2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Architectural Support for Efficient Large-Scale Automata Processing

Hongyuan Liu*, Mohamed Ibrahim*, Onur Kayiran†, Sreepathi Pai‡, and Adwait Jog*
*College of William & Mary †Advanced Micro Devices, Inc. ‡University of Rochester
Email: {hliu08,maibrahim}@email.wm.edu, onur.kayiran@amd.com, sree@cs.rochester.edu, ajog@wm.edu

***Abstract*—The Automata Processor (AP) accelerates applications from domains ranging from machine learning to genomics. However, as a spatial architecture, it is unable to handle larger automata programs without repeated reconfiguration and re-execution. To achieve high throughput, this paper proposes for the first time architectural support for AP to efficiently execute large-scale applications. We find that a large number of existing and new Non-deterministic Finite Automata (NFA) based applications have states that are never enabled but are still configured on the AP chips leading to their underutilization. With the help of careful characterization and profiling-based mechanisms, we predict which states are never enabled and hence need not be configured on AP. Furthermore, we develop SparseAP, a new execution mode for AP to efficiently handle the mis-predicted NFA states. Our detailed simulations across 26 applications from various domains show that our newly proposed execution model for AP can obtain 2.1× geometric mean speedup (up to 47×) over the baseline AP execution.**

I. Introduction

Many applications from domains such as genomics, malware detection, machine learning, and data analytics exhibit high levels of parallelism and are being accelerated through the use of *spatial architectures* that can exploit higher levels of parallelism than CPUs and also can significantly reduce data movement [1]–[9]. Spatial architectures usually consist of many interconnected processing elements that expose a very high degree of parallelism. Field-programmable gate arrays (FPGAs) are a classic example; the systolic-array-based Matrix Multiply Unit in Google's Tensor Processing Unit [10] is also a spatial architecture. One of the fundamental challenges with spatial architectures is that program size is a first order concern – there are a fixed number of states available and a spatial program must fit *completely* to begin execution. Otherwise, execution may be impossible, or in the best case multiple rounds of reconfiguration and re-execution may be required that can incur significant performance penalties [11]. On traditional von Neumann architectures, these issues can typically be handled by traditional mechanisms such as context switching and virtualization. However, the large size of the spatial program state means that these techniques do not transfer directly. Some of these issues affect also traditional architectures like the Graphics Processing Units (GPUs), whose massive parallelism also means that the amount of state is often prohibitively large to support efficient multitasking [12]–[15].

In this paper, we focus on providing architectural support for executing large-scale tasks on a special class of spatial architectures, known as automata processors (APs) [16]. These architectures accelerate the processing of Non-deterministic Finite Automata (NFA), a widely used representation of Finite State Machines (FSMs). FSMs are foundational in a wide range of application domains such as DNA sequence matching, network intrusion detection and machine learning [17]–[22]. Although many existing approaches [23]–[26] accelerate NFA processing on CPUs or GPUs, none of them completely solve the problem of data movement caused by irregular accesses due to NFA transition table lookups. In comparison, the AP executes NFAs natively and achieves significant performance speedup [27], [28] primarily because of: a) AP's massive parallelism where NFA states are mapped to columns in DRAM and can be activated independently and simultaneously in a given cycle; and b) AP's in-memory processing capability that handles NFA transitions without data movement between processor and memory.

An AP half-core (the basic processing unit of AP) can hold up to 24K states. However, in future, we expect that the NFA-based applications are going to scale both in terms of the number of NFAs per application and the number of states in an NFA. We expect this scaling from at least two aspects. First, in the era of big-data, the new applications will likely be mining even larger databases. For example, ClamAV [29], an anti-virus application, uses a variant of regular expression to specify each virus signature in an ever-enlarging database. The number of NFA states constructed from these signature regular expressions is consequently larger and state-of-the-art AP chips can no longer hold all the states at once. Second, a number of existing and newly proposed techniques enhance the throughput of FSM processing, but only by increasing the number of states. For example, existing AP supports duplicating NFAs to run multiple input symbol streams in parallel [30]; newly proposed Parallel Automata Processor [31] duplicates NFAs for parallel enumeration; and the Multi-stride NFAs [32], [33] transformation increases the number of transitions for processing multiple symbols at one step. Current AP chips execute these applications with a large number of NFAs/states by making independent batches of NFAs and executing each batch on the entire input while reconfiguring the AP between each batch.

To address the performance inefficiencies from repeated re-executions, we propose hardware and software support for large-scale NFA-based applications that currently do not fit in the AP chips. Our mechanisms are based on our key observation that not all states of an NFA are enabled during execution, and

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00078

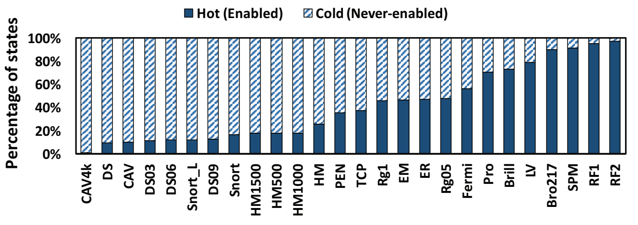


Fig. 1: A large portion of NFA states are cold (never-enabled) but are still configured on the AP leading to its underutilization.

hence need not be configured to the AP. Specifically, a large fraction of states unnecessarily take space in the AP chip but are not part of any state transitions. We refer to such never enabled states as *cold* states and the remaining (enabled) states as *hot* states. Figure 1 quantitatively shows our observation across 26 diverse applications [27], [34] sorted in the increasing order of their percentage of hot states (across all NFAs in an application). We find that on average 59% of states are cold and it can be up to 99% in applications such as `CAV4k`.

These observations can be explained by revisiting the way NFAs process inputs. NFA behavior is highly input dependent. A state can attempt to match a symbol of input only if it is enabled. In the most general case, a state is enabled only if at least one of its predecessor states matched a symbol of input (the exceptions being starting states, which are always enabled). A match indicates that the current input string is plausibly still a valid prefix of the regular language recognized by the NFA. States stop matching as soon as the input string is definitely not in the language. However, the AP must still process all input symbols as long as there is one state enabled (which is always true for an NFA with at least one starting state that is always enabled), thus leaving many states never enabled. Section III shows that this is indeed the case for the NFAs running on the AP.

Based on the above key insight, we first develop software-based mechanism to predict which states are cold and hence need not be configured on the AP. Next, we propose changes in the AP hardware to efficiently execute the mis-predicted cold states. To the best of our knowledge, this is the first work that proposes architectural support for efficiently executing large-scale NFA-based applications on the AP. In summary, this paper makes the following contributions:

- We demonstrate that a large number of NFA states are cold during execution but are still configured on the AP. This leads to its severe underutilization.
- We develop a prediction mechanism to classify the NFA states into *predicted hot* and *predicted cold* sets. We use properties of NFA execution to develop a simple and effective partitioning scheme based on a state's topological order and profiling information.
- We develop efficient hardware mechanisms to execute *predicted cold* states using a new *sparse execution mode* for the AP (called as SparseAP). Our detailed evaluation shows that we can achieve 2.1× geometric mean speedup (up to 47×) over the baseline AP execution across a wide range of 26 applications.

II. Background and Terminology

In this section, we provide a brief background on NFAs and their processing on the AP.

A. NFA-based Pattern Matching

An NFA is represented by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, where Q is a set of states, Σ is the alphabet (set of input symbols), Δ is a transition function which maps $\Sigma \times \Delta$ pairs to a new set of states, q_0 is the set of *starting* states, and F is a set of *accepting* or *reporting* states. Because there can be more than one possible state on a transition, such FSM is called *non-deterministic*. The NFAs used by APs are *homogeneous*[1]. These NFAs can be visualized as a directed graph where each node represents a state and each edge represents a state transition. Each state in the NFA has a *symbol-set* that represents what symbols can be accepted by this state. Each state has one or multiple successors connected by directed edges. In each step, the NFA has a number of *enabled states*. The *starting states* are *enabled* prior to the execution. The matching process is driven by a stream of input symbols. Each cycle, an enabled state compares the input symbol with its symbol-set for matching; when the symbol matches, the state is *activated*, and all its successor states are enabled in the next cycle. When a *reporting state* is activated, it generates a report showing that a relevant pattern has been observed in the input symbol stream.

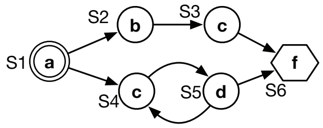


Fig. 2: A homogeneous NFA that accepts regular expression `a((bc)|(cd)+)f`: the doubled circle represents starting state and the hexagon represents reporting state.

Figure 2 shows the NFA of the regular expression `a((bc)|(cd)+)f`. At first, the starting state S_1 is enabled. `abcf` is the input symbol stream. `a` activates state S_1, resulting in the successors of S_1 (i.e., S_2 and S_4) to be enabled in the next cycle. `b` activates state S_2 (S_4 is not activated since it does not accept symbol `b`), then the successor of S_2 (i.e., S_3) is enabled. The process repeats until all input symbols are consumed. In this case, since reporting state S_6 is activated by input symbol `f`, a report is generated indicating a successful match.

B. Baseline Automata Processor (AP)

Figure 3 shows a schematic of the considered baseline AP chip. The AP is a DRAM-based spatial architecture in which each state of NFA is stored in a memory column of the DRAM, namely a state transition element (STE). A bit in the column represents whether the STE can accept the corresponding input symbol represented by each row. The maximum size of the alphabet is 256 as this is the width of the address decoder in the current AP architecture. Therefore, there are 256 rows

[1] In homogeneous NFAs [16], [35], [36], all incoming transitions to any given state must accept the same set of input symbols (symbol-set). In the rest of this paper, we treat *homogeneous* NFA synonymous with NFA, because they have the same computational ability and time complexity.

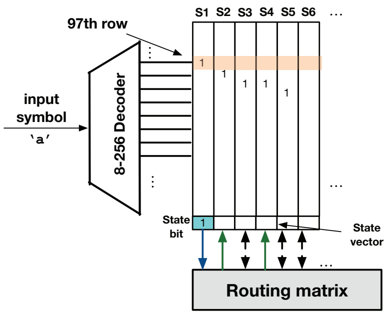


Fig. 3: The figure illustrates the first execution cycle of an AP configured with the NFA shown in Figure 2. S1 is enabled when input symbol `a` arrives, which activates S1, and enables S2 and S4 in the next cycle. Downward arrows represent the enable signal being fed to routing matrix in the current cycle. Upward arrows enable successor states for the next cycle. The physical connections between STEs and routing matrix are bi-directional, which are represented by the dashed arrows.

in total. An AP chip consists of two *half-cores*. The state transition cannot go across half-cores due to the limitation of the interconnect. The state transitions are compiled to the reconfigurable interconnecting network namely *routing matrix*.

The entire input stream is processed sequentially with the rate of one symbol per cycle. Each cycle, one input symbol is fed into the address decoder, which selects a whole row (out of 256) of the DRAM (orange shaded part in Figure 3). Each STE column has a bit that represents whether the STE is enabled or not, namely *state bit*. The state bits for all STEs are combined as a *state vector*. This information is available from the previous cycle. An *AND* operation is performed between the selected row (e.g., shaded part) and the state vector resulting in a vector that determines the activated states. This activation information is sent to the routing matrix, which updates the state vector with the enabled states for processing next symbol. Such a process is repeated until the entire input symbol stream is processed.

To understand the working of AP, we illustrate the execution of previously considered NFA (Figure 2) via Figure 3. We previously observed in Figure 2 that S_1 accepts symbol `a`. Accordingly, the bit stored in the 97th row (corresponding to the ASCII of `a`) and the column of STE that stores S_1 is set to 1 and the others remain 0. The state bit of S_1 is 1 *and* $\{a\}$ is in the symbol-set of S_1, therefore, S_1 is activated and it broadcasts the enable signals to the successor states (S_2, S_4) via the routing matrix (upward arrows in Figure 3).

III. Motivation and Analysis

In this section, we analyze why a high percentage of states are cold, which states are more likely to be cold, and how avoiding these states from being configured to AP can improve the performance.

A. Topological Order and Normalized Depth

In general, it is hard to predict which states will be enabled in NFAs [37]. Clearly, all starting states will be enabled at least once and this does not depend on the input. The states that are further away from the starting state, however, depend on the input. Each subsequent state transition in a homogeneous NFA must match a symbol of input (homogeneous NFAs do not have ε-transitions [38]). Intuitively, a state that is further away from the starting state is less likely to be enabled since each additional state on the path to it increases the chances of a mismatch.

To verify if this intuition holds on NFAs from real-world applications executing on the AP, we study whether states are hot or cold with respect to their depths in the NFAs. For simplicity of exposition, we first consider only NFAs that are also directed-acyclic graphs (DAGs). In this case, the depth of a state is simply its topological order (i.e., the maximum steps from the starting state to itself in the matching process). Thus, the matching process goes from states with a lower topological order to states with a higher topological order but cannot go back as DAGs do not have cycles. Such an NFA can be viewed as a graph with layers, where all starting states are in the first layer (i.e., their topological order is one), states in the second layer (i.e., states with topological order of two) are reachable from the first layer, states in the third layer are reachable from the first and second layers, and so on.

However, NFAs are not always DAGs, because they can contain back edges (i.e., from a later layer to an earlier layer) and cycles. For example, the NFA in Figure 4 (❶) contains a cycle between states S_4 and S_5. Topological sort cannot be performed on such graphs. Therefore, we pre-process an NFA by identifying all its strongly connected components (SCC) [39]. Each state s is marked with a connected component number $\mathrm{SCC}(s)$, such that the states belonging to the same SCC are marked with the same number. We construct graph G' from directed graph G (i.e., the NFA) by treating each SCC in G as a single node in G' (e.g., in Figure 4, the SCC that includes states S_4 and S_5 is considered as a single node in G'). For each edge (u,v) in G, an edge $(\mathrm{SCC}(u),\mathrm{SCC}(v))$ is added in G' if nodes u and v are in different SCCs. The resulting G' is a DAG on which we can run a topological sort. Figure 4 (❷) shows the results of identifying SCCs and topological sort. The topological order of each state is indicated as a number right to the state. Since S_4 and S_5 belong to the same SCC, they are assigned with the same topological order.

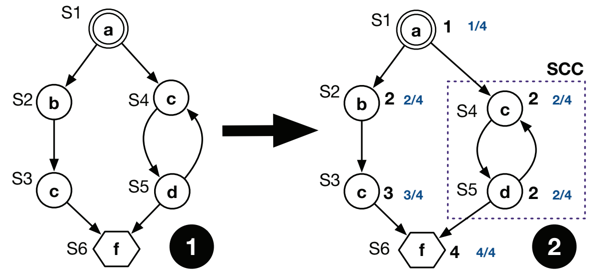
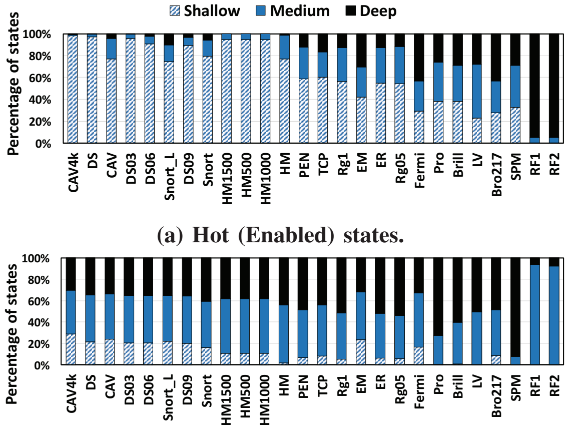


Fig. 4: Illustration of topological ordering and normalized depth.

The absolute topological order or depth of a state is uninformative as different NFAs can have a different number of layers, even within the same application. Therefore, we normalize the depth of a state to the maximum depth in the NFA it belongs to, resulting in *normalized depth*. For example, in Figure 4 (❷), because the maximum topological order is 4 (S_6), the normalized depth of each state s is $\mathit{topoorder}(s)/4$ (e.g., for S_4 and S_5, it is 2/4 or 0.5) where *topoorder* is a function that returns the topological order of a state. A normalized depth closer to 1 indicates the state is at the bottom of the NFA (or relatively deep), while a value closer to 0 indicates the state is closer to the top (or relatively shallow).

B. Analysis of Normalized Depth and Enabled NFA States

Figure 5(a) shows the normalized depth distribution of enabled (hot) states for our evaluated applications. Each application is comprised of many NFAs, each representing a different pattern. We find that for the majority of applications, the hot states have low normalized depth (i.e., they are closer to the starting state of the NFAs). Furthermore, for the same set of applications, Figure 5(b) shows the normalized depth distribution of cold (never enabled) states. We observe that the cold states in the majority of the applications have high normalized depth (i.e., they are in deeper regions of the NFAs). To confirm this conclusion further, we also find that there is a significant negative correlation (average correlation coefficient is -0.82) between normalized depth and percentage of hot states for all applications, except ER.



(a) Hot (Enabled) states.

(b) Cold (Never-enabled) states.

Fig. 5: Distribution of normalized depth for NFA states. For presentation purposes only, normalized depth is classified as: i) shallow ([0–0.3)), ii) medium ([0.3–0.6)), and iii) deep ([0.6–1]).

We conclude that whether a state is hot or cold is highly correlated with its normalized depth. Overall, "shallow" states are more likely to be hot while "deep" states are more likely to be cold.

C. Analysis of Performance Benefits

We analyze the ideal performance benefits when we completely eliminate the cold states from being configured on the AP. We show the potential benefits using a performance model assuming oracular knowledge of which states are cold and not configured on the AP.

Performance Model. Consider the case of the baseline AP execution, where the application has S states (across all NFAs) and the number of states the AP half-core can hold (capacity) is C_{AP}. Without loss of generality, we only discuss the case of one AP half-core. If the number of states (S) is larger than the size of AP (C_{AP}), it is not possible to configure the entire application at once to the AP and will require configuring the AP multiple times. Each configuration places a set of NFAs that can collectively fit in the AP. Suppose the size of each NFA in the application is less than the size of AP, therefore, the number of configurations to the AP would be $N_{config} = \lceil \frac{S}{C_{AP}} \rceil$, under the assumption that individual NFAs can be split at state granularity. In the current AP architecture, batches (partitions) usually contain whole NFAs, so the number of configurations may be even higher.

To maintain semantics, each configuration batch must see the same input stream. The matching process finishes after all batches of NFAs are executed on the same input stream. Thus, the total number of cycles spent on the same input stream is $N_{config} \times n$, where n is the length of the input stream and N_{config} is the number of batches. Under a perfect scenario where we can identify cold states (S_{cold}) with 100% accuracy, we can reduce N_{config} by not configuring the cold states to the AP. We define the resource saving $p = \frac{S_{cold}}{S}$. Therefore, the speedup over the baseline case is $\lceil \frac{S}{C_{AP}} \rceil / \lceil \frac{(1-p)\cdot S}{C_{AP}} \rceil$. If the number of states is sufficiently large, the speedup we can get is proportional to $\frac{1}{1-p}$, $p \neq 1$. Thus, the larger the proportion of cold states that can be correctly identified and eliminated, the more speedup we can have over the baseline execution scenario.

Illustrative Example. To illustrate the benefits of configuring the AP with only hot states, Figure 6 shows two scenarios: a) the baseline AP execution, and b) the AP that only executes hot states. The execution in both cases considers the same application (Ⓐ). In the baseline scenario, if the number of total states is more than the AP capacity, the execution will need to be done in batches as discussed before. In this example, the compiler partitions the application into two batches, where each batch can individually fit in the AP (Ⓑ). Hence, the same input stream is executed twice in a sequential manner (Ⓓ). However, with the oracular knowledge of cold states, the compiler can generate a *perfect partition* of the application with only the hot states (Ⓒ). If this perfect partition fits in the AP, it can execute on it by consuming the same input stream only once (Ⓔ), resulting in significant savings in the execution cycles.

In summary, significant speedup can be achieved if cold states are not configured to AP. In the next section, we propose a simple and effective profiling-based mechanism to identify such states in realistic scenarios and then leverage the profiling information to efficiently partition them from the NFAs.

IV. Design and Implementation of NFA Partitioning

Any realistic implementation that eliminates cold states from NFAs (i.e., partitions NFAs into cold and hot states, and only configures AP with hot states) has to deal with at least three challenges. First, although it is not possible to predict cold

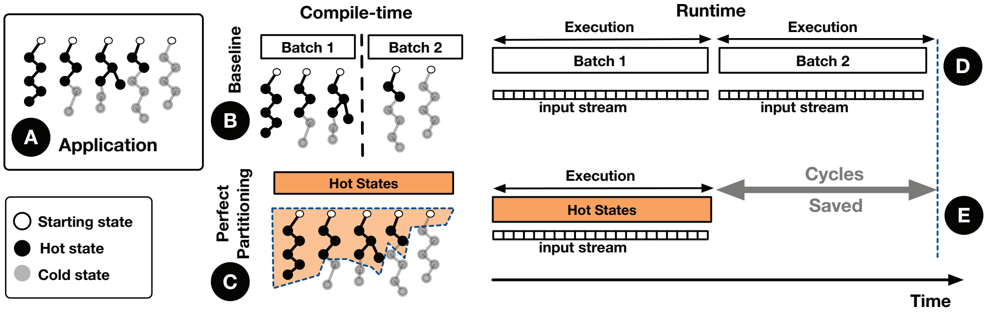


Fig. 6: An illustrative figure showing that by not configuring cold states on AP, all the hot states can fit onto an AP at the same time, reducing the number of re-executions over the input and hence saving time.

states with 100% accuracy in general, we need to develop low-overhead techniques to improve the accuracy of prediction as much as possible. Second, in the case of a mis-prediction, some transitions may require states that are not configured on the AP. To this end, we need a mechanism working as a safety net to handle a transition from a state on the AP to a state that is not on the AP. Third, to minimize the cost of such mis-predictions, transitions should be unidirectional to avoid re-executions of inputs on the AP.

Our proposed partitioning scheme systematically addresses these challenges. First, we use a profiling-based scheme to identify the topological layer that acts as a *partition layer* for each NFA in the application. Second, our proposed scheme handles transitions out of the AP by adding *intermediate reporting states* that piggyback on existing AP reporting hardware. Finally, to ensure unidirectional transitions, we partition the NFA at a specific topological order. Since the matching always proceeds from a lower to a higher topological order, edges that cross partitions go only in one direction.

A. Profiling-based Hot/Cold State Prediction

We use a small portion of input for each application as profiling input. Basically, at compile time, we run the profiling input on the NFAs of the application and determine whether a state is hot or cold. We assume that this profiling information holds true during the actual execution and hence are able to predict which states will be hot or cold. In the following parts of this sub-section, we evaluate the effectiveness of our profiling-based prediction.

Profiling and Testing Inputs. Each application that we evaluate has a 1MB input. We divide this 1MB input into two equal parts of 512KB. The first 512KB of input is used for creating different sizes of profiling inputs and the last 512KB is used for testing input. We create different sizes of profiling inputs by using the first 0.2%, 2%, 20%, 100% symbols of the 512KB portion, which is essentially 0.1%, 1%, 10%, 50% of the entire input.

Methodology for Evaluating the Effectiveness of Profiling. In our evaluation, we treat hot as *positive* (P) and cold as *negative* (N). Therefore, *true positives* (TP) are states that are hot both under profiling input and testing input. Similarly, *false positives* (FP) are states that are hot under profiling input but actually cold under testing input. True negatives (TN) and false negatives (FN) are defined similarly. We define: a) $accuracy = \frac{TP+TN}{P+N}$, which measures overall how well is the profiling-based prediction; b) $recall = \frac{TP}{TP+FN}$, which measures how complete our prediction is terms of predicting hot states; and c) $precision = \frac{TP}{TP+FP}$, which measures how well the prediction could realize the resource saving scope (p).

TABLE I: The effectiveness of profile-based prediction

Percentage of the entire input ⇒	0.1%	1%	10%	50%
Accuracy	87%	90%	93%	97%
Recall	64%	76%	87%	97%
Precision	94%	92%	90%	92%

Effectiveness of Profiling. Table I shows the average numbers for accuracy, recall, and precision when we use different sizes of profiling inputs. We evaluate all applications except `Fermi` and `SPM`. Specifically, using only 2% prefix of the first 512KB (i.e., 1% of the entire input) can achieve 76% recall, which means 76% of hot states under testing input are also hot with the small profiling input. The results are consistent across 24 applications (recall varies from 49% to 100%). In addition, the prediction also has good results in terms of accuracy and precision. To conclude, only a small profiling input can identify most of the hot states during the actual execution. Therefore, we use 0.1% and 1% of the entire input for profiling and the remaining for the actual evaluation[2] (Section VII).

B. Where to Partition?

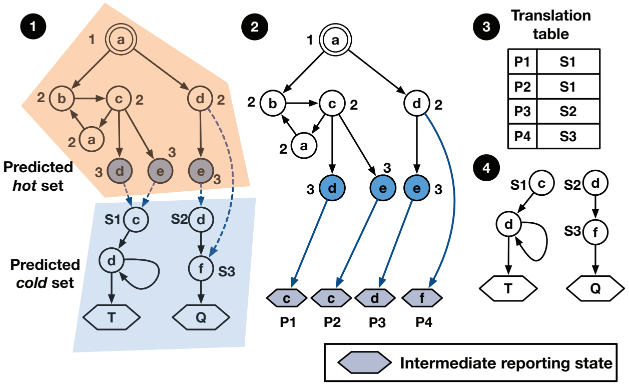
In current AP architecture, the application is split at NFA granularity into batches. In contrast, we partition the NFAs at topological-order granularity. There are two reasons that we use topological-order as our partition granularity. First, our previous analysis (Section III-B) shows there is a correlation between normalized depth and percentage of hot states. Second, partition at topological-order granularity can guarantee the unidirectional transition between predicted cold and hot states. In this sub-section, we show how do we obtain partition layer k_U for each NFA U of the application. We will show how to partition each NFA at the topological-order granularity in Section IV-C.

[2] For `Fermi` and `SPM`, we use the entire input for the actual execution because their starting states are only enabled at position 0 (*start-of-data* in ANML configuration).

Choosing Partition Layer. At compile time, we functionally simulate all NFAs of the application using the profiling input and predict whether a state is hot or cold. After simulation, for each NFA U, we set $k_U = \max\{topoorder(s)\}$, $\forall s$ is a hot state in NFA U under the profiling input. We define the *predicted hot set* $= \{s \mid s \in U \wedge topoorder(s) \leq k_U, \forall U\}$. Accordingly, the *predicted cold set* $= \{s \mid s \in U \wedge topoorder(s) > k_U, \forall U\}$. We divide the predicted hot set at NFA level into batches that can fit in AP and configure each batch sequentially.
Optimization. As an optimization, to make each batch fill the AP completely, we assign additional states to the predicted hot set from predicted cold set. This is achieved by incrementing k_U, which adds the states of the subsequent partition layers for each NFA U. This process terminates when the capacity of AP is met for each batch.

C. How to Partition?

In this sub-section, we demonstrate how to partition an NFA into two parts at a given partition layer k calculated based on the description presented in Section IV-B and how to handle state transitions when the partitioning is imperfect. For brevity, we describe our partitioning scheme for a single NFA, which then can be separately applied to each NFA in the application. Figure 7 illustrates NFA partitioning using the partition layer $k = 3$ and cut the edges that connect states with $k \leq 3$ to states with $k > 3$ (indicated as dashed lines in Figure 7 (❶)). However, the prediction may not be perfect – a state in the predicted cold set could end up being enabled during matching. Since only states in the predicted hot set are present on the AP, the matching process must transition out of the AP.



P1	S1
P2	S1
P3	S2
P4	S3

Fig. 7: Partitioning an NFA by the partition layer.

To handle such cases, for each edge (u, v) we cut in the original NFA, we introduce an *intermediate reporting state* v' and an edge (u, v'). The state v' matches exactly the same input symbols (symbol-set) as v but is also a *reporting state*. During execution, the AP contains these *intermediate reporting states* along with the predicted hot set. Therefore, when the matching process tries to enable a state that is not on the AP (i.e., in the predicted cold set), it activates the corresponding intermediate reporting state instead. Consequently, an *intermediate report* is generated that notifies a handler (Section V). The handler will enable corresponding states in predicted cold set to continue the matching process. Since we use topological order to partition, after the matching process continues, it will never go back to the predicted hot set. In Figure 7 (❷), the intermediate reporting states are P_1 through P_4. When activated, these states enable their corresponding states S1, S2 and S3 as indicated in the translation table (Figure 7 (❸)), which lie in the predicted cold set shown in Figure 7(❹).

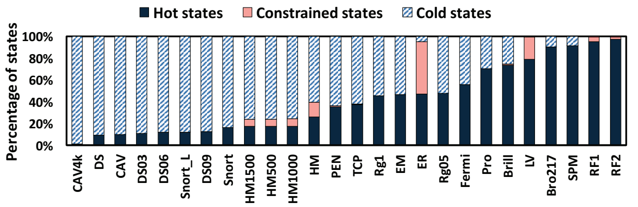


Fig. 8: Constrained states are cold states but configured on the AP due to the constraints in our topological-order-based partitioning scheme. Consequently, some AP resources are underutilized with a few applications.

D. Discussion

The use of SCC and topological-order-based partitioning imposes constraints that lead to more states than necessary being added to the predicted hot set. Specifically, (1) even if only one state in an SCC is hot, the whole SCC must be included in predicted hot set, and (2) a cold state with topological order less than the partition layer k is still included in the predicted hot set. This might reduce the AP resource savings.

To study the extent of this underutilization, Figure 8 shows that for all the 26 evaluated applications, our topological-order based perfect partitioning constrains only 4% on average more states to the predicted hot set (which in reality are not going to be enabled), compared with perfect partitioning that can cut NFAs at arbitrary edges. Two exceptions are `LV` and `ER` whose large SCCs prevent effective partitions. In summary, we still have a significant opportunity for resource savings if we can accurately identify the partition layer for each NFA.

V. Hardware Support for Intermediate Report Handling and Partitioned NFA Processing

In this section, we discuss how to efficiently handle the intermediate reports generated from the execution of the predicted hot set. To this end, we propose to: a) enable the states that intermediate reporting state directs to, and b) continue the matching process from the cycle (i.e., the input position) where the intermediate report was generated at. Although both steps can be performed on CPU, it incurs significant performance slowdown (Section VII), therefore we propose a new execution mode for the AP.

A. Analysis of New Execution Modes for AP

In order to support the aforementioned steps, we propose an augmented AP which supports two modes: BaseAP mode, and SparseAP (SpAP) mode. The BaseAP mode execution is similar to the baseline AP execution, however, AP in this mode is configured with only the predicted hot set. Once the

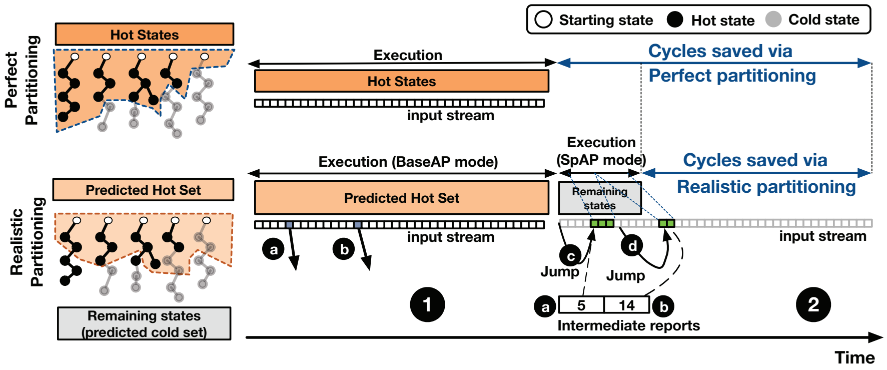


Fig. 9: Illustration of performance benefits under realistic partitioning: because of the *jump* operation, only a portion of input symbols are executed in the SpAP mode execution.

execution of BaseAP mode finishes, the generated intermediate reports are handled in the SpAP mode. In the SpAP mode, the AP is configured with the predicted cold set. The AP in this mode not only consumes input symbols but is also driven by the intermediate reports.

In this context, we develop two major operations for the SpAP mode: *enable* and *jump*. The enable operation allows each intermediate report to enable the appropriate state in the predicted cold set. The jump operation skips over the input symbols that are not necessary for handling the intermediate reports. Since no back-edge exists from predicted cold states to predicted hot states (discussed in Section IV), no back and forth switching between BaseAP and SpAP modes is required.

Each intermediate report in the list of intermediate reports (L) is represented by a tuple: input position and state ID (c, sid) denoting that the intermediate report is generated at input position c (i.e., cycle c in the BaseAP mode execution) and the state to be enabled is sid. Algorithm 1 shows the pseudo code for the SpAP mode execution. In each cycle, if no state is enabled (Line 4), it performs a *jump* operation setting the current input position i to the input position where next intermediate report was generated. The *enable* operation (Line 9 to Line 11) is performed due to either scenario: current input position i reaches the input position in next intermediate report or the current input position i was just set to $L[j].c$ by the jump operation. The remaining functionality of the SpAP mode is the same as the BaseAP mode. We describe next how these operations are used to handle realistic partitioning scenarios with the help of an illustrative example.

Illustrative Example. Figure 6 earlier discussed the performance benefits of perfect partitioning. Under realistic partitioning, inaccurate predictions of cold states require intermediate report handling. Figure 9 shows an illustrative example demonstrating the benefits of executing AP in BaseAP and SpAP modes. The execution starts in the BaseAP mode (❶) that is configured with the predicted hot set. During its execution, two intermediate reports are generated at input position 5 and input position 14, respectively and are stored (ⓐ, ⓑ). Once all the input symbols are consumed, the SpAP mode begins (❷), which is driven by both the input stream and the intermediate reports. If no state is enabled, SpAP mode jumps to the input position where the next intermediate report was generated. In this example, initially, it jumps to the input position 5 of the first intermediate report directly (ⓒ). During the execution, when there is no enabled state (at input position 8), the SpAP jumps to input position (14) of the next intermediate report (ⓓ). Therefore, under SpAP, only a portion of the input symbols are executed (green shaded part in ❷).

Algorithm 1 Functionality of SpAP mode

Input: L, the list of intermediate reports. Each element in L contains (c, sid) showing the input position where the report was generated, and the state id to be enabled.
Input: *input*, the input symbol stream.
Output: *out_list*, the list of reports.

```
 1: $i \Leftarrow 0$
 2: $j \Leftarrow 0$   ▷ $i$ is the index (input position) of $input$, $j$ is the index of $L$.
 3: while $i < input.length$ do
 4:     if $E$ is $\emptyset$ then          ▷ $E$ is the set of enabled states.
 5:         if $j < L.length$ then
 6:             $i \Leftarrow L[j].c$                ▷ Jump operation.
 7:         else
 8:             break
 9:     while $L[j].c = i$ and $j < L.length$ do
10:         enable $L[j].sid$                ▷ Enable operation.
11:         $j \Leftarrow j + 1$
12:     $A \Leftarrow$ {states in $E$ that accept $input[i]$}
13:                                 ▷ $A$ is the set of activated states.
14:     $E \Leftarrow \emptyset$
15:     for all $s$ in $A$ do
16:         if $s$ is a reporting state then
17:             append ($i$, $s$.id) to $out\_list$
18:         $E \Leftarrow E \cup$ {successors of $s$}.
19:     $i \Leftarrow i + 1$
```

B. Implementation Details

We describe the required hardware implementation supporting SpAP mode by implementing the *jump* and *enable* operations on top of the current AP architecture. We start by the implementation of the SpAP operations. Then we estimate the execution time overhead of these operations. Finally, we demonstrate the storage requirements for the intermediate reports.

Jump Operation. The *jump* operation modifies a register that tracks the current input position. Specifically, if no STE is enabled, the *jump* operation updates the register value with the input position from the next intermediate report. Since no state configured to SpAP is always enabled, the enabled states in next cycle are only determined by the activated states in the current cycle. Therefore, given that the routing matrix routes the enable signal from the activated states, we assume that the routing matrix provides a flag that is set if no STE is enabled.

Enable Operation. Given an intermediate report, we use the state ID information to enable the corresponding STE. Since STEs are connected to the routing matrix, and the routing matrix follows a hierarchical design (block, rows, and STEs) [16], we utilize such hierarchy to perform the *enable* operation. The routing matrix consists of 96 blocks per half core. Each block is a group of 16 rows, and each row is a group of 16 STEs. Since state ID is represented by 16 bits, we divide these bits to enable the required STE in a hierarchical manner. We use the first 8 bits to select the block, the middle 4 bits to select the row, and the last 4 bits to select the required STE within the row. We use a total of three decoders to select the required block, row, and STE, respectively. Specifically, a 7×128 decoder is used to select the block. Then, a 4×16 decoder selects the row. Finally, a 4×16 decoder enables the required STE. The *enable* operation works in parallel with the processing of input symbols during SpAP mode.

Enable Operation Execution Overhead. We can overlap the *enable* operation of only one intermediate report with the processing of the input symbols in SpAP mode. Thus, if multiple intermediate reports were generated in the same input position during BaseAP mode, the input processing is stalled until all the states in the simultaneous intermediate reports are enabled. In SpAP mode, to do that, we compare the input position of the head intermediate report with the next input position (current input position + 1). Similarly, we compare the input position of the second intermediate report with the next input position. If both of these comparisons are set, we pause the processing of the input symbols. After enabling the states in all simultaneous intermediate reports, the input processing resumes. The cycles spent to enable the simultaneous intermediate reports are considered overhead to the overall SpAP mode execution and are accounted for in our evaluation methodology.

Intermediate Reports Storage Overhead. The list of intermediate reports is stored in the off-chip device memory. Only a portion of the reports is loaded to the on-chip memory to be consumed during the SpAP mode. We use a queue of 128 entries to store the loaded intermediate reports. Because each intermediate report is a (input position, state ID) tuple, we need 6 bytes per intermediate report (4 bytes for the input position, and 2 bytes for the state ID). Thus, the overall storage required for the intermediate reports queue is 128×6 bytes.

VI. Evaluation Methodology

A. Applications

We evaluate our mechanisms with all applications in the ANMLZoo benchmark suite [27] and the Regex benchmark suite [34]. Table II shows that these applications have states ranging from approximately 2K to 100K, and several of them have states more than 24K, which is the size of our baseline AP half-core. In order to evaluate applications with an even larger number of states, we generate multiple applications based on three sources: ClamAV [29], Hamming [40], and Snort [41].

ClamAV4k (CAV4k). We convert the regular expressions in `main.cvd` of the Q1 2018 ClamAV Virus Database to ANML format. We select the first 4,000 patterns from the virus database. We use the same input of ClamAV in ANMLZoo [27].

Hamming. We generate Hamming automata using the same approach as the ANMLZoo benchmark suite [40]. To keep it consistent with Hamming in ANMLZoo, we also create the automata in the BMIA (Bounded Mismatch Identification Automaton) format. We created three different workloads from Hamming that contain different number of NFAs, namely `HM500`, `HM1000` and `HM1500`. For each workload we generate, we create a mix of different expected pattern lengths (8, 12, 20, 30), each with a distance of 2 to 20% of the pattern length (e.g., $0.2 \times 30 = 6$). Similar to Hamming in ANMLZoo [27], we generate the inputs randomly.

Snort_L. Our `Snort_L` application includes 3,126 rules from both community rules and registered rules of the Snort network intrusion detector [41]. We convert the regular expressions to ANML format. We use the same network traffic input as the Snort application in ANMLZoo.

We consider a total of 26 applications and divide them into three groups based on the number of states they contain. The high resource requirement (high) group contains applications with states more than the capacity of an AP chip (49K). The medium resource requirement (medium) group contains applications with states more than the capacity of an AP half-core (24K). The rest of the applications are grouped into low resource requirement (low) group.

B. Experimental Setup

We build our mechanisms on top of the open-source virtual automata simulator – VASim [42]. As we mentioned in Section V, we evaluate both AP–CPU and BaseAP/SpAP execution. In the AP–CPU execution, the states that are executed in the SpAP mode are instead executed on the CPU. Table III shows a summary of the evaluated scenarios. We model different timing mechanisms for AP–CPU and BaseAP/SpAP in the simulator as detailed below.

Timing AP–CPU Execution. We record the total amount of time that the CPU spends to handle the intermediate reports by using `std::chrono` in C++ library. Therefore, we use the real time when we calculate the speedup in the AP–CPU execution. We run our experiments on a machine with Intel(R) Xeon(R) CPU E5-2683 v3. We use 7.5 ns as the cycle time per symbol [31] for the BaseAP execution.

TABLE II: List of evaluated applications: “RStates” stands for reporting states and “MaxTopo” stands for maximum topological order across NFAs. “Grp” stands for resource requirement groups: High (H), Medium (M), Low (L).

Application	Abbr.	Grp.	#States	#NFAs	MaxTopo	#RStates
ClamAV4000 [29]	CAV4k	H	1124947	4000	2080	4015
Hamming1500 [40]	HM1500	H	366000	3000	32	6000
Hamming1000 [40]	HM1000	H	244000	2000	32	4000
Snort_big [41]	Snort_L	H	132171	3126	4509	4043
Hamming500 [40]	HM500	H	122000	1000	32	2000
SPM [27]	SPM	H	100500	5025	16	5025
Dotstar [27]	DS	H	96438	2837	95	2838
EntityResolution [27]	ER	H	95136	1000	64	1000
RandomForest1 [27]	RF1	H	75340	3767	3	3767
Snort [27]	Snort	H	69029	2687	133	4166
ClamAV [27]	CAV	H	49538	515	542	515
Brill [27]	Brill	M	42658	1962	38	1962
Protomata [27]	Pro	M	42009	2340	123	2365
Fermi [27]	Fermi	M	40783	2399	13	2399
PowerEN [27]	PEN	M	40513	2857	44	3456
RandomForest2 [27]	RF2	M	33220	1661	3	1661
TCP [34]	TCP	L	19704	738	100	767
Dotstar06 [34]	DS06	L	12640	298	104	300
Ranges05 [34]	Rg05	L	12621	299	94	299
Ranges1 [34]	Rg1	L	12464	297	96	297
ExactMath [34]	EM	L	12439	297	87	297
Dotstar09 [34]	DS09	L	12431	297	104	300
Dotstar03 [34]	DS03	L	12144	299	92	300
Hamming [27]	HM	L	11346	93	20	186
Levenshtein [27]	LV	L	2784	24	23	96
Bro217 [34]	Bro217	L	2312	187	84	187

TABLE III: Summary of Execution Scenarios

System	Software	Hardware		
		Execution of entire NFAs	Execution of predicted hot set	Execution of predicted cold set
AP	Partition (at NFA granularity)	BaseAP Mode	N/A	N/A
AP–CPU	Partition (hot/cold set)	N/A	BaseAP Mode	CPU
BaseAP/SpAP	Partition (hot/cold set)	N/A	BaseAP Mode	SpAP mode

Recording the Cycles in BaseAP/SpAP Execution. In the BaseAP/SpAP execution, we record the execution cycles via the simulator. The number of cycles in BaseAP/SpAP execution is the sum of cycles spent on BaseAP mode and SpAP mode. Therefore, $Speedup_{BaseAP/SpAP} = \frac{Number\ of\ cycles\ on\ AP\ baseline\ execution}{Number\ of\ cycles\ on\ BaseAP\ Mode+Number\ of\ cycles\ on\ SpAP\ Mode}$.

Performance per STE. We define a metric called *performance per STE* to show how much throughput each STE can provide on average. Specifically, $performance\ per\ STE = \frac{throughput}{C_{AP}}$, where $throughput = \frac{number\ of\ input\ symbols}{number\ of\ cycles}$. This allows us to compare APs with different capacities while also considering techniques that improve performance solely by increasing the AP size. Because each STE in the AP occupies die area, we can also consider this metric as a proxy for performance/area.

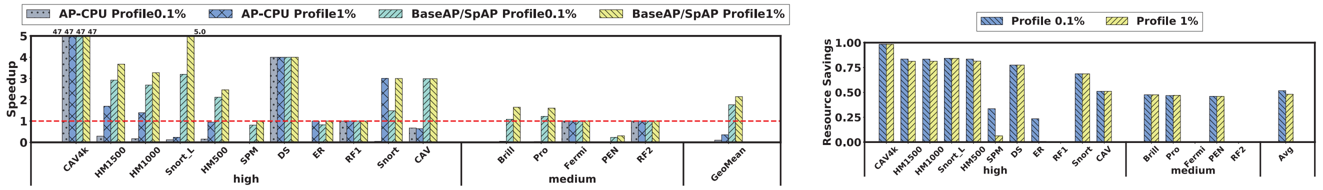
Overheads. In this paper, we focus on reducing the re-execution overhead as we found it is the major performance bottleneck in AP. The new SpAP mode incurs the stall cycles due to simultaneous intermediate reports (Section V-B). Our final results include these stall cycles. There are two more generic overheads related to output and reconfiguration. In our evaluations, we do not include the output overhead [30] and rely on existing work [43] that proposes both hardware and software techniques to address it. We also do not include the reconfiguration overhead (50 ms [44], [45] for reconfiguring a full AP board) in our results as we believe it can be amortized over AP execution, especially when it executes very large inputs.

VII. Experimental Results

Effect on Performance. To show the benefits of our schemes, we evaluate the speedup for applications in the high and medium groups. Our mechanisms do not change the throughput of AP for applications in the low category since the sizes of applications are smaller than our baseline AP with 24K STEs. Figure 10(a) shows the performance results of our proposal, from which we draw four major observations. First, The AP–CPU execution shows a significant geometric mean slowdown of 9.8× and 2.9× under 0.1% and 1% profiling input, respectively. However, five applications out of 16 applications (`CAV4k`, `HM1500`, `HM1000`, `DS`, `Snort`) achieve a 4.2× geometric mean speedup at no cost of hardware modification. Second, we find that BaseAP/SpAP execution shows a speedup in the majority of evaluated applications. It can achieve 1.8× and 2.1× geometric mean speedup using 0.1% and 1% of input as profiling input, respectively. Third, BaseAP/SpAP execution can be slower than the AP in a few applications (e.g., `PEN`), since these applications generate many simultaneous intermediate reports, leading to lengthy enable stalls on the SpAP mode (shown in Table IV). Fourth, in applications with large SCCs that prevent efficient partitioning (e.g., `ER`, see Figure 8), our scheme configures all the states to the BaseAP mode execution with no change in execution time.

Effect on Performance per STE. In order to evaluate the efficiency of our schemes across a wider set of system sizes and configurations, we show *performance per STE* in Figure 11, from which we draw two major observations. First, although different sizes of AP chips can execute the same application with the same performance (e.g. an application in *low* group fits and runs on both an AP chip or an AP half-core), larger AP chips have less performance/STE, because fewer STEs in the larger AP are utilized for the same application size. Such underutilization leads to less performance/STE. Second, on average, our scheme not only increases performance/STE by 32.1% under the scenario of AP half-core and using 1% profiling input, but consistently achieves better performance/STE under different sizes of AP as well. There are two major reasons: (1) we predict cold states and eliminate them from being configured, which increases AP utilization; (2) we use fewer cycles in the SpAP mode for mis-prediction handling than re-execution by batches hence increasing the throughput.

Resource Savings and Speedup. We show the results of resource savings in Figure 10(b). By comparing it with Figure 10(a), we make three observations. First, generally, the applications with high resource savings also have good speedups. Second, `PEN` shows slowdown although it has good resource savings. This is because its SpAP mode execution has lots of enable stalls due to a large amount of simultaneous intermediate reports (Table IV). Third, although the resource savings may be the same under different profiling inputs, the speedup may be different (e.g., `Snort`). It is because the original size of the predicted hot set was different, but due to the optimization in Section IV-B, each batch was extended with a part of the predicted cold states to match the capacity of AP. Consequently, this leads to the same resource savings.



(a) Speedup with AP–CPU and BaseAP/SpAP execution using 0.1% and 1% profiling input (capacity = 24K).

(b) Resource savings (i.e., the portion of states that are not configured in the BaseAP mode)

Fig. 10: Speedup and Resource Savings on AP.

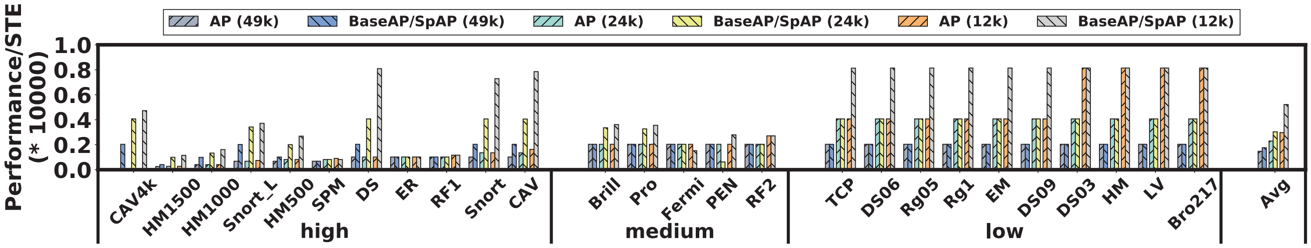


Fig. 11: Performance per STE of various AP sizes with BaseAP/SpAP execution considering 1% profiling input.

TABLE IV: Runtime statistics for AP and BaseAP/SpAP (under 1% profiling input): The first three columns show the number of executions on the AP, BaseAP mode and SpAP mode, respectively. "EStalls" stands for the stalls caused by enable operations for handling simultaneous intermediate reports. "JumpRatio" is defined as the proportion of cycles skipped in the SpAP mode.

	#Baseline Execution	#BaseAP/SpAP Execution		BaseAP/SpAP Runtime Statistics		
App	AP	BaseAP Mode	SpAP Mode	#Intermediate Reports	#EStalls	JumpRatio
CAV4k	47	1	0	0	0	-
HM1500	15	4	13	80680	248	99.37%
HM1000	10	3	9	54075	180	99.39%
Snort_L	6	1	5	172665	87882	97.99%
HM500	5	2	5	27815	79	99.43%
SPM	5	4	1	63490	119	2.11%
DS	4	1	0	0	0	-
ER	4	4	0	0	0	-
RF1	4	4	0	0	0	-
Snort	3	1	2	70	4	99.99%
CAV	3	1	1	3215	0	99.67%
Brill	2	1	1	68125	23997	81.51%
Pro	2	1	1	89733	15862	77.43%
Fermi	2	2	0	0	0	-
PEN	2	1	1	5450318	4509743	1.96%
RF2	2	2	0	0	0	-

However, since larger profiling input has higher *recall* for hot states (Section IV-A), the speedup is also higher. To conclude, the speedup is generally related to resource savings as we explained in Section III-C, but the speedup also depends on other factors such as the quality of prediction and the number of enable stalls.

Intermediate Reporting States. The addition of intermediate reporting states increases the total number of states which could increase the total number of configurations and executions (e.g., HM500 in Table IV). Figure 12 shows the effect on the number of reporting states in BaseAP mode normalized to that of the baseline. In the BaseAP/SpAP mode, the total number of reporting states includes both original reporting states and intermediate reporting states (stacked bars in the figure). We make two observations. First, the total number of reporting states in BaseAP mode could be more than the baseline AP execution that only contains original reporting states. For example, the number of reporting states in ER increases by 3.6×, because it has a large number of crossing edges between predicted hot set and predicted cold set. Second, the number of reporting states could decrease (e.g., Snort and Snort_L) in the BaseAP mode execution because the number of crossing edges is smaller than the number of original reporting states. Although our scheme may increase the number of reporting states, we are aware that an effective software-based reporting state compression technique [43] could be applied on top of our scheme.

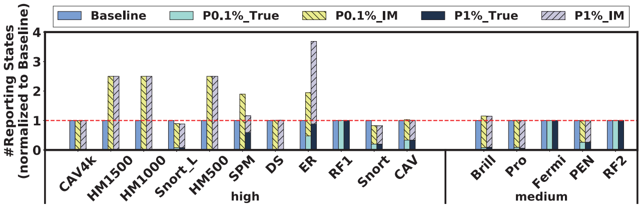
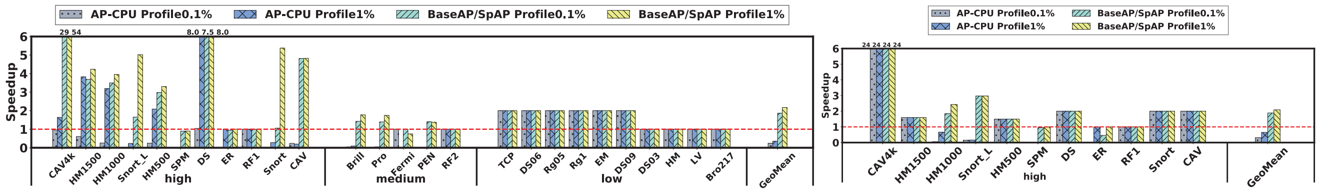


Fig. 12: Comparison of number of reporting states: "IM" stands for intermediate reporting states. "True" stands for original reporting states on BaseAP mode. "P" stands for profiling.

Effect of Jump Operations. In Table IV, although for some applications (e.g., HM500, Brill) the number of executions of BaseAP/SpAP may be greater than or equal to the baseline, we still obtain speedups on them because SpAP mode execution can reduce total number of cycles due to the *jump* operations. To show the effect of jump operations, we define *JumpRatio* as the proportion of cycles skipped in the SpAP mode. Formally, $JumpRatio = 1 - \frac{Total\ cycles\ on\ SpAP\ mode}{Number\ of\ batches\ on\ SpAP\ mode \times Length\ of\ input\ stream}$. Higher *JumpRatio* indicates better effect of *jump* operations. We show *JumpRatio* in Table IV for the applications that use SpAP mode. To conclude, the majority of the applications only execute a few percent of input symbols with the help of jump operations.



(a) Speedup on a small AP (capacity = 12K)

(b) Speedup on an AP chip (capacity = 49K)

Fig. 13: Sensitivity on the different capacities of AP chip.

Sensitivity of speedup on capacity of AP. The applications in the low resource requirement group require fewer states than the capacity of AP half-core. Figure 13(a) shows the speedup achieved by our schemes when the capacity of AP is 12K. Similar observations still hold as discussed in Figure 10(a). Specifically, BaseAP/SpAP achieves 1.9× and 2.2× speedup using 0.1% and 1% profiling input, respectively. In addition, we demonstrate another sensitivity study on AP with 49K STEs for the applications in the high group. Figure 13(b) shows BaseAP/SpAP execution achieves 1.9× and 2.1× speedup using 0.1% and 1% profiling input in the 11 applications of this group.

VIII. Related work

To the best of our knowledge, this is the first work that designs an efficient architectural support for large-scale NFA applications on AP.

Spatial Architectures. Multitasking on spatial architectures is usually carried out through the use of multiple contexts [46], which can consume extra memory. In contrast, our BaseAP/SpAP proposal relies on the ability to eliminate dynamically unused states from NFAs to improve AP utilization. We rely on a mechanism to transfer control to a spatially distinct partition to accommodate larger than device NFAs, though these could be implemented as multiple contexts. Recently, gate removal has been proposed to eliminate unused logic gates from general purpose processor IPs to customize processors to specific applications [47]. In our approach, we only eliminate states from the NFA (i.e., the program), and not the hardware. There are also alternative implementations of AP [48]–[50]. For example, cache automaton [49] re-purposes the last-level cache for automata processing. We believe our techniques are complementary as we propose hardware/software mechanisms to make the automata processing itself more efficient.

DFA and NFA Acceleration. Deterministic finite automata (DFA) have been characterized previously – with respect to implementing special machines [51] and for parallelization [37], [52]–[55]. Parallel execution of NFAs on the AP processor has been proposed by trading AP resources for higher throughput [31]. However, our characterization of the dynamic execution properties of NFAs specific to the AP execution model is, to our knowledge, the first of its kind. Our elimination of dynamically unused states can free up AP resources to complement parallel execution.

FSM Decomposition. FSM decompositions [56]–[59] could reduce the complexity of placement and routing in the routing matrix by simplifying the layout. While cascade decompositions are the closest to our studies, they are often static, for deterministic machines only, and are mostly not based on dynamic state behavior (i.e., predicted hot vs. predicted cold states). In contrast, our proposed approach (which uses graph-theoretic techniques, rather than sequential machine theory) is focused on increasing the AP throughput by allowing only predicted hot states to be configured to the AP. We believe both approaches are complementary and can be applied to different bottlenecks in the AP execution pipeline. For example, FSM decomposition can make the *reconfiguration* process efficient while our technique can accelerate the NFA execution on AP by reducing the number of *re-executions* of the input symbol stream.

IX. Conclusions

Automata processors (AP) are very efficient in executing Non-deterministic Finite Automata (NFAs). However, like other types of spatial architectures, AP faces major challenges in its execution model to efficiently execute very large tasks. In this paper, we make use of the inherent properties of NFAs to avoid using compute resources for states that are never used during execution by a low-cost software/hardware-coordinated approach. Consequently, this results in a new execution model for APs that enables efficient and high-performance processing for large-scale tasks. We believe this work will be helpful towards wider adoption of APs and will open up new research directions for enabling efficient NFA processing.

Acknowledgment

The authors thank the anonymous reviewers and members of the Insight Computer Architecture Lab at the College of William and Mary for their feedback. This material is based upon work supported by the National Science Foundation (NSF) grants (#1657336, #1717532, and #1750667). This work was performed in part using computing facilities at the College of William and Mary which were provided by contributions from the NSF, the Commonwealth of Virginia Equipment Trust Fund and the Office of Naval Research. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[2] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A Digital Neurosynaptic Core using Embedded Crossbar Memory with 45pJ per Spike in 45nm," in *IEEE Custom Integrated Circuits Conference (CICC)*, 2011.

[3] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[4] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "GenAx: A Genome Sequencing Accelerator," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[5] O. Temam, "A Defect-tolerant Accelerator for Emerging High-performance Applications," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2012.

[6] 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 International Symposium on Microarchitecture (MICRO)*, 2015.

[7] D. Sidler, Z. István, M. Owaida, and G. Alonso, "Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017.

[8] R. Mueller, J. Teubner, and G. Alonso, "Data Processing on FPGAs," *Proceedings of the VLDB Endowment*, 2009.

[9] S. R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, and E. Sedlar, "A Many-core Architecture for In-memory Data Processing," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2017.

[10] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[11] X. Yu, K. Hou, H. Wang, and W. C. Feng, "Robotomata: A Framework for Approximate Pattern Matching of Big Data on an Automata Processor," in *Proceedings of the International Conference on Big Data (BigData)*, 2017.

[12] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.

[13] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[14] Z. Lin, L. Nyland, and H. Zhou, "Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[15] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[16] P. Dlugosch, D. Brown, P. Glendenning, L. Leventhal, and H. Noyes, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2014.

[17] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the Micron Automata Processor," in *Proceedings of the International Conference on Semantic Computing (ICSC)*, 2015.

[18] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the Automata Processor," in *Proceedings of the International Conference on High Performance Computing (HiPC)*, 2016.

[19] I. Roy, N. Jammula, and S. Aluru, "Algorithmic Techniques for Solving Graph Problems on the Automata Processor," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[20] C. Bo, K. Wang, J. J. Fox, and K. Skadron, "Entity resolution acceleration using the automata processor," in *Proceedings of the International Conference on Big Data (BigData)*, 2016.

[21] M. Putic, A. J. Varshneya, and M. R. Stan, "Hierarchical Temporal Memory on the Automata Processor," *IEEE Micro*, 2017.

[22] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron, "Generating efficient and high-quality pseudo-random behavior on Automata Processors," in *Proceedings of the International Conference on Computer Design (ICCD)*, 2016.

[23] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "iNFAnt: NFA Pattern Matching on GPGPU Devices," *ACM SIGCOMM Computer Communication Review (CCR)*, 2010.

[24] X. Yu and M. Becchi, "GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space," in *Proceedings of the International Conference on Computing Frontiers (CF)*, 2013.

[25] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong, "GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[26] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte, "SIMD parallelization of applications that traverse irregular data structures," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2013.

[27] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, "ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2016.

[28] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi, "Demystifying Automata Processing: GPUs, FPGAs or Micron's AP?" in *Proceedings of the International Conference on Supercomputing (ICS)*, 2017.

[29] "Clamav net," https://www.clamav.net/, 2018.

[30] "ANML Documentation," http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html, 2018.

[31] A. Subramaniyan and R. Das, "Parallel Automata Processor," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[32] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2006.

[33] M. Becchi and P. Crowley, "Efficient Regular Expression Evaluation: Theory to Practice," in *Proceedings of the Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2008.

[34] M. Becchi, M. Franklin, and P. Crowley, "A Workload for Evaluating Deep Packet Inspection Architectures," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2008.

[35] "Characterization of Glushkov automata," *Theoretical Computer Science*, 2000.

[36] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron, "MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem," *IEEE Computer Architecture Letters (CAL)*, 2018.

[37] Z. Zhao, B. Wu, and X. Shen, "Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-based Computations Through Principled Speculation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[38] M. Sipser, *Introduction to the Theory of Computation*, 1st ed. International Thomson Publishing, 1996.

[39] T. T. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 1990.

[40] I. Roy and S. Aluru, "Finding Motifs in Biological Sequences Using the Micron Automata Processor," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[41] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Proceedings of the USENIX Conference on System Administration (LISA)*, 1999.

[42] J. Wadden and K. Skadron, "VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research," University of Virginia, Tech. Rep. CS2016-03, 2016.

[43] J. Wadden, K. Angstadt, and K. Skadron, "Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing

Architectures," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[44] I. Roy, "Algorithmic Techniques for the Micron Automata Processor," Ph.D. dissertation, Georgia Institute of Technology, 2015.

[45] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy, J. Wadden, M. Stan, and K. Skadron, "An Overview of Micron's Automata Processor," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2016.

[46] M. Gao, C. Delimitrou, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and C. Kozyrakis, "DRAF: A Low-power DRAM-based Reconfigurable Acceleration Fabric," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[47] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Bespoke Processors for Applications with Ultra-low Area and Power Constraints," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[48] R. Karakchi, L. O. Richards, and J. D. Bakos, "A Dynamically Reconfigurable Automata Processor Overlay," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017.

[49] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache Automaton," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[50] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "REAPR: Reconfigurable engine for automata processing," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[51] L. Vespa and N. Weng, "Deterministic Finite Automata Characterization and Optimization for Scalable Pattern Matching," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2011.

[52] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel Finite-state Machines," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[53] J. Qiu, Z. Zhao, and B. Ren, "MicroSpec: Speculation-Centric Fine-Grained Parallelization for FSM Computations," in *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2016.

[54] Z. Zhao and X. Shen, "On-the-Fly Principled Speculation for FSM Parallelization," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[55] 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 (ICS)*, 2017.

[56] J. C. Monteiro and A. L. Oliveira, "Finite state machine decomposition for low power," in *Proceedings of the Design Automation Conference (DAC)*, 1998.

[57] Y. Liu, S. Sezer, and J. McCanny, "NFA Decomposition and Multi-processing Architecture for Parallel Regular Expression Processing," in *Proceedings of the International SOC Conference*, 2009.

[58] P. Ashar, S. Devadas, and A. R. Newton, "A Unified Approach to the Decomposition and Re-decomposition of Sequential Machines," in *Proceedings of the Design Automation Conference (DAC)*, 1990.

[59] P. Ashar, S. Devadas, and A. R. Newton, *Finite State Machine Decomposition*. Springer US, 1992, pp. 117–168.