SimdFSM: An Adaptive Vectorization of Finite State Machines for Speculative Execution (Supplementary Material)

Le Li[1,2][0000−0002−7334−1602] and Kenjiro Taura[1][0000−0001−5224−382X]

[1] The University of Tokyo, Bunkyo, Tokyo, Japan
[2] lile@eidos.ic.i.u-tokyo.ac.jp

1 Introduction

This document details the following information that are not practical to include in the manuscript

- Implementation of several methods, including instructions that cannot be precisely introduced in the manuscript
- Large tables that involve the profiling results under some other microarchitectures (Haswell, Broadwell, etc.), number of speculated states ($|\mathcal{K}|$), and range of FSM size ($|Q|$)

2 Implementation

This section shows the implementation of several methods provided by SimdFSM. Different versions (vector length, indexing type, etc.) in each category (gather, shuffle, permute, igather) share the same computation logic but may differ in the following low-level details due to what the vector extension supports,

- Instructions used for the merge step in shuffle and permute (e.g., 128- and 512-bit shuffle)
- Necessary type casting, e.g., `_mm_loadu_si128 (__m128i const* mem_addr)` and `_mm512_loadu_si512 (void const* mem_addr)`

The code is implemented based on the following assumption,

- In the transition table $\mathcal{T}$, the first index c is the next character and a row of the table $\mathcal{T}[c]$ is a one-dimensional array indexed by the current state s, i.e., $\mathcal{T}[c][s]$ is the next state.

2.1 gather (32-bit indexing)

```
#if defined(__AVX512F__)
#define I32GATHER _mm512_i32gather_epi32
#define VECTOR_T __m512i
#elif defined(__AVX2__)
#define I32GATHER _mm256_i32gather_epi32
#define VECTOR_T __m256i
#endif

#if defined(__AVX512F__) || defined(__AVX2__)
/**
 * @param (vstates) vector packing all speculated states
 * @param (tablerow) a row in the transition table
 */
VECTOR_T inline gather(VECTOR_T vstates, uint32_t const* tablerow) {
  return I32GATHER(vstates, tablerow);
}
#endif
```

2.2 128-bit shuffle (8-bit indexing, 16 states per vector)

Instruction `_mm_blendv_epi8 (__m128i a, __m128i b, __m128i mask)` selects values in `a` or `b` according to the most significant bit (MSB) in `mask` ; if the MSB of `mask[i]` is 0, the *i*-th slot of the resulting vector is `a[i]` ; otherwise, `b[i]` . This is why we need to subtract some values (Line 13) into negative integers (the MSBs are 1s).

```
1  #if defined(__SSE4_1__)
2  /**
3   * @tparam (NUM_STATES) number of states of the FSM
4   * @param (vstates) vector packing all speculated states
5   * @param (tablerow) a row in the transition table
6   */
7  template <size_t NUM_STATES>
8  __m128i inline shuffle128(__m128i vstates, uint8_t const* tablerow) {
9    __m128i vlane = _mm_loadu_si128((__m128i*)tablerow); // the first 128-bit lane in <tablerow>
10   __m128i vnexts = _mm_shuffle_epi8(vlane, vstates); // the first shuffle
11   for (size_t i=16; i<NUM_STATES; i+=16) { // remaining part of <tablerow>
12     vlane = _mm_loadu_si128((__m128i*)(tablerow + i)); // the next 128-bit lane in <tablerow>
13     vstates = _mm_sub_epi8(vstates, _mm_set1_epi8(16)); // used to control the merge
14     __m128i vnexts2 = _mm_shuffle_epi8(vlane, vstates); // the next shuffle
15     vnexts = _mm_blendv_epi8(vnexts2, vnexts, vstates); // merge the two shuffles
16   }
17   return vnexts;
18 }
19 #endif
```

2.3 512-bit shuffle (8-bit indexing, 64 states per vector)

The difference to the 128-bit version is that it additionally calculates a mask to control the merge step (Line 15-16).

```
1  #if defined(__AVX512VBMI__)
2  /**
3   * @tparam (NUM_STATES) number of states of the FSM
4   * @param (vstates) vector packing all speculated states
5   * @param (tablerow) a row in the transition table
6   */
7  template <size_t NUM_STATES>
8  __m512i inline shuffle512(__m512i vstates, uint8_t const* tablerow) {
9    __m512i vlane = _mm512_loadu_si512(tablerow); // the first 512-bit lane in <tablerow>
10   __m512i vnexts = _mm512_permutexvar_epi8(vlane, vstates); // the first shuffle
11   for (size_t i=64; i<NUM_STATES; i+=64) { // remaining part of <tablerow>
12     vlane = _mm512_loadu_si512(tablerow + i); // the next 512-bit lane in <tablerow>
13     vstates = _mm512_sub_epi8(vstates, _mm512_set1_epi8(64)); // used to calculate <msk>
14     __m512i vnexts2 = _mm512_permutexvar_epi8(vlane, vstates); // the next shuffle
15     __mmask64 msk = _mm512_cmplt_epi8_mask(vstates, _mm512_set1_epi8(0)); // merge control mask
16     vnexts = _mm512_mask_blend_epi8(msk, vnexts, vnexts2); // merge the two shuffles
17   }
18   return vnexts;
19 }
20 #endif
```

Note that we use cross-lane shuffle `_mm512_permutexvar_epi8(__m512i idx, __m512i val)` instead of `_mm512_shuffle_epi8 (__m512i val, __m512i idx)` , which shuffles within 128-lanes only, as follows,

```
[test]$ cat shuffle512.cpp
#include <x86intrin.h>
#include <stdio.h>
#include <stdint.h>

int main(){
  __m512i tablerow = _mm512_set_epi8(
    0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
    32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63
  );
  __m512i vstates = _mm512_set1_epi8(2); // current states are 2s -> tablerow[2] = 61
  __m512i shuffle = _mm512_shuffle_epi8(tablerow, vstates); // 16*[2], 16*[2+16], 16*[2+32], 16*[2+48]
  printf("shuffle ");
  for (size_t i=0; i<64; i++)
    printf("%d ", ((uint8_t*)&shuffle)[i]);
  printf("\n");
  __m512i permutexvar = _mm512_permutexvar_epi8(vstates, tablerow); // 64*[2]
  printf("permutexvar ");
  for (size_t i=0; i<64; i++)
    printf("%d ", ((uint8_t*)&permutexvar)[i]);
  printf("\n");
}
[test]$ g++ -mavx512vbmi shuffle512.cpp
[test]$ ./a.out
shuffle 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 29
    29 29 29 29 29 29 29 29 29 29 29 29 29 29 29 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13
permutexvar 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
    61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
```

2.4 512-bit permute (8-bit indexing, 64 states per vector)

Note that the used permute instruction `_mm512_permutex2var_epi8 (__m512i a, __m512i idx, __m512i b)` is different from `_mm512_permutexvar_epi8(__m512i idx, __m512i val)` used in the 512-bit shuffle. The latter functions as cross-lane shuffle, while the former takes two register operands to extract values from. Therefore, we can deduce the times of merge based on the number of vectors needed for packing the entire row of the transition function,

- If it is even (2/4/6/8/...), we need fewer merge steps (0/1/2/3/...) than shuffle (1/3/5/7/...).
- While, if it is odd ($n = 3/5/7/...$), we permute every two of the first $n - 1$ vectors and shuffle the last one (Line 21-27), which needs $\lfloor n/2 \rfloor = 1/2/3/...$ merge steps.

```
 1  #if defined(__AVX512VBMI__)
 2  /**
 3   * @tparam (NUM_STATES) number of states of the FSM
 4   * @param (vstates) vector packing all speculated states
 5   * @param (tablerow) a row in the transition table
 6   */
 7  template <size_t NUM_STATES>
 8  void inline permute512(__m512i vstates, uint8_t const* tablerow) {
 9    assert(NUM_STATES > 64 && "only need one vector to pack an entire row (use shuffle instead)");
10    __m512i vlane = _mm512_loadu_si512(tablerow); // the first 512-bit lane in <tablerow>
11    __m512i vlane2 = _mm512_loadu_si512(tablerow + 64); // the second 512-bit lane in <tablerow>
12    __m512i vnexts = _mm512_permutex2var_epi8(vlane, vstates, vlane2); // the first permute
13    for (size_t i=64*2; i+64*2<=NUM_STATES; i+=64*2) { // remaining part of <tablerow>
14      vlane = _mm512_loadu_si512(tablerow + i); // the n-th lane (3,5,7,...)
15      vlane2 = _mm512_loadu_si512(tablerow + i + 64); // the (n+1)-th lane (4,6,8,...)
16      vstates = _mm512_sub_epi8(vstates, _mm512_set1_epi8(128)); // used to calculate <msk>
17      __m512i vnexts2 = _mm512_permutex2var_epi8(vlane, vstates, vlane2); // the next permute
18      __mmask64 msk = _mm512_cmplt_epi8_mask(vstates, _mm512_set1_epi8(0)); // merge control mask
19      vnexts = _mm512_mask_blend_epi8(msk, vnexts2, vnexts); // merge the two permutes
20    }
21    if constexpr(NUM_STATES / 64 % 2) { // if the number of 512-bit lanes is odd, shuffle the last one
22      vlane = _mm512_loadu_si512(tablerow + NUM_STATES / 64 * 64); // the last lane in <tablerow>
23      vstates = _mm512_sub_epi8(vstates, _mm512_set1_epi8(128)); // // used to calculate <msk>
24      __m512i vnexts2 = _mm512_permutexvar_epi8(vlane, vstates); // shuffle the last lane
25      __mmask64 msk = _mm512_cmplt_epi8_mask(vstates, _mm512_set1_epi8(0)); // merge control mask
26      vnexts = _mm512_mask_blend_epi8(msk, vnexts2, vnexts); // merge the last permute and shuffle
27    }
28    return vnexts;
29  }
30  #endif
```

2.5 512-bit igather (32-bit indexing, 16 states per vector)

We show the optimized implementation that replaces `_mm512_i32gather_epi32` with `_mm512_set_epi32` for collecting input symbols (Line 20-25), which avoids the input conversion overhead and reduces memory capacity pressure. It is efficient because the indices ($I/K*B$, where I is the index of the slot) in the input sequence (variable `data`) can be determined during compile time. Note that we need to ensure that Line 26 and 27 do not cause overflow by, for example, satisfying the following requirements,

- Line 26: $|Q| \le 2^{31}/2^{7} = 2^{24} = 16777216$
- Line 27: $|Q| \le 2^{31}/(1 + 2^{7}) = 16647160$

where 2^{7} is the 8-bit lane for encoding input symbols under ASCII ($0 \sim 127$).

```
 1  #define B 8196 // BLOCK SIZE
 2  
 3  #if defined(__AVX512F__)
 4  /**
 5   * @tparam (NUM_STATES) number of states of the FSM
 6   * @tparam (K) number of speculated states
 7   * @param (vstates) vector packing all speculated states
 8   * @param (table) the entire transition table
 9   * @param (data) pointer to input + offset + iteration
10   */
11  template <size_t NUM_STATES, size_t K>
12  __m512i inline igather512(__m512i vstates, uint32_t const (*table)[NUM_STATES], char const* data) {
13    // for example, if K == 4, it is 4-way (each way has K=4 speculated states), as follows,
14    // __m512i vsymbols = _mm512_set_epi32(
15    //  data[3*B], data[3*B], data[3*B], data[3*B],      (the 4th way reads data[3*B])
16    //  data[2*B], data[2*B], data[2*B], data[2*B],      (the 3rd way reads data[2*B])
17    //  data[B], data[B], data[B], data[B],              (the 2nd way reads data[B])
18    //  data[0], data[0], data[0], data[0],              (the 1st way reads data[0])
19    // );
20    __m512i vsymbols = _mm512_set_epi32(
21      data[15/K*B], data[14/K*B], data[13/K*B], data[12/K*B],
22      data[11/K*B], data[10/K*B], data[9/K*B], data[8/K*B],
23      data[7/K*B], data[6/K*B], data[5/K*B], data[4/K*B],
24      data[3/K*B], data[2/K*B], data[1/K*B], data[0/K*B],
25    );
26    __m512i vrows = _mm512_mullo_epi32(vsymbols, _mm512_set1_epi32(NUM_STATES)); // row-entry table[symbol]
27    __m512i vrowcols = _mm512_add_epi32(vstates, vrows); // column-entry table[symbol][state]
28    return _mm512_i32gather_epi32(vrowcols, table); // perform vectorized transition
29  }
30  
31  /**
32   * @tparam (NUM_STATES) number of states of the FSM
33   * @tparam (K) number of speculated states
34   * @tparam (N) number of ways
35   * @param (table) the entire transition table
36   * @param (input) pointer to the input sequence
37   * @param (len) length of the input
38   */
39  template <size_t NUM_STATES, size_t K, size_t N>
40  __m512i run(void const *table, char const* input, size_t const len) {
41    uint32_t (*_table)[NUM_STATES] = (uint32_t(*)[NUM_STATES])table;
42    __m512i vstates = ... // init the vector that packs speculated states
43    for (size_t offset=0; offset+N*B<len; offset+=N*B) {
44      for (size_t i=0; i<B; i++) {
45        vstates = igather512<NUM_STATES, K>(vstates, _table, input + offset + i);
46      }
47      ... // reduction (chaining neighboring ways)
48    }
49    return vstates;
50  }
51  #endif
```

2.6 The Adaptive Strategy

```
/**
 * @param (T) reciprocal throughput (cycles per byte) of the method
 * @param (R) speculation success rate
 * @param (T_serial) reciprocal throughput (cycles per byte) of the serial execution
 */
double inline model(double T, double R, double T_serial) {
  return T + (1 - R) * T_serial;
}

// function to call the method (void const* table, char const* input, size_t const len)
typedef void(*F)(void const* table, char const*, size_t const);
struct Optimal {
  F method; // the chosen method
  vector<int> specStates; // organize states in discerning order of frequency
  int k = 0; // use the first <k> states in <specStates> for speculation
};

/**
 * @tparam (NUM_STATES) number of states of the FSM
 * @param (stateCount) state frequency sampling
 * @param (totalCount) sum of elements in <stateCount>
 * @param (T_serial) reciprocal throughput (cycles per byte) of the serial execution
 * @param[in] (optimal) any
 * @param[out] (optimal) chosen method, states ordered in frequency, and number of speculated states
 */
template <size_t NUM_STATES>
void adapt(int const* stateCount, size_t const totalCount, double const T_serial, Optimal& optimal) {
  optimal.specStates.clear();
  optimal.k = 0;

  // sort states by frequency using max heap
  typedef pair<int, int> P; // <frequency, state>
  priority_queue <P, vector<P> > pq;
  for (size_t s=0; s<NUM_STATES; s++)
    pq.push(make_pair(stateCount[s], s));

  // choose the optimal method and the set of speculated states
  vector<int>& specStates = optimal.specStates;
  int k = 0; // speculate <k> states
  int freq = 0; // sum of the frequencies of the states stored in vector <specStates>
  double T_optimal = DBL_MAX; // modeled reciprocal throughput of <optimal.method>
  for (auto k2m : profiler) { // the profiling result of each method <m> when speculating <k> states
    for (; k<k2m.first && !pq.empty(); k++) { // calculate frequency CDF of the top <k> frequent states
      P freq2state = pq.top(); pq.pop();
      freq += freq2state.first;
      specStates.emplace_back(freq2state.second);
    }
    for (auto m : k2m.second) { // predict the performances of methods and update if finding a better one
      double predicted = model(m.T, (double)freq / totalCount, T_serial);
      if (predicted < T_optimal) {
        k = k2m.first;
        T_optimal = predicted;
        optimal.method = m.funcion;
        optimal.k = k;
      }
    }
  }
  assert(optimal.k > 0 && "illegal k");
}
```

3 Profiling Data

This section documents the data profiled for several methods under the following microarchitectures

- Haswell — Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
- Broadwell — Intel(R) Xeon(R) CPU E7-8890 v4 @ 2.20GHz
- SkyLake-X — Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz
- Ice Lake — Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz

Note that we adopt the same notations $M\_L/I(K)$ and igather_$L/I(C, K)$ as used in the manuscript itself, where M is the method name, L is the length of the vector, I is the indexing type, K is the number of speculated states ($K = L/I$), and C is the number of simultaneously processed chunks (the number of ways). In order to place the data (cycles per byte) as succinct as possible, we organize them as follows,

- **baseline**: The average values are calculated based on the type of state encoding, e.g., 8- and 16-bit integers when $|Q| = [1, 256]$ and $[257, 1024]$, respectively.
- **shuffle & permute**: The average values are calculated based on the number of merge steps; for example, shuffle_128/8(16) needs 0 and 1 merge steps if $|Q| \in [1, 16]$ *and* $[17, 32]$, respectively, i.e., the interval of $|Q|$ for average calculation is 16. On the contrary, that of shuffle_512/8(64) is 64.
- **gather & igather**: The average values are calculated across the entire range of $|Q|$ for not observing significant fluctuation, since they need no merge step and encode states with 32-bit integer in all cases (we do not consider the case when $|Q| \geq 2^{31}$).

3.1 Haswell — Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz

$\|Q\|$	**baseline** $\|\mathcal{K}\|$ = 2/4/8/16	SSE4.1 **shuffle_128** $\|\mathcal{K}\|$ = 16	AVX2 **igather_256/32** $\|\mathcal{K}\|$ = 2/4/8
1 ~ 16	5.00 / 5.04 / 6.21 / 14.64	2.12	5.44 / 9.31 / 16.53
17 ~ 32		4.07	
33 ~ 48		7.06	
49 ~ 64		10.03	
65 ~ 80		13.07	
81 ~ 96		16.04	
97 ~ 112		19.05	
113 ~ 128		22.03	
129 ~ 144		25.02	
145 ~ 160		28.05	
161 ~ 176		31.13	
177 ~ 192		34.15	
193 ~ 208		37.18	
209 ~ 224		40.16	
225 ~ 240		43.14	
241 ~ 256		46.12	
257 ~ 1024	5.54 / 5.58 / 7.13 / 15.96	N/A [3]	

[3] 16-bit indexing not supported, and 32-bit indexing is too inefficient

3.2 Broadwell — Intel(R) Xeon(R) CPU E7-8890 v4 @ 2.20GHz

$\lvert Q\rvert$		SSE4.1	AVX2
	baseline	**shuffle_128**	**igather_256/32**
	$\lvert\mathcal{K}\rvert = 2/4/8/16$	$\lvert\mathcal{K}\rvert = 16$	$\lvert\mathcal{K}\rvert = 2/4/8$
1 ~ 16	5.00 / 5.03 / 6.04 / 14.59	2.07	4.20 / 6.20 / 10.75 [4]
17 ~ 32		4.07	
33 ~ 48		7.05	
49 ~ 64		10.02	
65 ~ 80		13.04	
81 ~ 96		16.05	
97 ~ 112		19.04	
113 ~ 128		22.02	
129 ~ 144		25.02	
145 ~ 160		28.07	
161 ~ 176		31.09	
177 ~ 192		34.11	
193 ~ 208		37.16	
209 ~ 224		40.14	
225 ~ 240		43.13	
241 ~ 256		46.12	
257 ~ 1024	5.55 / 5.59 / 7.11 / 15.85	N/A [5]	

3.3 SkyLake-X — Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

$\lvert Q\rvert$		SSE4.1	AVX512BW	AVX512BW	AVX512F
	baseline	**shuffle_128**	**shuffle_512/16**	**permute_512/16**	**igather_512/32**
	$\lvert\mathcal{K}\rvert = 2/4/8/16/32$	$\lvert\mathcal{K}\rvert = 16$	$\lvert\mathcal{K}\rvert = 32$	$\lvert\mathcal{K}\rvert = 32$	$\lvert\mathcal{K}\rvert = 2/4/8$
1 ~ 16	5.00 / 5.03 / 5.38 / 14.23 / 37.73	2.69	6.02	N/A [6]	3.98 / 5.18 / 7.84
17 ~ 32		4.04			
33 ~ 48		5.04	12.97	6.05	
49 ~ 64		6.06			
65 ~ 80		7.45	16.02	10.27	
81 ~ 96		8.74			
97 ~ 112		10.57	18.86	11.02	
113 ~ 128		11.09			
129 ~ 144		12.86	22.02	14.51	
145 ~ 160		13.82			
161 ~ 176		16.33	25.02	15.40	
177 ~ 192		16.43			
193 ~ 208		19.53	28.06	18.82	
209 ~ 224		19.64			
225 ~ 240		20.43	30.98	18.49	
241 ~ 256		21.14			
257 ~ 1024	5.63 / 5.67 / 6.34 / 14.82 / 34.00	N/A [7]	34.12 ~ 126.63	23.15 ~ 106.86	

[4] more efficient than those on Haswell
[5] 16-bit indexing not supported, and 32-bit indexing is too inefficient
[6] the number of vectors needed for packing an entire row of the transition table is less than 2 (each vector has 512/16 = 32 slots)
[7] the 16-bit indexing version provided in AVX512BW is much worse than shuffle_512/16

3.4 Ice Lake — Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz

$\|Q\|$	**baseline/8** $\|\mathcal{K}\|$ = 2/4/8/16/32/64	SSE4.1 **shuffle_128/8** $\|\mathcal{K}\|$ = 16	AVX512VBMI **shuffle_512/8** $\|\mathcal{K}\|$ = 64	AVX512VBMI **permute_512/8** $\|\mathcal{K}\|$ = 64	AVX512F **igather_512/32** $\|\mathcal{K}\|$ = 2/4/8
1 ~ 16	5.00 / 5.01 / 5.03 /13.11 / 30.56 / 50.73	1.70	3.77	N/A [8]	3.92 / 4.79 / 8.16
17 ~ 32		3.53			
33 ~ 48		4.84			
49 ~ 64		6.39			
65 ~ 80		7.89	12.21	5.98	
81 ~ 96		9.22			
97 ~ 112		10.47			
113 ~ 128		11.55			
129 ~ 144		12.84	15.21	9.30	
145 ~ 160		13.89			
161 ~ 176		15.49			
177 ~ 192		16.13			
193 ~ 208		18.09	18.24	11.65	
209 ~ 224		19.31			
225 ~ 240		19.56			
241 ~ 256		20.59			

$\|Q\|$	**baseline/16** $\|\mathcal{K}\|$ = 2/4/8/16/32	AVX512BW **shuffle512/16** $\|\mathcal{K}\|$ = 32	AVX512BW **permute512/16** $\|\mathcal{K}\|$ = 32	AVX512F **igather512/32** $\|\mathcal{K}\|$ = 2/4/8
257 ~ 1024	5.42 / 5.43 / 5.51 / 14.16 / 26.09	35.56 ~ 152.70	26.54 ~ 151.55	3.92 / 4.79 / 8.16

[8] the number of vectors needed for packing an entire row of the transition table is less than 2 (each vector has $512/8 = 64$ slots)