Parallel Automata Processor

Arun Subramaniyan   Reetuparna Das
University of Michigan-Ann Arbor
{arunsub,reetudas}@umich.edu

ABSTRACT

Finite State Machines (FSM) are widely used computation models for many application domains. These embarrassingly sequential applications with irregular memory access patterns perform poorly on conventional von-Neumann architectures. The Micron Automata Processor (AP) is an in-situ memory-based computational architecture that accelerates non-deterministic finite automata (NFA) processing in hardware. However, each FSM on the AP is processed sequentially, limiting potential speedups.

In this paper, we explore the FSM parallelization problem in the context of the AP. Extending classical parallelization techniques to NFAs executing on AP is non-trivial because of high state-transition tracking overheads and exponential computation complexity. We present the associated challenges and propose solutions that leverage both the unique properties of the NFAs (connected components, input symbol ranges, convergence, common parent states) and unique features in the AP (support for simultaneous transitions, low-overhead flow switching, state vector cache) to realize parallel NFA execution on the AP.

We evaluate our techniques against several important benchmarks including NFAs used for network intrusion detection, malware detection, text processing, protein motif searching, DNA sequencing, and data analytics. Our proposed parallelization scheme demonstrates significant speedup (25.5× on average) compared to sequential execution on AP. Prior work has already shown that sequential execution on AP is at least an order of magnitude better than GPUs, multi-core processors and Xeon Phi accelerator.

CCS CONCEPTS

• **Hardware** → **Emerging architectures**; • **Computer systems organization** → *Parallel architectures*; • **Theory of computation** → *Formal languages and automata theory*;

KEYWORDS

Emerging technologies (memory and computing), accelerators

ACM Reference format:
Arun Subramaniyan   Reetuparna Das. 2017. Parallel Automata Processor. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080207

1 INTRODUCTION

Finite State Machines (FSM) are widely used as a computation model in a number of application domains such as data analytics and data mining [9, 33], network security [13, 21, 24, 38], bioinformatics [11, 28, 36], tokenization of web pages [25], computational finance [1, 4] and software engineering [5, 10, 26]. These applications require processing tens to thousands of patterns for a stream of input data.

NFAs form the core of many end-to-end applications that utilize pattern matching. These pattern matching routines are typically implemented as *if-else* or *switch-case* nests in conventional CPUs and contribute to a significant fraction of the overall execution time because of poor branch behavior and irregular memory access patterns. For example, FSM-like computations form the core of many activities inside a web browser, taking about 40% of the loading time for many web pages [17]. The *oligo_scan* routine used in Weeder 2.0, an open-source tool for motif discovery in DNA sequences contributes 30-62% of the total runtime [35]. In the *Apriori* algorithm for frequent itemset mining, NFA processing accounts for 33-95% of the execution time, based on the frequency threshold [32]. Prior work [40] has shown that without accelerating FSM operations, it is infeasible for these applications to achieve sustained performance improvement, no matter how well other parts of these applications are parallelized (Amdahl's law).

FSM computation, especially Non-Determinstic Finite Automata (NFA) computation is inherently hard to speedup. Modern multicore processors are limited by the number of transitions they can do per thread in a given cycle, limiting the number of patterns they can identify. Their processing capability is also limited by the available memory bandwidth. GPGPUs have limited success with automaton processing because it is inherently dominated by irregular memory access patterns.

In comparison, custom architectures which facilitate in-situ computation in memories can facilitate highly parallel and energy efficient processing of finite state automata in hardware. For instance, Micron's Automata Processor (AP) [12] has been shown to accelerate several applications like entity resolution in databases [9](by 434×) and motif search in biological sequences [28] (by 201×). Recent efforts from Virginia's Center for Automata Processing has demonstrated that AP can outperform GPGPUs by 32×, and accelerators such as XeonPhi by 62×, across a wide variety of automata benchmarks [31]. Some key problems in bioinformatics like (28, 12), i.e., matching protein motifs of length 28, within edit distance 12 were previously unsolvable by von-Neumann architectures [28].

The Micron AP is a generalized accelerator supporting many application domains which can benefit from fast NFA processing and is not limited to regular expressions. The success of AP relies on three factors: massive parallelism, eliminating data movement (between memory and CPU) overheads, and reducing instruction processing overheads significantly. Massive parallelism follows from the fact that all state elements (mapped to columns in DRAM arrays) can be independently activated in a given cycle. An AP chip can support up to 48K transitions in each cycle. Thus it can efficiently execute massive Non-deterministic Finite Automata (NFA) that encapsulate hundreds to thousands of patterns.

While AP significantly improves the state-of-art, our work aims to further improve its performance by custom parallelization of FSM processing on AP. To our knowledge, no existing work parallelizes NFA FSMs for AP. Parallelization of FSM is known to be a hard problem due to its inherent sequential nature and high computational complexity. A logical way to parallelize FSM traversal is by partitioning the input string into segments, and processing these segments concurrently. The problem with this approach is that starting states for each segment are unknown except the first segment (which starts from the FSM's designated start states). The starting states for a segment are essentially the ending states of the previous segment. Prior work [25] has solved this by executing the input segment for *every state* of the FSM by leveraging classic parallel prefix-sum [22]. This method is referred to as enumerative computation as it enumerates all possible start states. We refer to the sequence of states visited by each enumeration start state as the *enumeration path*. Once the first segment is finished, we know the correct start states of the second segment and can pick the results of enumerated paths belonging to the correct start states (Section 2.2, and Figure 2 discuss an example enumeration).

While enumerative approach is promising, there are several challenges to realize it in AP. In a conventional processor a SIMD thread can process enumeration paths and thread's local variables keep track of the start state for each enumerated path. Tracking the start state of an enumerated path is important for combining the results of individual input segments as discussed above. In the AP, there is no notion of software threads or local variables which can keep track of start states of enumerated paths. A processing unit or half core simply accepts a stream of input symbols and does transitions via a routing matrix (custom interconnect) each cycle. Thus it can be challenging to execute concurrently and keep track of all enumeration paths. Another critical challenge to be solved is taming the enormous computational complexity of enumeration. Enumerations can be highly inefficient because in the worst case each state has to be enumerated. NFAs can have several thousands of states (See Table 1). In general enumeration of an FSM with n states, over k input segments can lead to an ideal speedup of k provided we have $n \times k$ independent computing resources. For typical NFAs, these resources far exceed what is available in AP. Current generation of Micron's D480 AP board supports up to 4 ranks, were each rank has 16 independent processing units or half cores. If each FSM occupies one half-core, we can afford to have at best, 64 independent processing units.

Our architecture solves the above problems by leveraging some unique properties of NFAs and unique features of the AP. For instance, we utilize the connected components (disconnected subgraphs) in an NFA to merge enumeration paths and thereby take advantage of the massive parallelism of the AP. Furthermore, the range (or all reachable states) of an input symbol can be utilized to prune the enumeration paths. Another NFA property we leverage is based on parent states in an NFA. If the start states of enumeration paths have a common parent state, they can be merged. Similar to prior work [25, 29], we observe enumeration paths converge at runtime and implement dynamic convergence checks in AP. To solve the start state tracking problem, we utilize AP flows. The flow abstraction also allows for near-zero overhead convergence checks. Our framework also discusses the details of combining the results from input segments, and hiding these processing overheads by leveraging the asymmetric finish times of input segments.

In summary this paper offers the following contributions:

- This is the first work to explore parallelization of non-deterministic FSMs on the Automata Processor (AP). In particular we examine enumerative approaches to parallelize processing of non-deterministic FSMs. AP's unique in-memory architecture and huge computational complexity of enumerations pose interesting challenges for parallelising FSMs.
- Our work systematically explores the challenges in parallelizing FSMs for AP, such as tracking of enumeration paths and explosion in computational complexity. The computational complexity is tackled by leveraging unique properties of NFAs, such as connected components, common parents, input symbol range, convergence, and monitoring unproductive enumeration paths. We utilize AP's flows to solve the state tracking problem. The flow abstraction also allows implementation of near-zero convergence checks. Our framework also discusses the details of composing the results at the host and proposes techniques to hide this overhead by utilizing asymmetric finish time of different input segments.
- We evaluate our techniques against several important benchmarks including NFAs used for network intrusion detection, malware detection, text processing, protein motif searching, DNA sequencing, and data analytics. Our proposed parallelization scheme demonstrates significant speedup (25.5× on average) compared to sequential execution on AP. Prior work has already shown that sequential execution on AP is at least an order of magnitude better than GPUs, multi-core processors and the Xeon Phi accelerator [31].

2 BACKGROUND

In this section we provide a brief background on the Automata Processor and enumerative techniques for parallelizing FSM processing.

2.1 NFA and Automata Processor

A Non-deterministic Finite Automata (NFA) is formally described by a quintuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is a set of states, Σ is the input symbol alphabet, q_0 is the set of start states and F is the set of *reporting* or *accepting* states. The transition function $\delta\left(Q, \alpha\right)$ defines the set of states reached by Q on input symbol α. The non-determinism is due to the fact that an NFA can have multiple states active at the same time and have multiple transitions on the same input symbol.

NFA computation entails processing a stream of input symbols one at a time, determining which of the current active states match an incoming input symbol (*state match*) and looking up the transition function to determine the next set of active states (*state transition*).

Conventional compute-centric architectures store the complete transition function as a lookup table in the cache/memory. Since a lookup is required for every active state on every input symbol, symbol processing is bottlenecked by the available memory bandwidth. This leads to performance degradation especially for large NFAs with many active states. With limited memory bandwidth, the number of state transitions that can be processed in parallel is also limited. Converting these NFAs to equivalent DFAs also cannot help improve performance since it leads to exponential growth in the number of states.

The memory-centric Automata Processor (AP) accelerates finite state automata processing by implementing NFA states and state transitions in memory. Each automata board fits in a DIMM slot and can be interfaced to a host CPU/FPGA using the DDR/PCIe interface. Figure 1 illustrates the automata processor architecture.

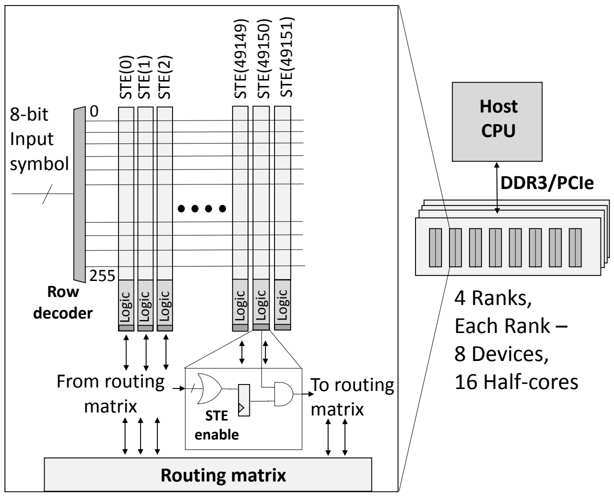


Figure 1: Automata Processor Overview.

For processing in AP, the classic representation of NFAs is transformed to a compact ANML NFA representation [12] where *each state has valid incoming transitions for only one input symbol*. Thus each state in an ANML NFA can be *labeled* by one unique input symbol. ANML NFA computation entails processing a stream of input symbols one at a time. Initially, all the start states are active states. Each step has two phases. In the *state match* phase, we identify which of the active states have the same label as the current input symbol. In the *state transition* phase, we look up the transition table to determine the destination states for these matched states. These destination states would become the active states for the next step.

In AP, the FSM states (called State-Transition Elements or STEs) are stored as columns in DRAM arrays (256 bits). Each STE is programmed to the one-hot encoding of the 8-bit input symbol (same as it is *label*) that it is required to match against. For example, for an STE to match the input symbol *a*, the bit position corresponding to the 97^{th} row must be set to 1.

Each cycle, the input symbol (ASCII alphabet) is broadcast to all DRAM arrays and serves as the *row address*. If an STE has a '1' bit set in the row, it means that the label of the state it has stored matches the input symbol. *State match* is then simply a DRAM row read operation, with the input symbol as the *row address* and the contents of the row determining the STEs that match against the input symbol. Thus, by broadcasting the input symbol to all DRAM arrays, it is possible to determine in parallel all the states which match with the current input symbol.

State transitions between currently active states to next states is accomplished by a proprietary interconnect (*routing matrix*) which encodes the transition function. Reconfiguring this interconnect requires a costly recompilation step. Only the states which matched with the current input symbol and are active, undergo state transition. The bits of a register (*active state mask*) at the bottom of STE columns determine the set of STEs that are *active* in a particular symbol cycle. These bits are initially set for only *start states*. All active bits for all STEs can be independently set in a given cycle, as they are all mapped to different columns of DRAM arrays. Thus, AP allows any number of transitions to be triggered in a given cycle, enabling massive parallelism and efficient NFA processing.

Due to physical routing constraints, each logical AP device (D480) is organized hierarchically as half-cores, blocks, rows and STEs with no state transitions across half-cores. Therefore, each half-core can be considered as the smallest unit of parallellization for partitioning into input segments. STEs configured as *reporting* have no outgoing transitions and their results are communicated to the CPU by writing to an *output event buffer*. Each entry in this buffer contains a report code and byte offset (in input stream) of the symbol causing the report. These entries are parsed in the host and communicated to the user.

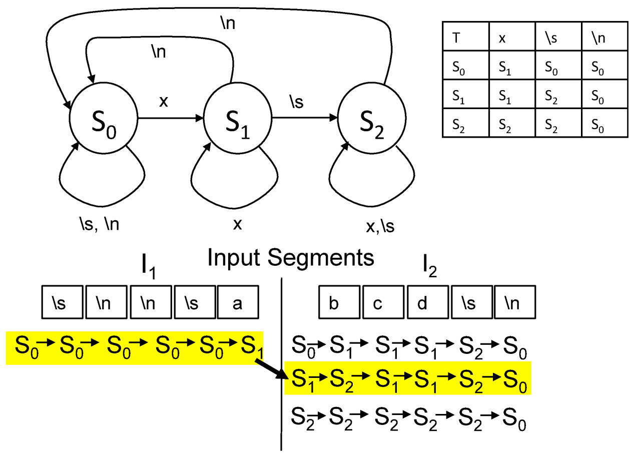
The current generation AP contains 4 ranks of 8 D480 devices each. Each device consists of 2 half-cores encompassing 49152 STEs, organized into 192 blocks. Each block further contains 256 rows and each row stores 16 STEs. The Micron AP also includes block-level power gating circuitry that disables a block with no active states.

In terms of the reporting hierarchy, each AP device is also partitioned into 6 output regions, with each output region storing a maximum of 1024 reporting elements. Also present are 768 counters and 2304 programmable boolean elements to augment pattern matching functionality.

2.2 Parallel FSM

Parallelizing FSM traversal is known to be extremely difficult due to the inherent sequential nature of computation arising because of dependencies between every consecutive state transitions. One way to parallelize FSM traversal is by partitioning the input string into segments, and processing these segments concurrently. This is feasible because FSM computation can be expressed as a *composition of transition functions* [15]. Parallelization is possible because transition function composition is *associative*. Figure 2 shows an example of parallelizing the FSM with two input segments (I_1 and I_2) each with five symbols. The FSM shown detects the first word in every line. The transition table is shown on right. Both these segments can be executed in parallel to provide a speedup of 2× over sequential baseline.

However, the starting states for each input segment are unknown except the first segment (which starts from initial start states). The starting states for a segment are essentially the ending states of the


\n
\n
x
\s
S_0
S_1
S_2
\s, \n
x
x,\s
I_1
Input Segments
I_2
\s \n \n \s a
b c d \s \n
$S_0 \to S_0 \to S_0 \to S_0 \to S_0 \to S_1$
$S_0 \to S_1 \to S_1 \to S_1 \to S_2 \to S_0$
$S_1 \to S_2 \to S_1 \to S_1 \to S_2 \to S_0$
$S_2 \to S_2 \to S_2 \to S_2 \to S_2 \to S_0$

T	x	\s	\n
S_0	S_1	S_0	S_0
S_1	S_1	S_2	S_0
S_2	S_2	S_2	S_0

Figure 2: An FSM example with enumeration.

previous segment. These dependencies prevent concurrent execution among threads.This problem can be solved by leveraging classic parallel prefix-sum [22]. The basic idea is to execute the second segment for *every state* of the FSM. This method is referred to as an enumerative computation as it enumerates all possible start states [25].

In Figure 2 the start state of the first segment is known (S_0) which is the start state of the FSM. However, the start states of input segment I_2 are unknown. Figure 2 shows an example enumeration for the second input segment, I_2. This example FSM has 3 states, so each segment (except the first) enumerates all 3 states. Once the first segment has finished, it can pick the correct or *true paths* from the enumerated paths of the second segment and discard *false paths*. Thus, final results can be obtained by combining the intermediate results of all input segments. The true path for I_2 in Figure 2 starts at S_1, the remaining two paths are false paths. The final path of the FSM is highlighted.

The evident disadvantage of this method is the exponential blowup in computational complexity for processing each input segment. Consider a benchmark `Protomata`, an NFA which encapsulates 2340 known string patterns called motifs in protein sequences. Matching with protein motifs is used to accelerate the discovery of unknown motifs in biological sequences in the field of bioinformatics. `Protomata` has 38,251 states. Enumerating all these states will make the parallel version orders of magnitude slower than the serial version.

Thus, unless we have the massive computational resources equivalent to n (states in a NFA) $\times$ k (number of input segments) processing units, enumeration can lead to slowdowns instead of speedup. In this paper we explore different techniques for enumerating NFAs on AP's unique architecture and taming the computational complexity of enumerations. For instance, we find that the set of reachable states of an input symbol, and number of connected components can drastically reduce the number of enumeration paths. Similar to prior works on parallelization of deterministic finite automata (DFAs) [25], we find that many enumeration paths converge and design an AP architecture which is capable of near-zero cost dynamic convergence checks. The next section discusses the above and other optimizations which make parallelization of NFAs on AP profitable.

3 PARALLEL AUTOMATA PROCESSOR

This section discusses our proposed framework and architectural enhancements needed for effective parallelization of NFAs on the Automata Processor (AP).

3.1 Range Guided Input Partitioning

Enumerating from all states of an FSM will lead to exponential computational complexity. Fortunately, many of these states are impossible start states for the particular input segment. The *range of an input symbol* is defined as the union of the set of all reachable states, considering transitions from *all states* in the FSM that have a transition defined for that symbol. During actual execution, the range of the *last input symbol* in a particular segment determines accurately the subset of start states for the next segment. Any states outside this range are impossible start states. Our proposed parallelization framework partitions the input such that input segments end at frequently occurring symbols with small ranges to take advantage of minimum range symbols. The symbol chosen for an FSM is obtained by offline profiling. Frequently occurring symbols are chosen to ensure that the size of input segments are roughly equal.

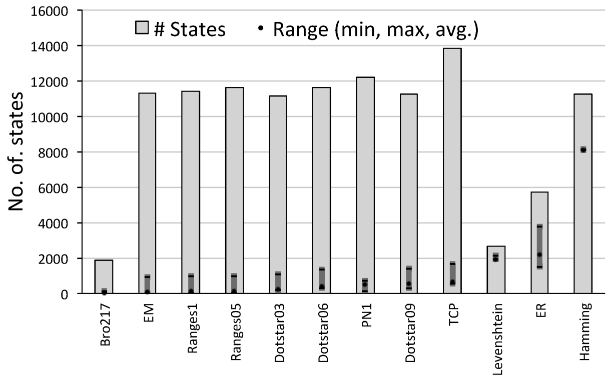
Figure 3 shows the average, and minimum range across 256 input symbols. Note that AP only accepts 8 bit symbols, limiting us to 256 symbols. The bar depicts the total number of states in the system and the dark line indicates the minimum, average and maximum range across 256 symbols. The figure demonstrates that ranges of input symbols is a small fraction of total states, greatly reducing the complexity of enumeration. For instance for `Protomata`, we can reduce the enumerated paths from 38251 start states to 667 start states. For some benchmarks the average range of symbols is almost as large as half the state space, example `SPM`. We discuss other optimizations for these benchmarks in the next section. Table 1 lists the range of the symbol chosen for input partitioning for each benchmark.

3.2 Enumeration using Flows

In this section, we provide an understanding of why we need flows to support enumeration, followed by a brief understanding of flows in AP, and how we map enumeration paths to flows.

Ideally, one can activate all start states and execute all enumerations simultaneously on one copy of the FSM. This is possible and *correct*, given that AP seamlessly implements any number of simultaneous transitions in a given cycle, and there is no limit on the number of start states that can be activated. This seems like a perfect solution, except that we lose all information about enumeration paths. After processing the input segments we know what are the end states for *all* enumeration paths, but there is no way of knowing which path lead to which end states. Recall that after an input segment finishes execution, it will inform the next input segment which enumeration paths were the *true paths* and which paths were *false paths*. The next input segment then must only use the results and end states of *true paths* and discard false paths.

In a conventional processor, enumeration paths are executed on SIMD threads and thread's local variables keep track of the start state



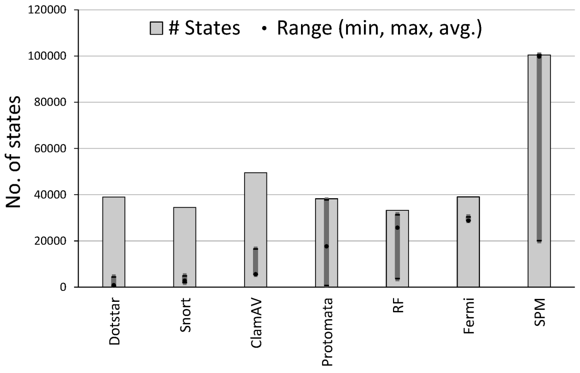


Figure 3: Range of symbols for different benchmarks. a) Smaller NFAs with state space limited to 15K states. b) Large NFAs

of each path. In the AP, however there is no notion of local variables or state tracking. The only way to implement state tracking is by propagating the start state via the routing matrix. Routing matrix currently just routes 1 bit per state element pair (which encodes transition between two state elements) and is already known to be a bottleneck in the system, both in determining the cycle time, and area complexity (occupies ≈30% of the chip). Augmenting the routing matrix with state information leads to exponential space complexity. Another possibility is replicating the FSM and executing the different enumeration paths in a separate replicated copy. Since each copy is mapped to a different half-core (or half-cores for large FSMs), we need as many half-cores as the number of enumeration paths. A typical AP D480 board has 64 half-cores which is far smaller than the number of enumeration paths for most of our benchmarks. This means that we can run at best one input segment at a time, leading to no speedup. Recall that speedup is proportional to number of input segments executing in parallel. Ideally, we would like to run 64 input segments one on each half-core and obtain a speedup of 64×. Furthermore, mapping enumerations to different half-cores also complicates checking for convergence between the paths, because there is no path of direct communication between half-cores on different dies or ranks.

Our solution leverages AP's flows to solve the above problem of tracking the start states of enumeration flows. Another unique advantage of using flows is that we can do low cost convergence checks, as we explain in Section 3.3.3. AP's flows allow multiple users to time multiplex the AP for independent input streams. Each chip is equipped with a state vector cache which can store up to 512 *state vectors*. A state vector represents the state of a FSM execution and consists of 59,936 bits [(256 enable bits per block + 56 counter bits per block) x 192 blocks + 32 count]. The state vector allows AP to context switch between two independent executions much like the register save/restore that allows tasks to context switch on traditional CPU architectures [3, 14]. The output match events also encapsulate a flow identifier.

In our architecture each enumeration path is mapped to an independent flow and time division multiplexed on the same half-core. By association to a flow identifier, we can easily track the enumeration paths that belong to each flow. The host CPU keeps a flow table which tracks which start states (or enumeration paths) are mapped to which flows. Each input segment comprising of several flows is processed in several Time Division Multiplexing (TDM) steps. Each flow processes k symbols before a context switch. Once all flows process k symbols, a TDM step is finished. Each TDM step thus processes k input symbols across all flows. The input symbols need to buffered until an entire TDM step is completed. A pointer in the input buffer is re-winded to the correct position after each context switch within a TDM step.

The context switch between flows in our system is as fast as 3 AP symbol cycles. This follows from the fact that in our architecture, each enumerated path (and hence each flow) utilizes the same FSM. Thus there is no need to load the memory arrays or configure the routing matrix during a context switch between flows. To change flows, AP transfers the current state to the state vector cache in the first cycle, then retrieves a previous state from the cache in the second cycle and finally loads it into the mask register (state-enable bits) and counters in the third cycle.

3.3 Merging Flows

The speedup which can be obtained from our parallelization techniques relies on two factors, the number of input segments executing in parallel and the time taken to complete each input segment. In general the speedup obtained is equal to number of input segments divided by slowdown experienced by the slowest input segment. Slowdown of an input segment is simply the time it takes when compared to the time it would have taken had we known the exact start states for that segment.

By utilizing flows we have maximized the number of input segments, however time division multiplexing of flows also slows down processing of each input segment. Specifically the processing time of each input segment is proportional to number of flows for that segment. The range guided partitioning method significantly reduces the number of enumeration paths and hence the number of flows needed. However, the number of flows remaining is still large. This section discusses several techniques to further reduce the number of active flows by merging flows.

3.3.1 ***Leveraging Connected Components.*** Intuitively, any two flows can be merged if we can guarantee that there would be no overlap between their state-spaces on any transition, i.e., they belong to different connected components. Since the AP supports any number of simultaneous transitions on a given cycle (subject to routing constraints), we can merge states belonging to different flows and execute them *simultaneously* in the same flow. This observation can be generalized to merge any number of flows as long as we can guarantee their state-spaces do not overlap.

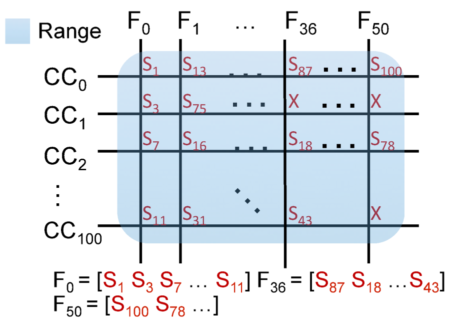


Figure 4: Merging paths belonging to separate connected components. Initially we start with the entire Range indicated by all the states in the shaded box. Horizontal lines are all states in the range which belong to connected component CC$_i$. Note 'X' means there is no state. Vertical line indicates a flow F$_j$ after merging.

Interestingly we find that many of our benchmarks have large number of connected components (or sub-graphs) which do not overlap with each other, i.e. there is no transition edge between the states belonging to different connected components. Intuitively this makes sense because each NFA collects a number of regular expressions or patterns. Patterns with common prefixes belong to the same connected component but patterns which do not share any common prefix belong to separate connected components. Table 1 lists the number of connected components in our benchmarks. Our insight is that we can merge flows belonging to separate connected components to reduce the number of flows. The AP compiler in one of its initial stages also partitions the FSM into distinct sub-graphs, however, to ease placement and routing [1]. Figure 4 shows our algorithm to merge flows belonging to separate connected components. We group the states obtained by range-guided input partitioning into different connected components. The range table consists of all these states as shown in the shaded box. All the 5000 states in the range are grouped into CC_i groups in the figure. The states on a vertical line through each group were previously mapped to separate flows. Note that we split the states in the same connected component across separate flows so that we can uniquely distinguish them (true paths vs false paths). It can be seen that the number of active flows is equivalent to the number of vertical lines. In the figure we have 50 vertical lines, hence 50 active flows. Thus we started from 5000 enumeration paths in the range and merged them into 50 flows. Once the flow finishes, the end states of each state belonging to the flow can be uniquely identified by simply masking with a bitmap consisting of the state space of each connected component.

3.3.2 ***Active State Group and Common Parent.*** NFAs usually have several states which are always active due to self-loops on all possible symbols (self loop labelled *). These states artificially increase the number of enumeration paths. Given that these states are always active, by definition they belong to the *true path* and can be all combined into one flow which we refer to as the Active State Group (ASG) flow. The output results of this flow are always reported.

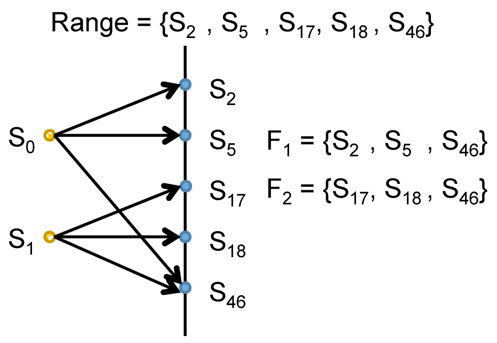


Figure 5: Merging states in the range with common parent.

We also observe that states in the range of an input symbol which originate due to the same parent state belong to the same enumeration path. This follows from the fact that in a NFA, there can be many outgoing transitions from a state on a given input symbol. Had we started the input segment one symbol earlier, all these states would have been part of the same enumeration path. Thus we map all enumeration paths with a common parent to the same flow. Figure 5 illustrates the concept. The range consists of states: S_2, S_5, S_{17}, S_{18} and S_{46}. Initially, this would lead to 5 flows. Since S_2, S_5, S_{46} have a common parent S_0, they can be merged into one flow. Similarly, S_{17}, S_{18}, S_{46} have a common parent S_1 and can be merged into one flow, resulting in only 2 flows. Note that for correctness S_{46} has to be included in both flows.

3.3.3 ***Dynamic Convergence Checks.*** Enumerations can be made more efficient by leveraging convergence. We can observe an example of convergence in Figure 2. Consider input segment I_2, starting with three different start states S_0, S_1, and S_2. The figure shows three enumeration paths for the state sequence, one for each starting state. After processing the first two symbols, the first two paths get into the same state S_1. After that, these two paths would keep producing the same state sequence as they will observe the same symbols. Hence there is no need to do redundant computation, and the first two paths can be merged into one path. Thus an enumeration which started with 3 paths reduces to 2 paths after processing the first two symbols. Prior work on parallelizing DFAs have observed that state convergence property widely exists in many FSMs [25].

Flow based enumeration allows for easy convergence checks. In our architecture convergence checks can be implemented by comparing the state vectors in the state vector cache. Comparison requires a simple bitwise logic comparator (one xor gate per state bit and a common wired and) to be augmented to the state vector cache. Accessing a state vector entry and comparing to a stored vector takes one symbol cycle. If we have f active flows, convergence checks over all the flows can take up to $f \times f$ symbol cycles. Fortunately, the convergence checks can be entirely overlapped with symbol processing because the state vector cache is not used while processing symbols. However, combining enumeration paths from different connected components into the same flow reduces the probability of convergence. Thus, we invoke convergence checks every ten TDM steps.

3.3.4 ***Deactivation Checks.*** Often many paths in an FSM are not productive. For instance an enumeration path may process a few symbols successfully, making transitions for each symbol until it comes across a symbol for which none of the active states match. In this case, the path is no longer productive and must be deactivated to save time. In practice, we find many enumerations paths become unproductive after processing few input symbols. If all the enumeration paths mapped to a flow are unproductive the entire flow can be deactivated to save time. We implement the flow deactivation logic by simply comparing the state bits in the state vector to a zero mask during a context switch and invalidate the state vector if there is a match to the zero mask. We observe that many flows get deactivated within processing few symbols (less than 20 symbols), so we do a few extra deactivation checks even before the first TDM step completes.

3.4 Composition of Input Partitions

Once the input segments finish, the final output results can be obtained by combining the results of true paths of each segment. The host CPU reads the final state vector from the AP and then constructs a Boolean array indicating which flow has results for true enumeration paths. This Boolean array is checked when reading the results out of the output buffer. Each output buffer entry has few bits indicating the flow identifier. *Only results for the output buffer entries which match with true flows are reported.* This computation is done by utilizing the pre-computed range table and masks for connected components (discussed in Sections 3.1 and 3.3.1).

It takes 1668 symbol cycles to transfer the final state vector from AP to the host CPU's save buffer [1]. It takes another few tens of symbol cycles to interpret the state vector to figure out which flows encompass true paths in the host CPU's core. We find that this overhead is not insignificant and thus explore methods to overlap this overhead with input segment processing time at AP. The *asymmetric finish times of input segments* can be leveraged for this purpose. In general different input segments finish at different times based on different rates of deactivations and convergence. Furthermore, the first input segment executes only the true path, so it is likely to finish quite ahead of others.

Thus the first input segment can read its final state vector and create a Boolean array indicating true flows, while the second input segment is still processing its symbols. Thus its composition overhead is overlapped with the second input segment's execution. In addition to this, the Boolean array can be utilized to create a Flow Invalidation Vector (FIV) which can be used to invalidate all false flows in the second input segment. In addition to overlapping of composition overhead whenever possible, this method can further reduce the active flows and speedup input segment processing. The concept can be generalized to all input segments. Figure 6 illustrates the above concepts. The first input segment I_1 has only one flow and completes first. The second input segment I_2 on the other hand has many flows (indicated by thickness of line) and is chugging along. The first flows takes T_{cpu} time to compute its Boolean array for true flows and FIV. The FIV is passed along to input segment I_2. Note T_{cpu} is hidden by I_2 processing time and after receiving the FIV the number of flows in I_2 reduce substantially. In some cases when all flows deactivate or converge to only one flow, there is no need to spend T_{cpu} cycles.

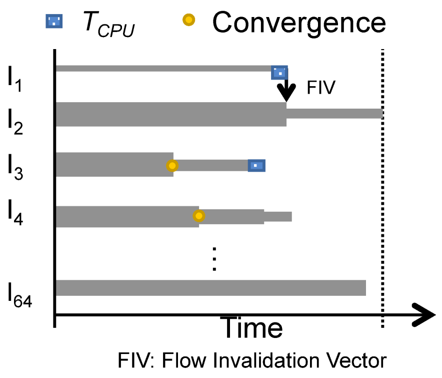


Figure 6: Overlapping T_{cpu} with next segment processing time and dynamically merging flows based on previous segments end results.

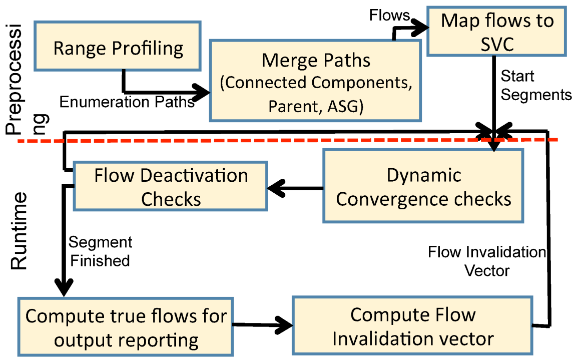


Figure 7: Overall framework.

3.5 Put It Together

This section describes our overall framework for parallelizing NFAs on the AP. Figure 7 brings together all the concepts discussed in this section to illustrate our overall framework. The parallelization framework consists of pre-processing steps and dynamic runtime steps. First, the range is computed for all input symbols and a frequently occurring symbol is chosen based on profiling (Section 3.1). This is followed by the merging the states in the range table into flows based on connected components (Section 3.3.1), common parents, and ASG (Section 3.3.2). This step generate the contents for the State Vector Cache (SVC). The state vector cache is then loaded onto the AP chips. This pre-processing can be augmented to the compilation and configuration process for the AP. Following this, the input is partitioned at boundaries of the chosen range symbol. Each input segment starts getting processed in parallel on AP half-cores. Deactivation and convergence checks occur dynamically to invalidate redundant or unproductive flows (Sections 3.3.4 and 3.3.3). A segment can also receive a flow invalidation vector from the previous segment during its runtime. Once an input segment finishes, the composition of output reports happens in the CPU (Section 3.4).

4 METHODOLOGY

The proposed approach and optimizations are evaluated on a wide range of benchmark FSMs from the *ANMLZoo* [31] and the *Regex* [8] benchmark suites. These real world benchmarks span multiple domains including network packet monitoring [8], gene sequence matching [28] and natural language processing [41]. Table 1 summarizes some of the important characteristics of these FSMs and the parameters used in our simulations. We first describe these workloads in detail, our modifications to these workloads, followed by a discussion on the experimental setup.

4.1 FSM workloads

The *Regex* suite consisting of 8 workloads, contains both real-world and synthetic regular expressions primarily meant for network intrusion detection. *ExactMatch* looks for exact pattern matches in the input stream. The *Dotstar* rulesets are parameterized by the fraction of unbounded repetitions of the wildcard $.*$. The *Ranges* dataset accounts for character classes in regular expressions. These are parameterized by the fraction of the ruleset that contains character classes. *Bro217* is an open-sourced set of 217 regular expressions used for packet sniffing. The *TCP* workload consists of regular expressions used for packet header filtering prior to actual packet inspection.

We use the synthetic trace generator tool from Becchi and others [8] to generate input traces for these workloads. We use traces with $p_m = 0.75$, which is the probability that a state matches on an input character and activates subsequent states as in a depth-wise traversal. $p_m = 0.75$ has been shown to be representative of real-world traffic [8]. Both 1 MB and 10 MB traces are used in our evaluation.

While the *Regex* suite targeted only the network security domain, several recent efforts have uncovered relatively diverse automata-based applications in bioinformatics, data mining and natural language processing that are not necessarily derived from regular expressions [28, 33, 41]. The *ANMLZoo* benchmark suite is one of the first attempts to group these benchmarks and create "standard candles" for comparing different automata architectures and algorithms. While these benchmarks were developed aiming to saturate the resources on one AP chip, newer versions of the AP compiler place and route some of these automata (e.g., Levenshtein, Entity Resolution) on multiple AP dies since several of these benchmarks are densely connected. We account for this physical automata distribution in our experimental results. The *ANMLZoo* benchmarks along with their input parameters are tabulated in Table 1.

The *Snort* ruleset for network intrusion detection is from Snapshot 2.9.7.0. *ClamAV* contains a set of regular expressions from an open source virus database. *Dotstar* in this suite contains a combination of 5%, 10% and 20% wildcard .* repetitions. *Levenshtein* implements the Levenshtein automata used for fuzzy string matching with deletions and insertions allowed. In this suite, strings are of length 24, with edit distance = 3. It is used to match against encoded DNA sequences. *Hamming* is similar to *Levenshtein* and counts the number of mismatches against input strings. *Entity Resolution* has applicability in databases, when the same entity represented with small differences is required to be resolved correctly. For example, names of individuals J. L. Doe and John Doe. *PowerEN* is part of a proprietary set of regular expressions from IBM. *Fermi* predicts high-energy particle paths by matching against known trajectories. *Random Forest* is a machine learning application that implements hand-written digit classification and *SPM* is a data-mining application that mines sequential relations between item transactions to predict future transactions.

It can be seen from Table 1 that the state-space of these benchmarks varies greatly and so does the average active set. Furthermore several of these benchmarks also exhibit potential for compression. Similar to the work in [31], we compress automata using the *common prefix merging* technique [7] prior to execution to remove redundant traversals from the automata. For *ClamAV*, *Fermi* and *Random Forest* we do not employ *common prefix merging* as it reduces the number of connected components with only minor benefits in terms of reduction in number of states. We use both the 1 MB and 10 MB representative input traces provided with each benchmark to evaluate our optimizations. The cost for pre-processing input stream and post-processing output reports is minor. Few symbols at the boundary of input segments (64kB for 1 rank and 16kB for 4 ranks) are compared to pre-chosen low-range symbol and chosen for partitioning.

4.2 Experimental Setup

We utilize the open-source virtual automata simulator *VASim* [31] to simulate the proposed architecture as well as implement the range-guided input partitioning, and all flow merging optimizations discussed in Section 3.3. *VASim* allows for fast non-deterministic finite automata emulation by traversing paths only for active states. It supports multi-threading and can partition the input stream and automata processing across many threads. We partition the input stream into nearly equal chunks at symbols with small range and execute a *VASim* context for each flow. Deactivation and convergence of flows is tracked in the simulator as described in Section 3.3.

The Automata Processor can deterministically process 1 symbol every 7.5 ns (as long as its output buffers to convey reports are not full), so the latency for symbol processing is known apriori. Context switching between flows requires writing out the old context into the State Vector Cache, reading the new context and loading the new state in the counters and STEs. This has been estimated as 3 cycles [3, 14]. Transferring the 59,936 bit state-vector to the CPU for dynamic invalidation of incorrect flows takes 1668 symbol cycles [2]. On the return path from the CPU, transferring the 512 bit-vector to invalidate flows takes 15 AP cycles. We find that in several of our benchmarks, flows are deactivated before the completion of execution of the previous chunk and we do not incur this extra

#	Benchmark	States	Range	Connected Components	Num. Half-Cores	Input Segments (1 Rank)	Input Segments (4 Ranks)
1	Dotstar03	11124	163	56	1	16	64
2	Dotstar06	11598	315	54	1	16	64
3	Dotstar09	11229	314	51	1	16	64
4	Ranges05	11596	1	63	1	16	64
5	Ranges1	11418	1	57	1	16	64
6	ExactMatch	11270	1	53	1	16	64
7	Bro217	1893	6	59	1	16	64
8	TCP	13834	550	57	1	16	64
9	PowerEN1	12195	466	62	1	16	64
10	Fermi	40783	30027	2399	2	8	32
11	RandomForest	33220	1616	1661	2	8	32
12	SPM	100500	20100	5025	2	8	32
13	Dotstar	38951	600	90	2	8	32
14	Hamming	11254	8151	49	2	8	32
15	Protomata	38251	667	513	2	8	32
16	Levenshtein	2660	2090	4	3	5	21
17	EntityResolution	5689	1515	5	3	5	21
18	Snort	34480	792	90	3	5	21
19	ClamAV	49538	5452	515	3	5	21

Table 1: Benchmark Characteristics

invalidation overhead in the common case. We assume a latency of 7.5 ns (1 symbol cycle) to determine if any two flows have converged.

We estimate the time taken to identify false paths (and false flows) on a Xeon E3-1240V5 workstation with 8 cores and 32GB RAM. It is also possible for our enumerative approach to falsely trigger reporting elements in some of its false paths. For each benchmark we also account for the overheads of removing these false positives in the output reports as described in Section 3.4.

5 RESULTS

In this section we first present the speedups obtained by the proposed Parallel Automata Processor Architecture (PAP), followed by a detailed explanation of the reasons for this speedup. We also present an analysis of the different sources of overhead introduced by the proposed optimizations.

5.1 Overall Speedup

Figure 8 shows the speedups obtained by our proposed Parallel Automata Processor Architecture (PAP), when compared to the baseline AP architecture. We present speedups for both 1 AP rank (8 D480 devices) and 4 AP ranks (32 D480 devices in the current AP generation) and 1 MB and 10 MB input streams. We also exploit the parallelism offered by each of the half-cores in a D480 device when our FSMs can fit in a single half-core. Table 1 details the AP footprint and number of input segments created for each of our benchmark FSMs. The *Ideal* legend in the figure equals the number of input segments that can be processed in parallel. Overall, across the complete range of 19 benchmark FSMs, for the 1 rank and 4 rank cases, PAP achieves 6.6× and 18.8× speedup for the 1MB input stream and 7.6× and 25.5× speedup for the 10MB input stream.

It can be seen from the figure that PAP outperforms the sequential AP baseline for most benchmarks. A noticeable trend is the larger performance gains with the 10 MB stream. This is because the larger stream provides opportunity for creating larger input segments. These larger input segments help in reducing in the number of active flows due to the deactivation and convergence properties of the FSMs discussed in Sections 3.3.3 and 3.3.4 and the associated flow switching overhead. Furthermore, large input segments also help amortize the cost of false path invalidation and input composition in the CPU. For benchmarks with small input symbol ranges, in particular *Ranges05*, *Ranges1* and *ExactMatch*, PAP achieves near ideal speedup both in the 1 rank and 4 rank cases. Even FSMs with significantly large number of initial flows like *SPM* (20101) and *Hamming* (8152), achieve greater than 16× speedup in the 10MB case because of the connected components and common parent optimizations for flow reduction discussed in Section 3.3. A detailed analysis of our optimizations for flow reduction is presented in the next section.

It is also important to note that the current generation of AP only supports 512 active flows in its State Vector Cache per D480 AP device. Several of the studied FSMs significantly exceed the 512 limit as can be noticed from the Range entries in Table 1. The proposed flow reduction optimizations are therefore essential to the success of the proposed parallelization approach. For benchmarks like *Fermi*, consisting of a large number of active states and *Entity Resolution* with highly dense connected components, our optimizations are unable to significantly reduce the initial number of active flows, limiting speedups.

Our parallel approach is never worse than the sequential baseline as the half-core processing the first input segment, after completion, continues to process the remaining segments (golden execution). In case this half-core finishes processing all input segments, we invalidate all other executing flows and report results for the golden execution. A more aggressive policy need not wait for completion of golden execution. It can invalidate all flows after the golden

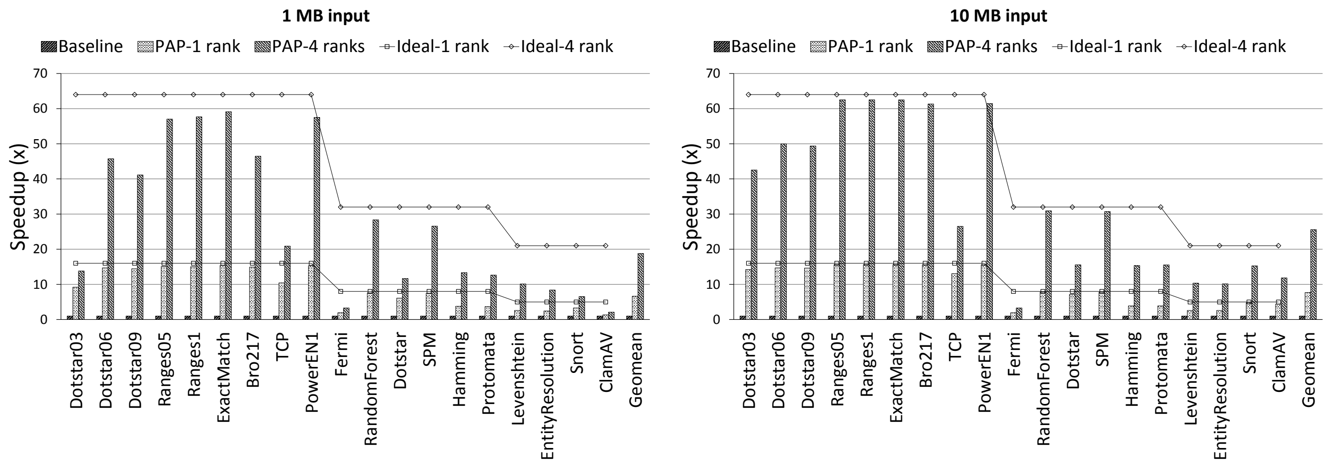


Figure 8: Speedups obtained by our proposed architecture Parallel Automata Processor Architecture

execution has finished *x* segments (with *x* calculated based on the minimum expected speedup).

5.2 Active Flow Set

Figure 9 shows the contribution of each of our flow reduction optimizations in achieving the speedups discussed in the previous section. We also plot the average number of active flows in different benchmarks for the 1 MB input stream case as an example. Note that the y-axis scale is logarithmic.

While benchmarks with small input symbol ranges are inherently good candidates for the flow-based enumeration scheme, it can be seen that several of our benchmarks have greater than 1000 states in their initial range. In particular *SPM* consisting of 20101 initial flows and 5025 distinct connected components greatly benefits from the connected components optimization which reduces these to 5 flows. Note that our pre-processing step identifies frequently occurring input symbols with small range. In their absence, other optimizations like connected components prune the number of enumeration flows as discussed above. All optimizations work synergistically to reduce the number of enumeration paths.

We noticed that even though the connected components optimization greatly helped reduce the number of flows (e.g., from 467 to 32 for *PowerEN*) several flows remained active for the complete execution. Investigating further revealed that the connected components optimization artificially creates more flows for states originating from the same parent as discussed in Section 3.3.2. With the proposed common parent merging algorithm, we achieved a 1.6× and 1.4× reduction in flows for *Levenshtein* and *Hamming*. Also, the dynamic flow convergence and deactivation checks discussed in Section 3.3 contribute to a great reduction in number of active flows for all benchmarks. In particular we see an order of magnitude reduction in number of flows for *Dotstar0x* and several orders of magnitude improvement for *RandomForest*, *Fermi* and *SPM*.

5.3 Overheads

This section discusses the different sources of overhead in the proposed PAP architecture.

Flow Switching and Dynamic Checks: It can be seen from Figure 10 that the overheads of context switching between flows are less than 2% for most benchmarks. As discussed before, since the number of active flows greatly reduces as input symbols are processed, the corresponding convergence and deactivation checking overheads also reduce. Furthermore these checks can be overlapped with symbol processing. *ClamAV* however has a large number of active flows and sees 2.4% overhead. This accounts for the relatively low speedup for *ClamAV* when compared to other benchmarks in Figure 8. Our speedup for 1 MB inputs reduces by on average, 0.5% (1.75% worst case) and 1.2% (5.04% worst case) for 2× (6 cycles) and 4× (12 cycles) context switch time respectively. The context switch overhead is proportional to the number of active flows, which greatly reduce as symbols are processed due to convergence and deactivations. Also, dynamic convergence checks can be overlapped with symbol processing, since these checks are carried out on state vector cache entries, which do not participate in symbol processing (state transitions).

False Path Decoding: Figure 11 illustrates the overheads of decoding false paths at the host CPU after an input segment finishes and sending a flow invalidation vector (FIV) to the next segment. On an average most benchmarks see around 2000 symbol cycles overhead. Fortunately, this cost is largely amortized because of two reasons: (1) it can be overlapped with symbol processing in subsequent segments, (2) these invalidations are infrequent since several flows have either already converged or have been deactivated and do not require this invalidation.

Output Reports: The AP uses reporting elements to inform the host CPU about pattern matches against the input. The host reads the output event buffer on the AP and decodes each entry to finally report matches to the user. Since our approach uses enumeration for parallelization, *false paths* are traversed and output events may be generated along these false paths. Figure 12 illustrates the increase in output reports due to false paths for each of the benchmarks. These false positives are filtered out on the host. We account for the time taken for post-processing the output reports in both baseline AP and PAP for our final speedup measurements shown in Figure 8.

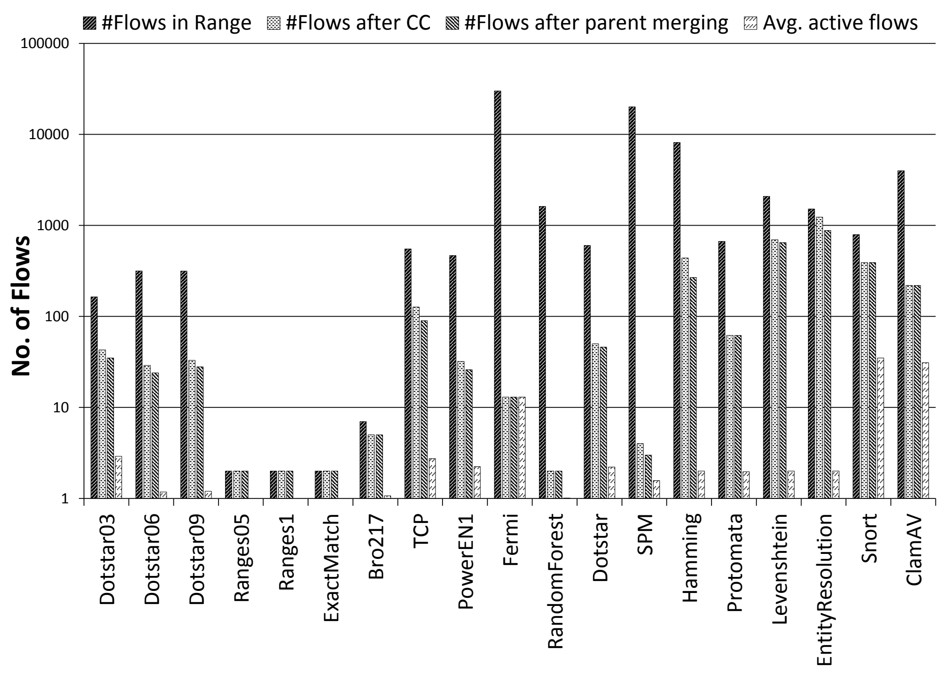


Figure 9: Average number of flows across benchmarks.

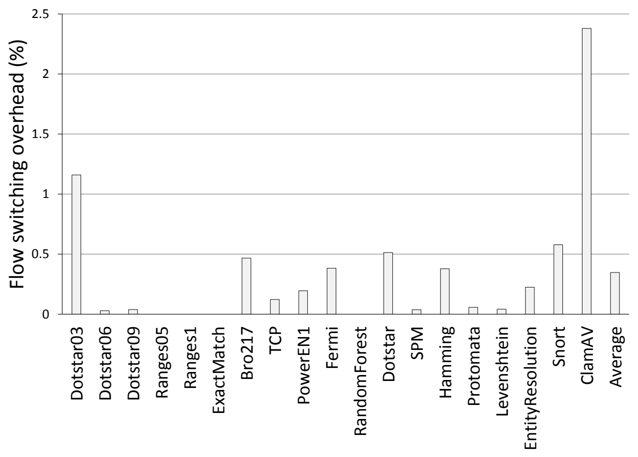


Figure 10: Costs of flow switching.

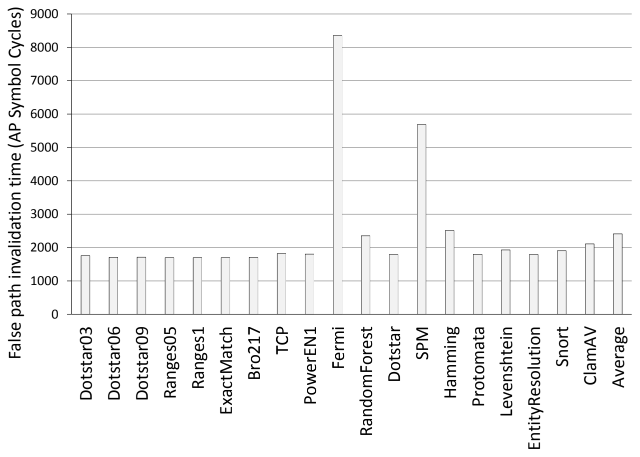


Figure 11: Costs of decoding false paths at the end of input segment.

On average, output reporting and worst case FIV on host CPU take $\sim$1% and $\sim$6% of total execution time respectively, without accounting for overlap of FIV computation. This is because output reporting and flow invalidation are infrequent.

Energy: Since the PAP architecture reduces overall execution time, we expect a reduction in static energy. However in the PAP architecture, we activate more state-transition-elements than the baseline due to traversal of false paths which can lead to increase in dynamic energy. On an average there are 2.4$\times$ extra transitions per input symbol. State activation only writes to multiple flip-flops (mask register) and does not require additional writes to the DRAM or activation of large number of additional DRAM rows. The AP activates an entire DRAM row for every input symbol and reads out different columns based on the bits stored in these flip-flops. Therefore, these additional activations do not lead to significantly increased dynamic energy costs.

6 RELATED WORK

To the best of our knowledge this work is the first to explore parallelization of non-deterministic FSMs (NFA) on the Automata Processor. Below we discuss the most closely related works:

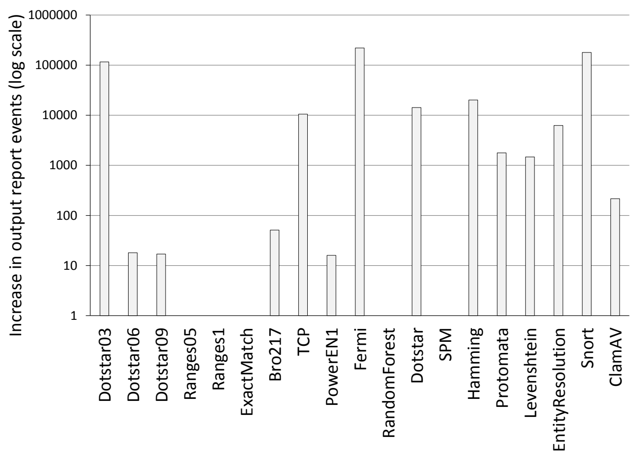


Figure 12: Increase in output reports due to false paths

Parallelization of FSMs: Parallelization of FSMs is a well studied problem. Ladner and Fischer [22] parallelize deterministic FSMs (DFA) using parallel prefix-sums. Hillis and Steele [15] present an improved parallel prefix algorithm that reduces the execution time from $O\left(log\left(m\right)\times n^3\right)$ to $O\left(log\left(m\right)\times n\right)$ when executing on m processors.

More recently, Todd and others [25] leverage classic parallel prefix sums to do enumeration of FSMs on modern hardware. The key contribution of their work is three fold: improving enumeration efficiency by reducing the dependence on n (number of states in the FSM) by cleverly leveraging convergence, demonstrating a scalable implementation on modern multi-core processors with vector SIMD units and careful data mapping of the transition table based on the range of input symbols to improve spatial locality of cache accesses. However, their work is limited to small DFAs, primarily due to the large computational complexity of enumerating NFAs.

To reduce the computational complexity and space footprint of conventional NFAs in multi-core architectures, modular NFA architectures have also been proposed [37]. To improve locality of access, several small regular expressions and regular expressions with *common prefixes* are merged into larger segments. Parallelism is achieved by mapping these segments to separate threads, with each thread processing either the same or different inputs in parallel. This paper builds on the above concepts, generalizes these insights to emerging applications with large NFAs and proposes custom parallelization for an entirely different memory centric architecture–Automata Processor.

An alternative to enumerating all states in the FSM is *speculation*, i.e. guessing the start states of input segments [27, 39, 40]. Speculation for parallelizing FSMs has been applied to specific application domains such as browser's front end [18], JPEG decoder that uses parallel Huffman decoding [20], intrusion detection using hot state prediction [23], and speculative parsing [19]. Notably, Zhao and others [39, 40] introduce the concept of principled speculation, which is the first rigorous approach to speculative parallelization. While our proposed architecture does not employ speculation, we believe this is a promising direction for reducing the number of active flows.

Automata Processor: The unconventional architecture of Automata Processor (AP) [1, 12] has sparked interest in the academic community. Many recent works have re-designed and analyzed critical algorithms for AP across diverse application domains, such as big data analysis [9], data-mining [33], bioinformatics [28], high-energy particle physics [34], machine learning [16], pseudo-random number generation and simulation [30], and natural language processing [41]. Wadden and others [31] have developed a multi-threaded data-flow automata simulator (VASim) and released a diverse automata benchmark suite (ANMLZoo). Angstadt and others [6] developed RAPID, a new high-level programming language that can be easily compiled to AP and tessellation techniques that can reduce the compilation time by up to four orders of magnitude by leveraging repeated NFA designs between FSMs. Complementary to the above works which have advanced the state-of-art by demonstrating the efficiency of the AP, our work aims to further improve the efficiency by designing new methods for parallelization of FSMs running on AP.

7 CONCLUSION

This paper attempts to break the sequential NFA execution bottleneck on the Micron Automata Procesor (AP). We identify two main challenges to applying enumerative NFA parallelization techniques on the AP: (1) high state-tracking overhead for input composition (2) huge computational complexity for enumerating parallel paths on large NFAs. Using the AP flow abstraction and properties of FSMs like small input symbol transition range, connected components and common parents we amortize the overhead of state-tracking and realize a time-mutliplexed execution of enumerated paths. To tackle the computational complexity, we leverage properties of the FSMs like path convergence to dynamically reduce the number of executed enumerated flows. The algorithmic insights about large real-word NFA provided in this work (e.g., presence of connected components, common parents, active state groups, range partitioning) are general and can be applied to parallelize NFA execution on any spatial, data-flow substrate with memory and interconnects, like FPGAs, cache sub-arrays or memristor crossbar arrays. What is required is an efficient state-encoding and state-mapping scheme along with a mechanism for supporting state-transitions using interconnects. Furthermore, different connected components and flow contexts may also be mapped to separate GPU threads for parallelism. We leave this exploration for future work. Our evaluation on a range of FSM benchmarks shows 25.5× speedup over the sequential baseline AP.

ACKNOWLEDGMENTS

We thank Daichi Fujiki, Harini Muthukrishnan, Subarno Banerjee and the anonymous reviewers for their feedback. This work was supported in part by the NSF under the CAREER-1652294 award and by C-FAR, one of the six SRC STAR-net centers sponsored by MARCO and DARPA.

REFERENCES

[1] Micron Automata Processing. Retrieved May 3, 2017 from http://www.micronautomata.com/
[2] Micron Automata Processing D480 Documentation Design Notes. Retrieved May 3, 2017 from http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html

[3] Micron Automata Processing D480 Software Development Kit. AP Flow Concepts. Retrieved May 3, 2017 from http://micronautomata.com/apsdk_documentation/latest/h1_ap.html

[4] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (June 1975), 333–340. https://doi.org/10.1145/360825.360855

[5] Rajeev Alur and Mihalis Yannakakis. 1998. Model checking of hierarchical state machines. In *ACM SIGSOFT Software Engineering Notes*, Vol. 23. ACM, 175–188. https://doi.org/10.1145/288195.288305

[6] Kevin Angstadt, Westley Weimer, and Kevin Skadron. 2016. RAPID Programming of Pattern-Recognition Processors. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 593–605. https://doi.org/10.1145/2872362.2872393

[7] Michela Becchi and Patrick Crowley. 2008. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 2008 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2008, San Jose, California, USA, November 6-7, 2008*. 50–59. https://doi.org/10.1145/1477942.1477950

[8] Michela Becchi, Mark A. Franklin, and Patrick Crowley. 2008. A workload for evaluating deep packet inspection architectures. In *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*. 79–89. https://doi.org/10.1109/IISWC.2008.4636093

[9] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. 2015. Entity Resolution Acceleration using Micron's Automata Processor. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[10] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*. Springer, 359–364. https://doi.org/10.1007/3-540-45657-0_29

[11] Sutapa Datta and Subhasis Mukhopadhyay. 2015. A grammar inference approach for predicting kinase specific phosphorylation sites. *PloS one* 10, 4 (2015), e0122294. https://doi.org/10.1145/2660193.2660231

[12] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098. https://doi.org/10.1109/TPDS.2014.8

[13] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. 2008. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review* 38, 5 (2008), 29–40. https://doi.org/10.1145/1452335.1452339

[14] Linley Gwennap. 2014. Micron Accelerates Automata:New Chip Speeds NFA Processing Using DRAM Architectures. In *Microprocessor Report*.

[15] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183. https://doi.org/10.1145/7902.7903

[16] Tommy Tracy II, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards Machine Learning on the Automata Processor. In *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, Vol. 9697. Springer, 200. https://doi.org/10.1007/978-3-319-41321-1_11

[17] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. 2009. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*.

[18] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanović, and Rastislav Bodík. 2009. Parallelizing the Web Browser. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar'09)*. USENIX Association, Berkeley, CA, USA, 7–7. http://dl.acm.org/citation.cfm?id=1855591.1855598

[19] Blake Kaplan. Speculative parsing path. Bug 527623. Retrieved May 3, 2017 from http://bugzilla.mozilla.org

[20] Shmuel Tomi Klein and Yair Wiseman. 2003. Parallel Huffman decoding with applications to JPEG files. *Comput. J.* 46, 5 (2003), 487–497. https://doi.org/10.1093/comjnl/46.5.487

[21] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Computer Communication Review*, Vol. 36. ACM, 339–350. https://doi.org/10.1145/1159913.1159952

[22] Richard E Ladner and Michael J Fischer. 1980. Parallel prefix computation. *Journal of the ACM (JACM)* 27, 4 (1980), 831–838. https://doi.org/10.1145/322217.322232

[23] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. 2009. Multi-byte regular expression matching with speculation. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 284–303. https://doi.org/10.1007/978-3-642-04342-0_15

[24] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. 2014. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 309–328.

[25] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. 529–542. https://doi.org/10.1145/2541940.2541988

[26] Alexandre Petrenko. 2001. Fault model-driven test derivation from finite state models: Annotated bibliography. In *Modeling and verification of parallel processes*. Springer, 196–205. https://doi.org/10.1007/3-540-45510-8_10

[27] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. MicroSpec: Speculation-Centric Fine-Grained Parallelization for FSM Computations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*. 221–233. https://doi.org/10.1145/2967938.2967965

[28] Indranil Roy and Srinivas Aluru. 2016. Discovering motifs in biological sequences using the micron automata processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 13, 1 (2016), 99–111. https://doi.org/10.1109/TCBB.2015.2430313

[29] Margus Veanes, Todd Mytkowicz, David Molnar, and Benjamin Livshits. 2015. Data-Parallel String-Manipulating Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 139–152. https://doi.org/10.1145/2676726.2677014

[30] Jack Wadden, Nathan Brunelle, Ke Wang, Mohamed El-Hadedy, Gabriel Robins, Mircea Stan, and Kevin Skadron. 2016. Generating efficient and high-quality pseudo-random behavior on Automata Processors. In *34th IEEE International Conference on Computer Design, ICCD 2016, Scottsdale, AZ, USA, October 2-5, 2016*. 622–629. https://doi.org/10.1109/ICCD.2016.7753349

[31] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*. 105–166. https://doi.org/10.1109/IISWC.2016.7581271

[32] Ke Wang, Yanjun Qi, Jeffrey J Fox, Mircea R Stan, and Kevin Skadron. 2015. Association rule mining with the micron automata processor. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 689–699. https://doi.org/10.1109/IPDPS.2015.101

[33] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016. Sequential pattern mining with the Micron automata processor. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 135–144. https://doi.org/10.1145/2903150.2903172

[34] Michael HLS Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted Zmuda. 2016. 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* 832 (2016), 219–230. https://doi.org/10.1016/j.nima.2016.06.119

[35] Qiong Wang, Mohamed El-Hadedy, Ke Wang, and Kevin Skadron. 2016. Accelerating Weeder: A DNA Motif Search Tool using the Micron Automata Processor. (2016).

[36] Zhen-Gang Wang, Johann Elbaz, Françoise Remacle, RD Levine, and Itamar Willner. 2010. All-DNA finite-state automata with finite memory. *Proceedings of the National Academy of Sciences* 107, 51 (2010), 21996–22001. https://doi.org/10.1073/pnas.1015858107

[37] Yi-Hua E Yang and Viktor K Prasanna. 2011. Optimizing regular expression matching with sr-nfa on multi-core systems. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 424–433. https://doi.org/10.1109/PACT.2011.73

[38] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. ACM, 93–102. https://doi.org/10.1145/1185347.1185360

[39] Zhijia Zhao and Xipeng Shen. 2015. On-the-Fly Principled Speculation for FSM Parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 619–630. https://doi.org/10.1145/2694344.2694369

[40] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "embarrassingly sequential": parallelizing finite state machine-based computations through principled speculation. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. 543–558. https://doi.org/10.1145/2541940.2541989

[41] Keira Zhou, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. 2015. Brill tagging on the micron automata processor. In *Semantic Computing (ICSC), 2015 IEEE International Conference on*. IEEE, 236–239. https://doi.org/10.1109/ICOSC.2015.7050812