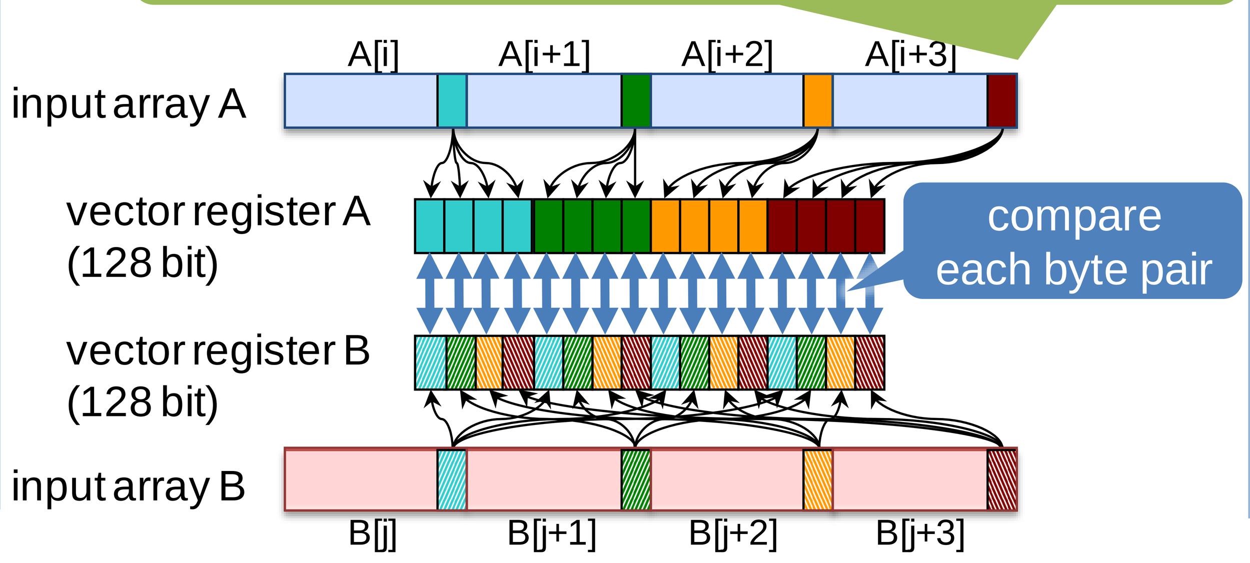
- In our block-based approach, the larger number of comparisons from these all-pairs comparisons is an obvious drawback

➔We use SIMD instructions to reduce these comparisons as follow

1. read only a part of each element and pack them into a vector register
2. compare them by SIMD comparison (partial comparison)
3. if no matching pair found, skip further comparisons for this block (common case)
4. execute full comparisons to find matching pairs (or repeat a partial comparison with a different part of each key)

- This partial comparison approach can yield higher data parallelism than comparing the entire key

pack only a part (e.g. least significant one byte) from elements into a vector register

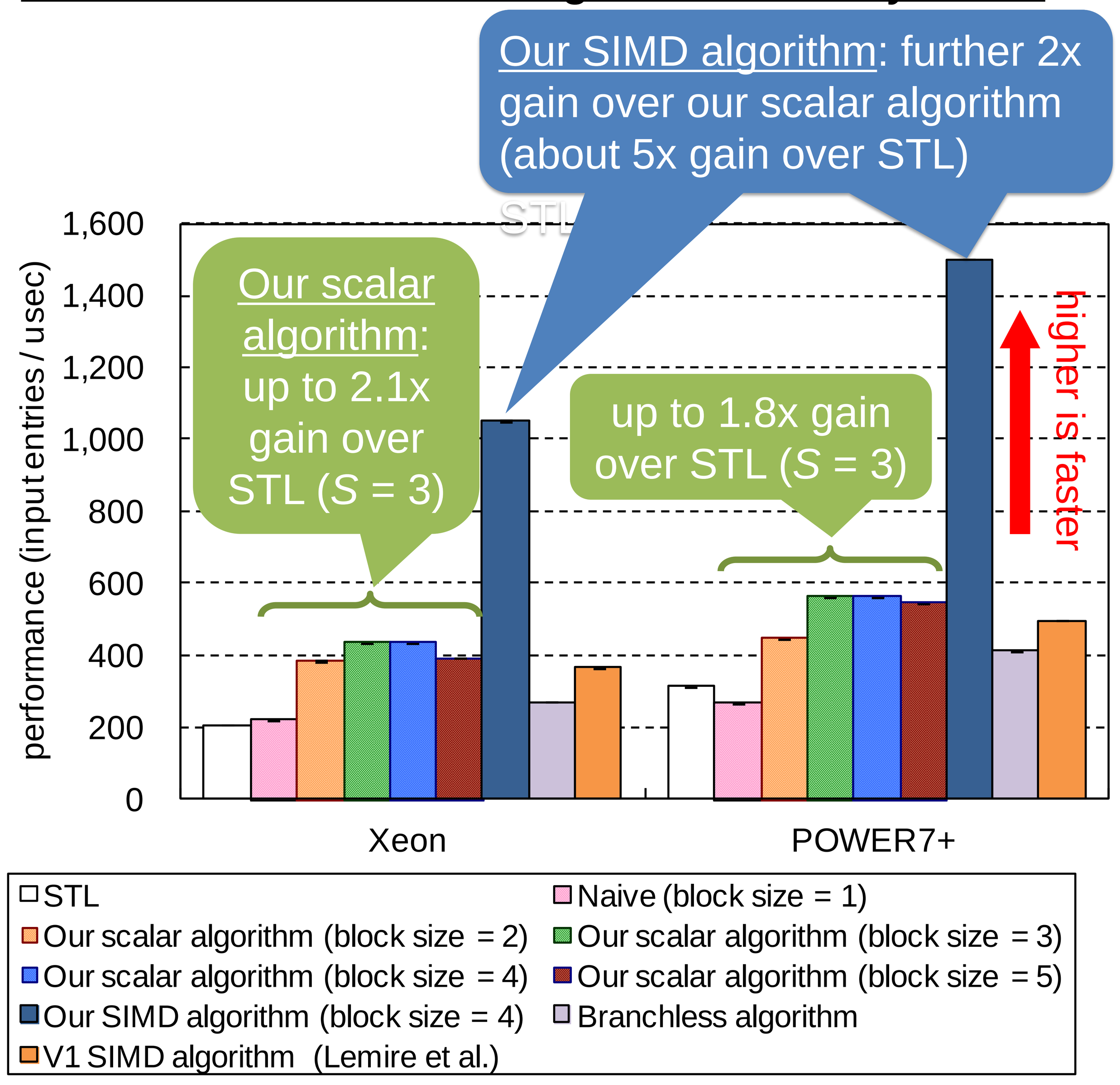


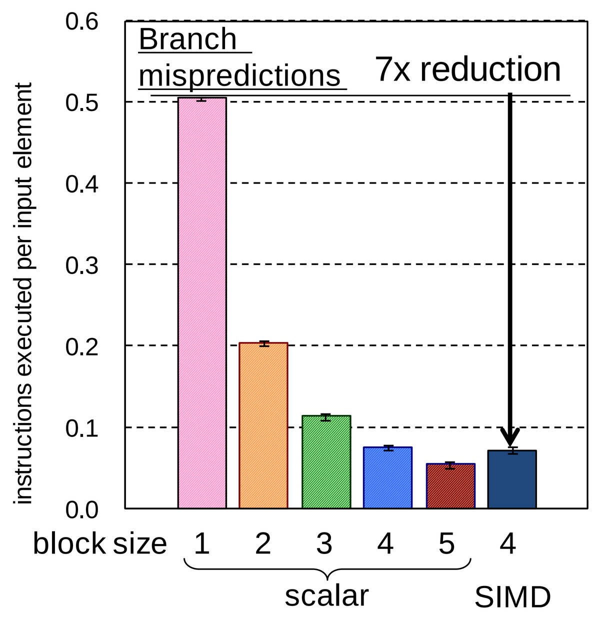
Performance results

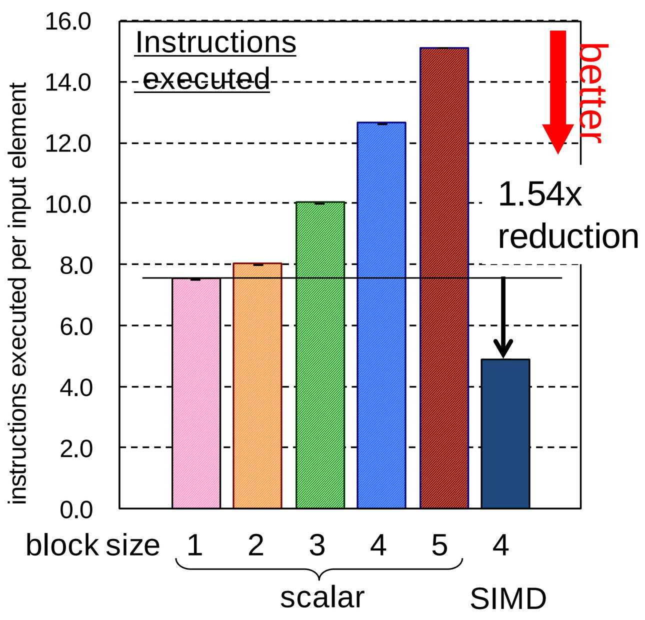
Evaluation with artificial dataset

Systems
- 2.9-GHz Xeon (SandyBridge) or 4.1-GHz POWER7+ / RHEL 6.4 / gcc-4.8
- using 128-bit SIMD (SSE or VSX)

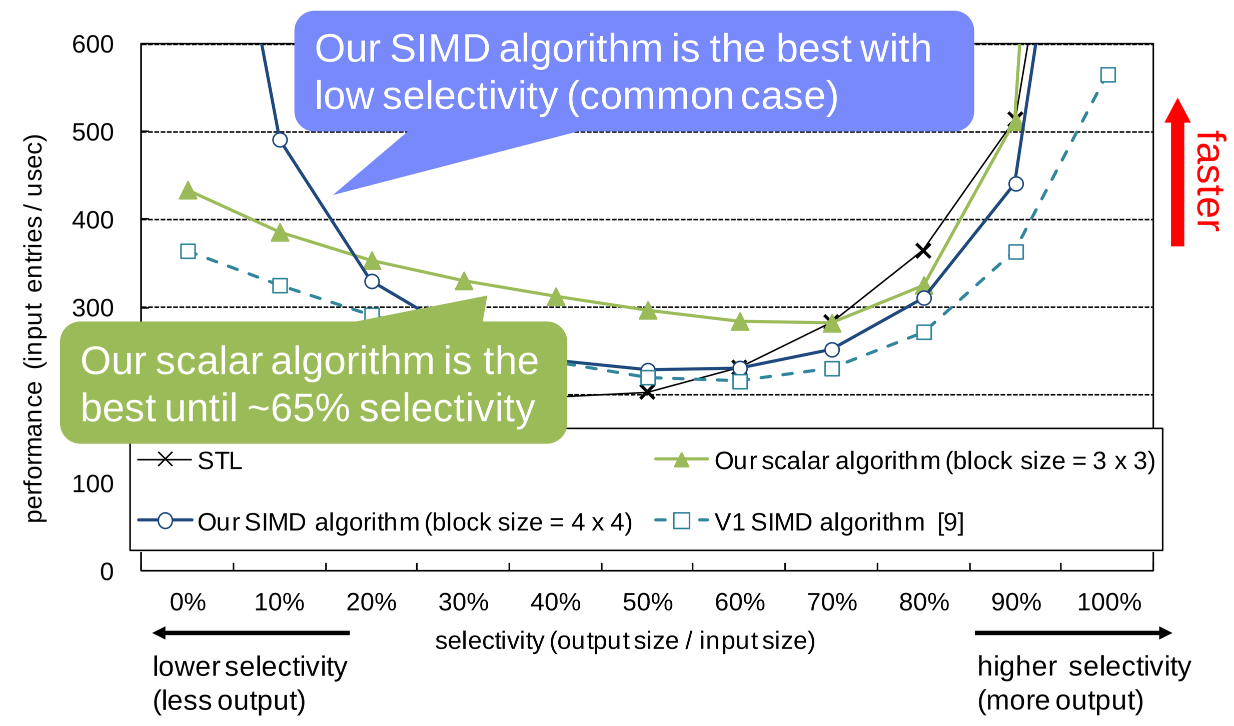
256k random 32-bit integers, selectivity = 0%







256k random integers, various selectivity



Evaluation with more realistic dataset

- emulated multi-word query from Wikipedia with a different number of words in a query
- each algorithm is combined with galloping (if the sizes of two sets are very different)