2025 34th International Conference on Parallel Architectures and Compilation Techniques (PACT) | 979-8-3315-8295-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/PACT65351.2025.00023

ANG: Accelerating NFA processing on GPUs via Exploring Multi-Level Fine-Grained Parallelism

Yuguang Wang
Michigan Technological University
yugwang@mtu.edu

Yunmo Zhang
City University of Hong Kong
yunmo.zhang@my.cityu.edu.hk

Zeyu Liu
City University of Hong Kong
zeyliu4-c@my.cityu.edu.hk

Junqiao Qiu
City University of Hong Kong
junqiqiu@cityu.edu.hk

Zhenlin Wang
Michigan Technological University
zlwang@mtu.edu

***Abstract*—Finite Automata (FA) processing is a core computation in various real-world applications. Over the past decades, extensive efforts have been dedicated to accelerating FA processing on modern parallel platforms, particularly GPUs, due to their high memory bandwidth and massive hardware parallelism. As Non-deterministic Finite Automata (NFA)-based applications have strong and growing demands for real-time data analytics nowadays, reducing latency in automata processing has become a critical priority. However, existing approaches face significant challenges when limited parallelism is exposed in NFA computations. In this work, we explore opportunities of introducing fine-grained parallelism from various sources and addressing the limitations of fast NFA processing. Specifically, by analyzing different NFA parallelization schemes, we identify the major performance issue caused by insufficient state-level parallelism in conventional designs. To overcome the bottleneck, this work introduces speculative parallelization tailored for GPU-based NFA processing, thus effectively exploiting fine-grained parallelism across multilevels, with a particular focus on input-chunk-level parallelism. To realize speculative parallelization in practice, we develop *ANG*, a latency-oriented NFA processing framework that overcomes key implementation challenges on GPUs. We evaluate the efficiency of *ANG* on a set of representative NFAs with diverse properties. Experimental results demonstrate that *ANG* achieves significant performance improvement compared to state-of-the-art techniques, with reaching 11.74× speedup on average (and up to 49.88× in extreme cases).**

***Index Terms*—NFA, GPU, Speculative Parallelization**

I. Introduction

Finite automata processing, including non-deterministic finite automata (NFA) and deterministic finite automata (DFA) processing, is a fundamental computation widely used in various real-world applications such as data analytics [1]–[3], malware detection [4], [5], system verification [6], [7], machine learning [8], [9], natural language processing [10], [11], and others. In recent years, with the increasing complexity and scale of data in emerging applications, low-latency automata processing is desired to handle massive amounts of data in real-time [12]–[14]. For instance, network intrusion detection systems need to identify prioritized threats with minimal delay, and high-frequency trading applications require real-time pattern matching over critical objects to prevent financial losses.

The growing demand for high-performance FA processing has motivated algorithmic and architectural solutions on different parallel platforms [12], [13], [15]–[23], ranging from von Neumann architectures such as CPUs and GPUs to domain-specific accelerators such as FPGAs, ASICs, and processing-in-memory architectures. Among these platforms, GPUs have emerged as a prominent choice due to their massive hardware parallelism and high memory bandwidth. Compared with DFA processing, NFA processing receives more attention for GPU acceleration as NFAs inherently exhibit more parallelism and are typically more memory-efficient.

The classical design of NFA processing on GPUs involves exhibiting parallelism at different levels. The first GPU-based NFA processing engine, *iNFAnt* [16], assigns a single thread block to each given NFA and an input sequence, which utilizes coarse-grained parallelism at automata-level and sequence-level, and then enables threads within a block to execute state transitions cooperatively, which introduces fine-grained parallelism at state level. Built upon *iNFAnt*, different variants are proposed to improve the GPU thread utilization and enable efficient data access on GPU memory hierarchy [24]–[26]. However, all these works can be categorized as using state-level parallelism and are not well-suited for achieving low-latency NFA processing when the number of states activated by an input symbol is inherently limited in the given NFA.

To enable efficient GPU-based processing of NFAs with varying properties, this work investigates how fine-grained parallelism utilized in a parallelization scheme determines the performance of NFA processing. By modeling the impact of various fine-grained parallelism, this work explores introducing chunk-level parallelism in NFA processing, with a key design of introducing speculative parallelization. Though previous studies claim that it is difficult to make speculation in NFA processing because multiple states may be activated in a step of state transitions, this work shows that achieving the most accurate speculation across the entire input sequence is unnecessary. Instead, a conservative hot-state-based speculation, which ensures only fully confident speculative starting states are included, can guarantee the benefits of speculation. Meanwhile, this work also proposes an aggregation-based par-

979-8-3315-8295-1/25/$31.00 ©2025 IEEE
DOI 10.1109/PACT65351.2025.00023

allel verification and recovery mechanism to bring significant performance improvement even when the speculation accuracy is low. To fully expose the potential benefits, this work also explores the design space of speculative NFA parallelization and determines the optimal configuration, which consists of thread assignment schemes and data structures of NFA topology and active states.

We develop a framework called *ANG* to **A**ccelerate **N**FA processing on **G**PU. This latency-sensitive framework utilizes fine-grained parallelism from different levels and automatically selects the best parallelization configuration for the given NFA and input sequences. To evaluate its efficiency, we use a set of NFA benchmarks with various characteristics and observe significant performance improvement over existing approaches, achieving 11.74× on average and up to 49.88× in benchmarks that have extremely limited state-level parallelism.

In summary, this work makes the following contributions:

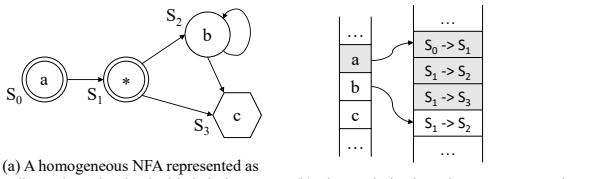
- By analyzing fine-grained parallelism at different levels applied to GPU-based NFA processing, it theoretically reveals efficiency issues in the state-of-the-art designs, and offers effective guidelines for high-performance NFA parallelization.
- It explores a new fine-grained parallelism for NFA processing by introducing speculation based chunk-level parallelism. To the best of our knowledge, this is the first work to enable speculative NFA parallelization on GPUs.
- It develops *ANG*, a latency-oriented multi-scheme framework which provides a unified interface and efficient implementation for NFA processing on GPUs.
- It evaluates *ANG* on real-world benchmarks with diverse properties, demonstrating the significant advancement over existing approaches and the benefits of utilizing fine-grained parallelism at multilevels.

II. Background and Motivation

A. Basics of Automata Processing

A finite automaton (FA) can be represented as a 5-tuple (Q, Σ, δ, q_0, F), where Q is a finite set of automaton states, Σ is the input alphabet, δ is a transition function which determines the next active states based on the current active states as well as the current input symbol, q_0 is the set of initial states, and F is a set of accepting or reporting states. An FA can be deterministic or non-deterministic. A DFA allows only one active state per input symbol, while an NFA may activate multiple states simultaneously. Although an NFA can be converted into an equivalent DFA [27], this conversion often leads to an exponential increase in FA state counts. Consequently, NFAs are preferred in practice [28]. In this paper, following previous work [17], [25], [29], [30], we focus on ϵ-free homogeneous NFAs[1]. Such an NFA is often represented in ANML [32] or MNRL [29] format in real-world applications, and the initial states in these formats are usually considered to be *always-active*.

[1]In a homogeneous NFA, transitions to any given state must occur on the same input symbol-set [31].



(a) A homogeneous NFA represented as a directed graph. The doubled circles represent the initial states, and the hexagon represents the accepting state.

(b) The symbol-oriented NFA representation where input symbols are used as indexes for locating all possible transitions.

Fig. 1. A running example of NFA.

Fig. 1 shows a simple NFA which can be used to find out all matches of the regular expression pattern "a∗.b∗c" on the given input. Following standard PCRE conventions [33], here '∗' denotes zero or more repetitions of the preceding symbol while '.' means the match of any character except a newline. A valid match of this regular expression is the string "*acbbc*". The processing of the corresponding NFA starts from two *always-active* initial states, S_0 and S_1. In each step, the NFA consumes a symbol from the given input sequence and activates certain states by following the transitions specified in the transition table or transition graph. When a reporting state is activated and accepts the current symbol, the NFA generates a report. For example, given the input sequence "*acb*. . . ", the NFA first consumes the symbol 'a'. Since both S_0 and S_1 are active and match the input, the successors of S_0 and S_1, i.e., S_1, S_2, and S_3, will become active in the next step. This process repeats until all input symbols are consumed. Note that after reading the second symbol '*c*', the active state S_3, which is also a reporting state in this NFA, matches with this symbol and thus generates a report to indicate a successful match (i.e., string "*ac*") for the given pattern.

B. NFA Parallelization

Given the fundamental importance of NFA in real-world applications, substantial efforts have been made to enable efficient NFA processing. An important direction is using GPU for acceleration due to the high degree of hardware parallelism and substantial memory bandwidth [13], [16], [25]. Because of the naturally parallel property in NFAs – multiple states may be active at the same time, a common design paradigm across a majority of these works involves assigning each thread block to process a single NFA and input sequence, thus exploiting both NFA-level and Sequence-level parallelism (i.e., coarse-grained parallelism). Threads within the same block collaborate to complete the processing. This design aligns well with the GPU architecture, effectively mapping different sources of parallelism to GPU grid, warp, and thread levels [12], [25], [34].

Using the above common design, the first GPU-based NFA processing engine is *iNFAnt* [16]. Each time it reads one symbol from the input sequence, it locates all possible transitions by looking up the symbol-based transition table and assigns one thread to handle a transition. For example, given the NFA shown in Fig. 1(b), *iNFAnt* assigns three threads concurrently

to process the shadow transitions when the input symbol is 'a'. Each thread checks whether the source state in its assigned transition is currently active. If so, it marks the destination state as the next active state. However, since threads may execute different operations, thread divergence can significantly degrade the performance of *iNFAnt*. To address this issue, *vNFA* [24] was proposed and aims to accurately identify active NFA states in each step, so that it can make each thread responsible for performing transitions from one of the active states. To well utilize the properties of GPU architecture, *HotStart* [25] optimizes NFA processing by identifying and prioritizing frequently activated (hot) states. Transitions from hot states are statically assigned to threads, with neighboring states and matchsets stored in registers to minimize global memory access. For less frequently activated (cold) states, a separate processing round is conducted only when these states are active. Building on this approach, recently *ngAP* [26] was introduced to further improve the thread utilization. By breaking the traditional "one-symbol-at-a-time" principle and associating each state transition with the symbol index in the input sequence, *ngAP* allows transitions triggered by different symbols to be processed simultaneously. Table I summarizes the characteristics of previous efforts in optimizing NFA processing on GPUs.

C. Motivation

Though previous studies have achieved high throughput for NFA-based applications running on GPUs, there has been limited focus on optimizing the peak performance (i.e., latency) of NFA processing [12], [13]. In fact, various real-world applications have strong and growing demands for real-time data analytics, thus it becomes increasingly important to reduce latency in automata processing. Moreover, many existing approaches assume that there are a large number of state transitions activated in each step of NFA processing [13], i.e., high parallelism cases. However, the number of transitions highly depends on the properties of the targeted application. For example, recent studies [14], [35] show that some specific small subsets of rules, instead of the entire Snort rule set, are more important for actively blocking or preventing malicious traffic from dedicated sources during certain timeframes. Most existing optimizations may fail to bring expected performance improvement in these latency-sensitive and low-parallelism NFA processing scenarios.

Fig. 2 shows the performance of running *HotStart* and *ngAP* on NFAs and input sequences from 3 representative NFA benchmarks. Details about these experiments can be found in Section VI. These benchmarks show various amount of parallelism (which is mainly indicated by the average number of transitions triggered by an input symbol) during the processing. For example, in the RF benchmarks, the average number of state transitions per symbol is 51.5 in low-parallelism cases, compared to over 16k in high-parallelism cases. In high-parallelism scenarios, *HotStart* presents superior performance due to its optimization strategies. However, in low-parallelism scenarios, these optimizations provide little benefit and introduce additional overhead. Though *ngAP* [26] tries to enable efficient NFA processing under low parallelism cases, it still focuses on state-level parallelism and it may potentially lead to increased thread divergence and poorer locality since different threads are assigned to manage transitions triggered by distinct symbols. For achieving low latency of running different NFAs on GPUs, we need to propose new designs to effectively exploit various sources of fine-grained parallelism and investigate how to provide the best parallelization under diverse scenarios.

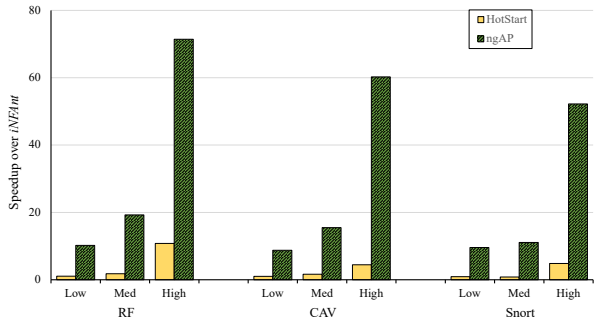


Fig. 2. Performance of *HotStart* [25] and *ngAP* [26] when processing NFAs with various amount of parallelism.

III. Exploring Fine-Grained Parallelism

In this section, we first present various types of fine-grained parallelism that are potentially applied to NFA processing, and then provide the formulation for modeling their expected benefits. which enables effective assessments of various designs.

A. Essence of Fine-Grained Parallelism

The fine-grained parallelism that has been widely utilized in NFA processing, as discussed in Section II-B, can be generally categorized as **parallelism at the state-level (PS)**. Previous studies have demonstrated their promising results when there is a sufficient number of state transitions in each processing step. However, in many NFA applications, the overall number of active states remains low. Profiling results (as reported in Table III) reveal that for individual NFAs from half of the tested real-world applications, fewer than five state transitions are triggered per step (excluding transitions originating from some special states).

To address this challenge and generate sufficient parallelism, inspired by existing DFA parallelization [15], [36], we propose speculative NFA parallelization, as outlined in Algorithm 1. For a given NFA and an input sequence, we first partition the input evenly into multiple chunks (line 1) and then assign a processing unit (PU) to concurrently handle the NFA processing on a chunk. Since we follow the common design mentioned in Section II, a PU here can be a thread block, a warp, or a single thread on GPU. We use N and N_t to represent the number of available PUs and threads, respectively. Then, we speculate the starting states of each chunk (line

TABLE I
PREVIOUS EFFORTS IN ENABLING EFFICIENT NFA PROCESSING ON GPUS.

Frameworks	Thread Assignment	Data Structures	
		NFA Topology & Indexing Options	Active States
iNFAnt [16]	a thread to a possible transition	symbol-oriented transition table similar to CSR	bit vectors
vNFA [24]	a thread to an active state	state-oriented transition table with fixed-size cells	state sets
HotStart [25]	a thread to a hot state/cold active states (if exist)	dedicated per-state information	state sets
ngAP [26]	a thread to a possible transition at different index	symbol-oriented transition table similar to CSR	state sets

Algorithm 1 Speculative NFA Parallelization.

▷ **Input:** an NFA fa and an input sequence in

1: $\{ck_1, \cdots, ck_N\}$ = input_partition(in, N)
2: **for** $i = 1 : N$ **do in parallel** ▷ Parallel Execution
3: Ss_i = spec(fa, ck_i); ▷ *Speculate* starting states Ss_i
4: $Es_i = Ss_i$; ▷ Es_i is a set of ending states
5: **while** ck_i.length() $\neq 0$ **do** ▷ *State Transitions*
6: c = ck_i.front();
7: $Es'_i = \emptyset$;
8: **for** state $st \in Es_i$ **do**
9: Es'_i.insert(fa.trans(c, st));
10: $Es_i = Es'_i$;
11: **for** $i = 2 : N$ **do** ▷ Seq. Validation and Recovery
12: **if** $Es_{i-1} \neq Ss_i$ **then**
13: Es_i = fa.rerun(ck_i, Es_{i-1}); ▷ As redo line 5-10

3) and use these states to run the NFA processing (line 5–10). Here different iterations of the outer for loop (line 2) can be executed in parallel. Finally, we verify the correctness of speculative starting states on each chunk. If a mis-speculation occurs, a recovery will be invoked for the current chunk (line 11–13). Through this speculation-based scheme, we can utilize **parallelism at the chunk-level (PC)** for fast NFA processing. However, implementing this speculative NFA processing on GPU architectures is not that straightforward. We leave the discussion about the challenges in Section IV. In this section, we mainly analyze the ideal benefits derived from this scheme.

Considering the characteristics of GPU architectures, we further explore **parallelism at both state- and chunk- levels (PB)**. GPU hardware parallelism is hierarchical with each level providing a different degree of parallelism and granularity. At the lowest level, thousands of threads execute instructions simultaneously. These threads are organized into warps, which run in SIMD lockstep and are grouped into thread blocks, and finally into a grid of thread blocks. A two-level parallelism in NFA processing can exploit both inter-warp data parallelism and intra-warp task parallelism. This is motivated by our observation that in Algorithm 1, the outer loop (line 2) operates on different chunks while the inner loop (line 8) handles all state transitions triggered by the same input symbol.

B. Performance Analysis

With the essence of fine-grained parallelism exposed, we aim to build a performance model for examining the efficiency of exploiting parallelism at different levels. In the following discussion, $T(\cdot)$ represents the processing time, L denotes the length of the entire input sequence, and W is the width of warps on GPU architecture. To simplify the analysis, we assume that (i) all threads run in lockstep during NFA state transitions, and (ii) shared memory size is limited, thus it can hold the active state sets but not the entire NFA transition table. We also ignore the impacts of thread divergence as well as resource contention. For PC and PB with speculations, we assume speculations can be perfect and recovery is unnecessary. This may not be the case for a lot of NFAs, but later we can show that with using a conservative hot-state-based speculation and an aggregation-based recovery, the influence of mis-speculation can be minimized and will not change the conclusion shown in this section. We discuss the efficiency modeling of each scheme as follows.

(i) for the scheme utilizing PS, the processing time

$$T(PS) = \sum_{i=1}^{L} T_i(PS) = \sum_{i=1}^{L} Max_{j=1}^{N} \{T_i^j(PS)\} \quad (1)$$

where $T_i(PS)$ is the global time span between synchronizations (which are used to ensure all active states in the current round are processed before moving to the next round) and $T_i^j(PS)$ is the time spent by the jth PU between the ith and $(i+1)$th synchronization. In fact, a PU in the scheme utilizing PS is usually a thread ($N = N_t$). Because the major operations done by a thread contain looking up the NFA transition table, which concerns irregular global memory access, and then updating the current active states based on the fetched transitions, $T_i^j(PS)$ can be computed as

$$T_i^j(PS) = T_g \times G_i^j(PS) + T_{act} \quad (2)$$

Factors on the right side of the equation respectively correspond to the execution time of one global memory access, the number of global memory accesses for thread j in iteration i, and the execution time of active state updates. Since accessing shared memory is much faster than global memory while tracking active state is basically done on shared memory, we can neglect the effect of T_{act}.

In the PS-based scheme, the number of transitions triggered by the input symbol read at step i determines the number of global memory access for each thread. Their relation can be defined as

$$\|Trans(in.\text{at}(i))\| = \sum_{j=1}^{N_t} G_i^j(PS) \quad (3)$$

By putting (1)-(3) together, we can infer how the number of transitions at each step affects PS-based processing. When $\|Trans(in.\text{at}(i))\|$ is much larger than N_t for most i, we have

$$T(PS) \approx T_g \times \sum_{i=1}^{L} \frac{\|Trans(in.\text{at}(i))\|}{N_t} \quad (4)$$

However, if $\|Trans(in.\text{at}(i))\|$ is smaller than N and task assignments for threads are imbalanced, meaning that only one or a subset of threads are responsible for the state transitions while others are idle, then the processing time becomes

$$T(PS) \approx T_g \times \sum_{i=1}^{L} \|Trans(in.\text{at}(i))\| \quad (5)$$

(ii) for the scheme utilizing PC, as the input has been divided into N chunks, the processing time

$$T(PC) = \sum_{i=1}^{L/N} T_i(PC) = \sum_{i=1}^{L/N} Max_{j=1}^{N} \{T_i^j(PC)\} \quad (6)$$

The definition and calculation of $T_i^j(PC)$ are similar to the ones in PS-based schemes. A PU in the PC-based scheme is also a thread. However, since PC-based schemes distribute all threads on the chunk level, which indicates that a thread handles all transitions triggered by a symbol, the relation between the number of transitions and the global memory access at each step is modeled as

$$\sum_{j=1}^{N} \|Trans(in.\text{at}(i + j \times \frac{L}{N}))\| = \sum_{j=1}^{N} G_i^j(PC) \quad (7)$$

Putting all together, we have the processing time estimation for the PC-based scheme:

$$T(PC) \approx T_g \times \sum_{i=1}^{L/N_t} \|Trans(in.\text{at}(i))\| \quad (8)$$

(iii) for the scheme utilizing PB, a PU running on a chunk is a warp while W threads in a warp work together for completing state transitions triggered by the same input symbol (on NVIDIA GPUs, $W = 32$), so the input is divided into ($N = N_t/W$) chunks for parallel execution and the processing time is

$$T(PB) = \sum_{i=1}^{L \times W/N_t} T_i(PB) = \sum_{i=1}^{L \times W/N_t} Max_{j=1}^{N_t} \{T_i^j(PB)\} \quad (9)$$

For threads in a group h ($h \in [0, N_t/W)$), we can infer

$$\|Trans(in.\text{at}(i + h \times \frac{L \times W}{N_t}))\| = \sum_{j=1}^{W} G_i^{j+h\times W}(PB) \quad (10)$$

Putting (10) into the calculation of (9), and similar to the analysis in **(i)**, when $\|Trans(in.\text{at}(i))\| \gg W$,

$$T(PB) \approx T_g \times \sum_{i=1}^{L \times W/N_t} \frac{\|Trans(in.\text{at}(i))\|}{W} \quad (11)$$

When $\|Trans(in.\text{at}(i))\|$ is smaller than W and the task assignment is imbalanced, then

$$T(PB) \approx T_g \times \sum_{i=1}^{L \times W/N_t} \|Trans(in.\text{at}(i))\| \quad (12)$$

Discussion. In general, when the state-level parallelism is sufficient, all schemes above have a similar ideal processing time. With the consideration of the GPU thread divergence and data movement, the PS-based scheme should outperform the others as it allows all threads to work on the state transitions triggered by the same input symbol, while in the other two schemes, threads may work on different input symbols. When the average number of state transitions at a step is moderate, i.e., it is much larger than W but smaller than N_t, then PB-based scheme is highly potential to be the best choice, as it can maintain high thread utilization (compared with PS-based scheme) and also introduce less thread divergence (compared with PC-based scheme). Finally, when there is limited state-level parallelism, ideally PC-based scheme can be N_t times and W times faster than the PS-based and PB-based schemes, respectively, as it effectively shortens the length of critical path. In practice, the actual processing time also highly depends on the characteristics of the underlying architecture. Note that here we do not need to provide a complicated calculation for the detailed processing time since our target is not its exact value; instead, we only want to figure out key factors determining the best parallelization scheme.

IV. Speculative Parallelization on GPU

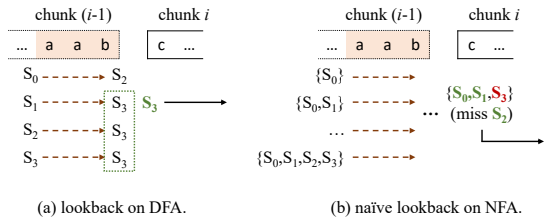
Analyses above present potentials of speculative NFA parallelization. This section explores the efficient implementation on GPUs.

A. Major Challenges

There are three major components when implementing the speculative NFA parallelization.

State Speculation. A widely used technique to speculate starting states for each chunk in DFA parallelization is called *all-state lookback* [15], [36]–[39]. It first performs DFA processing by starting from all states on the suffix of the prior chunk, and then selects the ending state appearing most as the speculative state. Fig. 3(a) shows an example. By enumerating transitions from all states (from S_0 to S_3) over the suffix with length 3 (i.e., "aab") of the chunk $i-1$, this technique speculates S_3 as the starting state of chunk i, because 3 out of 4 lookback transitions end in this state.

However, directly applying this lookback-based speculation to NFA processing is not feasible, since speculating a set of NFA states poses significant overheads [13]. Moreover, while the outcome of a DFA speculation is relatively simple (either hits the ground truth or not), the speculation results in NFAs are more complicated, preventing an efficient verification and recovery. For example, in Fig. 3(b), the ground truth starting states in chunk i are $\{S_0, S_1, S_2\}$, but the speculative starting states are $\{S_0, S_1, S_3\}$ (assume this state set appears most)



(a) lookback on DFA. (b) naïve lookback on NFA.

Fig. 3. An illustrative example of applying the basic *all-state lookback* speculation. States in green are the true states.

when directly following the DFA-based lookback, which leads to a false starting state (S_3) and misses a targeted state (S_2). To enable efficient and effective speculation, the key is to develop a mechanism to understand and estimate how the speculation affects the overall performance of NFA parallelization.

State Transition. The primary operation in each step of NFA processing is the state transition, which involves looking up the NFA transition table or transition graph, checking current active states, and updating next active states. To achieve optimal transitions on GPUs, it is crucial to determine the data structures used.

There are three basic parameters about the data structures affecting the performance of NFA processing. First, the topology of an NFA can be represented as a *CSR* format or a two-dimensional table with *fixed-size entries*. Second, there are two indexing options for accessing the NFA transition table, including a *state-oriented* and a *symbol-oriented* indexing, depending on whether transitions are indexed by the NFA state or the input symbol. Third, either *bit vectors* where each bit corresponds to a distinct NFA state, or *state sets* which directly contain active state IDs, are used to track current and next active states. In each of existing frameworks, a fixed combination of the above three parameters is used without systematically investigating how it works under different NFAs and inputs, thus may lead to variable and suboptimal performance.

Mis-Speculation Recovery. To ensure the correctness of NFA processing, conventional speculative parallelization performs an iterative verification and recovery as shown in line 11–13 of Algorithm 1. Such a sequential recovery leads to low GPU thread utilization and thus overwhelms the benefits of chunk-level parallelism, particularly when mis-speculation occurs frequently. Previous studies [15], [34], [40] have investigated parallel recovery for speculative DFA parallelization by letting each unverified chunk start the verification and recovery earlier with speculating the ending state of its direct prior chunk as the ground truth. However, directly applying such a parallel design to NFA processing may be less efficient. A major issue is that several ground truth starting states on a chunk may be missed in the original speculation, requiring multiple iterations of verification and recovery to cover all targeted states.

B. Hot-state-based Speculation

To evaluate how accurate the speculation is and how it affects the overall performance of NFA processing, we first provide the following definitions: (1) the speculative states which are the real starting states are referred to as *true positives* (TPs); (2) the speculative states which are not the real starting states are referred to as *false positives* (FPs); and (3) the missed real starting states in the speculation are referred to as *false negatives* (FNs). Then two correctness metrics can be introduced to enable a comprehensive evaluation of a speculation method, i.e., True Positive Rate (TPR = $||TP||$ / ($||TP||$ + $||FN||$)) and False Discovery Rate (FDR = $||FP||$ / ($||TP||$ + $||FP||$)).

Analysis. The ranges of both TPR and FDR are [0, 1]. Ideally, the optimal NFA speculation in a divided chunk should ensure TPR is 1 while FDR is 0, otherwise, recovery is needed. However, different TPR and FDR will bring varying recovery costs. When FDR is 0, which indicates that no FP is involved in the speculation, the higher the value of TPR is, the less workload is needed during recovery, thanks to the fact that only a partial number of the execution paths are missed. The recovery procedure can just re-run the missing starting states and merge the corresponding processing results with existing results. However, when FDR is not 0, the recovery procedure has to redo the entire chunk by starting from all ground-truth states, no matter what the value of TPR is. This is because the mapping relationship between starting states and ending states on a chunk is not recorded during the processing (since this is time- and space-consuming), and then the recovery procedure cannot just exclude the wrong ones and reuse existing processing results.

Algorithm 2 Hot-state-based Speculation.

▷ **Input:** an NFA fa and a divided input chunk ck_i

1: $Ss_i = (fa.\text{initials}() \cap fa.\text{always\_actives}())$;
2: **for** $i = LB$; $i > 0$; $i = i - 1$ **do** ▷ lookback with length LB
3: c = ck_i.at($-i$); ▷ Access the suffix of previous chunk
4: $Ss'_i = \emptyset$;
5: **for** state $st \in Ss_i$ **do**
6: Ss'_i.insert(fa.trans(c, st));
7: $Ss_i = Ss'_i$;

Based on analysis above, instead of trying to achieve the highest TPR, a more effective speculation should avoid FDR being non-zero. In homogeneous NFAs explored in this paper, some states will always be active once they are activated (referred to as *always-active states*). Initial states are usually always-active (referred to as *always-active initial states*) and considered as hot states in the previous studies [13], [25]. They are explicitly marked in the ANML format, thus can be easily determined. We propose a *hot-state-based lookback* for conservative speculation, as shown in Algorithm 2. For chunk i, the speculative states will be generated in the most conservative way by applying lookback from those always-active initial states only. Such a design ensures FDR in each chunk is 0.

C. GPU-oriented State Transition

To reduce the excessive data movement, a common principle [13], [24], [25], [34], [41] about utilizing GPU memory hierarchy is putting active states as well as the frequently accessed portions of NFA topology onto GPU shared memory, and leaving the rest stored in the global memory. Such a principle and NFA properties affects the selection of data structures used for state transitions.

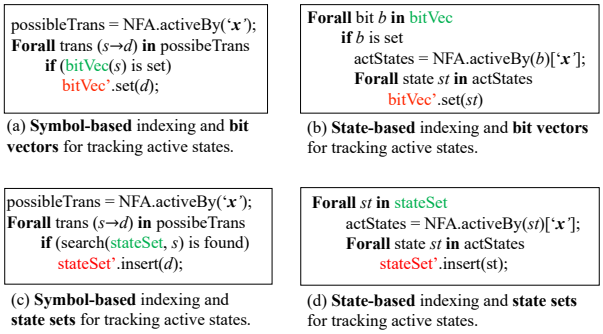
Tracking of active states. Assume that M is the number of NFA states while M_a is the maximum number of active states during runtime for the given NFA. The memory usage of applying bit vectors for a chunk is $(M \times 2)$ bits, where each bit indicates whether the corresponding state is active. The memory usage of applying state sets is $(M_a \times 16 \times 2)$ bits when 16 bits are used for a state ID. Note that $M_a \leq M < 2^{16}$ for all NFAs evaluated in this paper, and "$\times 2$" indicates that we need to maintain both current and next active states. As active states of N chunks should be fully stored in shared memory, the data structure causing less memory cost should be chosen. For example, when M is expected to be much larger than $M_a \times 16$, state sets should be used.

Indexing options. Under different combinations of indexing options and data structures used for tracking active states, the procedures of looking up the NFA transition table vary, as shown in Fig. 4. When bit vectors are used (Fig. 4(a) and (b)), applying the symbol-oriented indexing performs better than the state-oriented one since the former is in $\mathbf{O}(\#Trans)$ while the latter is in $\mathbf{O}(M \times M_a)$. On the other hand, the state-oriented indexing is preferred when state sets are used, as it takes $\mathbf{O}(M_a^2)$ while the symbol-oriented one takes $\mathbf{O}(\#Trans \times M_a)$. It's important to note that $\#Trans \geq M_a$.

NFA representation. Various NFA representations have been explored to enhance the efficiency of data movement. In an ideal scenario, fetching state transitions would not necessitate global memory access. Basically, the CSR format is employed for better space efficiency when the distribution of transitions is highly skewed, i.e., a majority of transitions in the NFA topology originate from one or a small set of NFA states. Conversely, when the transitions exhibit a uniform distribution, a transition table with fixed-size cells is preferred to avoid additional memory overhead incurred by using pointer arrays.

Considering all above parameters, in this paper, we choose CSR-based symbol-oriented transition table with bit vectors (CYB), fixed-size symbol-oriented transition table with bit vectors (FYB), and fixed size state-oriented transition graph table with state sets (FTS) as the candidate configurations of speculative NFA parallelization.

Optimization with unrolling. Loop unrolling is one of the major ways to expose instruction-level parallelism (ILP). However, exposing such low-level parallelism is unexplored for automata processing on GPUs. Our observation shows that NFA processing still has capabilities to get benefits from unrolling to some degrees. We only explore applying loop unrolling for state transitions in each step, i.e., the inner for loop in line 8–9 of Algorithm 1. We also disable unrolling



(a) **Symbol-based** indexing and **bit vectors** for tracking active states.

(b) **State-based** indexing and **bit vectors** for tracking active states.

(c) **Symbol-based** indexing and **state sets** for tracking active states.

(d) **State-based** indexing and **state sets** for tracking active states.

Fig. 4. Using different indexing methods for looking up the NFA transition table when giving an input symbol 'x'.

in PC-based schemes because the number of iterations of the inner loop is large only when the parallelism in state level is enough. A key question in optimization with unrolling is the selection of the size of the unrolling factor. When choosing a large unrolling factor, it takes risks of bringing more branches and thread divergence, but if choosing a small unrolling factor, it may not fully utilize registers and cache on GPU architectures for better locality.

D. Aggregation-based Recovery

Unlike the existing sequential recovery and ending-state-based parallel recovery, which both rely on the ending states of the directly preceding chunk, we propose an aggregation-based recovery algorithm to minimize the iterations needed for covering all real starting states, as shown in Algorithm 3.

Algorithm 3 Aggregation-based Recovery.

```
...                                   ▷ Replacing the seq. recovery in Algorithm 1
11: for i = 2 : N do in parallel      ▷ Parallel recovery
12:     while Es_{i-1} ≠ Ss_i do      ▷ When a recovery is needed
13:         RSs_i = (Es_{i-1} \ Ss_i);
14:         for j = 1 : (i − 1) do    ▷ Aggregation
15:             RSs_i.insert(Es_j.always_actives());
16:         REs_i = fa.rerun(ck_i, RSs_i);   ▷ As redo line 5-9
17:         Es_i.insert(REs_i);
18:     sync();                       ▷ GPU Global Synchronization
```

By leveraging the proposed hot-state-based speculation, all ending states of the running NFA processing are guaranteed to be the ground-truth states. Furthermore, once an *always-active* state is activated, it remains active throughout the rest of the processing [13]. Built upon these properties, the aggregation-based recovery method enables collaborative work among all threads on GPUs. Specifically, this recovery method continuously checks for any missed targeted states in every unverified chunk (line 11–12). When mis-speculation occurs, instead of solely re-running the missed starting states just discovered (line 13), it also gathers the missed *always-active* states from all previous chunks for recovery (line 14–15), resulting in faster convergence towards the ground truth.

TABLE II
NFA APPLICATIONS USED FOR EVALUATION.

Application	Abbr.	Connected Components (CCs)		
		Nums	Max #ST	Avg. #ST
Brill	Brill	5,946	40	19.4
CRISPR-1	CRP1	2,000	37	37
CRISPR-2	CRP2	2,000	101	101
ClamAV	CAV	33,171	22,075	71.6
EntityResolution	ER	10,000	75	41.3
Hamming	HM	1,000	108	108
Levenshtein	LV	1,000	109	109
Protomata	Pro	1,309	123	18.4
RandomForest	RF	16,000	62	62
Snort	Snort	2,486	4,509	81.3
YARA	YARA	23,530	1,017	44.5
Ranges1	Ran1	297	96	42
Ranges05	Ran5	299	94	42.2
TCP	TCP	738	391	26.7

V. IMPLEMENTATION

Guided by performance analyses and discussions of design space presented in previous sections, we implement *ANG*, a framework that explores fine-grained parallelism at multiple levels to maximize the efficiency of NFA processing on GPUs. It consists of two major components: (i) a C++ library which provides a uniform interface to various NFAs through a set of APIs. The major arguments to these APIs include NFA objects and input sequences. Other parameters such as the detailed parallelization scheme, data structures of NFA topology and active states can be automatically or manually configured; (ii) a profiler which can provide profiling to estimate NFA properties, including the number of active states and transitions during NFA processing, and the NFA topological information. It supports two modes: (1) *Offline*, which statically analyzes the given NFA by inspecting its transition table, and (2) *Online*, which processes the targeted NFA over a small prefix (e.g., $<$0.1%) of one randomly selected testing input sequence to collect runtime information. This runtime profiling overhead is negligible (less than 1% of the end-to-end latency). The implementation of online profiling is similar as that in *HotStart* [25] (though collected properties are different) and has been optimized with techniques from prior work [15], [25], [37]. The profiling results are used to guide the selection of parallel schemes and processing configurations. In the following evaluation, we use the online profiler. We also enable dynamic switching between *ANG* and other works based on runtime profiling to allow users get the optimal performance across diverse NFAs and inputs.

VI. EVALUATION

A. Methodology

Our evaluation includes the following GPU-based designs:

- *iNFAnt* [16], the first NFA processing engine for GPUs;
- A hot-state-based NFA processing design (*HotStart* [25]);
- *ngAP* [26] which tries to improve thread utilization by enabling parallel processing on different input symbols;
- *ANG* which contains PC-based and PB-based parallelization schemes introduced in this paper: speculative NFA parallelization with using thread per chunk (*TPC*) and warp per chunk (*WPC*).

The implementations of *iNFAnt*, *HotStart* and *ngAP* are based on their published codes[2,3]. We also enable all other optimizations proposed in their original works, such as interwaved input storage layout and multistriding, whenever these optimizations are beneficial for the given NFA and input sequence. Besides the above approaches, the other design we considered but not included for direct comparison is *vNFA* [24]. It needs a preprocessing procedure to divide states into multiple compatible groups where states belonging to the same group cannot be active simultaneously. However, the time complexity of this procedure is at least quadratic in the number of NFA states. We found that the published codes of this procedure could not finish within 1 hour for NFAs with more than 1k states. Under this situation, we leave addressing this challenge and conducting fair comparisons with *ANG* to our future work.

Benchmarks. Table II lists the source applications of NFA benchmarks used in our evaluation. These applications come from AutomataZoo [42] (row 2-12) and Regex [43] (row 13-15). Each application contains multiple small NFAs, which are also considered as connected components (CCs) from the perspective of representing each NFA as a directed graph. Prior works aiming to maximize throughput typically combine multiple small NFAs into a single large NFA by packing a subset of CCs together. This aggregated NFA is then processed using one thread block per input sequence on GPUs. As prior studies [13], [26] show that the properties of aggregated NFA vary significantly, to simulate different combination scenarios, we construct NFAs by randomly selecting varying numbers of CCs from source applications. For example, to evaluate medium parallelism scenarios in the Snort application, 1% of the CCs (also representing a subset of regular expressions) are randomly selected to build the target NFA. This setup aligns with recent Snort-based packet inspection applications [44], [45], and we extend it to other applications, as we believe it reflects many real-world scenarios. This NFA construction process is repeated 10 times for each application under low, medium, and high parallelism settings. The static profiling results of the evaluated benchmarks under various amounts of parallelism are summarized in Table III. We also observe that our online profiling gets similar results when using the first 1k symbols from testing input sequences for each NFA.

Datasets. We provide 82 input sequences with each of size 30MB for an NFA. These input sequences are collected by following guidelines from their respective benchmark suites. For instance, the inputs to CAV benchmarks are binary executables from a Linux machine and the inputs to Snort ones are network traffic traces collected from a Linux server with using `tcpdump`. For NFAs without specific input collection

[2]https://github.com/bigwater/gpunfa-artifact
[3]https://github.com/getianao/ngAP

TABLE III
STATIC PROFILING RESULTS OF EVALUATED BENCHMARKS AND THE EXECUTION TIME (IN SECONDS) OF APPLYING *iNFAnt*.

App.	Low Parallelism (0.1% or 1 CCs)				Medium Parallelism (1% CCs)				High Parallelism (10% CCs)			
	#ST	#AS	#Trans	*iNFAnt*	#ST	#AS	#Trans	*iNFAnt*	#ST	#AS	#Trans	*iNFAnt*
Brill	11.8	1	4.8	17.1	1113.2	59.0	418.2	23.8	11585.8	594.0	4225.2	121.1
CRP1	37.0	2	12.2	16.2	740.0	40.0	243.6	17.5	7400.0	400.0	2435.9	90.8
CRP2	101.0	2	72.2	15.9	1919.0	38.0	1371.6	30.6	19998.0	396.0	14293.1	245.7
CAV	39.1	1	0.2	13.5	1932.4	18.7	62.1	23.9	19971.6	285.9	345.7	95.8
ER	43.8	1	17.7	16.6	1995.0	48.0	814.2	30.2	19977.0	483.3	8161.5	199.7
HM	108.0	2	79.1	16.0	1080.0	20.0	791.0	21.3	10800.0	200.0	7909.8	152.6
LV	109.0	4	164.1	16.2	1090.0	40.0	1641.0	36.9	10137.0	372.0	15261.4	225.5
Pro	10.2	1	0.9	16.9	216.1	13.0	92.3	16.9	2494.5	129.3	1347.7	39.2
RF	62.0	1	51.5	16.2	1984.0	32.0	1651.1	34.9	19964.0	322.0	16621.8	298.4
Snort	10.4	1	0.8	14.7	1569.8	26.8	265.4	23.6	18383.9	244.6	7951.5	185.6
YARA	10.4	1	2.4	14.7	1803.7	51.3	164.0	22.3	19974.6	464.3	2447.1	145.5
Ran1	27.0	1	0.4	14.8	88.4	2.0	0.5	14.4	1185.7	29.0	7.2	18.8
Ran5	36.6	1	0.2	14.0	89.2	2.0	0.3	14.4	1236.8	29.0	6.8	18.9
TCP	28.7	1	0.7	14.7	172.3	7.3	9.6	15.9	1821.5	77.9	145.3	20.6

Note: "#ST", "#AS", and "#Trans" stand for the average number of states, the average number of always-active states, and the average number of transitions per input symbol, respectively.

guidance, we generate inputs by concatenating multiple copies of the example input streams provided in the original suites.

Evaluation Platform. We evaluate *ANG* on a machine that consists of two Intel 3.0GHz Xeon Gold 6248R processors with 256 GB of RAM and an NVIDIA RTX 3090 GPU. The GPU device is powered by the Ampere architecture. It contains 24GB of global memory and 82 Streaming Multiprocessors (SMs), with each consisting of 128 cores and 100KB of shared memory. The host operating system for our experiments is Linux 3.10.0. All CUDA/C++ programs are compiled with GCC 9.5 and CUDA 11.1 with using `-O3` flag.

The latency of processing each constructed NFA on a given input sequence is measured with the geometric mean latency reported as the average performance across all runs. Each set of experiments is performed 10 times, and we do not report the 95% confidence interval of the average as the variation is not significant. For a fair comparison with prior works, our results do not include the I/O time and data structure preparation time.

B. Overall Performance

The overall latency performance across the tested benchmarks is presented in Fig. 5, which reports the speedups of various designs compared to *iNFAnt*. On average (geometric mean), *HotStart*, *ngAP*, *TPC*, and *WPC* all outperform *iNFAnt*, achieving 1.13×, 11.3×, 9.56×, and 14.39× speedups among all tested NFAs. The average latency of applying *iNFAnt* on each benchmark is reported in Table III. When comparing the two newly proposed designs in *ANG* with state-of-the-art (SOTA) techniques (where the best among *iNFAnt*, *HotStart*, and *ngAP* is considered), the experimental results show that *ANG* yields speedups ranging from 0.05× to 49.88× across various NFAs. The major reason is that the parallelization performance of each framework highly depends on state-level parallelism exposed in the given NFAs.

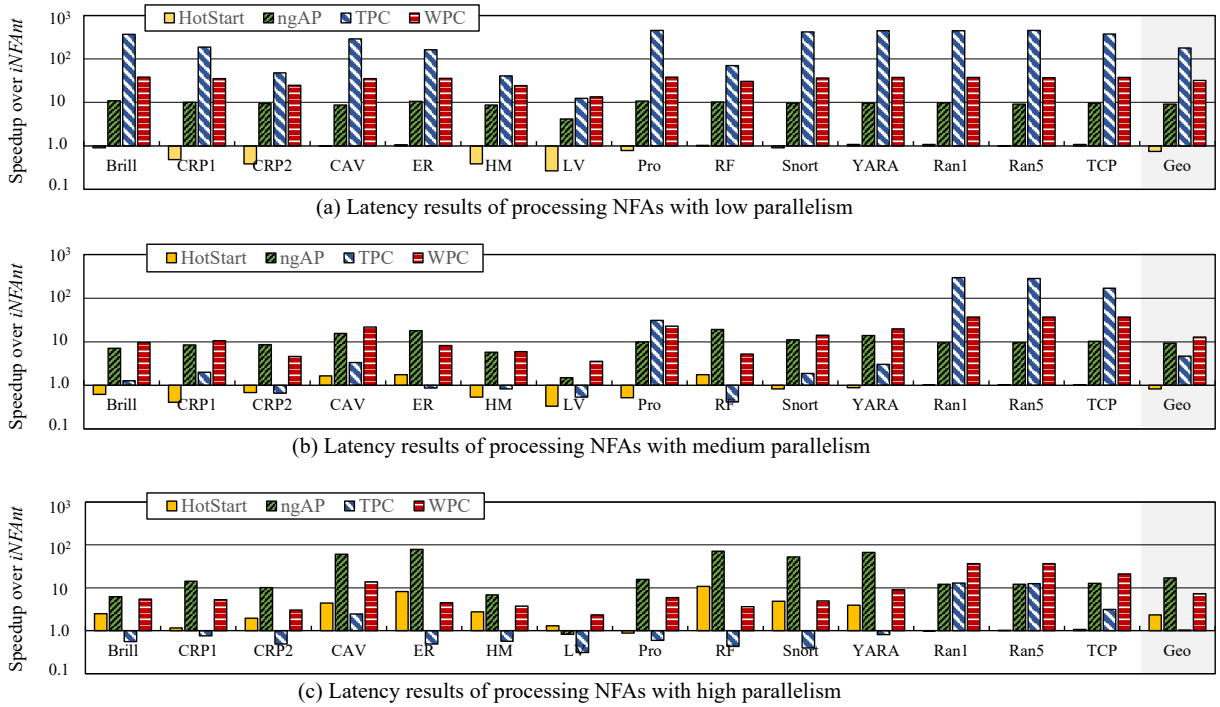
Specifically, for benchmarks in the low parallelism construct setting, where most have fewer than 100 transitions per step occurring on average during processing, *TPC* achieves the best performance, with the geometric mean speedup over SOTA being 27.7×. In NFA benchmarks from CAV, Pro, YARA, Ran1, Ran5, and TCP applications, the speedups are over 40×. Meanwhile, *WPC* also performs well in general, with a 3.49× average speedup over SOTA. This demonstrates the effectiveness of introducing chunk-level parallelism when state-level parallelism is insufficient. In benchmarks under the medium parallelism setting, where the ones from Brill, CRP1, CAV, LV, Snort, and YARA have the average number of transitions per step below 1k, *WPC* performs the best. However, *TPC* still outperforms both *WPC* and SOTA on specific medium-parallelism benchmarks like those from Pro, Ran1, Ran5, and TCP, where state-level parallelism remains insufficient despite 1% of CCs being aggregated to generate an NFA. For most benchmarks with high parallelism, the performance results of *ngAP* and *HotStart* are significantly better than *iNFAnt* and *TPC*, as they well utilize sufficient state-level parallelism and GPU memory hierarchy to enable high thread utilization and efficient data movement. This was also demonstrated in prior studies [13], [25], [26].

TABLE IV
IMPACT OF THE PROPOSED SPECULATION AND RECOVERY.

NFAs	Avg. Spec. Results		Speedup over Seq.	
	Accu. (%)	TPR (%)	AG	ED
CAV-M1	45.36	97.83	23.62	1
CAV-M2	19.45	95.89	23.68	16.78
CAV-H1	3.62	95.77	29.17	1
CAV-H2	14.56	95.61	24.57	2.66
Snort-M1	6.65	97.78	24.83	24.16
Snort-M2	0	97.82	24.91	1.21
Snort-H1	0.09	99.60	18.63	2.28
Snort-H2	0	98.35	21.46	1

C. Breakdown Analysis

Effect of Optimized Speculation and Recovery. To evaluate the impact of the proposed speculation and recovery methods, we conduct case studies on two applications, CAV and Snort,



(a) Latency results of processing NFAs with low parallelism

(b) Latency results of processing NFAs with medium parallelism

(c) Latency results of processing NFAs with high parallelism

Fig. 5. Overall Performance of running different GPU-based designs over NFAs with different state-level parallelism.

with the breakdown results shown in Table IV. For each application under medium and high parallelism scenarios, we selected two benchmarks and analyzed their speculation accuracy and true positive rate (TPR), as shown in Columns 2–3. The results indicate that the average speculation accuracies for these benchmarks are relatively low, ranging from 0% to 45.36%. The accuracy of Snort NFAs is low because those non-initial but always-active states are hardly captured by lookback techniques. Despite this, the recovery mechanisms can reuse most of the existing execution results due to their high TPRs, which range from 95.61% to 99.6%. Additionally, all tested NFAs report false discovery rates (FDRs) of zero, demonstrating the reliability of the proposed conservative speculation methods. On the other hand, Column 4-5 report the efficiency of two recovery strategies: our aggregation-based recovery (AG) and ending-state-based recovery (ED) [15], [40]. These parallel recovery designs achieve significant speedups over naive sequential recovery, with improvements of 23.57× and 10.65×, respectively. Case studies on CAV and Snort NFAs with low parallelism were excluded, as these benchmarks exhibit high speculation accuracy (>90%) and therefore achieve strong performance without requiring recovery optimizations. Our observation reveals that the proposed recovery optimizations provide minimal benefit for NFAs with generally high speculation accuracy, as these cases already leverage parallelism effectively without worrying about the recovery overhead.

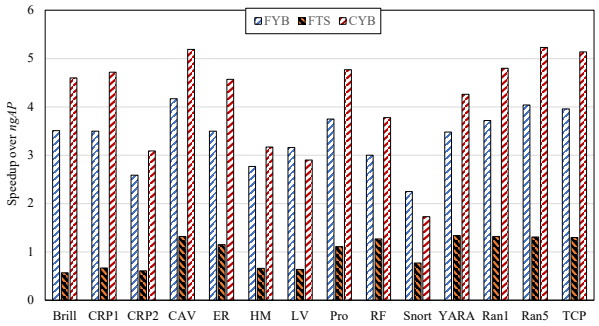


Fig. 6. Efficiency of different configuration choices.

Impact of design parameters. Fig. 6 reports the performance impact of three selected configuration combinations under the *WPC* scheme. Among these configurations, CYB generally shows superior performance, achieving up to 5.23× speedup over *ngAP* and outperforming FYB and FTS across most benchmarks. However, FYB has better performance than others when the proportion of transitions with currently active source states is significantly low, as observed in the LV and Snort benchmarks under low-parallelism scenarios (as reported in Table III). This is because the fixed-size table design in FYB reduces memory access overhead associated with those sparse and uniformly distributed transition patterns.

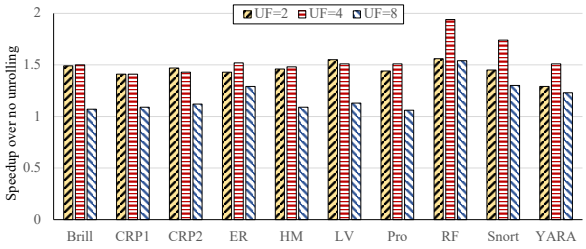


Fig. 7. The effect of unrolling factor.

TABLE V
L1 HIT RATE WHEN APPLYING UNROLLING.

NFAs	L1 Hit Rate (%)			
	No Unroll	UF=2	UF=4	UF=8
Brill	32.3	59.6	77.0	87.6
CRP1	91.5	94.3	96.7	92.5
CRP2	46.0	55.6	71.5	75.7
ER	32.5	55.5	73.0	76.8
HM	92.2	94.8	94.9	87.8
LV	92.2	65.6	78.8	87.4
Pro	50.6	65.9	79.7	88.3
RF	33.1	57.7	74.6	77.4
Snort	33.0	56.0	72.6	76.5
YARA	31.4	50.2	66.5	71.5

Unrolling Factor. Fig. 7 shows the performance improvements achieved by *WPC* when using different unrolling factors across various NFAs. As discussed in Section IV-C, unrolling is not applied to *TPC* scheme or to cases where the profiler identifies insufficient state-level parallelism. The results demonstrate that an unrolling factor of 4 consistently delivers the best performance across all tested NFAs, achieving up to a 94% improvement and an average improvement of 54.8% compared to disabling unrolling.

To further analyze the impact of unrolling, we measured the L1 cache hit rates under different unrolling factors, as summarized in Table V. The results indicate that larger unrolling factors improve data locality by better utilizing registers and cache, leading to higher L1 hit rates. However, as the unrolling factor increases beyond the optimal point, the benefits of improved locality are offset by increased thread divergence, ultimately limiting performance gains.

VII. RELATED WORK

Accelerating Automata Processing. For both NFA and DFA processing tasks, a wide range of algorithmic and architectural solutions have been proposed. DFA processing can be accelerated through techniques like transition compression, fast lookup tables, and SIMD-friendly designs on different parallel platforms and specialized hardware accelerators [17], [21], [22]. NFAs introduce computational challenges because of multiple active states in each step. However, their compact representation and expressive power make them attractive [12]. Recent works explore asynchronous execution [13], space efficient FPGA designs [19], and overlay-based reconfigurable acceleration [20]. Furthermore, software-hardware co-design approaches integrate optimized in-memory data structures with hardware support [18], [23]. These systems are designed to exploit the inherent automata parallelism by optimizing data movement, resource utilization, and concurrency control across heterogeneous environments. Most prior efforts focus on maximizing throughput for batch processing, often neglecting latency-critical scenarios, especially when state-level parallelism is limited. Our work explores multi-level parallelism to enable fast NFA processing. While we do not directly compare with specialized hardware, the high-level idea of the proposed conservative speculation and aggregation-based parallel recovery are architecture-agnostic. thus can be integrated into other existing designs and parallel platforms.

NFA parallelization on GPUs. While GPU is commonly used as NFA accelerators, it also introduces significant challenges in minimizing data movement and enhancing thread utilization. Different optimization techniques have been proposed, including transition table optimizations [16], compression techniques [41], compiler-generated memory-efficient code [12], direct memory access for input execution [46], and on-chip storage of topology data [25]. Recent approach partitions input chunks across dedicated threads [13] but incur redundant computations and elevated worst-case time complexity. Though speculative parallelization has been widely used for accelerating DFA processing over different hardware platforms [15], [34], [47], speculative NFA parallelization remains significantly more challenging, either on GPUs or other architectures, due to its inherently non-deterministic nature [13]. Our approach demonstrated a practical and generalizable speculative NFA execution to exploit chunk-level parallelism while avoiding heavy overheads and maintaining bounded time complexity.

VIII. CONCLUSION

This work focuses on accelerating latency-sensitive NFA processing on GPUs. To address the performance bottleneck from the insufficient state-level parallelism, it explores opportunities of introducing fine-grained parallelism at multiple levels. By analyzing different NFA parallelization schemes, this work, for the first time, introduces speculative parallelization into NFA processing. To make speculative parallelization practical on GPUs, it further investigates how to address the key implementation challenges and then develops *ANG*, a latency-oriented NFA processing framework on GPU. Experiments show that *ANG* outperforms the state-of-the-art by up to 49.88×, demonstrating the benefits of exploring fine-grained parallelism at multi-levels.

ACKNOWLEDGMENT

We are grateful to the anonymous reviewers for their constructive comments and suggestions. The work was partially supported by City University of Hong Kong internal and donation fundings (No. 9610598 and No. 9220148), and the National Science Foundation (NSF) Grant (No. 2105006).

REFERENCES

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

[2] C. Bo, K. Wang, J. J. Fox, and K. Skadron, "Entity resolution acceleration using the automata processor," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 311–318.

[3] J. Qiu, L. Jiang, and Z. Zhao, "Challenging sequential bitstream processing via principled bitwise speculation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 607–621.

[4] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE transactions on information forensics and security*, vol. 11, no. 2, pp. 289–302, 2015.

[5] Y. Qi, J. Zhong, R. Jiang, Y. Jia, A. Li, L. Huang, and W. Han, "Fsm-based cyber security status analysis method," in *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*. IEEE, 2019, pp. 510–515.

[6] S.-H. Bae, S.-H. Joo, J.-W. Pyo, J.-S. Yoon, K. Lee, and T.-Y. Kuc, "Finite state machine based vehicle system for autonomous driving in urban environments," in *2020 20th International Conference on Control, Automation and Systems (ICCAS)*. IEEE, 2020, pp. 1181–1186.

[7] K. Esper, S. Wildermann, and J. Teich, "Enforcement fsms: specification and verification of non-functional properties of program executions on mpsocs," in *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*, 2021, pp. 21–31.

[8] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "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*. Springer, 2016, pp. 200–218.

[9] E. Sadredini, R. Rahimi, K. Wang, and K. Skadron, "Frequent subtree mining on the automata processor: challenges and opportunities," in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1–11.

[10] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the micron automata processor," in *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*. IEEE, 2015, pp. 236–239.

[11] E. Sadredini, D. Guo, C. Bo, R. Rahimi, K. Skadron, and H. Wang, "A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 665–674.

[12] 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*, 2017, pp. 1–11.

[13] H. Liu, S. Pai, and A. Jog, "Asynchronous automata processing on gpus," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 1, pp. 1–27, 2023.

[14] N. S. K. Bashah, T. S. Simbas, N. Janom, and S. R. S. Aris, "Proactive ddos attack detection in software-defined networks with snort rule-based algorithms," *International Journal of Advanced Technology and Engineering Exploration*, vol. 10, no. 105, p. 962, 2023.

[15] J. Qiu, X. Sun, A. H. N. Sabet, and Z. Zhao, "Scalable fsm parallelization via path fusion and higher-order speculation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, p. 887–901.

[16] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "infant: Nfa pattern matching on gpgpu devices," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 20–26, 2010.

[17] H. Liu, M. Ibrahim, O. Kayiran, S. Pai, and A. Jog, "Architectural support for efficient large-scale automata processing," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 908–920.

[18] L. Kong, Q. Yu, A. Chattopadhyay, A. Le Glaunec, Y. Huang, K. Mamouras, and K. Yang, "Software-hardware codesign for efficient in-memory regular pattern matching," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 733–748.

[19] X. Wang, L. Gong, J. Cao, W. Lou, W. Wang, C. Wang, and X. Zhou, "Hap: A spatial-von neumann heterogeneous automata processor with optimized resource and io overhead on fpga," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2023, pp. 185–196.

[20] R. Karakchi and J. D. Bakos, "Napoly: A non-deterministic automata processor overlay," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 3, pp. 1–25, 2023.

[21] F. Carloni, D. Conficconi, I. Moschetto, and M. D. Santambrogio, "Yarb: a methodology to characterize regular expression matching on heterogeneous systems," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2023, pp. 1–5.

[22] A. Le Glaunec, L. Kong, and K. Mamouras, "Regular expression matching using bit vector automata," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 492–521, 2023.

[23] L. Cicolini, F. Carloni, M. D. Santambrogio, and D. Conficconi, "One automaton to rule them all: Beyond multiple regular expressions execution," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 193–206.

[24] 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 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 129–140.

[25] H. Liu, S. Pai, and A. Jog, "Why gpus are slow at executing nfas and how to make them faster," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 251–265.

[26] T. Ge, T. Zhang, and H. Liu, "ngap: Non-blocking large-scale automata processing on gpus," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024, pp. 268–285.

[27] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001.

[28] F. A. Siddique, T. J. Tracy II, N. Brunelle, and K. Skadron, "Deterministic vs. non deterministic finite automata in automata processing," *arXiv preprint arXiv:2210.10077*, 2022.

[29] 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*, vol. 17, no. 1, pp. 84–87, 2017.

[30] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An open-source, full-stack, and customizable automata processing on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 138–147.

[31] V. M. Glushkov, "The abstract theory of automata," *Russian Mathematical Surveys*, vol. 16, no. 5, p. 1, 1961.

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

[33] J. Friedl, *Mastering regular expressions*. O'Reilly Media, Inc., 2006.

[34] Y. Wang, R. Watling, J. Qiu, and Z. Wang, "Gspecpal: Speculation-centric finite state machine parallelization on gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 481–491.

[35] M. Sheeraz, H. Durad, S. Tahir, H. Tahir, S. Saeed, and A. M. Almuhaideb, "Advancing snort ips to achieve line rate traffic processing for effective network security monitoring," *IEEE Access*, 2024.

[36] Z. Zhao, B. Wu, and X. Shen, "Challenging the embarrassingly sequential: Parallelizing finite state machine-based computations through principled speculation," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, p. 543–558.

[37] Z. Zhao and X. Shen, "On-the-fly principled speculation for fsm parallelization," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, p. 619–630.

[38] J. Qiu, Z. Zhao, and B. Ren, "Microspec: Speculation-centric fine-grained parallelization for fsm computations," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016, pp. 221–233.

[39] 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, 2017, pp. 1–10.

[40] Y. Xia, P. Jiang, and G. Agrawal, "Scaling out speculative execution of finite-state machines with parallel merge," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel ProgrammingFebruary*, 2020.

[41] X. Yu and M. Becchi, "Gpu acceleration of regular expression matching for large datasets: exploring the implementation space," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2013, pp. 1–10.

[42] J. Wadden, T. Tracy, E. Sadredini, L. Wu, C. Bo, J. Du, Y. Wei, J. Udall, M. Wallace, M. Stan *et al.*, "Automatazoo: A modern automata processing benchmark suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 13–24.

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

[44] L. Turoňová, L. Holík, I. Homoliak, O. Lengál, M. Veanes, and T. Vojnar, "Counting in regexes considered harmful: Exposing {ReDoS} vulnerability of nonbacktracking matchers," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4165–4182.

[45] C.-M. Ou, Y.-X. Huang, M.-H. Chen, I.-H. Chung, and J. Chou, "Fast malicious packets inspection framework using converged accelerator," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2025, pp. 153–161.

[46] V. Yaneva, A. Rajan, and C. Dubach, "Gpu acceleration of finite state machine input execution: Improving scale and performance," *Software Testing, Verification and Reliability*, vol. 32, no. 1, p. e1796, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1796

[47] P. Jiang and G. Agrawal, "Combining simd and many/multi-core parallelism for finite state machines with enumerative speculation," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 179–191.