# CSE: Parallel Finite State Machines with Convergence Set Enumeration

Youwei Zhuo*, Jinglei Cheng*, Qinyi Luo*, Jidong Zhai†, Yanzhi Wang‡, Zhongzhi Luan§, Xuehai Qian*
*University of Southern California, †Tsinghua University, ‡Syracuse University, §Beihang University
{youweizh, chen520, qinyiluo, xuehai.qian}@usc.edu, zhaijidong@tsinghua.edu.cn, ywang393@syr.edu, luan.zhongzhi@buaa.edu.cn

***Abstract*—Finite State Machine (FSM) is known to be "embarrassingly sequential" because the next state depends on the current state and input symbol. Enumerative FSM breaks the data dependencies by cutting the input symbols into segments and processing all segments in parallel. With unknown starting state (except the first segment), each segment needs to calculate the state transitions, i.e., state→state, for all states, each one is called an enumeration path. The current software and hardware implementations suffer from two drawbacks: *1)* large amount of state→state computation overhead for the enumeration paths; and *2)* the optimizations are restricted by the need to correctly performing state→state and only achieve limited improvements.**

**This paper proposes *CSE*, a Convergence Set based Enumeration based parallel FSM. Unlike prior approaches, CSE is based on a novel computation primitive set(N)→set(M), which maps N states to M states without giving the specific state→state mappings (which state is mapped to which). The set(N)→set(M) has two key properties: *1)* if M is equal to 1, i.e., all N states are mapped to the same state, the state→state for all the N states are computed; *2)* using one-hot encoding, the hardware implementation cost of state→state is the same as set(N)→set(M).**

**The convergence property ensures that M is always less than N. The key idea of CSE is to partition the original all S states into n state sets $CS_1$,$CS_2$,...,$CS_n$, i.e., convergence sets. Using set(N)→set(M) to process each $CS_i$, if the states converge to a single state, then we have successfully computed the enumeration path for each state in $CS_i$; otherwise, we may need to re-execute the stage when the outcome of the previous stage falls in $CS_i$. CSE is realized by two techniques: *convergence set prediction*, which generates the convergence sets with random input based profiling that maximizes the probability of each $CS_i$ converging to one state; *global re-execution algorithm*, which ensures the correctness by re-executing the non-converging stages with known input state.**

**Essentially, CSE reformulates the enumeration paths as set-based rather than singleton-based. We evaluate CSE with 13 benchmarks. It achieved on average 2.0x/2.4x and maximum 8.6x/2.7x speedup compared to Lookback Enumeration (LBE) and Parallel Automata Processor (PAP), respectively.**

## I. Introduction

Finite-State Machine (FSM) is an important mathematical concept in automata theory. FSMs are widely used as a computation model in several important applications such as data analytics and data mining [1], [2], [3], [4], [5], bioinformatics [6], [7], [8], [9], network security [10], [11], computational finance [12] and software engineering [13], [14], [15], [16]. These applications use FSMs as the essential computational model to process tens to thousands of patterns on a large amount of input data. Therefore, the performance of FSM is crucial.

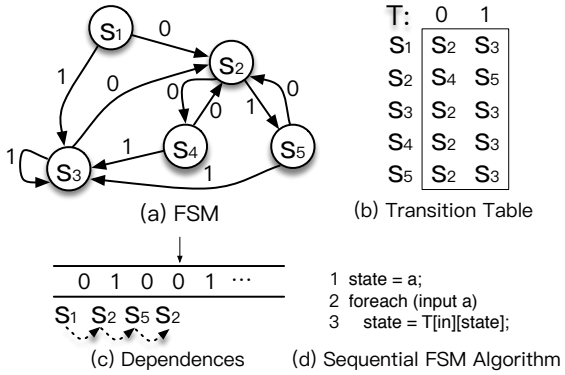


Fig. 1. (Sequential) FSM Example

There are two types of FSM: Deterministic Finite-state Automata (DFA) and Nondeterministic Finite-state Automata (NFA). This paper considers DFA (every NFA can be converted to an equivalent DFA). Figure 1 (a) shows an FSM with five states and input symbols $\{0, 1\}$. The state transition rules could be represented as the *state transition table*, as shown in Figure 1 (b). During execution, the FSM processes the input symbols *sequentially* due to the state transition dependency (Figure 1 (c)). Specifically, each time the FSM reads one symbol from the input string, it looks up the transition table based on the current state and the read symbol to find a transition to the next state (line 3 of Figure 1 (d)). Besides the sequential bottleneck, transition table lookups also cause irregular memory accesses, which motivates the recent works on hardware FSM accelerators [17], [18].

To parallelize the sequential computation, *enumerative FSM* [19], [20], [21], [22], [23], [24], [25], [26] is proposed, in which the input is divided into segments that can be processed in parallel. All segments (except the first) have unknown starting state, therefore the computation has to calculate the state transitions for *all* states, which are called enumeration paths. The number of enumeration paths is equal to the number of states in the FSM, which poses a key challenge due to the high computation cost. Fortunately, FSM has the *convergence* property, which states that the number of enumeration paths to compute is non-increasing due to state convergence, — after two states transition to the same state, the following state paths become the same. This property is the foundation of various hardware and software implementations of enumerative FSM, which are reviewed in detail in Section II. Essentially, they

all compute *state→state transitions* for the non-increasing enumeration paths.

The hardware implementations of enumerative FSM and NFA (e.g., [25]) typically use *one-hot encoding*, where N states are represented by N-bit vector (i.e., active mask), and a bit is set when the state is active. Based on one-hot encoding, the combinational logic (i.e., state transition matrix) selects the next states, based on the input symbol and the matched and activated states (more details in Section III-A).

In the normal state→state, only one bit in active mask is set any time. Lifting this restriction requires no change in combinational logic and hardware cost but enables a novel computation primitive set(N)→set(M), which maps a state set with N states to another state set with M states without giving the specific state→state mappings. The key property of set(N)→set(M) is that, when *M = 1*, it computes the enumerative paths for *all N states* with the cost of computing one path in state→state. This is because all N states are mapped to the same state. Importantly, the convergence property of enumerative FSM ensures M < N. While it is unlikely that all N states of an FSM can converge to a single state, it is highly possible that a subset of N states can converge to a single state. Our solution in this paper is based on this insight.

This paper proposes *CSE*, a Convergence Set based Enumerative FSM. We use set(N)→set(M) as the building block to compute multiple enumeration paths of a state set in parallel speculating that the states will converge to a single state. When the speculation is correct, CSE achieves significant speedup, otherwise, one of more stages may be re-executed based on the concrete output from previous stage. To increase the likelihood of correct speculation, the whole state set ($S$) needs to be partitioned into $n$ disjoint convergence sets $\{CS_i | i \in [1,n]\}$ such that $S = \bigcup_{i=1}^{n} CS_i$ and $CS_i \bigcap CS_j = \varnothing$. The quality of the partition is a major factor determining the performance.

CSE is supported by two techniques. First, *convergence set prediction* generates $\{CS_i | i \in [1,n]\}$ with random input based profiling. Each profiling input will produce one convergence set partition. By profiling a large number of inputs, we can use the partition with the maximum frequency (MFP) as the predicted partition for a given FSM. To improve the prediction accuracy, we apply a partition refinement algorithm to merge distinct partitions. The refined partition *covers* all or most (99% to 100%) partitions generated in profiling input, which leads to high prediction accuracy and low re-execution ratio for real input. Second, we propose a *global re-execution algorithm* and its hardware implementation. It minimally triggers the re-execution of certain stages to ensure the correctness. In essence, CSE reformulates the enumeration paths as set-based rather than singleton-based.

To evaluate CSE, we use an automata simulator VASim [27] and 13 benchmarks in *Regex* [28] and *ANMLZoo* [27]. CSE is compared to lookback enumeration and parallel automata processor [25] with all optimizations implemented. We directly compare different designs based on symbol/sec. We observe that CSE achieved on average 2.0x/2.4x and maximum 8.6x/2.7x speedup compared to lookback enumeration and parallel automata processor, respectively.

The remainder of this paper is organized as follows: Section II reviews the FSM basics and the current works on enumerative FSMs; Section III introduces set(N)→set(M) and analyzes its unique properties; Section IV proposes CSE, focusing on convergence set generation and re-execution algorithm; Section V discusses evaluation methodology and Section VI shows evaluation results; Section VII discusses other related works; Section VIII concludes the paper.

## II. Background and Motivations

This section first explains the concepts of basic and enumerative FSMs. Then we review two recent approaches to improve the efficiency of enumerative FSMs. In the end, we outline the drawback of existing works and motivate our approach.

### A. FSM Basics

There are two forms of FSM: Deterministic Finite-state Automata (DFA) and Nondeterministic Finite-state Automata (NFA). As every NFA can be converted to an equivalent DFA, this paper focuses on DFA. In the remainder of the paper, we will use FSM and DFA interchangeably.

A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, in which $Q$ is a finite set of states, $\Sigma$ is a finite set of input symbols called the alphabet, $\delta$ is a transition function that maps a state and an input symbol to another state, $q_0 \in Q$ is the initial state and $F$ is a set of reporting/accepting states. With a sequence of input symbols $W = w_1, w_2, ...w_T (w_t \in \Sigma)$, the DFA goes over a sequence of states $q_0, q_1, q_2, ...q_m$, where $q^{t+1} = \delta(q^t, w_t)$. The computation is sequential, because the next state depends on the current one.

### B. Enumerative FSM

Sequential bottleneck is the major challenge for FSM computation. *Enumerative FSM* is an effective technique to parallellize sequential FSM by breaking data dependence. In this method, input symbols are partitioned into a sequence of segments that are processed in parallel.

Clearly, only the first segment has a concrete initial state; for all other segments, while the inputs are known, the starting states are unknown. Therefore, they have to perform enumerative execution to compute the state transition for every state (i.e., enumeration path). When all segments finish execution, the first segment generates the concrete output state, while the others generate state→state mapping of all states. At this point, the stages can be "*chained*" together by selecting the output state of each segment $i$ from the computed state→state mapping of segment $i + 1$.

Figure 2 (a) illustrates an example of enumerative FSM with four segments. Figure 2 (b) shows a conceptual view of a segment. With unknown starting state, it needs to compute the state transitions for all $N$ states. Each curve is an enumeration path that is a sequence of state transitions. Given the $t$ input symbols (shown at the bottom), each state follows a different path, finally reaching the eventual states in this stage. Each

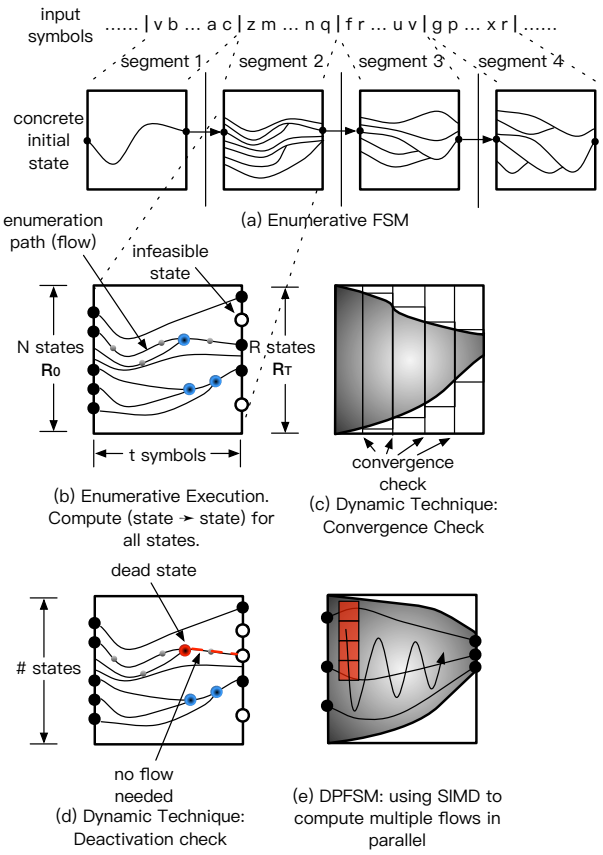


Fig. 2. Enumerative FSM

enumeration path computes the state→state mapping starting from one state.

A *flow* is a logical concept that computes an enumeration path. Let the number of flows to perform an enumerative execution be $R$. Initially $R$ is equal to N, the total number of states in an FSM. Figure 2 (b) also indicates $R_0$ and $R_T$, $R$ at the start and end of a stage. To compute multiple enumerative paths, the component implementing the state→state is time-multiplexed among all flows.

In enumerative FSM, the convergence property implies that $R$ is non-increasing. Intuitively, it is because when two states transition to a same state, then the paths for the two will be the same afterwards. The concept is illustrated in Figure 2 (c). At the output of a segment, there are some infeasible states (marked as the empty dots). Another concept is the dead states, which are deemed not to lead to a pattern matching. The enumeration path terminates after reaching a dead state, which is shown in Figure 2 (d).

Data Parallel FSM (DPFSM) [20] is the first to study enumerative FSM. The implementation uses SIMD instructions in CPU to enable parallel computation of multiple flows. It performs convergence check during execution, thereby reducing R dynamically. This approach is illustrated in Figure 2 (e), each compute unit is marked as a red square. The compute resources (e.g., SIMD or SMs in GPGPU) are devoted to the computations corresponding to the shaded region.

In principle, smaller $R_0$ and $R_T$ lead to faster execution. To accelerate processing, the general guidance is to reduce R before the computation (to get a smaller $R_0$) and during enumeration (to get a smaller $R_T$). Besides convergence check, dead state elimination (i.e., deactivation) is another *dynamic* optimization to reduce $R_T$, The recent works also perform different *static* optimizations to reduce $R_0$. Next, we discuss their key ideas based on the just discussed terminology.

### C. Lookback Enumeration

Lookback Enumeration (*LBE*) [23], [22], [19], [21] is illustrated in Figure 3. The key idea is to "lookback"-reduce the initial N states to a smaller $R_0$ using suffix symbols of previous input segment.

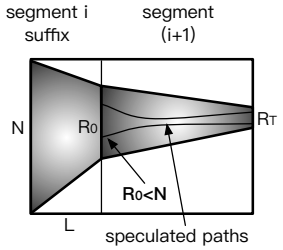


Fig. 3. Lookback Execution.

LBE of each segment is divided into two phases. The first step is lookback, which performs enumerative execution starting from N flows using L prefix symbols in the suffix. The result is a smaller set of possible starting states ($R_0 \leq N$) for the current segment. The second step is to compute the enumeration paths of one or more predicted states in $R_0$. If the predicted starting state(s) contain the actual final state of the previous segment, the speculation is successful; otherwise this segment is re-executed sequentially with a concrete state.

Without prediction, LBE is essentially the same as the basic enumerative FSM but with longer input symbols (with L suffix symbols). Recent works [23], [22] use probabilistic analysis and profiling to study how to select starting states from $R_0$ to reduce re-execution. [19], [21] considered choosing multiple states from $R_0$.

It is worth noting that the existing probabilistic methods are designed for software implementation. For example in [22], the "stochastic speculation scheme" requires tracking the feasibility (probability) of each state at runtime. Such computation can be extremely inefficient on hardware because at least we need to update floating point values.

### D. Parallel Automata Processor

Parallel Automata Processor (PAP) [25] supports NFA enumeration using Automata Processor (AP) [29]. In PAP, $R_0$ flows in a segment are computed by a hardware component for state→state with time-multiplexing. Targeting NFA, PAP needs to compute state→set mapping, so one-hot encoding is also used to represent multiple active states. The key difference between NFA and DFA is that R is not monotonically decreasing. Based on the results in [25], over a long symbol sequence, R still decreases.

An important effort of PAP is to reduce $R_0$. PAP proposed four *static* optimizations. They are illustrated in Figure 4. The first optimization is range guided input partition, by which $R_0$ can be reduced by cutting inputs at frequent symbols with small feasible range (Figure 4 (a)). The second optimization is connected component analysis, in which all feasible states for a stage are partitioned into connected components (CCs). Two

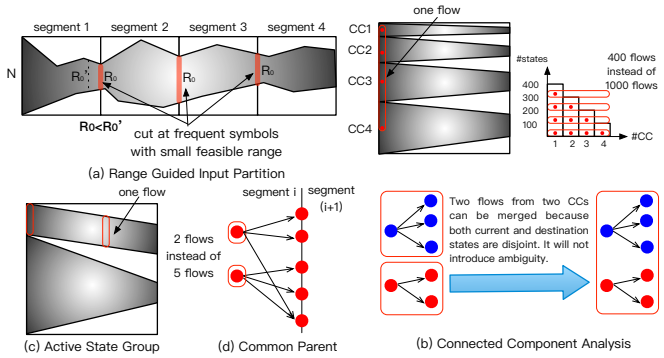


Fig. 4. Parallel Automata Processor Optimizations

flows from two CCs can be merged because both current and destination states are disjoint. It allows one flow to compute several state→state transitions in parallel. In Figure 4 (b), the total number of states is 1000, which is partitioned into four CCs with sizes: 400,300,200,100. Thus, only 400 flows are needed, a reduction of 600 flows. Figure 4 (c) shows active state group optimization. Since NFAs usually have several states which are always active due to self-loops on all possible symbols, they can be assigned into one flow. Figure 4 (d) shows common parent optimization, by which the number of flows is reduced to the number of parents. In the example, the intuition is that, if the boundary of the two segments have been one symbol earlier, only 2, instead of 5, flows are needed.

### E. Drawbacks

While recent works achieve certain speedups, they share a common assumption: computing each enumeration path through state→state. For PAP, although several state→state transitions can be computed in parallel in certain cases (see Section II-D), it still stores state→state mapping in memory so that they can be used when the computation for the corresponding states are resumed in time-multiplexing. The large amount of state→state computation for the enumeration paths poses a great challenge. Although PAP can statically parallelize and reduce flows, the optimizations need to ensure correctly computing state→state and achieve limited improvements. We explore a new enumerative FSM design, which is inspired by DPFSM and PAP but uses set(N)→set(M) primitive, — an unique opportunity enabled by the one-hot encoded architecture.

## III. SET(N)→SET(M) Computation Primitive

This section discusses the insight, definition and application of set(N)→set(M), the novel and key computation primitive that leads to efficient implementation of enumerative FSM.

### A. Motivation

The memory-centric Automata Processor (AP) [29] accelerates finite state automata processing by implementing NFA states and state transitions in memory. Figure 5 shows the AP architecture, which accepts the input symbols one at a time and performs state transitions. The current states are represented using one-hot encoding since multiple states can be active in

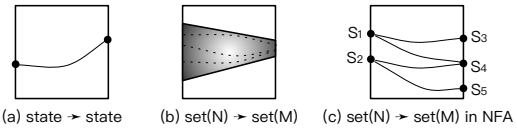


Fig. 6. New Computation Primitive: State Set Transition

an NFA. The computation is divided into two phases: *1) state matching*, which reads the row based on the current input symbol, a bit one means that the state matches the symbol; *2) state transition*, which generates the next states in *active mask*. The state transition is only performed when a state can match the input symbol and is active. The active mask is initially set to the start states. In one-hot encoding, all active mask bits for all states can be *independently* set in a given cycle.

The general architecture can implement both NFA and DFA. In NFA, multiple bits in active mask can be set and multiple state transitions can be performed in parallel. In DFA, only one bit in active mask is set and one state transition is performed.

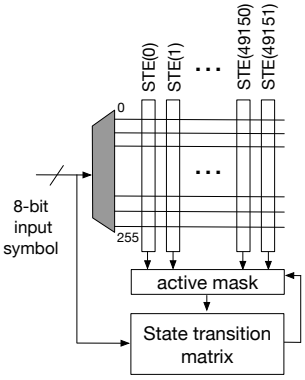


Fig. 5. Automata Processor

We consider the scenario that multiple bits in active mask are set. Clearly, the hardware implementation does not require any change. However, it can no longer compute the state→state transitions for individual state in DFA. For example, if we have two states $S_0$ and $S_1$, and after an input symbol 'a', they transition to two different new states, $S_0 \rightarrow S_2$, $S_1 \rightarrow S_3$. If the bits for $S_0$ and $S_1$ in active mask are both set, then the bits for $S_2$ and $S_3$ are set in the updated active mask. From this information, we know the state set $\{S_0, S_1\}$ transitions to $\{S_2, S_3\}$, but we do not know the specific state mapping (which state transitions to which).

We define this new computation primitive as set(N)→set(M), which transitions a state set with N states to another state set with M states. The concept is shown in Figure 6. The curve in Figure 6 (a) is an enumeration path, a sequence of state→state. The dotted curves in Figure 6 (b) are the enumeration paths for the N starting states. However, set(N)→set(M) cannot provide the states sequences in each path, but can just give the M states that the N starting states are mapped to. As shown in Figure 6 (c), two active states $\{S_1, S_2\}$ are transitioned to $\{S_3, S_4, S_5\}$, but the information that $S_1$ is mapped to $\{S_3, S_4\}$ is lost. Next, we show that in a special case, set(N)→set(M) is very useful.

### B. Application to Enumerative FSM

The most natural application is to compute the lookback in LBE. Referring to Figure 3, the existing LBE uses state→state to compute the enumeration paths on the suffix (length L) of the previous segment. We only need to know the state set after looking back the suffix ($R_0$), but there is no need to know

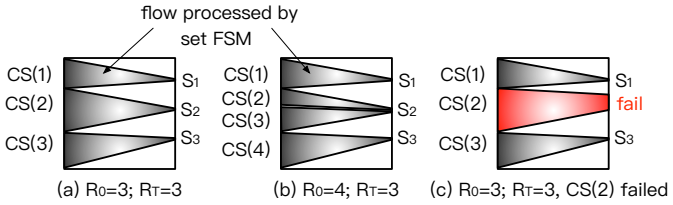


Fig. 7. CSE Approach

how these states are reached. In this scenario, set(N)→set(M) perfectly matches the goal: we can directly reduce state set of size N at the beginning of the suffix to $R_0$ with overhead of just computing one enumeration path of length L.

Most importantly, set(N)→set(M) can compute enumerative paths in a special case. If M = 1, all N states converge to the same state, so all N enumerative paths are computed in parallel. Moreover, it is achieved with the same hardware cost. The convergence property of enumerative FSM ensures M < N. While it is unlikely that all N states of an FSM can converge to a single state, it is highly possible that a subset of N states can converge to a single state. Next, we describe out solution based on this insight.

## IV. CSE Approach

This section presents *CSE*, Convergence Set Enumeration. We first explain the insights and then focus on the convergence set generation and re-execution algorithm.

### A. Insights

CSE uses set(N)→set(M) as the building block to construct enumerative FSM. To increase the probability of converging to one state, we partition all states ($S$) into $n$ *convergence sets*, {CS(i)|i ∈ [1,n]} such that $S = \bigcup_{i=1}^{n}$ CS(i). We speculate that each CS(i) converges to one state, if it is true, set(N)→set(M) computes |CS(i)| enumeration paths in parallel. Otherwise, one or more stages need to be re-executed. Figure 7 illustrates the CSE approach.

In Figure 7(a), we partition S states into three convergence sets: CS(1), CS(2) and CS(3), so $R_0$ is directly reduced to 3. At the end of the segment, each CS(i) converges to one state successfully, so $R_T$ is also 3. We mentioned earlier that smaller $R_0$ and $R_T$ lead to faster execution, and in this case, $R_0$ is directly reduced to the *lower bound*, the same as $R_T$. It is difficult to achieve with the optimizations in PAP.

Figure 7(b) shows a more conservative partition. Instead of partitioning into three CSs, we partition S states into four CSs ($R_0$ = 4). In the end, each CS(i) converges successfully into one state and CS(2) and CS(3) converge to the same state $S_2$, so $R_T$ is equal to 3. In this example, $R_0$ is not reduced to the lower bound, but is still significantly less than N.

Figure 7(c) shows a different scenario. Here, CS(2) does not converge to a single state, so we do not successfully compute the state→state mapping of all states in CS(2). This example illustrates an incorrect partition and suggests a possible re-execution, which will execute state→state on the concrete state from previous stage. In certain cases, re-execution might be avoided: if the concrete output state from the previous stage belongs to CS(1) or CS(3), then the re-execution is unnecessary. The details of re-execution algorithm is described in Section IV-B2.

The above discussion is about executing one stage in CSE. After finishing the stage, we need to composite the flows of all stages, i.e., distinguish the true/false state of enumerative execution in each stage. PAP simply selects the flow starting from the concrete result state from previous stage as the true one, because in state→state computation, results starting from all states are available. CSE does it similarly *at convergence set granularity*. We select the result of the corresponding CS of the concrete state. If CS converges, all states (including the true concrete output state) produce the same report state. If CS diverges, we will perform re-execution with the concrete state. [1] Therefore, CSE can produce exactly the same result state and output report as performing sequential execution.

However, in some cases, we not only need the terminal state but the whole state transition path. Because set(N)→set(M) did not keep the path of individual state, CSE can not distinguish the state transition path within a convergence set. We can still recover such path information with another sequential execution. In real-world services (e.g., FSMs in network intrusion detection), *computing the terminal state is latency sensitive while state transition path is not*. Thus, CSE can still accelerate FSM processing by speeding up latency critical tasks.

### B. Convergence Set Prediction

The quality of convergence sets generated is a key factor determining performance. With fewer convergence sets (smaller $R_0$), the successful execution is faster because the number of flows that need to be enumerated by set(N)→set(M) is smaller. However, it is more likely to trigger re-execution, which incurs extra latency. If we are more conservative and generate more convergence sets (larger $R_0$), we may end up performing certain redundant computations similar to enumerative FSMs based on state→state transition. Figure 7(b) is an example of this case.

We propose a profiling-based prediction method and partition refinement algorithm to merge partitions. By carefully selecting the merge strategy, our convergence set prediction can produce both *concise* (small number of converge sets) and *accurate* (low re-execution rate) state partition for real-world FSM applications.

*1) Profiling:* The idea of profiling is intuitive: we can use synthetic input sequences to discover the convergence behavior. The profiling input is randomly generated based on two characteristics in real input strings: string length and symbol range. 1) String length: While input strings may have varying sizes, real-world applications can always split a string into independent strings of similar length. Section V-B describes the input length of each benchmark used in evaluation. 2) Symbol range: Symbol range can be obtained

[1] If at time point during the execution, CS arrives at more than one reporting/accepting states, this CS will diverge at the end of this segment and we also need re-execution to distinguish report states.

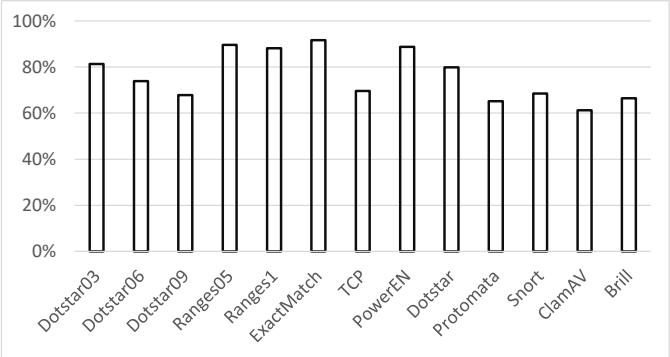


Fig. 8. Maximum Frequency Partition (MFP)

from FSM specification. For example, some FSM benchmarks only accept visible ASCII codes, thus the profiling symbols are sampled on a subset of ASCII.

After generating profiling input, we emulate the execution. One input will produce one convergence partition. We count the frequency of appearances of distinct partitions. The more frequent a partition appears, the more likely the real-world inputs could converge with convergence sets in this partition.

In the evaluation, we profile with 1k input strings. The profiling time is less than 5 minutes for FSM benchmark on one PC. We also tried to profile with 10k input strings and the profiling result does not change. It is because the frequency distribution has unnoticeable change across *all* benchmarks. Note that although the profiling strings are randomly generated, the profiling result is *consistent* across *all* benchmarks. Since only one convergence partition has to be selected for a given FSM, we can simply choose the *maximum frequency partition (MFP)* to minimize the probability of re-execution.

*2) Convergence Partition Merge:* Among the many partitions generated from profiling, we found that even MFP does not achieve sufficiently high frequency. Figure 8 shows the frequency of MFP after profiling. For example, frequency of MFP in *Clamav* (one of the benchmarks we used) is only 61%. These MFPs are far from "accurate": If we choose them, the convergence sets generated will diverge in around 39% executions, leading to frequent re-execution and performance degradation.

Instead of simply choosing MFP with low accuracy, we propose to refine partitions: merging multiple partitions to a new partition with refined subsets. This is based on a classic algorithm known as partition refinement [30] (Figure 10). It takes two two partitions $P1$ and $P2$ as input and output a merged partition. Note that the refinement is commutative operation, so the order of selecting elements in $P1$ or $P2$ does not affect the result. Output $P$ will have the following property: any pair of states both belonging to the a convergence set in $P1$ and a convergence set $P2$ will fall in a convergence set in $P$. It implies that if an input string can converge in $P1$ or $P2$, it will converge under $P$. Thus, the frequency of $P$ is the sum of $P1$ and $P2$.

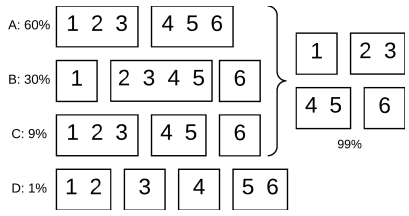


Fig. 9. Merge Example

**Require:** $P1$ contains subsets $\{S1_1, ..., S1_m\}$
**Require:** $P2$ contains subsets $\{S2_1, ..., S2_n\}$
1: **for** $S2$ *in* $P2$ **do**
2:  **for** $S1$ *in* $P1$ **do**
3:  Intersection $I \leftarrow S1 \cap S2$
4:  Difference $D \leftarrow S1 \setminus S2$
5:  **if** $I \neq \varnothing$ **then**
6:  Split $S1$ into $I$ *and* $D$
7:  **end if**
8:  **end for**
9: **end for**
10: **return** $P1$

Fig. 10. Partition Refinement Algorithm

In this case, we say that partition $P$ *covers* $P1$ and $P2$. In essence, we trade off the increased number of convergence sets for the benefit of increased frequency. Figure 9 is an example of 4 partitions after profiling. To merge A, B and C, a new partition of 4 subsets is created. If we only want to merge A and C, the result is still C.

Based on the above algorithm that merges two partitions, we can merge all partitions that appear in profiling. Our objective is to find a refined partition with higher frequency without increasing the number of subsets significantly. For simplicity, we propose an effective heuristic merge strategy:

- Merge all partitions if one covers another. The reason to do this compatible check first is that such merge does not increase the number of subsets.
- Merge from partition with higher frequency.
- Stop when the merged partition reaches some cut-off frequency.

We explored several cut-off frequency, from 90% to 100% (i.e. merge all partitions that appear in profiling). For most benchmarks, even merging to 100% will not increase the number of subsets drastically. However, *Protomata* using 100% MFP will generate a partition with 61 subsets, a prohibitive number of subsets. The cut-off frequency used in evaluating performance is shown in the MFP column of Table I and detailed discussion on selecting the best merge strategy is in Section VI-E.

### C. Correctness Guarantee with Re-Execution

**Formalization.** To uniformly specify the transitions in each segment, we define a new type called *State Type (ST)*, which can be either a *state (st) or a convergence set (CS)*.

Assume we have M segments and N convergence sets, then after all segments are executed in parallel based on the corresponding input symbols, we have m *transition functions, T: ST* $\rightarrow$ *ST* defined as follow:

$$T_s(CS(i)) = \begin{cases} st(j) & \text{if CS(i) converges} \\ \bigcup_{j=1}^{k} CS(j) & \text{otherwise} \end{cases}$$

Each segment has such transition function for each convergence set, so we have i = 1,2,..,n. In addition, the transition

function for each segment is different and depends on its input, so we have s = 1,2,..,m. If a CS does not converge, the output states can be included in one or more convergence sets. Our definition includes both cases: k = 1 when converging to one CS, k $>$ 1 otherwise. In the first segment, the starting state is concrete. We can trivially modify the function to have special treatment for this segment: $T_1(s(i)) = s(j)$, which means that it always performs the normal state→state.

After all segments finish the execution, we essentially need to compute the composition function, $T_1 \circ T_2 \circ ... \circ T_m$. *If the outcome is a single state, then no re-execution is required*, otherwise, some segments need to re-execute. Before presenting how to identify the re-execution segments and organize the procedure, we first discuss function composition.

We consider the rules to compose two functions $T_i$ and $T_{i+1}$. Note that we only need to compose the transition functions for the consecutive segments since FSM has a linear structure. $T_{i+1}$ can have several possible input types.

(1) The input of $T_{i+1}$ (output of $T_i$) can be $\bigcup_{j=1}^{k} CS(j)$ based on our definition. In this case, we have $T_i \circ T_{i+1} = T_{i+1}(\bigcup_{j=1}^{k} CS(j)) = \bigcup_{j=1}^{k} T_{i+1}(CS(j))$. The output is the union of the results of each convergence set's transition function.

(2) Based on the result in (1), the input of $T_{i+1}$ (output of $T_i$) can be one state, multiple states or a mix of different states and convergence sets. Let us consider a case that the input is k states. We first convert the states to convergence sets, find the union and then apply (1). Specifically, for states st(j), where j=1,2,..,k, we find the convergence set CS(j) that each state st(j) belongs to. Then, we have $T_i \circ T_{i+1} = T_{i+1}(\bigcup_{j=1}^{k} st(j)) = T_{i+1}(\bigcup_{j=1}^{k} CS(j)) = \bigcup_{j=1}^{k} T_{i+1}(CS(j))$. If states are mixed with convergence sets, the procedure is the same because the states are already treated as convergence sets.

**Re-Execution Algorithm.** As discussed earlier, if the output of $T_m$ (the last segment) is a single state, re-execution is not required. It is easy to see that such condition can be satisfied even if some previous segments do not converge. Our formalization can naturally capture this behavior by the transition function of each segment. However, this condition does not tell how to identify the segments for re-execution, which is discussed below.

(1) Basic approach. Re-executing a segment means that a concrete output state is generated when the concrete input state is known. Obviously, it requires that the output of the previous segment is a single state. For example, we can always re-execute segment 2 because the first segment can always produce a concrete output state. Thus, a simple approach is that, when re-execution is required, segment 2 to segment m sequentially perform the re-execution. This guarantees that the output of segment m is a concrete state.

(2) Last-concrete optimization. We can improve the basic approach by realizing the fact that, even if the input of a segment is not a single concrete state, the output can be. The next segment can perform the re-execution based on the concrete output. In the segment chain from 2 to m, there could be multiple such segments, we call them *concrete points*. They

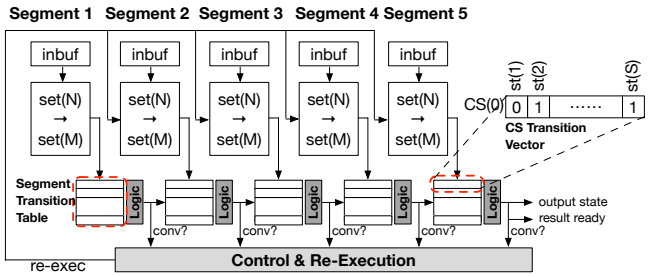


Fig. 11. CSE Hardware Implementation

are depending on the input symbols of a segment, so it can be only determined dynamically. The re-execution is always possible after these points. To ensure that the output of the last segment is a concrete state, we can perform *backward* checks through the segment chain (segment m → (m-1) → ...), and find the first concrete point. Let that point be segment r (r $<$ m), the re-execution can be performed sequentially from segment r to segment m. We call it "last-concrete" optimization.

(3) Opportunistic transition function re-evaluation. After re-executing the segment r, instead of directly re-executing all successor segments sequentially, we can "re-evaluate" the transition functions $T_{r+1}$, $T_{r+2}$,...,$T_m$ in sequence with the changed input of $T_{r+1}$. Since these functions are already computed in parallel in all segments, re-evaluating the functions is faster than the actual re-execution which depends on input length. There are two benefits: *a)* With function re-evaluation, it is possible that $T_m$ produces a concrete output, and no further re-execution is needed. This situation can be reached faster than re-executing multiple segments. *b)* Even if $T_m$ still do not converge, after function re-evaluation, we may identify a different last-concrete point (segment r') "later" than segment r, where r' $>$ r, using the same method. Essentially, we can skip re-execution of segments between r and r'. The hardware implementation to be discussed next is based on the most advanced design.

**Hardware Implementation.** Figure 11 shows the hardware implementation based on the re-execution algorithm with opportunistic transition function re-evaluation. Here, there are five segments. The input buffer of each segment stores the input symbols until no re-execution is needed. At this point, the input buffer of all segments are cleared together. The *Control & Re-Execution* module determines the segment required to re-execute and controls the timing. Note that even in re-execution, the number of cycles needed is predictable, so the cycles that different operations are performed can be easily determined, therefore, the control logic is not complex.

The interface between each segment and Control & Re-Execution module is *Segment Transition Table*, which essentially specifies the transition functions for each convergence set. The table is composed of n *CS Transition Vectors*, each corresponds to one convergence set. The length of the vector is the same as the number of states in the FSM, the vector is simply copied from the active mask for each convergence

set after the last input symbol for a segment is processed [2]. To make the logic design clean, we choose not to directly indicate the vector for CSs, the simple operation to generate CSs from states is implemented in the *Logic* module after the segment is finished. The Logic module of a segment generates the output, which are connected to the Logic module of the next segment. It also outputs a signal (*conv*) indicating whether this segment converges. Therefore, by chaining Logic modules of all segments together, we can get the final output state (if the output of the last segment is a single state) and a global result ready signal if it is true.

The Control & Re-Execution module takes all *conv* signals and finds the last-concrete point, which is a simple backward search for the first *conv* signal set to 1. The re-execution is warranted when the *conv* of the last segment is not set, and the re-execution signal (an M-bit vector) is sent to all segments. Only one bit is set among the M bits, indicating the segment selected to be re-executed (found by the backward search). The re-evaluation can be also performed by the Logic modules of the re-executed segment (r) and its successors, to segment m. The logic is the same as chaining all segments, except that the segment sequence is shorter, from segment r $\rightarrow$ m, instead of from segment 1 $\rightarrow$ m. Further re-execution may be necessary and can be determined by the same logic.

## V. Evaluation Methodology

In this section, we evaluate different enumerative FSM designs across a wide range of FSM benchmarks including *ANMLZoo* and *Regex* suites. These benchmarks contain multiple real-world FSMs and input sequences. We will first describe environment setup in detail and how we validate our results with PAP.

### A. FSM Benchmark

*Regex* [28] benchmark consists of both real-world and synthetic regular expressions. *ExactMatch* represents the simplest patterns which may appear in a rule-set. Dotstar contains a set of regular expressions matching wildcard ".*". The *Ranges* rulesets (*Range01*, *Range5*) contain character ranges (randomly selected between the different \x and \x+ groups), with average frequency of 0.5 and 1 per regular expression. The *TCP* regular expressions filter network packet header before deep packet inspection.

*ANMLZoo* [27] is a much more diverse benchmark of automata-processing engines. *Brill* is short for Brill rule tag updates in the Brill part-of-speech tagging application. *ClamAV* is an open source repository of virus signatures intended for email scanning on mail gateways. It is a subset of the full signature database. *Dotstar* is a set of synthetic automata generated from 5%, 10% and 20% ".*" probability regular expression rulesets. *PowerEN* is a synthetic regular expression benchmark suite developed by IBM. *Snort* is an open-source network intrusion detection system (NIDS) software. It monitors the network packages and analyzes them against a rule set defined by the community and the user. *Protomata* converts the 1307 rules based on protein patterns in PROSITE into regular expression patterns. *Regex* and *ANMLZoo* provide two representations for the same FSMs: NFA and regular expression. While PAP focuses on NFA, we use the regular expression and convert it to DFAs with *RE2* [31], an open-source regular expression library. State blow-up is possible when compiling to DFA. However, this is not the case for *Regex* and *ANMLZoo* benchmark (In *ANMLZoo* paper, both DFA and NFA representations are evaluated). Table I shows characteristics of converted FSMs.

[2]The active mask for each convergence set needs to be saved and restored during context switch, similar to PAP

TABLE I
Benchmarks

| Benchmark | #FSM | #State | #Half-Core Per Segment / #Segment | L | MFP |
|---|---|---|---|---|---|
| Dotstar03 | 300 | 19038 | 1/16 | 30 | 100% |
| Dotstar06 | 300 | 24821 | 1/16 | 30 | 100% |
| Dotstar09 | 299 | 29442 | 1/16 | 30 | 99% |
| Ranges05 | 300 | 13391 | 1/16 | 20 | 100% |
| Ranges1 | 299 | 13324 | 1/16 | 10 | 100% |
| ExactMatch | 300 | 13212 | 1/16 | 10 | 100% |
| TCP | 733 | 38107 | 1/16 | 30 | 100% |
| PowerEN | 2860 | 73140 | 1/16 | 20 | 100% |
| Dotstar | 3000 | 119468 | 2/8 | 20 | 100% |
| Protomata | 2340 | 1949014 | 2/8 | 20 | 99% |
| Snort | 3379 | 152616 | 3/5 | 10 | 99% |
| Clamav | 515 | 185181 | 3/5 | 40 | 99% |
| Brill | 2050 | 256349 | 3/5 | 50 | 100% |

### B. FSM Input

For *Regex* benchmark, we generate the input using trace generators provided by Becchi[28]. We set $p_m$ to 0.75, the probability that a state matches and activates subsequent states. For *ANMLZoo* benchmark, input traces have been provided for each application. PAP takes one input file as one input string. However, for applications in *ANMLZoo*, the input can be easily split up by *delimiter* symbols into smaller input with provably no dependencies. Here are two examples: In *Brill*, no matches can occur across sentence boundaries, so we can split the input file by periods. In *Snort*, one input file contains many packets, and processing them should be independent and done in parallel. In our evaluation, we use the same input file as in PAP but split them according to real-world practice. The performance number is averaged over all input strings. Note that in profiling convergence sets, we do not use any of the dataset mentioned above and only use random inputs.

### C. Environment Setup

We use VASim [27], the same automata simulator as utilized in PAP and we implement PAP optimizations described in Section II-D. VASim is a widely-used open-source library with fast NFA emulation capability. It also supports multi-threading and therefore is able to partition input stream and process them simultaneously. We divide the input stream into segments and execute each flow using context switch. We utilize different optimizations mentioned in PAP such as Range Guided Input

TABLE II
PARALLEL ENUMERATIVE FSMS EVALUATED

| FSM | Basic FSM | Static Optimization | Dynamic Optimization |
|---|---|---|---|
| Baseline | state FSM | NA | NA |
| LBE | state and set FSM | NA | lookback |
| PAP | state FSM | Four optimizations in Section 4 | convergence check and deactivation check |
| CSE | set FSM | convergence set prediction | |

Partition, Deactivation and Dynamic Convergence. We also take into consideration the overhead of path-decoding part which varies with the benchmark. We implemented a baseline sequential FSM and three enumerative FSMs for comparison. Table II shows the name of each together with the hardware building block, static and dynamic optimizations applied. Note that for LBE, we use set FSM to perform the lookback, so it is better than the basic LBE. Also, we use LBE without prediction since the probabilistic methods are not suitable for hardware implementation.

We assume that enumerative FSM designs are running on Micron Automata Processor (AP) [29]. In AP, half-core is the smallest unit of parallelization. AP in the current generation has 4 ranks, while each rank contains 16 half-cores. We choose to run on 1 rank (16 half-cores). For most benchmarks, one segment is assigned to one half-core. According to PAP, some benchmarks are densely connected that AP compiler will place these FSMs on multiple AP half-cores. We put the same resource constraint on LBE and CSE. Table I lists the number of half-cores assigned for each segment and the total number of segments. We estimate that sequential FSM on AP processes 1 symbol per cycle (7.5ns). Context switching between flows requires 3 cycles. And we assume that it takes 1 cycle for dynamic convergence check of every two flows.

In summary, we adopt the same evaluation methodology as in PAP, except that we split the input differently according to real-world practice. To further verify that we implement the static and dynamic optimizations correctly, we make sure that our $R_0$ and $R_T$ number is the same or less than those reported in PAP paper.

## VI. EXPERIMENTAL RESULTS

In this section, we first present the speedup of different enumerative FSM designs. We will then analyze the sources of speedup by looking at the enumeration overhead $R_0$ and $R_T$ and explain how we explore the parameter space of LBE and CSE to achieve the best performance. Finally, we investigate the predictive power of our profiling based convergence set prediction method by comparing the re-execution rate with different cut-off partition coverage frequency.

### A. Performance

Figure 12 demonstrates our proposed CSE performance compared to baseline, LBE, PAP and ideal speedup. The baseline throughput is 1 symbol per cycle. The ideal throughput is 1 symbol per cycle in each segment, so the speedup over baseline equals total number of segments. For LBE and CSE, parameters like lookback length and merge strategy are shown

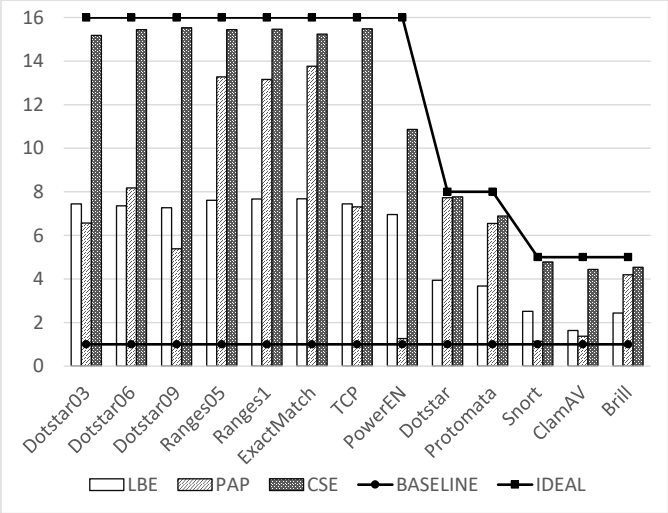


Fig. 12. Speedup

in Table I. Our proposed CSE is better than LBE and PAP in all applications. On average, CSE speedup is 2.0x to LBE and 2.4x to PAP.

CSE achieves near ideal speedup in all applications except *PowerEN*. PAP also reaches near 16x speedup in *ExactMatch*, *Ranges1* and *Ranges5*. Note that CSE still outperforms PAP by a very small margin (4.7% on average). However, the speedup of PAP is not consistent. For *Snort* and *ClamAV*, PAP speedup is 1.14x and 1.37x respectively. Considering that ideal speedup should be 5x, PAP is only a little faster than baseline. Our design offers consistent speedup of 4.9x and 4.4x respectively.

### B. Initial Flow Number $R_0$

Figure 13 shows $R_0$ for each application evaluated in this paper. Recall that $R_0$ is an intuitive indication of initial enumeration overhead.

For CSE, all applications has a small $R_0$. In fact, except *PowerEN*, $R_0$ is reduced to 1 dynamically within less than 10 symbols (20 cycles). When $R_0$ becomes 1, it follows that $R_T$ is 1 as well. During the whole execution process, there is only one flow, no enumeration overhead and no time multiplexing. There is no doubt that such applications will match the ideal performance. As for *PowerEN*, it takes 565 symbols for $R_T$ to become stable. The large enumeration overhead makes CSE *PowerEN* much lower than ideal speedup. As we have mentioned, PAP also performs well in 3 applications. The static optimization of PAP has reduced $R_0$ to 1 for the applications. Sinilarly, PAP has no enumeration overhead when running these applications.

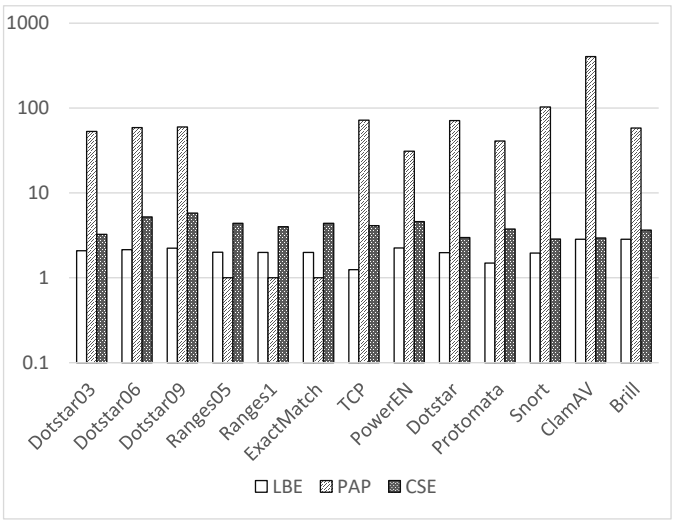


Fig. 13. $R_0$

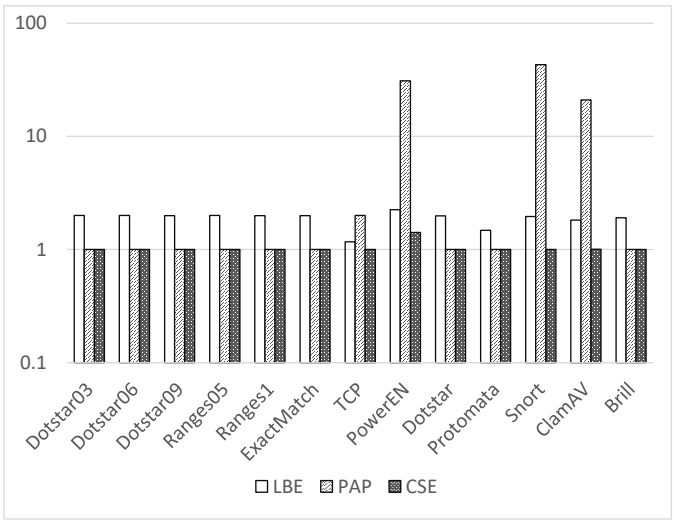


Fig. 14. $R_T$

Here is one intricate question: why CSE still outperforms PAP in the 3 applications when all of them have small $R_0$? The problem is in the "range guided input partition" optimization of PAP (Figure 4(a) and Section II-D). The input string is divided by a certain symbol to make sure a smallest feasible state range. This is a clever technique which greatly reduces $R_0$ from tens of thousands to hundreds. However, such special symbol boundary might appear at any position and one can not expect the symbol to divide the input string evenly. The longer segment will take more cycles to execute and determine the total execution cycle. In our CSE, we do not rely on this technique to reduce $R_0$ and always divide into equal segments. This accounts for the marginal 4.7% performance benefit.

Unfortunately, if we look at other applications, $R_0$ of PAP is larger than that of LBE and CSE. For example, *Snort* and *ClamAV* for PAP has 103 and 404 initial flows respectively, comparing to 2.9 and 2.9 for LBE and 3.9 and 4.8 for CSE. In PAP, the authors argue that large $R_0$ is not a problem. They will rely on dynamic optimization to reduce the number of active flows quickly. There are two problems with this claim: First, their observation is true based on input of 1 million or 10 million symbols. However, as we have discussed in the evaluation methodology (Section V-B), the input file should be split into independent pieces and processed in parallel. In practice, dependent input sequence length rarely exceeds ten thousand. The time spent on initial enumeration will become a major overhead, impacting the overall performance. Second, the capability of PAP dynamic convergence check is limited by some static optimization performed. We will discuss it in the following section.

### C. Last Flow Number $R_T$

Figure 14 demonstrates $R_T$. Due to dynamic optimization (convergence check and deactivation check), $R_T$ is always no greater than $R_0$. The value of $R_T$ tells us the enumeration flow number at the end of computation. $R_T$ and $R_0$ together give us some hint on FSM performance.

For CSE, $R_T$ becomes around 1 for all applications. There is almost no enumeration overhead at the end of computation.

For LBE, $R_T$ is 1.9 on average, which means that there are still about 2 active flows running in LBE and it takes 2 cycles to execute a symbol. $R_T$ of LBE confirms the fact that LBE performance will be lower than half of ideal throughput. The reason of smaller $R_T$ in CSE compared to LBE is the weak convergence condition. (Section IV-C)

However, in PAP, some applications have $R_T$ much larger than 2. While $R_T$ in *TCP*, *Dotstar*, and *Protomata* is 3, *ClamAV* and *Snort* have as many as 21 and 30 active flows at the end. $R_T$ clearly answers the question of why PAP is extremely slow in these 2 cases: dynamic optimization is not working effectively and there is large enumeration overhead.

From the experiment results, we can reveal the real cause behind inefficient dynamic optimization: the "connected component analysis" optimization in PAP. This optimization merge states in different connected components in the same flow. It is intuitively appealing to do so because it reduces $R_0$. (See Section II-D and Figure 4(d) for details) However, after merging, it becomes more difficult to merge these flows dynamically. Here is a simple example. There are two connected components and two states in each component, State 1 and 2 in component A and state 3 and 4 in component B. PAP will merge 4 states to 2 flows: A:(1,3) and B:(2,4). As for static optimization, $R_0$ decreases from 4 to 2. If we consider dynamic convergence, flow A and B will be merged only when state 1 and 2 converge *and* 3 and 4 converge. In general, if there are H connected components, flows will be merged only if all H pairs of states converge.

### D. LBE and Lookback Length

In the previous section, we have learned that the overall LBE performance is limited by $R_T$ to around half of ideal. However, we can still tune lookback length parameter to find the best performance for LBE. We explored 10 lookback lengths from 10 to 100. For simplicity, Figure 15 shows the speedup of 4 configurations.

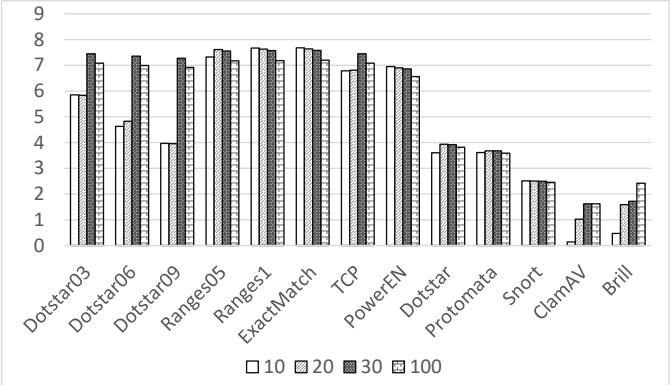


Fig. 15. LBE Speedup w.r.t. Lookback Length

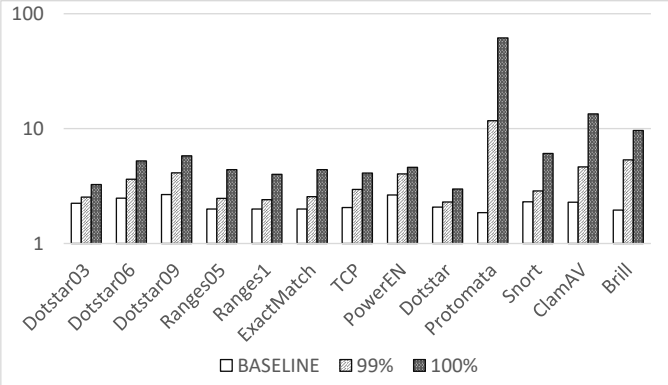


Fig. 16. CSE $R_0$ w.r.t. Merge Strategy

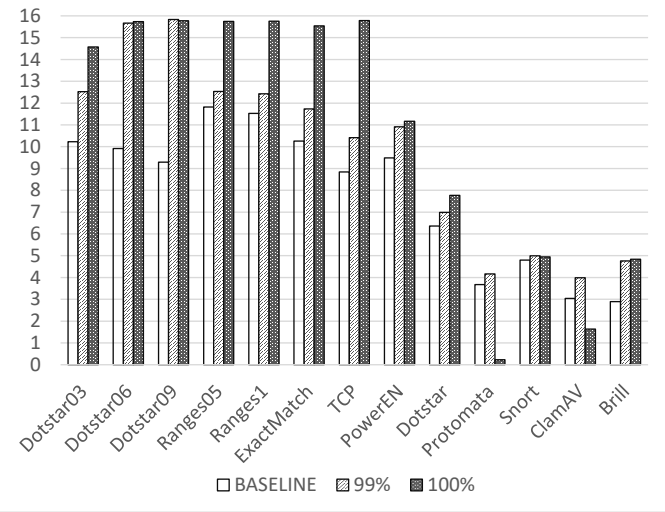


Fig. 17. CSE Speedup w.r.t. Merge Strategy

For applications like *Brill*, we can see that lookback is extremely beneficial with 5.2 times speedup. For *ClamAV*, lookback of 10 symbols will perform worse than sequential baseline, because $R_0$ is still large to introduce enumeration overhead. For all benchmarks, lookback prefix longer than 100 steps will bring about diminishing benefit or even slow-down, because $R_0$ cannot be reduced further and lookback cycles become unneglected overhead.

### *E. CSE and Convergence Set Generation*

As discussed in Section IV-B, convergence set generation is essential to CSE performance. We have known that profiling alone will not select MFP with a large frequency and partition merge is necessary. We evaluate the effectiveness of different merge strategies in this section.

Figure 16 shows the number of convergence sets in MFP. Note that the number is equal to $R_0$ in CSE, which determines the initial enumeration overhead. For almost all benchmarks except *Protomata*, merge to 100% will not increase $R_0$ significantly. For *Protomata* and *ClamAV*, merging all partitions that have shown up in profiling will increase $R_0$ to 61 and 13. This can possibly incur considerable overhead. Before partition merge, the average $R_0$ is 2.2. Merge to 99% and 100% increases $R_0$ to 3.9 and 9.9 respectively. We choose different merge strategies for different applications, shown in the last column of Table I, with $R_0$ equals 5.2 on average. The overhead introduced by merge can be handled by dynamic optimization checks in a few cycles. Meanwhile, the partition prediction becomes much more reliable and avoids many re-execution situations. Figure 17 shows speedup with respect to merge strategy. For *Protomata*, *ClamAV* and *Brill*, merge to 99% is more desirable. Partition merge technique proves to be effective across all benchmarks, boosting performance 1.4x over baseline on average.

### *F. The Predictive Power and Re-Execution*

Apart from $R_0$ and $R_T$, re-execution rate is another important factor in determining CSE performance. It is determined by the quality of predicted convergence sets. As we have merged MFP to a large frequency like 99%, partition is expected to re-execute with less than 1% possibility on our random test inputs. The key question is how well the convergence sets can predict the convergence behavior for *real* inputs. It can be measured by re-execution rate as a proxy. Figure 18 illustrates the re-execution rate for each application, evaluated by representative real-world input.

The MFP generated by basic profiling suffers from high re-execution rate. For *TCP*, *Protomata* and *Brill*, it is 23.2%, 19.0%, and 26.3% respectively. With the merge optimization, re-execution drops significantly. The re-execution rate does not deviate from our expectation, staying lower than 0.5%. On average, re-execution rate is only 0.2% and will have unnoticeable affect on the final CSE performance. It also shows that the convergence sets generated with profiling on random inputs indeed predict very well the convergence behavior of real inputs.

## VII. Related Work

CSE is the first work on parallelizing FSM by convergence set enumeration. Next, we review the related work in FSM acceleration in software, hardware and automata processor.

**FSM Software Parallelization.** FSM parallelization is a classical problem. [32] proposed DFA parallelization by parallel prefix sum. An optimized algorithm [33] reduced the time complexity to $O(nlogm)$ with m processors.

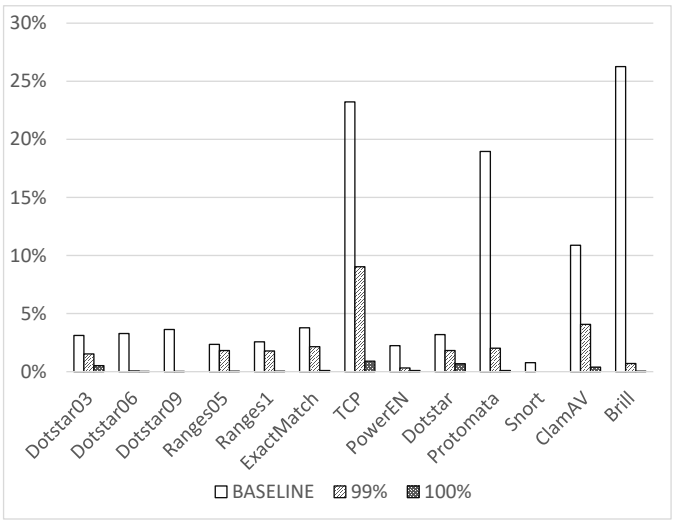


Fig. 18. CSE Re-execution rate w.r.t. Merge Strategy

Our work is inspired by the recent reasearch [20], [34] which performs enumeration on multicore architecture (with SIMD) based on prefix sums. It leverages state convergence to reduce the computation overhead. In this paper, we refer to this method as enumerative FSM, summarized in Section II-B. [26], [23], [22], [21] proposes "speculation" method and the concept of principled speculation to speculate on the possible start state. In 3, we introduce the idea and how we map it to hardware, LBE. Leveraging one-hot encoding and hardware features of AP, CSE proposes a new enumeration method based on set(N)→set(M) not included in any software parallelization literature.

**FSM Hardware Accelerators.** HAWK [17] is a hardware accelerator for ad hoc queries against large in-memory logs. HARE [18] targets regular expression matching against in-memory logs and optimizes for memory bandwidth. Unified Automata Processor (UAP) [35] is a new architecture for efficient automata processing. Unstructured Data Processor (UDP) [36] is a processor supports general data transformation including automata processing. Cache automation [37] proposes novel last-level cache for automata proccessing acceleration. This paper, as well as PAP, is distinguished from the above hardware accelerators because we consider data parallelization and dependency breaking.

**Automata Processor.** Automata Processor architecture [29] is an emerging research topic. Many applications and algorithms have been designed for AP, including high-energy particle physics [38] bioinformatics [39], pattern recognition [40], machine learning [41] and natural language processing [42]. The most related work is PAP [25], the state-of-the-art NFA parallelization on AP. Although the evaluation of this paper is based on AP simulation, the idea of convergence set enumeration is not specific to AP architecture but applies to all hardware accelerators and SIMD implementations.

## VIII. Conclusion

This paper proposes *CSE*, a Convergence Set based Enumerative FSM. Unlike prior approaches, CSE is based on a new computation primitive set(N)→set(M), which maps N states (i.e., a state set) to M states without giving the specific state→state mappings (which state is mapped to which). CSE uses set(N)→set(M) as an essential building block to construct enumerative FSM. Essentially, CSE reformulates the enumeration paths as set-based rather than singleton-based. We evaluate CSE with 13 benchmarks. It achieved on average 2.0x/2.4x and maximum 8.6x/2.7x speedup compared to Look-back Enumeration (LBE) and Parallel Automata Processor (PAP), respectively.

## Acknowledgment

We would like to thank the anonymous reviewers for their helpful feedback. This research is supported by the National Science Foundation grants NSF-CCF-1657333, NSF-CCF-1717754, NSF-CNS-1717984, and NSF-CCF-1750656. It is also partially supported by the National Key R&D Program of China (Grant No. 2016YFB0200100), National Natural Science Foundation of China (Grant No. 61722208). Jidong Zhai (zhaijidong@tsinghua.edu.cn) and Zhongzhi Luan (luan.zhongzhi@buaa.edu.cn) are the corresponding authors of this paper.

## References

[1] R. A. Van Engelen, "Constructing finite state automata for high performance web services," in *IEEE International Conference on Web Services*, Citeseer, 2004.

[2] C. Chitic and D. Rosu, "On validation of xml streams using finite state machines," in *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, pp. 85–90, ACM, 2004.

[3] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu, "Processing xml streams with deterministic automata," in *International Conference on Database Theory*, pp. 173–189, Springer, 2003.

[4] Y. Pan, Y. Zhang, K. Chiu, and W. Lu, "Parallel xml parsing using meta-dfas," in *e-Science and Grid Computing, IEEE International Conference on*, pp. 237–244, IEEE, 2007.

[5] Y. Pan, Y. Zhang, and K. Chiu, "Simultaneous transducers for data-parallel xml parsing," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–12, IEEE, 2008.

[6] Y. Chen, D. Che, and K. Aberer, "On the efficient evaluation of relaxed queries in biological databases," in *Proceedings of the eleventh international conference on Information and knowledge management*, pp. 227–236, ACM, 2002.

[7] S. Datta and S. Mukhopadhyay, "A grammar inference approach for predicting kinase specific phosphorylation sites," *PloS one*, vol. 10, no. 4, p. e0122294, 2015.

[8] I. Roy and S. Aluru, "Finding motifs in biological sequences using the micron automata processor," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 415–424, IEEE, 2014.

[9] Z.-G. Wang, J. Elbaz, F. Remacle, R. Levine, and I. Willner, "All-dna finite-state automata with finite memory," *Proceedings of the National Academy of Sciences*, vol. 107, no. 51, pp. 21996–22001, 2010.

[10] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pp. 93–102, IEEE, 2006.

[11] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved dfa for fast regular expression matching," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 5, pp. 29–40, 2008.

[12] C. L. Giles, S. Lawrence, and A. C. Tsoi, "Rule inference for financial prediction using recurrent neural networks," in *Computational Intelligence for Financial Engineering (CIFEr), 1997., Proceedings of the IEEE/IAFE 1997*, pp. 253–259, IEEE, 1997.

[13] R. Alur and M. Yannakakis, "Model checking of hierarchical state machines," in *ACM SIGSOFT Software Engineering Notes*, vol. 23, pp. 175–188, ACM, 1998.

[14] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *International Conference on Computer Aided Verification*, pp. 359–364, Springer, 2002.

[15] A. Petrenko, "Fault model-driven test derivation from finite state models: Annotated bibliography," in *Modeling and verification of parallel processes*, pp. 196–205, Springer, 2001.

[16] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley, "Efficient bdd algorithms for fsm synthesis and verification," *IWLS95, Lake Tahoe, CA*, vol. 253, p. 254, 1995.

[17] P. Tanden, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch, "Hawk: Hardware support for unstructured log processing," in *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pp. 469–480, IEEE, 2016.

[18] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 10 2016.

[19] J. Qiu, Z. Zhao, and B. Ren, "Microspec: Speculation-centric fine-grained parallelization for fsm computations," in *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pp. 221–233, IEEE, 2016.

[20] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 529–542, ACM, 2014.

[21] J. Qiu, Z. Zhao, B. Wu, A. Vishnu, and S. L. Song, "Enabling scalability-sensitive speculative parallelization for fsm computations," in *Proceedings of the International Conference on Supercomputing*, p. 2, ACM, 2017.

[22] Z. Zhao and X. Shen, "On-the-fly principled speculation for fsm parallelization," in *ACM SIGPLAN Notices*, vol. 50, pp. 619–630, ACM, 2015.

[23] Z. Zhao, B. Wu, and X. Shen, "Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation," in *ACM SIGPLAN Notices*, vol. 49, pp. 543–558, ACM, 2014.

[24] L. Jiang and Z. Zhao, "Grammar-aware parallelization for scalable xpath querying," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 371–383, ACM, 2017.

[25] A. Subramaniyan and R. Das, "Parallel automata processor," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 600–612, ACM, 2017.

[26] Z. Zhao and B. Wu, "Probabilistic models towards optimal speculation of dfa applications," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 220–220, IEEE, 2011.

[27] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, *et al.*, "Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pp. 1–12, IEEE, 2016.

[28] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 79–89, IEEE, 2008.

[29] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[30] R. Paige and R. E. Tarjan, "Three partition refinement algorithms," *SIAM Journal on Computing*, vol. 16, no. 6, pp. 973–989, 1987.

[31] "Re2." https://github.com/google/re2/.

[32] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.

[33] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM - Special issue on parallelism*, vol. 29, no. 12, pp. 1170–1183, 1986.

[34] M. Veanes, T. Mytkovicz, D. Molnar, and B. Livshits, "Data-parallel string-manipulating programs," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 139–152, ACM, 2015.

[35] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: The unified automata processor," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 533–545, ACM, 2015.

[36] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "Udp: A programmable accelerator for extract-transform-load workloads and more," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 55–68, ACM, 2017.

[37] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 259–272, ACM, 2017.

[38] M. Wang, G. Cancelo, C. Green, D. Guo, K. Wang, and T. Zmuda, "Using the automata processor for fast pattern recognition in high energy physics experiments - a proof of concept," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 832, no. C, pp. 219–230, 2016.

[39] I. Roy and S. Aluru, "Discovering motifs in biological sequences using the micron automata processor," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 13, no. 1, pp. 99–111, 2016.

[40] K. Wang, E. Sadredini, and K. Skadron, "Sequential pattern mining with the micron automata processor," in *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, (New York, NY, USA), pp. 135–144, ACM, 2016.

[41] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the automata processor," in *International Conference on High Performance Computing*, pp. 200–218, Springer, 2016.

[42] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the micron automata processor," in *Semantic Computing (ICSC), 2015 IEEE International Conference on*, pp. 236–239, IEEE, 2015.