Scalable FSM Parallelization via Path Fusion and Higher-Order Speculation

Junqiao Qiu*[1], Xiaofan Sun[2], Amir Hossein Nodehi Sabet[2], and Zhijia Zhao[2]

[1]Michigan Technological University, [2]University of California, Riverside

1. Motivation

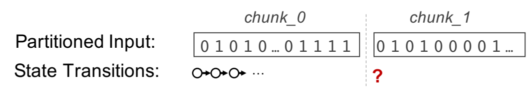
As a basic computation model, finite-state machine (FSM) embodies a variety of important applications, ranging from intrusion detection [21, 11, 18, 3] and data decoding [10, 17] to motif searching [16, 4], rule mining [19], and textual data analytics [13, 6, 5]. Unfortunately, the execution of an FSM is known to be "embarrassingly sequential" [2, 23], 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.

2. State of The Art

To address the inherent dependences in FSM computations, prior work [7, 9, 23, 12, 22, 14, 15] has mainly followed two basic parallelization schemes: *state enumeration* and *state speculation* (see Figure 1-b). 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:

(i) *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 [7, 12]. Obviously, maintaining all the execution paths may bring significant overhead. To reduce it, prior work [12] checks if some paths transition to the same state, known as *path merging*, in which case only one of them 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.

(ii) *State Speculation*. Instead of considering all the states, one can guess the starting state of the second chunk [23, 22, 14, 15]. 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).



(a) State Dependence

	State Enumeration [7, 12]	State Speculation [14, 15, 22, 23]
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	Higher-Order Speculation
	Scheme Selection	

(b) Two Basic Parallelization Schemes

Figure 1: FSM Parallelization: Challenges and Solutions

These serialized validations form a fundamental scalability bottleneck in the speculative FSM parallelization [15].

In addition, a hybrid scheme may enumerate a subset of states [8, 20], which makes a tradeoff between the limitations of both schemes. In summary, the existing FSM parallelization schemes face fundamental scalability challenges.

3. Major Contributions

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.

3.1. Path Fusion

For state enumeration, we propose to fuse different execution paths into a single path to lower down the overhead.

Intuition. An interesting observation we made is that state enumeration suffers from a similar kind of inefficiency as the execution of *nondeterministic finite automaton* (NFA). The former needs to maintain a *vector* of states for all the execution paths, while the latter needs to track a *subset* of active states. A well-known solution to the inefficiency of NFA execution is to convert the NFA to an equivalent DFA (deterministic finite automaton) using the subset construction algorithm [1]. *Can we design a similar technique to address the inefficiency in state enumeration?* In fact, we find that, by developing a *vector construction algorithm* similar to the subset construction algorithm, we can generate a new FSM whose single execution path mimics multiple execution paths of the original FSM. We call this technique *path fusion*.

Static Path Fusion. The key to path fusion is to construct a *fused FSM* where each state corresponds to a vector of states

*This work was performed when the author was a Ph.D. student at UCR.
[1]Here, an FSM refers to a deterministic finite automaton (DFA).

in the original FSM. Like NFA to DFA conversion [1], we can statically construct the fused FSM. First, we map the initial fused state to state vector $[S_0, S_1, \cdots, S_N]$ which corresponds to the enumerated execution paths. Then, we feed every input symbol to the existing fused states to iteratively discover new fused states and valid fused state transitions, until no new fused states can be found. In theory, the fused FSM can be very large as it traverses a space of N^N, where N is the size of the original FSM. However, in practice, their sizes are often well below N^3 and even N^2. Despite the promises, it might still be desired that the fused FSM can fit into a memory budget.

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 consists of states and transitions for a single input. The idea of dynamic path fusion resembles the just-in-time (JIT) compilation used in modern compilers. It consists of two execution modes: the *basic* mode where different paths are enumerated and fused state transitions are generated, and the *fused* mode which only makes fused state transitions. An execution starts from the *basic* mode, then switches between the two modes based on the availability of the fused state transitions.

3.2. Higher-Order Speculation

To address the serial validation bottleneck in state speculation, we introduce the concept of *speculation order*.

Speculation Order. Formally, we denote the speculation at the beginning of input *chunk_i* as:

$$\textsc{Spec}(i, S, C) \tag{1}$$

where S is the *predicted starting state* and C is the correct starting state, also referred to as the *correctness criterion*. By feeding a *speculated* correctness criterion to the speculation, we can raise the speculation to higher orders. For example,

$$\textsc{Spec}^{k+1}(i, S, C) \xrightarrow{validate} \textsc{Spec}^{k}(i, C, C') \tag{2}$$

means that a $k+1$-th order speculation, after its validation, becomes a k-th order speculation. The correctness criterion of the former becomes the predicted state of the latter.

Based on the above formalization, it is not hard to find that all prior FSM speculation techniques [9, 23, 22, 14, 15], in fact, belong to *first-order* speculation, as the correctness criteria used in their validations are always non-speculative.

Benefits of Higher-Order Speculation. Raising the order of speculation may bring benefits in two aspects:

- Chunks with higher-order speculation no longer need to wait for the ground truth, thus can be validated earlier;
- The validation of higher-order speculation introduces a new speculated state which, in theory, is more likely to the correct starting state, thus improving the accuracy.

Iterative Speculation. Based on the above findings, we design a higher-order *iterative speculation* scheme which organizes

Table 1: 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

the FSM computations into a series of iterations, gradually improving the speculation in a naturally parallel manner.

3.3. Scheme Selection

Together, we consider five FSM parallelization schemes:

- `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

Which scheme works the best depends on the characteristics of the FSM and its inputs. We design a set of heuristics, as a decision tree, to guide the scheme selection.

4. Key Results

We integrated the above FSM parallelization schemes along with the scheme selector into a *multi-scheme* parallelization framework, named BoostFSM. Table 1 reports the speedups of different schemes over the sequential FSM execution on a 64-core machine. The FSMs are collected from a widely used open-source network intrusion detection system (Snort), carrying diverse properties. First, `S-Fusion` is found feasible for five FSMs, for which it raises the speedups from 12.9× (`B-Enum`) to 31.0× on average. Next, `D-Fusion` shows varying speedups, from 3.6× to 25.5×, as its efficiency highly depends on the skewness of the fused state transitions and the state vector size. Third, comparing to `B-Spec`, `H-Spec` shows consistent improvements (from 3.1× to 19.5×), thanks to its two benefits mentioned earlier. Finally, the last column reports the results of scheme selection. Among 16 FSMs, it successfully finds the best scheme for 15 FSMs. The failed case is due to our coarse-grained performance modeling.

5. Why ASPLOS

This work fits ASPLOS for its focus on the parallelization and scalability of a fundamental class of computations and for its uses of speculation and compiler techniques.

References

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[2] 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. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

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

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

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

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

[7] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[8] Peng Jiang and Gagan Agrawal. 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*, pages 179–191, 2017.

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

[10] Shmuel Tomi Klein and Yair Wiseman. Parallel Huffman decoding with applications to JPEG files. *The Computer Journal*, 46(5):487–497, 2003.

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

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

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

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

[15] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. Enabling scalability-sensitive speculative parallelization for FSM computations. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery.

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

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

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

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

[20] Yang Xia, Peng Jiang, and Gagan Agrawal. 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*, pages 160–172, 2020.

[21] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. 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*, pages 93–102. ACM, 2006.

[22] Zhijia Zhao and Xipeng Shen. On-the-fly principled speculation for FSM parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Zhijia Zhao, Bo Wu, and Xipeng Shen. 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*, ASPLOS '14, page 543–558, New York, NY, USA, 2014. Association for Computing Machinery.