Scalable FSM Parallelization via Path Fusion and Higher-Order Speculation

Junqiao Qiu*
junqiaoq@mtu.edu
Michigan Technological University
USA

Xiaofan Sun
xsun042@ucr.edu
University of California, Riverside
USA

Amir Hossein Nodehi Sabet
anode001@ucr.edu
University of California, Riverside
USA

Zhijia Zhao
zhijia@cs.ucr.edu
University of California, Riverside
USA

ABSTRACT

Finite-state machine (FSM) is a fundamental computation model used by many applications. However, FSM execution is known to be "embarrassingly sequential" due to the state dependences among transitions. Existing solutions leverage enumerative or speculative parallelization to break the dependences. However, the efficiency of both parallelization schemes highly depends on the properties of the FSM and its inputs. For those exhibiting unfavorable properties, the former suffers from the overhead of maintaining multiple execution paths, while the latter is bottlenecked by the serial reprocessing among the misspeculation cases. Either way, the FSM parallelization scalability is seriously compromised.

This work addresses the above scalability challenges with two novel techniques. First, for enumerative parallelization, it proposes ***path fusion***. Inspired by the classic NFA to DFA conversion, it maps a vector of states in the original FSM to a new (fused) state. In this way, path fusion can reduce multiple FSM execution paths into a single path, minimizing the overhead of path maintenance. Second, for speculative parallelization, this work introduces ***higher-order speculation*** to avoid the serial reprocessing during validations. This is a generalized speculation model that allows speculated states to be validated *speculatively*. Finally, this work integrates different schemes of FSM parallelization into a framework—`BoostFSM`, which automatically selects the best based on the relevant properties of the FSM. Evaluation using real-world FSMs with diverse characteristics shows that `BoostFSM` can raise the average speedup from 3.1× and 15.4× of the existing speculative and enumerative parallelization schemes, respectively, to 25.8× on a 64-core machine.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; • **Theory of computation** → **Regular languages**; **Parallel algorithms**.

KEYWORDS

Finite State Machine, FSM, Parallelization, Speculation, Scalability

ACM Reference Format:
Junqiao Qiu, Xiaofan Sun, Amir Hossein Nodehi Sabet, and Zhijia Zhao. 2021. Scalable FSM Parallelization via Path Fusion and Higher-Order Speculation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3445814.3446705

1 INTRODUCTION

As a basic computation model, finite-state machine (FSM) embodies many important applications, ranging from intrusion detection [6, 27, 53, 64] and data decoding [26, 52] to motif searching [11, 49], rule mining [62], and textual data analytics [12, 15, 36]. However, the execution of an FSM is known to be "embarrassingly sequential" [5, 67], due to the inherent dependences among state transitions—in each state transition, the current state always depends on the prior state [1]. These state dependences fundamentally limit the performance of FSM-based computations on modern processors, where parallelism plays an increasingly critical role.

To address the inherent dependences in FSM computations, prior works [16, 24, 33, 41, 42, 66, 67] fall into two basic parallelization schemes: (i) *state enumeration* and (ii) *state speculation*. Figure 1-(b) summarizes them. Without loss of generality, assume the input to an FSM (e.g., a binary sequence) is partitioned evenly into two chunks, as shown in Figure 1-(a). Due to the dependences among state transitions, the starting state for the second chunk would be *unknown*, until the first chunk has been processed—the ending state of the first chunk is the starting state of the second chunk. To process the two chunks in parallel, one can choose:

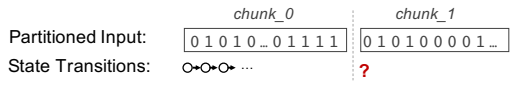
(1) *State Enumeration.* As the unknown starting state must be one of the states in the FSM, we can enumerate all of them by forking an execution path for each state [16, 33], but maintaining all the execution paths may bring significant

*This work was performed when the author was a Ph.D. student at UC Riverside.

[1]Here, it refers to a deterministic FSM; Nondeterministic FSMs can be converted to deterministic ones via a classic conversion algorithm [2].

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446705



(a) State Dependence

	State Enumeration [16, 27]	State Speculation [33, 34, 57, 58]
Dependence Handling	Fork an execution path for each state	Execute speculatively from a predicted state
Issue	Maintaining multiple paths	Sequential validations
Solution (this work)	Path Fusion → Scheme Selection ← Higher-Order Speculation	

(b) Two Basic Parallelization Schemes

Figure 1: FSM Parallelization: Challenge and Schemes

overhead. To reduce it, prior work [33] checks if some paths transition to the same state, known as *path merging*, in which case only one of the merged paths needs to be kept. However, the effectiveness of this approach highly depends on the state convergence property of the FSM. When some of the execution paths exhibit slow convergence or fail to converge, the overhead of this scheme would be high.

(2) *State Speculation.* Instead of considering all the states, one can guess the starting state of the second chunk [41, 42, 66, 67]. To ensure correctness, the predicted state must be validated against the ending state of the prior chunk—the *ground truth*. If the validation fails (i.e., misspeculation), the chunk needs to be reprocessed. However, when the input is partitioned into multiple chunks, the ending state of the prior chunk may not be the ground truth until its own speculation has been validated (with needed reprocessing). These serialized validations form a fundamental scalability bottleneck in the existing speculative FSM parallelization [42].

In addition, a hybrid scheme may choose to enumerate a subset of states [23, 63], which in fact inherits both the advantages and limitations of the above two schemes. In summary, the existing FSM parallelization schemes face fundamental scalability challenges.

This work introduces two novel techniques: *path fusion* and *higher-order speculation*, to address the scalability challenges in the two basic FSM parallelization schemes, respectively. For state enumeration, we propose to fuse different execution paths into a single path. Note that, unlike path merging, path fusion is not based on the state convergence. Instead, its idea stems from the classic NFA to DFA [2] conversion [2]—a way to remove the inefficiency of nondeterministic NFA execution. Rather than mapping a *subset* of NFA states to a DFA state, path fusion encodes a *vector* of states in the original FSM into a *fused state*, based on which it generates a *fused FSM*. Thus, a single execution path of the fused FSM simulates multiple execution paths of the original FSM. In principle, the fused FSM could be much larger than the original. To address this, we also propose to *dynamically* generate a partial fused FSM which captures the states and transitions only for the current input to minimize the memory requirement.

For state speculation, to address the bottleneck of sequential validations, we introduce the concept of *speculation order* and show that the existing FSM speculation solutions are in fact of the first order. By raising the speculation to *higher orders*, we find not only that the validations can be naturally parallelized, because chunks of higher-order speculation no longer need to wait for the ground truth, but also that the speculation accuracy might get improved—the validation of higher-order speculation may introduce a better speculated state that is more likely to be the correct starting state (see Section 4.2). Based on these findings, we propose a higher-order *iterative speculation* scheme which organizes the FSM computations into a series of iterations, gradually improving the accuracies of speculation in a parallel fashion.

Finally, to cover FSMs exhibiting diverse properties, we integrate different parallelization schemes into a *multi-scheme* parallelization framework, called BoostFSM. Based on a series of heuristics, BoostFSM automatically selects the best parallelization scheme for the given FSM and its inputs. Using a set of FSM benchmarks with various characteristics, our evaluation shows that path fusion improves the speedup of enumerative parallelization from 15.4× to 31.0× (static fusion [3]) and 18.3× (dynamic fusion) on a machine with 64 cores; for speculative parallelization, high-order speculation raises the speedup from 3.1× to 19.5×. With parallelization scheme selection, BoostFSM achieves 25.8× speedup on average.

In summary, this work makes a three-fold contribution.

- First, it proposes *static and dynamic path fusion* techniques to reduce the overhead of maintaining multiple execution paths in enumerative FSM parallelization (Section 3).
- Second, it introduces *higher-order speculation* in the context of FSM parallelization and designs an *iterative speculation* scheme to address the serial validation bottleneck in the existing speculative FSM parallelization (Section 4).
- Finally, this work offers a set of heuristics to help select the parallelization scheme for the given FSM (Section 5) and confirms the effectiveness of the proposed techniques with a systematic evaluation (Section 6).

2 BACKGROUND

We first provide the background of this work.

2.1 FSM and Its Dependences

As shown in Figure 2-(a), an FSM can be represented as a directed graph, where nodes represent *states*, edges represent *transitions* among states, and labels on the edges indicate the *conditions* for the transitions to happen. The transitions can be stored in the memory as a *transition table*, as shown in Figure 2-(b). The size of the table is $N \times |\Sigma|$, where N is the number of states and $|\Sigma|$ is the number of symbols (Σ is known as the *alphabet*).

As shown in Figure 2-(c), an FSM execution starts from the *initial state* (S_0) and makes transitions by consuming input symbols one by one. Once moving into an *accept state* (a node with double circles), the FSM may trigger some *action*, like emitting a code in Huffman decoding [26] or incrementing a counter in pattern matching [64].

[2]Nondeterministic finite automaton and deterministic finite automaton.

[3]Note that static path fusion is not applicable to all FSMs.

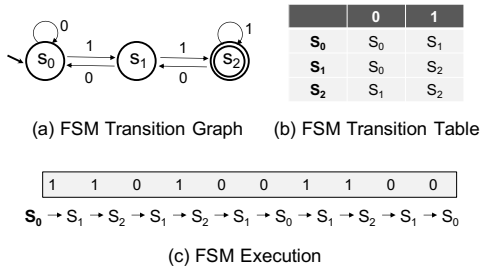


Figure 2: FSM Example

According to its execution model, every state transition in the transition sequence depends on not only the corresponding input symbol but also the prior state. Together, they form a *dependence chain*, inherently preventing the FSM from running in parallel. Many prior studies [16, 24, 33, 41, 42, 66, 67] tried to "break" the dependence chain. Despite the differences in detail, they fall into two basic categories: *state enumeration* and *state speculation*. Next, we elaborate each of them with examples.

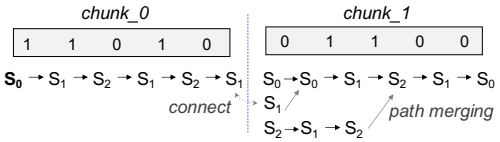


Figure 3: State Enumeration

2.2 State Enumeration

Assume we partition the input to an FSM into two chunks, as shown in Figure 3. For the first chunk, we start the FSM execution from the initial state S_0. But, for the second chunk, we do not know its starting state, as it depends on the processing of the first chunk. The basic idea of state enumeration [16, 33] is to fork an execution path for each state in the FSM. Certainly, one of the paths must be correct. As demonstrated in Figure 3, S_1 is later found to be the actual starting state, based on the ending state of *chunk_0*. Hence, its execution path will be finally selected.

However, maintaining all execution paths may create significant overhead, which compromises or even outweighs the benefits of parallelization. Prior work [33] observed that, after certain number of transitions, different execution paths may merge. As shown in Figure 3, the paths started with S_0 and S_1 both transition into S_0 after reading the first 0. Later, the path started with S_2 also merges with the rest. After path merging, only one of them needs to be maintained, thus lowering the overhead. However, the effectiveness of path merging highly depends on the state convergence properties of the FSM. In fact, for many real-world FSMs, most states tend to converge quickly, but a few states fail to converge after a large number of transitions [33]. As an example, Figure 4 shows an FSM slightly different from that in Figure 2, but no states in this FSM would converge for any given input.

One way to address the limitation of poor state convergence is to explore SIMD parallelism [33]—using different SIMD lanes for

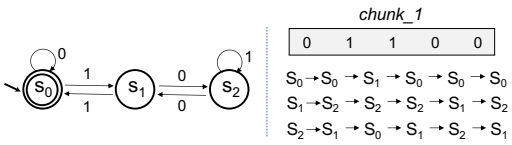


Figure 4: FSM Example with Poor Convergence

running different FSM execution paths. However, such fine-gained hardware parallelism can be otherwise used to enable extra data-level parallelization—partitioning the input into more chunks [41]. Moreover, its efficiency is restricted by the SIMD width.

Besides path merging, it is often beneficial to separate the state actions into a second pass after the state enumeration, so that they do not have to be "multi-versioned" [33]. However, this two-pass processing introduces non-negligible overhead even when the FSM shows ideal state convergence (see Section 6).

2.3 State Speculation

Instead of enumerating all the states, the other strategy is to predict the starting state. As illustrated in Figure 5, state S_2 is predicted to be the starting state, from which *chunk_1* is processed. However, if the prediction turns out to be incorrect—*misspeculation*, the chunk would need to be reprocessed. In the example from Figure 5, S_1 is later found to be the correct starting state. As a result, *chunk_1* gets reprocessed. Luckily, path merging may be detected between the reprocessing path and the speculated processing path. Once they merge, the reprocessing can safely stop.

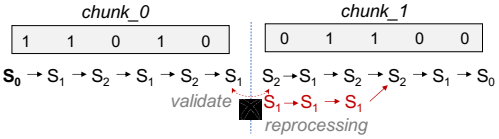


Figure 5: Speculative Parallelization

To predict the starting state for *chunk_i*, prior work [41, 42] runs state enumeration on a suffix of chunk *chunk_i−1*, namely *lookback*, then selects the ending state that appears most frequently among the enumerated paths. More principled reasoning of probabilities can further improve the accuracy [67].

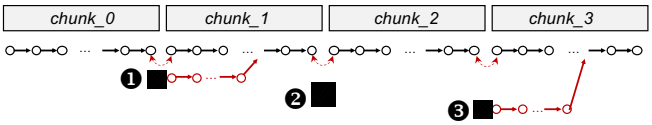


Figure 6: Sequential Validations

The scalability bottleneck in the speculative parallelization lies in its *sequential validations*. When the input is partitioned into multiple chunks, the validations have to be conducted in order from the second chunk to the last, as shown in Figure 6, because, before

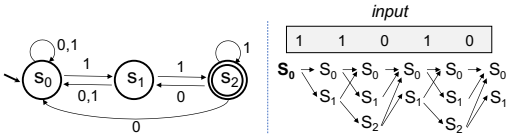


Figure 7: NFA and Its Execution

the prior chunk is validated (and reprocessed as needed), we are not sure if its ending state is correct. This is less a concern when the speculation accuracy is high or the reprocessing lengths are short. But, when the speculation accuracy drops or the reprocessing paths fail to converge with their speculative paths quickly, the scalability of speculative parallelization could be seriously limited.

In summary, the efficiencies of both FSM parallelization schemes depend on the properties of the FSM and its inputs. For those FSMs exhibiting unfavorable properties, they both suffer from high overhead and poor scalability. In the following, we will introduce two techniques to address the scalability issues in each of the two schemes, namely *path fusion* and *higher-order speculation*. After that, we will present a set of heuristics to facilitate the parallelization scheme selection in the presence of FSMs that carry a wide range of diverse properties.

3 PATH FUSION

This section presents *path fusion*, a technique that fuses different FSM execution paths into a single path, to boost the efficiency of enumerative parallelization. Note that, unlike path merging (see Section 2), path fusion does not rely on FSM's state convergence property. Instead, its idea is inspired by the classic NFA to DFA conversion [2]. Next, we first provide the intuition of path fusion, then present its basic algorithm, and finally discuss how to adopt it dynamically during the enumerative parallelization.

3.1 Intuition

An interesting observation we made is that the enumerative FSM parallelization suffers from a similar kind of inefficiency as NFA execution. As shown in Figure 7, an NFA execution in general needs to track multiple current states (bounded by the total number of states) due to its nondeterministic behaviors, which leads to poor execution efficiency in a way similar to that of the enumerative parallelization. Despite the similarities, there are a couple of key differences between the two scenarios:

- First, state enumeration maintains a *vector* of states (i.e., ordered), each for an FSM execution path. The ordering is essential to selecting the right execution path later during the merging phase. By contrast, an NFA only maintains a *subset* of states, without any ordering.
- Second, the number of current states in an NFA execution may increase or decrease (see Figure 7), but the number of current states in state enumeration may only decrease, which happens in the cases of path merging.

A well-known solution to the inefficiency of NFA execution is to convert the NFA to an equivalent DFA using the subset construction algorithm [2]. Thus, a natural question is: *can we design a similar technique to address the execution inefficiency in state enumeration?* Fortunately, we find that, by adopting a worklist-based strategy like the one used in the subset construction algorithm [2], we can generate a new FSM, called *fused FSM*, whose single execution path simulates multiple execution paths of the original FSM. Next, we explain how to statically (i.e., offline) construct the fused FSM.

3.2 Static Path Fusion

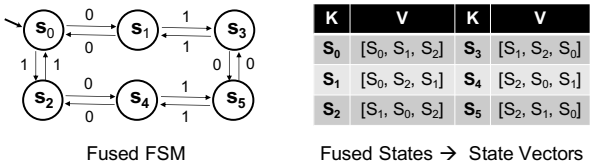
A state in a fused FSM corresponds to a vector of states in the original FSM. Like NFA to DFA conversion [2], we can statically construct a fused FSM without any actual inputs.

Algorithm 1 Static Fused FSM Construction

```
1:  Input: FSM with trans[S_j][c_i], j ∈ [0, N), c_i ∈ Σ
2:  Output1: Fused FSM with Trans[S_j][c_i], j ∈ [0, M), c_i ∈ Σ
3:  Output2: Mapping from fused states S to state vectors V: map
4:
5:  initialize Trans[][] and map
6:  V_0 = [S_0, S_1, ··· , S_N] /* starting state of fused FSM */
7:  map.add(V_0, S_0) /* initialize the map */
8:  worklist = {V_0} /* initialize the worklist */
9:  cnt = 1 /* counter of fused states */
10: while worklist is not empty do
11:     remove an item V from the worklist
12:     S = map.find(V)
13:     for each input symbol c_i do
14:         initialize V_next[]
15:         for each state S_j in V do
16:             V_next[j] = trans[S_j][c_i]
17:         if map.find(V_next) == null then /* first time meet it? */
18:             map.add(V_next, S_cnt)
19:             cnt = cnt + 1
20:             add V_next to worklist
21:         S_next = map.find(V_next)
22:         Trans[S][c_i] = S_next /* record fused state transition */
23: reverse the key and value in map
```

Algorithm. Algorithm 1 presents a worklist-based strategy to construct the fused FSM with states $\{S_0, S_1, \cdots, S_M\}$ from a given FSM with states $\{S_0, S_1, \cdots, S_N\}$. Initially, it maps V_0, a special state vector $[S_0, S_1, \cdots, S_N]$ that corresponds to the N enumerated execution paths, to the initial state of the fused FSM S_0 (Line 6). Then, it initializes a worklist with V_0. After that, the algorithm iteratively removes state vectors from the worklist, computes their next state vectors (V_{next}) for each symbol $c_i \in \Sigma$ (Lines 15–16), maps new state vectors to new fused states (Line 18), and finally records the fused state transitions (Line 20). In addition, the algorithm creates a *map* from fused states to state vectors (Line 22) which will be used to decode a fused state back to a state vector.

By only adding new fused states to the worklist (Lines 17 and 20), the algorithm will always terminate, as the number of states in the fused FSM is bounded by the size of the N-dimensional vector space N^N. However, in practice, the algorithm usually traverses only a very small fraction of the entire vector space, as we will demonstrate shortly after a quick example.



K	V	K	V
S_0	[S_0, S_1, S_2]	S_3	[S_1, S_2, S_0]
S_1	[S_0, S_2, S_1]	S_4	[S_2, S_0, S_1]
S_2	[S_1, S_0, S_2]	S_5	[S_2, S_1, S_0]

Figure 8: Static Fused FSM for the FSM in Figure 4

Example. Figure 8 shows the fused FSM generated for the FSM example in Figure 4. The fused FSM consists of 6 states, whose IDs follow the order that the states are created. With the fused FSM, the three execution paths on *chunk_1* shown in Figure 4 can be reduced to a single execution path of the fused FSM: $S_0 \rightarrow S_1 \rightarrow S_3 \rightarrow S_1 \rightarrow S_0 \rightarrow S_1$. Later, if S_2 turns out to be the actual starting state of *chunk_1*, we can then immediately find that the actual ending state of *chunk_1* is S_1, the third element in the state vector that is mapped to S_1. This shows the importance of using vectors instead of subsets as the states of fused FSM—preserving the correspondence between the starting states and ending states. Note that, though the vector space for the FSM example in Figure 4 is $3^3 = 27$, the statically generated fused FSM only consists of 6 states. Next, we show this is not a special case, but a prevalent property of the fused FSMs.

Size in Practice. Similar to the NFA to DFA conversion [2], the sizes of the fused FSMs for real-world FSMs are often significantly less than the theoretical bound. To demonstrate this, 377 FSMs from the Snort library [48] are chosen so that the fused FSM for each consists of less than 10^6 states. Figure 9 reports the actual number of fused states (note that logarithmic scales are used on both axes). We find that the sizes of the fused FSMs are usually well below N^3 and even below N^2, where N is the number of states in the original FSM. These results confirm the feasibility of static fused FSM generation for many real-world FSMs. Correspondingly, the time complexity of Algorithm 1 in practice is often below $O(N^4 \cdot |\Sigma|)$ and even $O(N^3 \cdot |\Sigma|)$, because, for each fused state, there are $|\Sigma|$ different transitions, and for each fused state transition, the algorithm needs N original FSM transitions.

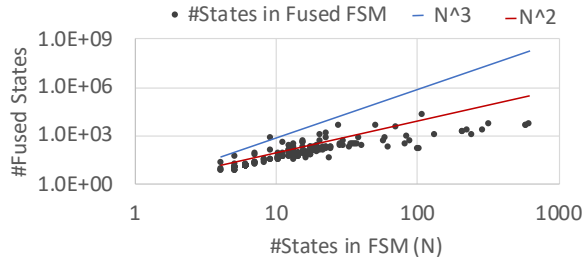


Figure 9: Number of States in Fused FSMs

Despite the promises of static fused FSM generation, we still found that, for many FSMs, the algorithm fails to generate fused FSMs in 3 minutes or generates fused FSMs with over 1 million states. In general, the size of the fused FSM should fit into the given memory budget. For this reason, we next explore the possibility of dynamic fused FSM generation.

3.3 Dynamic Path Fusion

Unlike static path fusion which builds the entire fused FSM for all possible inputs, dynamic path fusion constructs a partial fused FSM that only captures the states and transitions for a single input. In this way, it may reduce the memory needs.

Algorithm. The application of dynamic path fusion resembles the just-in-time (JIT) compilation strategy used in modern compilers. It consists of two execution modes:

- `basic` mode. Given a vector of current states V and an input symbol c, this mode makes individual transitions for each state in the vector to obtain the next state vector V_{next}:
$$V_{next}[i] = trans[V[i]][c], 0 \leq i < N \quad (1)$$
In addition, it generates fused states and transitions: $S_{next} = Trans[S][c]$, where S and S_{next} correspond to V and V_{next}, respectively. Once it finds a visited state vector V, it will map it to its fused state S and switch to the *fused* mode.
- `fused` mode. Given the current fused state S, this mode tries to make a fused state transition $S_{next} = Trans[S][c]$. If the transition is unavailable, it switches to `basic` mode.

With dynamic path fusion, the state enumeration scheme starts from the `basic` mode, then switches between the two modes based on the availability of fused state transitions.

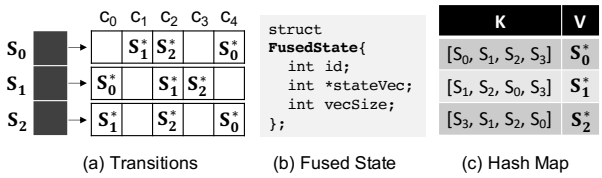


Figure 10: Data Structures for Dynamic Path Fusion
(S_i^* is a pointer to a fused state; $[S_i, \cdots, S_j]$ is a state vector)

Data Structures. A key design question in implementing the dynamic path fusion is how to store the transition information $Trans[S][c]$. A straightforward solution is to use a hash map, where the key is a combination of S and c, and the value is the next fused state S_{next}. While being intuitive, it requires an invocation of a hash function for each fused state transition. Comparing to the transition table (Figure 2-b), we found the cost of hash-map-based state transitions is about 7× higher. Instead, we store the fused state transitions into a *vector of arrays*. As illustrated in Figure 10-a, each array is of fixed length once allocated, but the vector is extensible at the end—each time a new fused state is created, a "row" is added to the vector of arrays. In addition, each "row" is indexed by the input symbol ID while the vector is indexed by the fused state ID. Each element in the "row" stores a pointer (like S_1^*) to the target state for an input symbol. In theory, if the transitions of an execution are scattered sparsely across many fused states, this data structure may waste space, similar to the transition table. However, in practice, we found that, for a single input, the transitions are often concentrated among a few "hot" states, leading to small memory footprints.

The `FusedState` shown in Figure 10-b consists of a state `id` and a pointer to its corresponding state vector to quickly switch back

to the basic mode once the fused state transition is unavailable. To find the chances of switching to the fused mode, a *hash map* from the state vectors to fused states (see Figure 10-c) is maintained. The sizes of the vector and the hash map equal to the number of generated fused states.

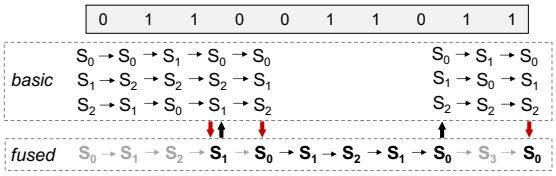


Figure 11: Example Execution with Dynamic Path Fusion

Example. Figure 11 illustrates an execution with dynamic path fusion using the FSM from Figure 4. The thick arrows indicate the switches between the two execution modes. Initially, the execution starts from the basic mode, meanwhile it generates fused states and transitions (in gray color) as it consumes input symbols. After consuming the third symbol ('1'), the execution switches to the fused mode, as it finds that current state vector $[S_0, S_2, S_1]$ has been observed (after consuming the first symbol '0'—its fused state exits ($\mathbf{S_1}$). However, after reading the fourth symbol ('0'), it notices that the fused transition for this symbol has not yet been established (i.e., unavailable), so it switches back to the basic mode, then records this fused transition. A similar process repeats until all the input symbols are consumed.

Cost Analysis. In general, the longer the execution stays in fused mode, the more efficiency benefits the dynamic path fusion brings. In fact, we can capture how long the execution stays in basic mode using the number of unique fused state transitions met in the execution, denoted as N_{uniq}, because (i) each unique fused state transition has to be generated in basic mode; and (ii) basic mode only generates each unique fused state transition once. On the other hand, the time spent in the basic mode also depends on the cost of state enumeration for processing each symbol, which is proportional to the state vector size $|V|$. By default, $|V|$ equals to the number of states in the original FSM, but often can be significantly reduced with some optimization (as we will show shortly). The product between these two factors, $N_{uniq} \times |V|$ captures the total cost of execution in basic mode.

In addition, there are also costs of generating the fused states and transitions, as well as the cost of switching between the two execution modes. However, our evaluation shows that they are usually negligible thanks to the relatively small numbers of fused states and unique state transitions.

Optimization. As mentioned earlier, path fusion is different from the path merging optimization (Section 2). In fact, we can integrate the latter into the former to further boost the efficiency. To achieve this, we separate state enumeration into two phases: *path merging phase* and *path fusing phase*. In the first phase, most paths tend to merge quickly [33]. Once the number of execution paths is below a threshold τ_n or remains unchanged for τ_l transitions, we move to the second phase and start dynamic path fusion. In cases where the path merging reduces the size of the state vector, the following dynamic path fusion will consume even less memory and make faster switches between the two execution modes.

So far, we have presented the path fusion for improving the scalability of state enumeration. Next, we move to the other FSM parallelization scheme, state speculation, which also suffers from a critical scalability issue related to the properties of FSMs.

4 HIGHER-ORDER SPECULATION

As explained in Section 2, the scalability issue in speculative FSM parallelization lies in the sequential validations. In this section, we address this issue by introducing the concept of *speculation order*. Note that though we are not aware of any existing definition for it, the ideas behind this concept have been intensively studied in the literature, especially in the context of thread-level speculation. More details of prior related work will be given in Section 7. Based on this concept, we show that the existing FSM speculation solutions belong to *first-order* speculation, and by raising the speculation to higher orders, it is possible to validate different input chunks in parallel, while ensuring the correctness.

4.1 Speculation Order

Formally, we denote the speculation at the beginning of *chunk_i* as:

$$\textsc{Spec}(i, S, C) \tag{2}$$

where S is the *predicted starting state* and C is the corresponding correct starting state, also referred to as the *correctness criterion*. Speculation $\textsc{Spec}(i, S, C)$ can be *validated* by replacing the predicted state S with the correctness criterion C:

$$\textsc{Spec}(i, S, C) \xrightarrow{validate} \textsc{Non-Spec}(i, C) \tag{3}$$

A validation makes the starting state of *chunk_i* non-speculative. If we refer to the above speculation $\textsc{Spec}(i, S, C)$ as the *first-order* speculation, we can then generalize the concept of "speculation" to *higher-order speculation*, recursively:

Definition 4.1. Speculation $\textsc{Spec}(i, S, C)$ is of

- $(k+1)$-th order, if and only if its validation leads to a k-th order speculation, denoted as

$$\textsc{Spec}^{k+1}(i, S, C) \xrightarrow{validate} \textsc{Spec}^{k}(i, C, C') \tag{4}$$

  where C' is the new correctness criterion corresponding to the new speculated state C;
- first order, if and only if its validation makes the starting state non-speculative, denoted as

$$\textsc{Spec}^{1}(i, S, C) \xrightarrow{validate} \textsc{Non-Spec}(i, C) \tag{5}$$

As shown in Equation 4, the predicted state in $\textsc{Spec}^{k}(i, C, C')$ is in fact the correctness criterion from $\textsc{Spec}^{k+1}(i, S, C)$. In another word, the correctness criterion C itself is speculative.

Based on the above formalization, it is not hard to find that all prior FSM speculation techniques [24, 41, 42, 66, 67], in fact, belong to *first-order* speculation, as the correctness criteria used in their validations are always non-speculative. This is the root cause to the sequential validations—first-order speculation requires all the prior chunks to be non-speculative before it validates the current. Next,

we show that by raising the speculation orders of different input chunks, the sequential validation issue can be effectively alleviated.

4.2 Benefits of Higher-Order Speculation

In general, raising the speculation order could bring benefits to speculative FSM parallelization in two aspects:

- ***Earlier & meaningful validation.*** To illustrate this benefit, let us reexamine the conventional (first-order) speculation in Figure 6, where the validation of *chunk_3* has to wait for the completion of *chunk_2*'s validation, in order to obtain the non-speculative ending state of *chunk_2*, S_{end_2}. However, if we raise the speculation at *chunk_3* to the 2nd order, as shown in Figure 12, and use the speculative ending state of *chunk_2*, S'_{end_2}, as the correctness criterion, then we can immediately start its validation and reprocessing, in parallel with those of *chunk_1*. If S'_{end_2} turns out to be the correct ending state of *chunk_2*, like the case in the figure, then the reprocessing of *chunk_3* would be valid. In another word, the sequential validations are optimistically parallelized.

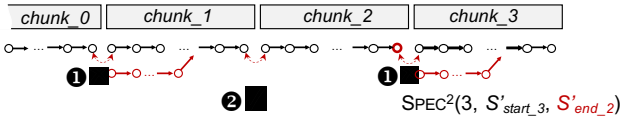


Figure 12: Earlier and Meaningful Validation

- ***Improved speculation accuracy.*** Besides extra parallelism, the other benefit of higher-order speculation comes from the improved speculation accuracy. Without loss of generality, consider *chunk_2* and *chunk_3* in Figure 13, whose starting states are predicted with some existing technique [24, 41, 67], denoted as S'_{start_2} and S'_{start_3}. Statistically speaking, their probabilities of being the correct starting states are the same. After the speculative processing of *chunk_2*, assume the ending state is S'_{end_2}, then the probability that S'_{end_2} is the correct starting state of *chunk_3* might be higher than S'_{start_3} thanks to the potential state convergence during the speculative processing of *chunk_2*. That is, *even if* S'_{start_2} *is incorrect, its execution path may converge with the correct path*, resulting in a correct ending state. If the speculation at *chunk_3* is of second order (see Figure 13), where the correctness criterion is S'_{end_2}, then after the validation (i.e., replacing S'_{start_3} with S'_{end_2}), the speculation accuracy can potentially be increased.

To take the above benefits from the higher-order speculation, we next present a new speculative FSM parallelization model, referred to as *iterative speculation*.

4.3 Iterative Speculation

Unlike existing speculative FSM parallelization [24, 41, 42, 66, 67], iterative speculation organizes the speculative FSM execution into a series of iterations. Algorithm 2 summarizes its basic ideas. First, it predicts the starting state for each chunk, just like the conventional

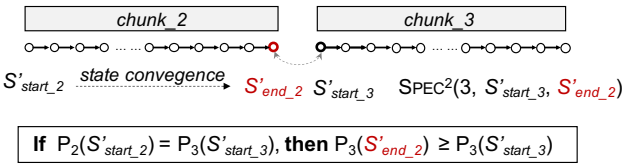


Figure 13: Improved Speculation Accuracy
($P_i(S)$: the probability that S is the correct starting state of *chunk_i*)

Algorithm 2 Iterative Speculation

```
1:  for each chunk_i, 0 ≤ i < N do /* initialization */
2:      if i == 0 then /* set up starting state for each chunk */
3:          s_i = s_init
4:      else
5:          s_i = predict(i) /* initial prediction of starting states */
6:      active[i] = true /* all threads are active initially */
7:  while some active[i] is true do /* iterations */
8:      for each chunk_i do in parallel /* (speculative) execution */
9:          if active[i] == true then
10:             e_i = process(chunk_i, s_i) /* run with path merging */
11:     barrier() /* synchronize */
12:     for each chunk_i do in parallel /* validation */
13:         if s_i ≠ e_{i-1} then /* e_{i-1} is ending state of chunk_i-1 */
14:             s_i = e_{i-1} /* reset starting state */
15:             active[i] = true
16:         else
17:             active[i] = false
18:     barrier() /* synchronize */
```

speculation, except that it also sets up an `active` flag for each thread (Line 6). After initialization, the algorithm enters into a series of iterations. In each iteration, *chunk_i* is processed only if `active[i]` is `true` (Lines 9–10). Note that the processing also checks if its current path merges with that of the prior iteration and stops when it happens. The values of `active` flags are set during the validations (Lines 15 and 17): if the starting state of *chunk_i* in the last iteration is different from the ending state of the prior chunk, `active[i]` is set to `true`; otherwise, `active[i]` is set to `false`. The algorithm terminates once all `active` flags become `false`.

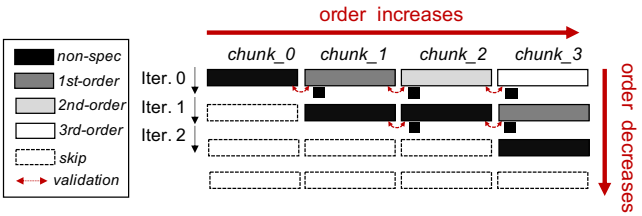


Figure 14: Illustration of Iterative Speculation
(for better clarity, path merging is not illustrated)

Next, we explain how higher-order speculation is reflected in the above algorithm and why the algorithm in fact always terminates within *#chunks* iterations. Figure 14 uses different grayscale levels to represent different orders of speculation, with the darkest used

for non-speculative processing. Initially, we assume that chunks are assigned with increasing orders of speculation: *chunk_i* is of *i*-th order speculation. Then, during each iteration, the latest speculation of each chunk is validated using the latest ending state from the prior chunk. As a result, its speculation order gets reduced by at least one. Once its speculation order becomes 0th (i.e., non-speculative), a chunk will stay inactive, as no ending states of its prior chunks are speculative. Obviously, the initial highest speculation order (of the last chunk) determines the maximum iteration number, thus the algorithm takes at most *#chunks* iterations.

So far, we have explained both path fusion and higher-order speculation. Still, a remaining question is which scheme works the best for a given FSM and its inputs. We address this next.

5 PARALLELIZATION SCHEME SELECTION

Including the two basic schemes (see Section 2), we have discussed five FSM parallelization schemes in total, denoted as follows:

- `B-Enum`: basic state enumeration
- `B-Spec`: basic state speculation
- `S-Fusion`: state enumeration with static path fusion
- `D-Fusion`: state enumeration with dynamic path fusion
- `H-Spec`: higher-order (iterative) speculation

We also refer to the last three as *augmented schemes*. Which scheme works the best depends on the characteristics of the FSM and its inputs. Based on their designs, we focus the scheme selection on four key properties: (i) state convergence rate, (ii) speculation accuracy, (iii) the feasibility to generate a static fused FSM, and (iv) the skewness factor of fused FSM, where the first and last properties are defined below:

Definition 5.1. For an enumerative execution over l symbols, the state convergence rate $conv(l) = 1/|V|$, where $|V|$ is the number of unique current states at the end of the execution.

Definition 5.2. For a (fused) FSM execution over l symbols, the skewness factor $skew(l) = 1/N_{uniq}$, where N_{uniq} is the number of unique (fused) state transitions met during the execution.

Note that our goal is NOT to precisely model the execution time of each scheme, which could be extremely challenging given the diverse and complex FSM transition behaviors. So, instead, we intend to *qualitatively* reason about the conditions for each scheme to work well in general, based on which we draw the heuristics to guide the scheme selection.

The decision tree in Figure 15 summarizes the heuristics used for selecting the parallelization scheme. It starts from the most favorable scenario, then moves to the more challenging ones. For speculative parallelization, the most favorable scenario is when the speculation accuracy is high (according to a threshold τ_{acc}). In this case, `B-Spec` and `H-Spec` are the best choices for their negligible overhead [4] (❶). By contrast, even in the most favorable conditions, enumerative schemes still suffer from the overhead of two-pass processing (see Section 2). If the speculation accuracy is not high enough, the next heuristic is to check the state convergence rate $conv(l)$. Even with a low speculation accuracy, `H-Spec` could still work well as long as the state convergence rate is higher than a

[4] Assuming the cost of starting state prediction is negligible.

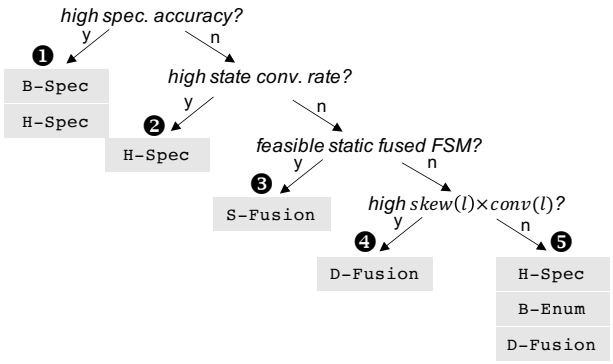


Figure 15: Decision Tree for Scheme Selection

predefined threshold τ_{conv} (❷), thanks to its capability in improving the speculation accuracy (Section 4). Further down the decision tree, if none of the above conditions are met, the next step is to check the feasibility of statically generating a fused FSM. In fact, `S-Fusion` works the best among the enumerative schemes for its single-path execution and offline fused FSM generation (❸). When this condition is unavailable neither, the last resort is `D-Fusion`, which works well when the skewness factor $skew(l)$ of the fused FSM is high and the state vector size is small (i.e., high $conv(l)$). In fact, at this point, the state convergence rate is already unfavorable (see the second heuristic). However, if the combined factor $skew(l) \times conv(l)$ is sufficiently high, `D-Fusion` might become the best option (❹). According to their definitions, $1/(skew(l) \times conv(l)) = N_{uniq} \times |V|$, which captures the major cost of execution for `D-Fusion`, as shown in its "Cost Analysis" (see Section 3.3). Finally, if none of the above conditions are met (i.e., the least favorable situation), one may choose among `H-Spec`, `B-Enum`, and `D-Fusion`(❺)—the best option depends on the specific values of the relevant FSM properties. More detailed performance modeling may help break the tie, but it would come with extra complexities.

Considering the cost of collecting the properties, we target the scheme selection for the given FSM and a group of its inputs, rather than a single input. That is, a few training inputs are (randomly) selected to collect the properties offline, based on which a scheme is selected and used online. In fact, we can instrument `D-Fusion` and `B-Spec` to collect these properties, thus the costs of profiling are slightly higher to their running time on the training inputs. For situations where input sensitivity is concerned, we can run the given FSM over a tiny portion (say 0.25%) of the actual input, though this will pay for a proportional amount of runtime overhead.

6 EVALUATION

In this section, we evaluate the effectiveness of path fusion and higher-order speculation, as well as the scheme selection heursitics.

6.1 Methodology

We implemented the five FSM parallelization schemes summarized in Section 5, in C++ language and used `Pthread` for their parallel executions. Then, we integrated these five schemes along with the scheme selector into one multi-scheme FSM parallelization

Table 1: FSM Benchmarks

(N: #states; $conv(l)$: state conv. rate; static: feasibility for static fusion; $skew(l)$: skewness factor; acc: spec. accuracy; time: profiling time)

FSM	N	$conv(10^3)$	$conv(10^6)$	acc	static	$skew(10^6)$	time(s)
M1	17	1/2	1/2	0%	Yes	1/843	0.87
M2	22	1/7.3	1/1	5%	No	1/98790	1.31
M3	30	1/1.9	1/1.6	78%	Yes	1/1049	0.89
M4	31	1/5	1/5	0%	Yes	1/823	0.86
M5	31	1/5.7	1/1	4%	No	1/73182	1.40
M6	34	1/7.9	1/1	4%	No	1/108896	1.83
M7	53	1/3.8	1/1	9%	No	1/30574	0.89
M8	65	1/2	1/2	100%	Yes	1/795	0.84
M9	145	1/5	1/5	0%	No	1/1089	0.89
M10	193	1/46.7	1/20.1	0%	No	1/10145	3.79
M11	207	1/2	1/2	0%	Yes	1/925	0.90
M12	507	1/2	1/2	0%	No	1/123618	5.38
M13	1045	1/2	1/2	0%	No	1/541	1.15
M14	1179	1/2	1/2	33%	No	1/1065	0.89
M15	2012	1/2	1/2	0%	No	1/819	0.93
M16	4736	1/1	1/1	100%	No	1/3880	1.36

framework, called `BoostFSM`. The memory budget for static fused FSM generation is set to 1GB/FSM, or equivalently 10^6 fused states.

Benchmarks. Table 1 lists the FSM benchmarks used in evaluation with their relevant properties. The 16 benchmarks are collected from the Snort library [48], a pool of signatures in PCRE format used by the state-of-the-art Network Intrusion Detection Systems (NIDS). We converted the signatures into FSMs using one of the off-the-shelf regex2DFA tools [1]. They are chosen to cover the diverse properties of FSMs.

The inputs to the FSMs are 20 traces of real-world network traffics collected from a Linux server using `tcpdump`. Each trace consists of 4×10^8 symbols (i.e., 400MB). For each FSM, five traces are randomly selected and their first 10^6 symbols (i.e., 0.25%) are used to collect the properties in Table 1 offline.

Platform. All experiments were performed on a 64-core machine equipped with an Xeon Phi 7210 processor and 96GB RAM, running Linux 3.10.0. All programs were compiled by GCC 4.8.5 with the "`O3`" flag. The timing results reported are the average of three repetitive runs over 20 inputs (unless specified otherwise).

6.2 Performance

Table 2 reports the speedups of different FSM parallelization schemes using 64 cores over the sequential FSM execution. Note that the sequential FSM execution times are similar across FSMs (the second column), despite their large variation in terms of the number of states (see Table 1). This is because the inputs to different FSMs are of the same size and also the frequently accessed state transitions often well fit into CPU caches. In the following, we first compare the three augmented schemes with each of the two basic schemes[5], then examine the effectiveness of the scheme selection.

Static Path Fusion. First, for benchmarks whose static fused FSMs can be generated (M1, M3-4, M8, M11), `S-Fusion` significantly raises the speedups comparing to `B-Enum`, from 12.9× to 31.0×

[5]We do not select the best of the two basic schemes for each individual FSM as our baseline, since we are not aware of any prior work that can automatically select between the two schemes for a specific FSM.

Table 2: Speedup Comparison

(Baseline: sequential execution; #threads: 64; input size: 4×10^8)

		Basic Schemes		Augmented Schemes			
FSM	Seq(s)	B-Enum	B-Spec	S-Fusion	D-Fusion	H-Spec	BoostFSM
M1	7.45	13.7	1.9	**30.9**	25.1	17.8	**30.9**
M2	7.48	29.1	20	-	19.6	**32.6**	**32.6**
M3	7.39	14.2	1.4	**30.8**	25.1	18.3	**30.8**
M4	7.43	11.1	0.6	**31.1**	25.5	13.9	**31.1**
M5	7.43	28.5	22.9	-	13.1	**30.1**	**30.1**
M6	7.57	26.9	21.6	-	16.1	**32.6**	**32.6**
M7	7.49	29.8	29.7	-	25.5	**32.7**	**32.7**
M8	7.46	13.0	**39.8**	30.9	24.9	39.2	**39.8**
M9	7.44	11.6	0.6	-	**23.9**	10.4	**23.9**
M10	7.37	7.3	1.9	-	8.5	**13.0**	7.3
M11	7.47	12.9	0.6	**31.2**	23.6	17.6	**31.2**
M12	7.53	**12.9**	0.5	-	3.6	8.7	**12.9**
M13	7.40	12.2	0.6	-	**22.5**	16.7	**22.5**
M14	7.46	12.7	0.9	-	**23.5**	11.2	**23.5**
M15	7.35	13.0	0.6	-	**23.4**	17.1	**23.4**
M16	7.51	19.3	**37.2**	-	17.9	36.5	**37.2**
Geo	-	15.4	3.1	31.0	18.3	19.5	25.8

Table 3: Statistics of Static Path Fusion

(N: #states; N_{fused}: #fused states; time: construction time)

FSM	N	N_{fused}	time(s)
M1	17	173	0.06
M3	30	2876	1.25
M4	31	486	0.22
M8	65	6655	4.80
M11	207	19899	37.1

on average, thanks to its (fused) single-path execution and offline fused FSM generation. According to Table 1, after consuming 10^6 symbols, there are still 2.3 paths left on average for these FSMs. `S-Fusion` completely avoids such overhead. On the other hand, for those FSMs whose static fused FSMs are too large to generate (i.e., over the memory budget), static path fusion cannot help. In addition, Table 3 reports the sizes of the static fused FSMs and the construction time. More results about the sizes of fused FSM were presented in Section 3.2 and Figure 9.

Dynamic Path Fusion. The speedups of `D-Fusion` vary a lot across benchmarks, ranging from 3.6× to 25.5×. For some FSMs (M2, M5-7, M12, M16), `D-Fusion` performs worse than `B-Fusion`. As discussed in Section 3.3, given an FSM execution with `D-Fusion`, its efficiency depends on the size of state vector $|V|$ in `basic` mode and the number of unique fused state transitions N_{uniq} encountered in the execution, which are shown in the second and third columns of Table 4, respectively. Note that the state vector size $|V|$ is the number of remaining active states after the path merging phase (see "Optimization" in Section 3.3). The product $N_{uniq} \times |V|$—capturing the cost in `basic` mode (see "Cost Analysis" in Section 3.3), roughly inversely aligns with the speedups of `D-Fusion`. Note that M16 is special in that its $|V|$ drops to one during the path merging phase, so no path fusion is needed.

The 4th column of Table 4 lists the numbers of fused states dynamically generated (ranges from 4 to 1209). For 10 out of 16

Table 4: Statistics of Dynamic Path Fusion

($|V|$: vector size; N_{uniq}: num. of unique fused state transitions; N_{fused}: num. of fused states; t_{merge}: time in merging phase; t_{basic}: time in `basic`; t_{fused}: time in `fused`; t_{pass2}: time in 2nd pass)

FSM	$\|V\|$	N_{uniq}	N_{fused}	t_{merge}(s)	t_{basic}(s)	t_{fused}(s)	t_{pass2}(s)
M1	2.0	1140	7	0.0056	0.0005	0.0957	0.1554
M2	1.4	17260	131	0.0116	0.0205	0.0933	0.1486
M3	1.9	1323	11	0.0069	0.0007	0.0933	0.1547
M4	5.0	966	5	0.0069	0.0004	0.0955	0.1545
M5	1.2	11130	149	0.0110	0.0117	0.0871	0.1527
M6	1.4	21368	246	0.0125	0.0293	0.0975	0.1409
M7	1.1	887	57	0.0095	0.0003	0.0670	0.1523
M8	2.0	902	5	0.0060	0.0004	0.0954	0.1542
M9	5.0	1350	10	0.0077	0.0009	0.0956	0.1552
M10	11.8	10581	116	0.4028	0.0082	0.0772	0.1595
M11	2.0	1236	14	0.0155	0.0007	0.0957	0.1512
M12	2.0	163005	1209	0.0497	0.8116	0.7748	0.1639
M13	2.0	644	4	0.0592	0.0002	0.0951	0.1546
M14	2.0	1253	8	0.0186	0.0004	0.0956	0.1583
M15	2.0	1031	6	0.0177	0.0003	0.0955	0.1578
M16	1.0	-	-	0.0156	0.0000	-	0.1795

Table 5: Speculation Accuracies

		Higher-Order Speculation			
FSM	B-Spec	Iteration-1	Iteration-2	Iteration-3	#Iterations
M1	61%	61%	100%	-	1.9
M2	5%	5%	100%	-	2.0
M3	0%	0%	100%	-	2.0
M4	0%	0%	98%	100%	2.4
M5	5%	5%	100%	-	2.0
M6	5%	5%	100%	-	2.0
M7	9%	9%	100%	-	2.0
M8	100%	100%	-	-	1.0
M9	0%	0%	2%	100%	3.0
M10	62%	62%	97%	100%	2.6
M11	0%	0%	100%	-	2.0
M12	2%	2%	57%	100%	3.0
M13	0%	0%	100%	-	2.0
M14	33%	33%	57%	100%	3.0
M15	0%	0%	98%	100%	2.1
M16	100%	100%	-	-	1.0
Avg	24%	24%	86%	100%	2.1

FSMs, the numbers of fused states are even less than those in the original FSMs, showing high space efficiency in practice.

The last four columns of Table 4 report the time breakdown of D-Fusion, where the first three columns are the time of merging phase (t_{merge}), the time spent in `basic` mode (t_{basic}), and the time spent in `fused` mode (t_{fused}). For most FSMs, t_{fused} is significantly higher than t_{basic}, indicating that the FSM runs mostly in the `fused` mode with a single transition path. M12 is the only FSM for which t_{basic} is higher than t_{fused}, which aligns with its high N_{uniq}—its execution encounters many different fused state transitions, so it has to frequently switch back to the `basic` mode. The summation of t_{merge}, t_{basic}, and t_{fused} roughly equals to the total time of the first pass in a two-pass enumerative scheme (see "State Enumeration" in Section 2). Note that the first pass also includes (partial) fused FSM generation and switchings between the two execution modes, however, as the number of dynamically generated fused states and transitions are relatively small (comparing to the input length), the cost is usually negligible, so as to the cost of mode switchings. The last column of Table 4 reports the time spent in the second pass, which counts the number of accept states encountered during an FSM execution. As the second pass naturally runs in parallel, it shows limited variation across FSMs.

Higher-Order Speculation. As shown in Table 2, `H-Spec` boosts the speedups from 3.1× (`B-Spec`) to 19.5× on average. Specifically, `H-Spec` offers better speedups across all benchmarks, except for M8 and M16, in which cases, both `H-Spec` and `B-Spec` work very well (over 36× speedups), with `B-Spec` showing marginally better speedups. These results are consistent with our earlier discussion, that is, `H-Spec` performs no worse than `B-Spec` in principle. The improvements come from two benefits of higher-order speculation: (i) earlier and meaningful validations and (ii) improved speculation accuracy (see Section 4.2). Table 5 reports the speculation accuracies of `B-Spec` and `H-Spec`. The initial speculation accuracies of `H-Spec` (Iteration-1) are the same as those of `B-Spec` (24% on average). But, over iterations, as it introduces new speculated starting states (based on the new ending states of the prior chunks), the speculation accuracies of `H-Spec` get improved quickly. By the third iteration, all benchmarks reach 100% speculation accuracy. On average, it takes 2.1 iterations for `H-Spec` to complete the processing.

In summary, the augmented schemes substantially boost the speedups over the basic ones. However, their beneifts vary across benchmarks. As shown in Table 2, the best schemes (in bolded font) for the benchmarks scatter across different parallelization schemes, which confirms the needs of scheme selection.

Scheme Selection. The FSM properties used for scheme selection are shown in Table 1. Following the heuristics in Section 5, the selector first checks the speculation accuracy against the threshold τ_{acc} (95%) and finds that only M8 and M16 meet the requirement. Thus, it selects `B-Spec` for these two FSMs. Then, it checks if the state convergence rate $conv(10^6)$ is one (i.e., a single current state is left). If so, it chooses `H-Spec`, which happens to M2 and M5-7. For the remaining benchmarks, the selector further examines the feasibility to generate a static fused FSM. It obtains positive answers for benchmarks M1, M3-4, and M11, thus assigns `S-Fusion` to them. Finally, the scheme selector compares the combined factor between the skewness factor and the state convergence rate $skew(l) \times conv(l)$ against the threshold (10^{-4}). As a result, the remaining benchmarks who satisfy the requirement (M9 and M13-15) are assigned with `D-Fusion`. At this point, there are still two benchmarks left: M10 an M12. Since our selector does not further examine the properties based on their specific values, by default, it chooses `B-Enum`. The last column of Table 2 shows the results of the scheme selection. Out of 16 cases, it only fails to pick the best scheme for M10. The failure is simply due to the fact that the heuristics stops reasoning about the performance at more fined-grained levels, which can be improved with more detailed performance modeling.

6.3 Scalability

In this section, we examine the scalability of different schemes in terms of both the number of cores and the input size.

Varying Number of Cores. The scalabilities over different number of cores for a subset of representative benchmarks are reported

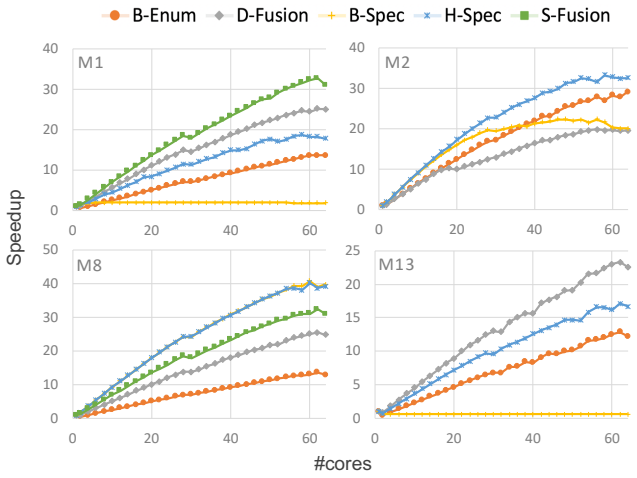


Figure 16: Scalability of Representative Cases
(M1, M2, M8, and M13 correspond to the cases where `S-Fusion`, `H-Spec`, `B-Spec`, and `D-Fusion` are selected, respectively)

in Figure 16. In general, when desired properties are present (see Section 5), all the five schemes can scale well. On the other hand, when the properties are not ideal, some schemes suffer from worse scalabilities than the others. Take `B-Spec` as an example, when the speculation accuracy drops and the state convergence rate is low, it scales poorly and may even run slower than the serial execution (see the curves of `B-Spec` in the cases of M1, M2, and M13), due to its serial validations. Another scheme that may not scale well is `D-Fusion`, as shown in the case of M2. This is because when the input is partitioned into smaller chunks, the number of unique state vectors may not decrease proportionally, thus the overhead becomes relatively higher, compromising the benefits of parallelization. Note that, for some cases, the speedups at 64 cores drop slightly, which is caused by some issue specific to the tested machine.

Varying Input Size. Figure 17 reports the speedups of the five schemes under different input sizes: small (1×10^8), medium (4×10^8), and large (16×10^8). Overall, there are clear trends that the speedups get improved as the input sizes increase for all the parallelization schemes. The trends reflect the Amdhal's law. In our context, the sequential components include thread creation (64 threads), thread synchronization (validations in speculative schemes or correct path selections in enumerative schemes), and I/O operations (printing out results). In addition, with larger inputs, `H-Spec` may also benefit from better convergence with longer input chunk (see Figure 13). For `D-Fusion`, as the input chunk becomes longer, the number of switches between the two modes may become relatively less, thus further improve the performance (happened to M7).

7 RELATED WORK

This section summarizes the related work into three categories: speculative, enumerative, and FSM-related parallelization.

Speculative Parallelization. There exists a rich body of work in realizing speculative parallelization using various methods, ranging from architecture extensions [9, 38, 43, 54, 68] and transactional memory [32, 45], to compiler supports [4, 13, 25, 46, 57, 58] and programming language constructs [39].

By modifying the architectures, thread-level speculation (TLS) [9, 29, 38, 43, 54, 68] spawns speculative threads along the dynamic execution path of a single-threaded application. For correctness, TLS must isolate the writes of "more speculative" threads from the "less speculative" threads and detect the data dependence violations at runtime [38]. As speculation contexts are typically established in a nested way—a speculative thread spawns another speculative thread, such architecture-based TLS schemes are naturally of the *higher-order* speculation as we defined in this work.

More specifically, the idea of high-order speculation is akin to some of the existing ideas like speculative data forwarding and eager recovery from misspeculation [37, 44, 54], as well as parallel ordered commits [18, 19]. Given the existence of these ideas, one contribution of this work is to bring them to solve the scalability problem of parallelizing FSM computations.

Software-based approaches [4, 13, 25, 46, 57, 58] achieve similar goals with software-managed thread state isolation and runtime data dependence analysis. For example, LRPD [46] speculatively applies privatization and reduction to transform sequential loops into DOALL loops, then validate them with runtime checks. Once failed, the loops would be re-executed sequentially.

In comparison, BOP [13] allows programmers or profiling tools to suggest possibly parallel regions (PPR) and leverages virtual memory (i.e., process mechanism) to protect the address space. Note that BOP defines "speculation depth"—a concept relevant to our definition of "higher-order speculation". The key difference is that, in BOP, the k-th level speculation is checked after the first $k-1$ speculative processes commit, which makes itself essentially *first-order* speculation. Tian and others [57] further separate the speculative state from the non-speculative and propose a "copy or discard" model to better manage the memory state in software speculation. According to its thread execution model, a speculative thread only synchronizes with non-speculative thread (called *main thread*), which makes the solution also *first-order* speculation. On the other hand, the above software speculation schemes may also be augmented to support higher-order speculation.

Besides architecture and programming system supports, Prabhu and others [39] propose two new language constructs: *speculative composition* and *speculative iteration*, for programmers to express speculative parallelism in programs declaratively.

In parallel discrete event simulation (PDES), several optimistic mechanisms [17], such as time warp with lazy cancellation and lazy rollback, also resemble the basic idea of higher-order speculation.

Enumerative Parallelization. By contrast, there are only a few prior works on enumerative parallelization. One reason could be the infeasibility in enumerating all the cases in general programs. Some early works [3, 60] studied the potential of enumerating different execution paths under control branches. If FSM transitions are hard-coded, rather than being stored in a transition table, the enumerative FSM parallelization would be similar to branch enumeration. In comparison, N-way programming model [10] enumerates different algorithms or implementations of the same tasks and selects the one that finishes earliest. For more specific application areas, Malki and others [31] leverage the rank convergence property of dynamic

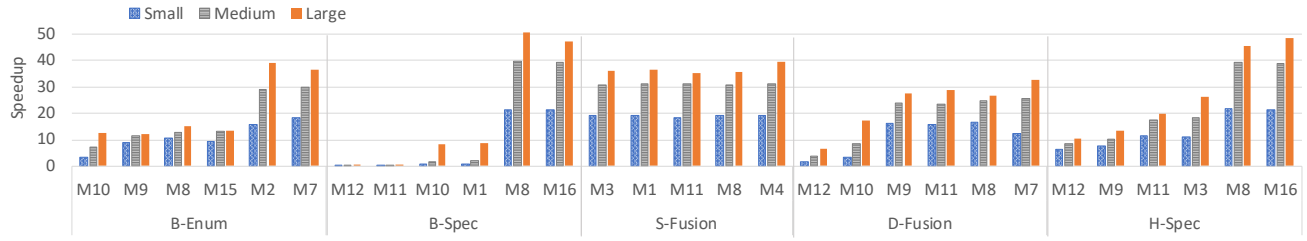


Figure 17: Speedups of Each Scheme with Inputs of Different Sizes
(For conciseness, for each scheme, we ranked the benchmarks by the speedup of Large input, then selected the bottom two, middle two, and top two, but for S-Fusion only five are shown, because they are the only benchmarks for which static fused FSMs can be generated.)

programming to enable coarse-grained parallelization, which can be viewed as a form of state enumeration. Similarly, Raychev and others [47] use symbolic execution to parallelize the user-defined aggregations in big data frameworks, where a symbolic value is an abstraction of all the enumerative cases. More related to FSMs, there are a series of works [21, 22, 35] on enumerative parallelization of pushdown automata, which consist of an FSM and a stack, for processing semi-structured data like XML and JSON.

Other FSM-related Parallelization. In addition to the related FSM parallelization work mentioned in Section 2, there are other works in parallel FSM computations. In particular, various hardware FSM implementations, usually in a form of NFA-like automaton, have been proposed, such as automata processors [14, 55, 61], cache automata [56], and FlexAmata [51]. In comparison, some other works choose to use GPUs to accelerate FSM computations [8, 30, 34, 59, 65, 69]. Like hardware FSMs, they also mostly focus on NFAs rather than DFAs for better space efficiency. In a recent work [63], Xia and others propose reduction-style validations to address the scalability limitation in speculative FSM parallelization on GPUs, which essentially is also of higher-order speculation. Besides the conventional character-by-character FSM processing, there are works that leverage bitwise parallelism and/or SIMD operations to accelerate FSM-related applications [7, 20, 28] or model bitwise/SIMD applications using FSMs to enable (speculative) parallelization [40].

Besides parallelization, the state convergence property in FSMs also makes their executions more tolerable to errors when they are executed in unreliable environments [50].

8 CONCLUSION

This work targets the scalability issues inherited in the two basic FSM parallelization schemes: (i) the cost of maintaining multiple execution paths in enumerative parallelization and (ii) the serial chunk-by-chunk validations in speculative parallelization. For the former, we propose *path fusion*, which can fuse different execution paths into a single one, either statically or dynamically, to lower down the runtime cost of enumerative parallelization. For the latter, we introduced *higher-order speculation* which allows a speculated state to be validated speculatively to enable additional parallelism and improve the speculation accuracy. Furthermore, we presented a set of heuristics to help select the parallelization scheme for practical use. Finally, we evaluated the proposed techniques using real-world FSMs with diverse properties. The results confirmed the effectiveness of the proposed techniques, substantially raising the speedups for a spectrum of FSM benchmarks on parallel processors.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their constructive comments and our shepherd Dr. Guy Steele for his time and efforts in helping with the paper revision. This material is based upon work supported by the National Science Foundation under Grant No. 1565928 and 1751392. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the source code of BoostFSM, including the five FSM parallelization schemes discussed in our paper and some benchmarks along with their inputs used for evaluation. In addition, this artifact provides bash scripts to compile the source code and reproduce the key experimental results reported in the paper.

Considering the software dependencies, a software environment with Linux Centos 7 or other similar Linux distributions, GCC, Bash, Pthread, CMake and Boost library, is needed before the evaluation. Moreover, to reproduce all results reported in the paper, especially the speedup comparison and scalability analysis, the artifact needs to run on Intel Xeon Phi processor (Knights Landing/KNL).

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Five FSM parallelization schemes (see Section 5): `B-Enum`, `B-Spec`, `S-Fusion`, `D-Fusion` and `H-Spec`.
- **Program:** The parallelization framework BoostFSM written in C++.
- **Compilation:** GCC 4.8.5.
- **Binary:** The source code of BoostFSM and compilation scripts are included to generate binaries.
- **Data set:** There are 20 input traces (around 400MB each) of network traffics collected from a Linux server using `tcpdump`. They are the inputs to network intrusion detection systems, and our FSMs.
- **Run-time environment:** The artifact has been developed and tested on Linux (CentOS 7) environment, with source code compiled by GCC using Pthread and Boost libraries.
- **Hardware:** To reproduce the results reported in our paper, the artifact is supposed to run on Intel Xeon Phi 7210 Processor (Knights

Landing/KNL, 1.3GHz), but it may also be compiled and run well on other Linux machines (yielding different results).
- **Execution:** Bash scripts are included for execution.
- **Output:** Results include FSM profiling information, speculation accuracy and performance comparison.
- **How much disk space required (approximately)?:** At least 20GB is recommended, for downloading the 8GB input data and its decompression.
- **How much time is needed to complete experiments (approximately)?:** It takes about 4 hours on the recommended KNL machine, assuming all required tools and libraries have been installed.
- **Publicly available?:** Yes

A.3 Description

A.3.1 How to Access. A file named `ASPLOS21_AE.zip`, containing the source code, scripts, and data sets, are available as a public repository on Zenodo (https://doi.org/10.5281/zenodo.4556045).

A.3.2 Hardware Dependencies. We recommend artifact evaluation on an Intel Xeon Phi architecture (Intel Xeon Phi 7210 with 1.3GHz in particular) to reproduce the results reported in the paper, but it may also be compiled and run well on other Linux machines (yielding different results). At least 20GB space is needed (mainly for the data sets decompression).

A.3.3 Software Dependencies. We recommend that the artifact runs on CentOS 7, but other similar Linux distributions should also work. To compile and run the source code with scripts, users need GCC 4.8.5, CMake 2.8 and Boost 1.66.0 library (or their later versions).

A.3.4 Data Sets. Benchmarks are collected from an open-source network intrusion detection system (Snort), where there is a pool of signatures in PCRE format. The evaluated FSMs are converted from the signatures with using a regular expression to DFA tool. The corresponding data sets are included in this artifact for testing. They are the traces of network traffics collected from a Linux server using `tcpdump` and zipped into the artifact file. There are totally 20 inputs, with size of about 400MB each.

A.4 Installation

Please ensure software dependencies are met before evaluating the artifact. Users need to download the source code and scripts which are zipped into `ASPLOS21_AE.zip` from Zendo. There is a script `compile.sh` under the directory `ASPLOS21_AE/` which can be used to compile the source code and generate the executables (please run command `bash compile.sh`).

A.5 Experiment Workflow

We have provided a script `run.sh` to generate all the results in one step, but we also support flexible evaluations or manual testing. The total evaluation time of this artifact is about 4 hours (on the recommended KNL machine).

To generate the results in Tables 1, 2, 3, 4, and 5, and Figure 16, users can run the following commands:

```
# cd scripts/
# bash GetTable1.sh InputConf_five.in 64
# bash GetTable2.sh InputConf.in 64
# bash GetTable3.sh
# bash GetTable4.sh InputConf.in 64
# bash GetTable5.sh InputConf.in 64
# bash GetFigure16.sh InputConf.in 64
```

For Figure 17, users can repeat the evaluation of Table 2, but over inputs with different sizes.

A.6 Evaluation and Expected Result

Results will be printed to the command console after finishing the evaluation for a table or a figure. Following the experiment workflow, users firstly get the properties of evaluated FSMs, then the speedup comparison results among different schemes in `BoostFSM`. After that, the statistics of path fusion and speculation accuracy for `B-Spec` and `H-Spec` are produced. Finally, users can get the scalability results (i.e., speedup curves) reported in Figure 16.

A.7 Experiment Customization

Please follow commands in the compilation and execution scripts to customize the testing. For example, to test the scalability of different parallelization schemes, users can follow the commands in `ASPLOS21_AE/scripts/GetFigure16.sh`.

REFERENCES

[1] [n.d.]. regex2dfa. https://github.com/kpdyer/regex2dfa.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Pritpal S Ahuja, Kevin Skadron, Margaret Martonosi, and Douglas W Clark. 1998. Multipath execution: Opportunities and limits. In *Proceedings of the 12th International Conference on Supercomputing*. 101–108.

[4] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. 2020. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 351–367.

[5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. *The landscape of parallel computing research: A view from berkeley.* Technical Report. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[6] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. 2015. Scalable algorithms for NFA multi-striding and NFA-based deep packet inspection on GPUs. *IEEE/ACM Transactions on Networking* 24, 3 (2015), 1704–1717.

[7] Robert D Cameron, Thomas C Shermer, Arrvindh Shriraman, Kenneth S Herdy, Dan Lin, Benjamin R Hull, and Meng Lin. 2014. Bitwise data parallelism in regular expression matching. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 139–150.

[8] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review* 40, 5 (2010), 20–26.

[9] Marcelo Cintra, José F Martínez, and Josep Torrellas. 2000. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 13–24.

[10] Romain E Cledat, Tushar Kumar, and Santosh Pande. 2011. Efficiently speeding up sequential computation through the N-way programming model. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 537–554.

[11] Sutapa Datta and Subhasis Mukhopadhyay. 2015. A grammar inference approach for predicting kinase specific phosphorylation sites. *PloS one* 10, 4 (2015), e0122294.

[12] Yanlei Diao, Peter Fischer, Michael J Franklin, and Raymond To. 2002. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 341–342.

[13] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. 2007. Software behavior oriented parallelization. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 223–234.

[14] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*. 533–545.

[15] Todd J Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. 2003. Processing XML streams with deterministic automata. In *International Conference on Database Theory*. Springer, 173–189.

[16] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183.

[17] David Jefferson and Peter Reiher. 1991. Supercritical speedup. *ACM SIGSIM Simulation Digest* 21, 3 (1991), 159–168.

[18] David R Jefferson. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 3 (1985), 404–425.

[19] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture*. 228–241.

[20] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. 2020. Scalable Structural Index Construction for JSON Analytics. *Proceedings of the VLDB Endowment* 14, 4 (2020), 694–707.

[21] Lin Jiang, Xiaofan Sun, Umar Farooq, and Zhijia Zhao. 2019. Scalable Processing of Contemporary Semi-Structured Data on Commodity Parallel Processors-A Compilation-based Approach. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 79–92.

[22] Lin Jiang and Zhijia Zhao. 2017. Grammar-aware Parallelization for Scalable XPath Querying. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 371–383.

[23] Peng Jiang and Gagan Agrawal. 2017. Combining SIMD and Many/Multi-core parallelism for finite state machines with enumerative speculation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 179–191.

[24] C. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. 2009. Parallelizing the web browser. In *HotPar*.

[25] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. 2009. Fast track: A software system for speculative program optimization. In *2009 International Symposium on Code Generation and Optimization*. IEEE, 157–168.

[26] Shmuel Tomi Klein and Yair Wiseman. 2003. Parallel Huffman decoding with applications to JPEG files. *Comput. J.* 46, 5 (2003), 487–497.

[27] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Computer Communication Review*, Vol. 36. ACM, 339–350.

[28] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1118–1129.

[29] Mikko H Lipasti and John Paul Shen. 1996. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 226–237.

[30] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. Why GPUs are slow at executing NFAs and how to make them faster. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 251–265.

[31] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2014. Parallelizing dynamic programming through rank convergence. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 219–232.

[32] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 166–176.

[33] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. 529–542.

[34] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. 2017. Demystifying automata processing: GPUs, FPGAs or Micron's AP?. In *Proceedings of the International Conference on Supercomputing*. 1–11.

[35] Peter Ogden, David Thomas, and Peter Pietzuch. 2013. Scalable XML query processing using parallel pushdown transducers. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1738–1749.

[36] Yinfei Pan, Ying Zhang, Kenneth Chiu, and Wei Lu. 2007. Parallel XML parsing using meta-DFAs. In *e-Science and Grid Computing, IEEE International Conference on*. IEEE, 237–244.

[37] Leo Porter, Bumyong Choi, and Dean M Tullsen. 2009. Mapping out a path from hardware transactional memory to speculative multithreading. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 313–324.

[38] Manohar K Prabhu and Kunle Olukotun. 2003. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

[39] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. 2010. Safe programmable speculative parallelism. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 50–61.

[40] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. 2020. Challenging Sequential Bitstream Processing via Principled Bitwise Speculation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 607–621.

[41] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. MicroSpec: Speculation-centric fine-grained parallelization for FSM computations. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 221–233.

[42] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. 2017. Enabling Scalability-Sensitive Speculative Parallelization for FSM Computations. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. https://doi.org/10.1145/3079079.3079082

[43] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M Tullsen. 2005. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 269–279.

[44] Hany E Ramadan, Christopher J Rossbach, and Emmett Witchel. 2008. Dependence-aware transactional memory for increased concurrency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 246–257.

[45] Arun Raman, Hanjun Kim, Thomas R Mason, Thomas B Jablin, and David I August. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 65–76.

[46] Lawrence Rauchwerger and David A Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (1999), 160–180.

[47] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 153–167.

[48] Martin Roesch et al. 1999. Snort: Lightweight Intrusion Detection for Networks.. In *LISA*, Vol. 99. 229–238.

[49] Indranil Roy and Srinivas Aluru. 2014. Finding motifs in biological sequences using the Micron automata processor. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 415–424.

[50] Amir Hossein Nodehi Sabet, Junqiao Qiu, Zhijia Zhao, and Sriram Krishnamoorthy. 2020. Reliability Analysis for Unreliable FSM Computations. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 2 (2020), 1–23.

[51] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. FlexAmata: A universal and efficient adaption of applications to spatial automata processing accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 219–234.

[52] Priti Shankar, Amitava Dasgupta, Kaustubh Deshmukh, and B Sundar Rajan. 2003. On viewing block codes as finite automata. *Theoretical Computer Science* 290, 3 (2003), 1775–1797.

[53] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. 2008. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. ACM, 207–218.

[54] J Gregory Steffan and Todd C Mowry. 1998. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*. IEEE, 2–13.

[55] Arun Subramaniyan and Reetuparna Das. 2017. Parallel automata processor. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 600–612.

[56] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 259–272.

[57] Chen Tian, Min Feng, and Rajiv Gupta. 2010. Speculative parallelization using state separation and multiple value prediction. In *Proceedings of the 2010 International Symposium on Memory Management*. 63–72.

[58] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. 2008. Copy or discard execution model for speculative parallelization on multicores. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 330–341.

[59] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P Markatos, and Sotiris Ioannidis. 2008. Gnort: High performance network intrusion detection using graphics processors. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 116–134.

[60] Steven Wallace, Brad Calder, and Dean M Tullsen. 1998. Threaded multiple path execution. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*. IEEE, 238–249.

[61] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016. An overview of Micron's automata processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP*

International Conference on Hardware/Software Co-design and System Synthesis. 1–3.

[62] Ke Wang, Yanjun Qi, Jeffrey J Fox, Mircea R Stan, and Kevin Skadron. 2015. Association rule mining with the Micron Automata Processor. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 689–699.

[63] Yang Xia, Peng Jiang, and Gagan Agrawal. 2020. Scaling out speculative execution of finite-state machines with parallel merge. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 160–172.

[64] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. ACM, 93–102.

[65] Xiaodong Yu and Michela Becchi. 2013. GPU acceleration of regular expression matching for large datasets: exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers*. 1–10.

[66] Zhijia Zhao and Xipeng Shen. 2015. On-the-Fly Principled Speculation for FSM Parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2694344.2694369

[67] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-Based Computations through Principled Speculation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 543–558. https://doi.org/10.1145/2541940.2541989

[68] Craig Zilles and Gurindar Sohi. 2002. Master/slave speculative parallelization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings*. IEEE, 85–96.

[69] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 129–140.