GSpecPal: Speculation-Centric Finite State Machine Parallelization on GPUs

Yuguang Wang, Robbie Watling, Junqiao Qiu, Zhenlin Wang
Department of Computer Science
Michigan Technological University
Email: {yugwang, rwatling, junqiaoq, zlwang}@mtu.edu

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00053

***Abstract*—Finite State Machine (FSM) plays a critical role in many real-world applications, ranging from pattern matching to network security. In recent years, significant research efforts have been made to accelerate FSM computations on different parallel platforms, including multicores, GPUs, and DRAM-based accelerators. A popular direction is the speculation-centric parallelization. Despite their abundance and promising results, the benefits of speculation-centric FSM parallelization on GPUs heavily depend on high speculation accuracy and are greatly limited by the inefficient sequential recovery.**

Inspired by speculative data forwarding used in Thread Level Speculation (TLS), this work addresses the existing bottlenecks by introducing speculative recovery with two heuristics for thread scheduling, which can effectively remove redundant computations and increase the GPU thread utilization. To maximize the performance of running FSMs on GPUs, this work integrates different speculative parallelization schemes into a latency-sensitive framework, *GSpecPal*, along with a scheme selector which aims to automatically configure the optimal GPU-based parallelization for a given FSM. Evaluation on a set of real-world FSMs with diverse characteristics confirms the effectiveness of *GSpecPal*. Experimental results show that *GSpecPal* can obtain 7.2× speedup on average (up to 20×) over the state-of-the-art on an Nvidia GeForce RTX 3090 GPU.

***Index Terms*—Finite State Machine, Speculative Parallelization, GPU**

I. Introduction

Finite State Machine (FSM) is a fundamental computational model that has been used in a wide range of real-world applications, including data analytics [1], [2], motif searching [3], [4], natural language processing [5], [6], malware detection [7], [8], and others. Two commonly used FSM representations in these applications include a non-deterministic finite automaton (NFA) and a deterministic finite automaton (DFA). Figure 1 shows an FSM example, $div7$, which can be used to check if a given binary number can be divided by seven. An FSM can be represented as a transition graph (Figure 1(a)) or a transition table (Figure 1(b)). Figure 1(c) illustrates how an FSM executes state transitions on the given input sequentially.

Previous research has tried to use domain-specific accelerators, for example, the Micron Automata Processor (AP), for FSM-based computations [9]–[11]. However, there are some limitations when accelerating FSM computations on AP, including high reconfiguration latency and limited support for arithmetic computations. AP has not been commercialized and in fact its development has been discontinued [12], [13]. In

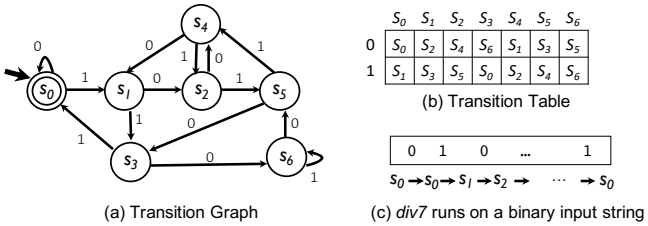

Fig. 1: An FSM example *div7*. In (a), state s_0 is the initial state (marked by an incoming edge) as well as an accepting state (shown in double-circles).

recent years, the research community has turned back to accelerate FSM-based computations on traditional von Neumann architectures (e.g., CPUs and GPUs). Compared with CPU, GPU shows more potential because of its massive power in data parallel computing.

In past decades, GPUs have been mainly utilized as throughput-oriented processors for accelerating FSM computations [8], [14]–[18]. A recent study [19] has proposed a speculation-centric technique to explore the latency-sensitive optimization for FSM computations on GPUs. It first divides the given input stream into multiple chunks, then relies on enumerative speculation [20] to enable parallel FSM processing on each chunk. Thanks to the high speculation accuracy and an efficient parallel tree-like verification technique, this approach can achieve significant speedups on a set of FSMs. However, when the high speculation accuracy cannot be guaranteed, it still requires the *sequential verification and recovery* to ensure the correctness, and thus leads to under-utilization of GPU threads and dilutes the benefits from parallelization. Though another recent research [21] develops a *higher-order speculation* to activate multiple threads during the recovery, in some cases, the number of idle threads is still large, which is a major source of inefficient FSM computation on GPUs.

In this paper, we solve the poor thread utilization problem by showing that the speculation in any chunk can be speculatively verified and then re-execution can start earlier (if it is needed). Therefore, we propose an aggressive parallelization design which breaks the one-to-one binding relationship between threads and chunks during verification and recovery. We also show that well organized thread scheduling in this aggressive parallel design is necessary for reducing redundant

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00053

computations. We thus introduce two heuristics, a round-robin based and a nearest first based algorithm, to guide threads working on different chunks for speculative recovery. In addition, to avoid excessive GPU global memory access, we propose an FSM transformation technique, and develop a hierarchical design for storing the speculative execution and recovery results. Finally, to enable shortest response time for any given FSM and input stream, we integrate different speculative parallelization schemes as well as a scheme selector into a framework, called *GSpecPal*. Our evaluation of *GSpecPal* on a set of real-world FSM benchmarks with various characteristics confirms its effectiveness in accelerating FSM computations on GPUs, yielding 7.2× speedup on average and maximum 20× speedup.

In summary, this work makes the following contributions.

- First, it proposes an aggressive speculative recovery design with two heuristics for thread scheduling, to enable efficient FSM parallelization on GPUs.
- Second, through performance analyses on various parallelization schemes, it reveals the efficiency issues on speculative FSM parallelization on GPUs, and guides the selection of optimal parallelization scheme.
- Finally, it designs and implements a latency-sensitive framework and confirms its effectiveness on a set of real-world benchmarks.

II. Background and Motivations

In this section, we first provide a brief background on FSMs and their parallelization. Then we present the previous efforts on accelerating FSM computations via GPUs, in particular, the state-of-the-art speculative DFA parallelization on GPUs. Finally, we point out the bottlenecks of the existing approaches and motivate our work.

A. FSM Basics and Parallel Strategies

An FSM can be represented as a tuple (Q, Σ, q_0, δ, F), where Q is a finite set of states, Σ is the alphabet, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function which indicates the output state(s) when given a state and an input symbol (usually represented as a transition table), and F is the set of reporting or accepting states. Depending on whether an input symbol can activate multiple output states in one transition, FSMs can be deterministic or non-deterministic. Every NFA can be converted to an equivalent DFA [22]. Lines 6-14 in Algorithm 1 show how an FSM runs on a given input stream. An FSM starts with the initial state(s) and consumes one input symbol at a time. At each symbol, the FSM may activate one or multiple states by looking up the transition table. This process repeats until all input symbols are consumed. Besides state transitions, the FSM may also invoke an output function ϕ on the current activated states and input symbol. In this paper, we follow the settings in previous work [19], [23], [24] and assume such a ϕ function is a void function at each step. We only invoke an output function after we know the end state, to report an accept or reject decision

TABLE I: Notations

Notations	Description
N	the number of threads
QS_i	a concurrent queue of ranked spec. states on chunk i
VR_i	a vector of spec. execution or recovery records on chunk i (each record is a state pair {`start`, `end`})
f	ID of the latest chunk being truly verified (i.e., frontier)
tid, cid	thread ID and chunk ID
end_c, end_p	end state of the current and predecessor chunk

[23]. To make the presentation easy to follow, we also include some other important notations in Table I.

FSM Parallelization. The existing mechanism of FSM computations exhibits parallelism at multiple levels, as shown in Algorithm 1. Various FSMs as well as input streams can be processed simultaneously (Lines 2-3). These two sources of parallelism are called *FSM-level parallelism* and *Stream-level parallelism*, respectively [18]. When running an FSM over one input stream, however, the FSM processing is considered as "embarrassingly sequential" [25], [26]. Data dependencies exist in each step of FSM state transitions, i.e., the next activated state(s) always depends on the current activated state(s) and the current input symbol. Prior studies have found that this FSM processing can still get benefits from parallelization. In NFA processing, since there may be more than one state being activated in each step, multiple transitions can be executed at the same time (Lines 9-10). Such a parallelism is called *State-level parallelism* [27]. On the other hand, in DFA processing (as shown in Figure 1), the inherent data dependencies prevent any parallelism from being exposed.

Algorithm 1 Parallelism in FSM Computations [18], [27]

▷ **Input:** a set of FSM benchmarks $Sfsm$ and a set of inputs Sin.

```
1:  procedure PARALLEL_FSM(Sfsm, Sin)
2:      Forall (FSMs fsm in Sfsm)
3:          Forall (Inputs in in Sin)
4:              Sact.insert(fsm.init_state());
5:              FSM_Processing(fsm, in, Sact);
6:  procedure FSM_PROCESSING(fsm, in, Sact)
7:      Table[···][···] = fsm.transition_table();
8:      for i = 1 : in.length do
9:          Forall (states st in Sact)          ▷ Sact is a set of states
10:             Sact'.insert(Table[st][in.at(i)]);   ▷ state transitions
11:         φ(Sact', in.at(i));                 ▷ assume φ is a void func.
12:         swap(Sact, Sact');
13:         Sact'.clear();
14:     return Sact;
```

To break data dependencies in DFA processing, existing efforts mainly fall into two directions: speculative [26], [28] and enumerative [23], [24] parallelization. Algorithm 2 shows the default 3-phase algorithm used for speculative DFA parallelization. After partitioning an input stream into T chunks evenly, the speculative method predicts a start state for each chunk (except the first one) to enable parallel execution, then relies on sequential verification and recovery to ensure the correctness. The enumerative method, on the other hand, enumerates all the possible states for each chunk and thus

allows different chunks to run in parallel. Then the real execution path can be determined by connecting the end state of each chunk to the start state of its successor. In recent years, a new approach called enumerative speculation is proposed to combine the above two ideas [19], [20]. In fact, all these approaches exploit *chunk-level parallelism*.

Algorithm 2 Default Speculative DFA Parallelization with Sequential Verification and Recovery [26], [29], [30]

```
1:  procedure SPEC_DFA_PROCESSING(fsm, in)
2:      Π = partition(in, N);                          ▷ partitioning
3:      Forall (thread i = 1 : N)                      ▷ parallel speculative exec.
4:          QS_i = predict(Π(i));
5:          end = FSM_Processing(fsm, Π(i), QS_i.front());
6:          VR_i.push_back({QS_i.front(), end});
7:      join();
8:      end_p = VR_1[0].end;
9:      for i = 2 : N do                               ▷ seq. verification and recovery
10:         if end_p ≠ VR_i[0].start then
11:             end_c = FSM_Processing(fsm, Π(i), end_p);
12:         else
13:             end_c = VR_i[0].end;
14:         end_p = end_c;
```

B. GPU Acceleration

The parallel FSM computations described in Algorithm 1 fit well for GPUs: different sources of parallelism can be deployed onto different levels on GPU architectures, i.e., grids, thread blocks, and warps [19], [27]. As massively parallel architectures, GPUs enable thousands of concurrent threads and have higher memory bandwidth (orders of magnitude more than CPUs), so in the past decade, a significant amount of work has been proposed on accelerating FSM computations via GPUs [8], [16]–[18]. Most existing GPU-based FSM parallelizations are built on FSM-level and stream-level parallelism to reach higher throughput, and ignore the peak performance (i.e., the response time) of running over a single input stream. Furthermore, their implementations usually rely on NFAs, considering that NFAs present state-level parallelism and NFAs are inherently memory efficient [27].

As the demand for real-time data analytics increases, achieving low latency becomes critical, and thus makes DFAs be more attractive since DFAs can provide faster per-character processing (only one state transition in each step). Meanwhile, even though GPUs are notorious for having poor latency, the massive compute power still makes GPUs great parallel platforms in exploring latency optimization for FSMs. In this paper, we focus on optimizing the peak performance of DFA parallelization on GPUs. In the following sections, the terms FSM and DFA will be used interchangeably.

State of The Art. Recently, Xia et al. [19] proposed a latency-sensitive technique, called *Parallel Merge (PM)*, for efficient DFA processing on GPUs. *PM* enables chunk-level parallelism by utilizing enumerative speculation and develops a *parallel tree-like merge* technique to reduce the overhead in sequentially verifying the speculation. Figure 2 shows an example of applying *PM* onto *Div7*. Each thread executes

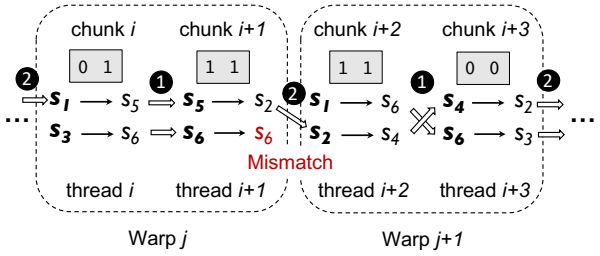

Fig. 2: An illustrative example of running *div7* with *PM*.

DFA processing over a divided chunk with starting from two speculative states, instead of just one state, and maintains both state transition paths. After threads in the same warp finish their speculative execution, intra-warp verification (❶) is performed. Since both end states in chunk i can find their matches to the speculative start states in chunk $i+1$ (thread i and $i+1$ are in the same warp), the execution results of these two threads can be merged together. After all warps finish the local verification and merge the results, inter-warp verification (❷) is performed. If a mismatch is found (e.g., the end state s_6 in chunk $i+1$), *PM* marks the path $s_3 \rightarrow s_6 \rightarrow s_6$ as invalid rather than re-executing chunk $i+2$ with start state s_6 immediately. Such a strategy is to avoid unnecessary recovery. To be more specific, assuming that the start state of chunk i is verified to be s_1, the real end state of chunk $i+1$ is s_2. In this case, missing the execution results starting from s_6 on chunk $i+2$ does not prevent us from getting the desired ground truth. Finally, when the inter-warp verification and merge are completed, the ground truth can be determined by checking the valid paths.

C. Motivations

While *PM* shows promising results on running DFA computations on GPUs, it turns out that this technique has two major limitations, which are discussed in the following.

The benefits brought by *PM* heavily rely on a high speculation accuracy. To ensure that, *PM* and some other prior studies [20] adopt *spec-k*, which maintains k transition paths starting from various speculated states during speculative execution. However, this enumerative speculation introduces some unnecessary executions. Figure 3 shows the normalized execution time of maintaining 4, 6, and 8 state transition paths on divided chunks, with ignoring the verification and recovery. More overheads can be seen with a larger k since more redundant state transitions are introduced. Moreover, in previous work of utilizing *spec-k* [19], [20], the value of k is determined statically and does not change across all divided chunks. As such, a thread may waste compute resources when k is set to be too large on an easy-to-predict chunk, or may need recovery later when k is too small to cover the desired ground truth on the assigned chunk.

On the other hand, even though the speculation accuracy could be generally high, the sequential verification and recovery may still occur and then reduce the benefits from

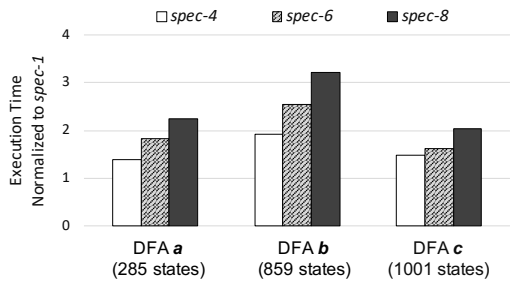

Fig. 3: Execution time of *spec-k* normalized to the *spec-1* solution (neither verification nor recovery is performed).

parallel execution. Though the state-of-the-art has proposed an architecture-aware parallel verification approach and a delayed recovery strategy which optimistically speculates a mismatch as invalid, the sequential recovery bottleneck remains when a mis-speculation really affects the determination of the ground truth and needs to be addressed. For instance, in Figure 2, if the execution path $s_3 \rightarrow s_6 \rightarrow s_6$ is part of the ground truth, recovery starting from s_6 over chunk $i+2$ is unavoidable. This sequential verification and recovery problem is more serious when running DFAs on GPUs, since it causes a large fraction of GPU threads to be idle and thus poor hardware utilization.

Ideally, if both verification and recovery can be done in parallel with all threads activated, speculative FSM parallelization on GPUs can be more efficient. In the next section we propose an advanced speculative parallelization, to address the performance bottlenecks described above.

III. Speculative Recovery

This section first presents the insights of speculative recovery. Then based on FSM properties and speculation characteristics, we discuss two heuristics that can be used to improve the performance. Finally we analyze the efficiencies of different parallel schemes for further building a scheme selector.

A. Insights

An interesting observation is that the speculative FSM parallelization suffers from a similar kind of inefficiency of serial mis-speculation recovery as the conventional Thread Level Speculation (TLS). A well-known solution to this inefficiency is adding nested speculation, which enables earlier verification and recovery by forwarding the speculative data [31]–[33]. *PM* in fact is a special variant of this speculative verification and recovery approach. It forwards multiple speculative execution results from a thread to the successor thread, but when a mismatch is found (by comparing the forwarded end state with the speculative start states in the successor), it optimistically speculates that the mismatch is harmless and thus delays the recovery, as described in Section II-B.

Another recent study [21] shows a more conservative approach, which introduces immediate **S**peculative **R**ecoveries activated by **E**nding states from predecessor threads (*SRE*). Though it is originally designed for multicores, it can also be implemented on GPUs with slight modifications. Algorithm 3 presents the verification and recovery kernel executed by each thread. Note that, *SRE* follows the first two phases in the default speculative parallelization (i.e., lines 2-7 in Algorithm 2). But similar to *PM*, it allows each thread to concurrently forward results of speculative execution and then start the earlier verification (lines 4-8). Instead of delaying the recovery when a mismatch is found, it triggers the re-execution immediately. Thanks to the FSM **state convergence** property (i.e., different FSM states may transit to the same state after running on a certain length of inputs), the end state from the previous chunk has a higher possibility to be the ground truth, compared with the original speculative start states [21]. More recoveries are executed and some of them may be redundant in *SRE*, but among these speculative recoveries, the one occurs in the frontier chunk, i.e., the chunk where the first mismatch is found, belongs to the *must-be-executed* recovery, thus the overhead brought by redundant recovery can be covered by the parallel execution (lines 12-18).

Algorithm 3 **S**peculative **R**ecovery activated by the **E**nding State from Predecessor (*SRE*)

```
1:  kernel SPEC_RECOVERY_END(fsm, in)
2:      f = 0;
3:      __shared__ mark = false;
4:      while f < N do
5:          found = false;
6:          end_p = end_state_comm(end_c);              ▷ thread comm.
7:          for record : VR_tid do
8:              if end_p == record.start then
9:                  found = true;
10:                 end_c = record.end;    ▷ update current end state
11:                 break;
12:         if tid == f then
13:             mark = found;
14:         f++;
15:         sync();
16:         if mark == false and found == false then
17:             end_c = FSM_Processing(fsm, Π(tid), end_p);
18:             VR_tid.push_back({end_p, end_c});
```

However, *SRE* also suffers from the low thread utilization: a thread will be activated and start a recovery task only when it finds a mismatch on the assigned chunk. To address this limitation, we propose an aggressive speculative recovery which breaks the one-to-one binding relationship between threads and chunks. When thread i finishes the speculative execution on chunk i, it may be assigned to any other chunk j ($j \neq i$) for a speculative recovery. Moreover, the speculative recovery task may start from a randomly selected state if no mismatch is found on the current chunk.

B. Two Thread Scheduling Heuristics

An aggressive speculative recovery may improve the thread utilization by randomly scheduling re-execution tasks to idle threads. However, this random scheduling may still yield little performance improvement: some threads may do redundant work in a chunk where the speculation accuracy is high and thus waste computing resources; while some others may start

the re-execution from the states which are nearly impossible to be the correct ones. To achieve the optimal performance of speculative recovery, we need to answer the following two thread scheduling problems: (1) *which chunk should a thread work on for recovery?* and (2) *which FSM states should a recovery start from?* In the following, we present two heuristics to address these two problems.

Round-Robin (RR). Round-robin scheduling is an algorithm widely used in process and network schedulers [34]. A set of tasks are assigned to each process in equal portions and in a circular order. The first heuristic is to introduce round-robin scheduling into speculative verification and recovery, as shown in Algorithm 4. When a *must-be-done* recovery appears on chunk i (*the frontier*), instead of letting the rest threads be idle or randomly assigning recovery tasks to threads, this round-robin based heuristic makes each thread work on a chunk with the following rules: if a thread originally works on the chunk j during speculative parallel execution and $j > i$ (such a thread is called *rear thread*), it stays on the current chunk and starts verification and recovery by speculating the end state from chunk $j-1$ as the ground truth (lines 20-21). In other words, the rear threads follow the same strategy in *SRE*. On the other hand, other threads are assigned to work on chunk $i+1$ to chunk N with round-robin scheduling applied. This design ensures that the maximum number of threads working on a chunk should be $1 + \lceil (i-1)/(T-i) \rceil$, and no thread is wasted on the chunks which have been verified. Because the speculation in each chunk generates a queue QS_i of speculative states, where states are ranked by their probabilities to be the ground truth, a straightforward way to answer the second scheduling problem for non-rear threads is to let them follow the order in QS_i to choose a speculative state (lines 24-25) and then execute the recovery. Note that QS_i is a concurrent queue to ensure thread-safety.

Algorithm 4 Round Robbin based Speculative Recovery (**RR**); derived from Algorithm 3

```
...
16: if mark == true then
17:     continue;
18: else
19:     if tid ≥ f then
20:         cid = tid;
21:         st = end_p;
22:     else
23:         cid = (f + 1) + (tid − 1) % (N − f);   ▷ let cid ≠ tid
24:         st = QS_cid.front();
25:         QS_cid.dequeue();
26:     end'_c = FSM_Processing(fsm, Π(cid), st);
27:     VR_cid.push_back({st, end'_c});
28:     if found == false and tid ≥ f then
29:         end_c = end'_c;
```

Nearest First (NF). Unlike the randomized and the RR based scheduling, this nearest-first heuristic assigns the non-rear threads to execute recovery on the chunks nearest to the frontier. All previous designs are built on a strong assumption that speculations on various chunks are usually similar (i.e., input insensitive). But we observe that in some cases, it is difficult to make an accurate speculation when running on a specific portion of the given input stream. That requires us to provide more resources for making speculation and recovery, so the nearest first heuristic is proposed. Algorithm 5 summarizes its basic idea. All non-rear threads are first scheduled to execute recoveries on the chunk $i+1$ (which is right after the frontier) with enumerating the rest start states in the speculation queue. If the number of the non-rear threads is larger than the size of speculation queue in chunk $i+1$, then chunk $i+2$, chunk $i+3$, and the rest rear chunks become the target in order, until all non-rear threads have been scheduled (lines 25 - 34).

Algorithm 5 Nearest First based Speculative Recovery (**NF**); derived from Algorithm 3

```
...
16: if mark == true then
17:     continue;
18: else
19:     (cid, st) = NF_Sched (tid, i, preEnd);
20:     end'_c = FSM_Processing(fsm, Π(cid), st);
21:     VR_cid.push_back({st, end'_c});
22:     if found == false and tid ≥ f then
23:         end_c = end'_c;
24:
25: procedure NF_SCHED(tid, f, end_p)
26:     if tid ≥ f then
27:         return (tid, end_p);
28:     else
29:         for cid = f + 1 : N do
30:             if QS_cid.size() ≠ 0 then
31:                 st = QS_cid.front();
32:                 QS_cid.dequeue();
33:                 break;
34:         return (cid, st);
```

C. Efficiency Analysis

At high level, the execution time of speculative FSM parallelization can be broken down into three parts: the time spent on predicting start states T_{pred}, the time spent on parallel speculative execution T_{par}, and the verification and recovery time $T_{v\&r}$.

$$T_{spec} = T_{pred} + T_{par} + T_{v\&r} \quad (1)$$

We utilize the same predictor (more details can be found in Section IV-A) for the given FSM and input stream in all speculation-centric parallelization, and the speculation over different chunks can be done in parallel, so we consider the prediction cost as a constant (denoted as C). During speculative execution, *PM* requires each thread maintain k state transition paths (i.e., *spec-k*), while other schemes demand each thread start FSM transitions from only one predicted state. To quantitatively denote the overhead, we introduce the metric *redundancy factor* $\alpha_k = T_{pk}/T_{p1}$, where T_{pk} and T_{p1} are the parallel speculative execution time in utilizing *spec-k* and *spec*-1, respectively. It is expected that $\alpha_k > 1$ when $k > 1$. As the major difference among various parallelization schemes locates in verification and recovery, in the following,

we elaborate the details of $T_{v\&r}$ and then analyze the total execution time for each scheme.

Analysis of *PM*. The verification and recovery phase in *PM* mainly consists of two stages. In the first stage, a **parallel** tree-like verification is performed and there are $logN$ rounds parallel verification in total. During each round, every thread needs to pass k end states to the successor (the time spent is denoted as $T_{comm}(k)$) and provides runtime checks for each end state received from its predecessor thread to determine whether there is a match or not (the cost is represented as $T_{ver}(k)$). No recovery will be performed in this stage. The second stage, on the other hand, is a **sequential** verification and recovery for handling any *must-be-done* recovery appearing after the first stage. During the sequential verification, if a mismatch is found on chunk i (assuming the probability of a mismatch occurs is P_i^{PM}), we have to provide a immediate recovery on the current chunk. Putting all together, the execution time of *PM* can be represented as

$$\begin{aligned} T_{PM} = C + T_{p1} \times \alpha_k + \sum_{i=1}^{\log N}(T_{comm}(k) + T_{ver}(k)) \\ + \sum_{i=2}^{N} P_i^{PM} \times (T_{comm}(1) + T_{ver}(k) + T_{p1}) \end{aligned} \quad (2)$$

In Equation 2, P_i^{PM} is fixed after we made the speculation on chunk i and it equals to $1 - accu_i^{spec\text{-}k}$, where $accu_i^{spec\text{-}k}$ denotes the speculation accuracy of utilizing *spec-k*. Note that speculation in different chunks is independent.

Analysis of *spec-1* based *Speculative Recovery*. For other schemes utilizing speculative recovery, a common strategy is: after speculative execution, there will be at least one thread getting an end state from its predecessor and then check if a mismatch is found until the entire input stream has been verified. Once a mismatch is found, a recovery is executed. As described above, the two heuristics make the threads busy even they don't find a mismatch, but this cost can be hidden by the *must-be-done* recovery in the frontier. In general, the total execution time for parallelization with speculative recovery is

$$T_{SR} = C + T_{p1} + \sum_{i=2}^{N}(T_{comm}(1) + T_{ver}(1) + P_i^{SR} \times T_{p1}) \quad (3)$$

In Equation 3, P_i^{SR} indicates the probability of starting a recovery in the frontier chunk i. Different from P_i^{PM},

$$P_i^{SR} = 1 - (accu_i^{spec\text{-}1} + \Delta_i^{End} + \Delta_i^{Specs}) \quad (4)$$

where Δ_i^{End} denotes the accuracy increment in chunk i due to starting an earlier recovery with end state from its predecessor, and Δ_i^{Specs} indicates that other threads are assigned to run on chunk i with other speculative states and thus we can increase the probability to avoid a recovery ($\Delta_i^{Specs} = 0$ when applying Algorithm 3).

Discussion. Based on the above analysis, we compare the performance of different parallelization schemes and get hints about when a specific scheme works most efficiently.

First, the quality of start state prediction is apparently critical for the performance. If the speculation technique used can ensure that $accu_i^{spec\text{-}1}$ is high, the dominant cost in every speculation-centric parallelization should be in the parallel speculative execution phase, and thus introducing least redundant work during speculative execution is the best choice. But if it is more likely to get a correct speculation only when trying multiple predicted states, applying *spec-k* should be beneficial even though it may introduce redundant computations.

Second, the FSM predictors used in GPUs are usually lightweight to avoid heavy overheads, so it is possible that the probability of a mis-speculation occurs is high even though k is set to be a large number. To reduce this probability, a potential direction is increasing the values of Δ_i^{End} and Δ_i^{Specs}. Based on their definitions above, the former one is affected by the FSM convergence property (Δ_i^{End} is large when any pair of FSM states easily converge [26], [28]), while the latter is still affected by the quality of speculation.

Overall, the efficiency of applying a parallel scheme depends on the speculation technique used and the FSM convergence property. However, FSM transition behaviors are complex and diverse [21]. Instead of precisely producing a closed-form solution for determining the optimal parallelization scheme, we propose a selector guided by the analysis above, to infer which scheme can work well in general for the given FSM and input data.

IV. GSpecPal

Following parallelization schemes discussed in Section III, we develop *GSpecPal*, a latency-sensitive framework that leverages speculative parallelization to maximize the peak performance of FSM processing on GPUs. At high-level, *GSpecPal* consists of four components, including *state prediction*, *state transition*, *verification and recovery*, and *parallel scheme selection*. We next present each of them in order.

A. State Prediction

As discussed in Section III-C, the speculation accuracy greatly affects the performance of FSM parallelization. The key challenge is to determine the likelihoods of different states to be the real start state on a divided chunk. Previous studies have proposed various FSM predictors [26], [28], but these predictors usually target CPU-based FSM processing, requiring tedious online or offline training runs in practice. In fact, the tradeoff between speculation accuracy and training overhead is still under exploration.

Recently, a relatively lightweight prediction technique called *all-state lookback-2* [19]–[21] has been utilized in FSM parallelization on GPUs. Basically, it executes FSM transitions with starting from all states over the last two characters of the predecessor chunk, and thus produces a set of end states. The state convergence property ensures that all impossible states are ruled out and the real start state on the current chunk must be contained in the produced end state set. The likelihood of a state to be the real start state is simply determined by its frequency of appearance in the end state set. Since start state prediction is not the focus of this work, we follow this technique and integrate it into *GSpecPal*.

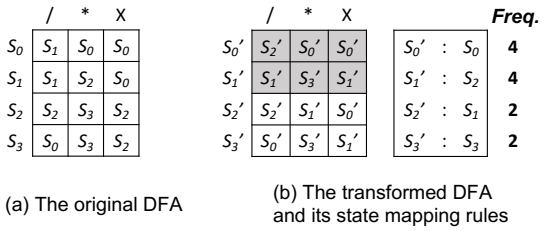

Fig. 4: Applying the Transformation Optimization to a DFA with 4 states. The shadowed area in (b) are considered as hot transitions and will be stored in GPU shared memory.

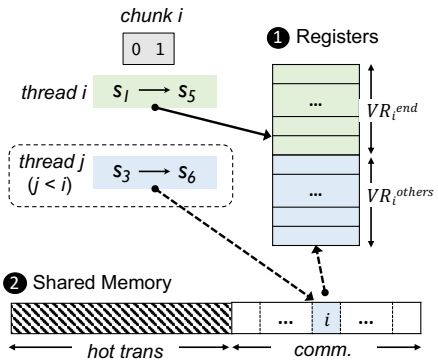

Fig. 5: Illustrating the data layout in GPU shared memory and register files for efficient verification and recovery.

B. State Transition

The major operation in DFA computations is the state transition $state = Table[state][symbol]$. A significant difficulty in FSM processing on GPUs is that the memory access pattern on the transition table is data-dependent and thus unpredictable. For current GPU architectures, the memory needed for storing FSM transition table is usually larger than the shared memory size. For instance, in NVIDIA GeForceRTX 3090, the available shared memory is no more than 100KB per SM, which cannot hold the entire transition table of an FSM with thousands of states. But storing the transition table in global memory may make the transition table lookup extremely expensive. To reduce excessive global memory accesses, existing work [18], [19] keeps a copy of a partial transition table (the *hot* portion) into shared memory before the execution starts, and expect most of the transition can be completed by accessing the copy.

The implementation of state transition in *GSpecPal* is similar to the one in *PM* [19]. For allocating hot state transitions in shared memory, an offline profiling is applied to count the frequency of each state in the original transition table. Then transitions activated by the states with the highest frequencies are promoted to shared memory until there is no more space. Note that once the allocation is done, it won't change during the FSM processing. In each step of state transitions, we need to examine whether the current transition has been cached – if it is, a shared memory access is enough, otherwise, a global memory access is needed. To efficiently perform the examination, *PM* introduces a hash table based method. More specifically, a hash table *Hots* is also constructed and put into shared memory. In every state transition, a hash table access $Hots[hash(state)]$ (with a constant time complexity) is performed at first, where $hash(state)$ is the hash function.

Frequency-Based DFA Transformation. The above hash table based approach in fact introduces one extra shared memory access (to the hash table) and one more computation (in the hash function) in each step. To avoid extra overheads and reduce global memory access as much as possible, we propose a new approach to store the hot state transitions in shared memory. The key idea is to transform the FSM transition table according to the state frequency.

More specifically, after ranking the states based on their frequency in the original transition table, we re-layout the table by grouping hot transitions (i.e., the transitions activated by states with higher frequencies) as well as replacing the state IDs with their corresponding ranks. To preserve the semantics of the original FSM, we also maintain a state ID mapping rule. Figure 4 shows an example of applying this transformation optimization to a DFA with 4 states. In this example, we just need to check if the state ID is less than 2 during state transitions, instead of accessing a hash table to determine whether the current state is a *hot* state.

C. Verification and Recovery

After the parallel speculative execution is completed, we invoke the verification and recovery to ensure the correctness. *GSpecPal* adopts the parallel verification approach used in *PM*. Since one-to-one mapping relationship between threads and chunks may be broken during the aggressive speculative recovery, i.e., a recovery task running on chunk i may be executed by thread j ($i \neq j$), a key challenge is about how to load and store the results of speculative recovery efficiently. Figure 5 shows how *GSpecPal* organizes data in GPU shared memory and register files for efficient verification and recovery. Here both threads i and j execute recovery tasks on chunk i and thread i is a rear thread. Because thread i keeps checking whether is a match for the end state from the predecessor until chunk i is verified, we build up VR_i on registers which can be efficiently accessed by thread i to store the speculative execution or recovery records on chunk i (as shown in Figure 5 ❶). Records in VR_i come from two sources. Different from VR_i^{end}, which stores records directly generated by thread i, VR_i^{others} maintains the speculative recovery records from other threads. Since registers in thread i are private to other threads, we use shared memory (as shown in Figure 5 ❷) for communication: thread j first stores recovery results to shared memory, then thread i loads those results back to registers for future verification. In fact, the number of registers used for VR_i^{others} affects the performance of speculative parallelization: if using few registers, we cannot hold all recovery results from other threads; but if using a large number of registers, the frequent load, store and runtime

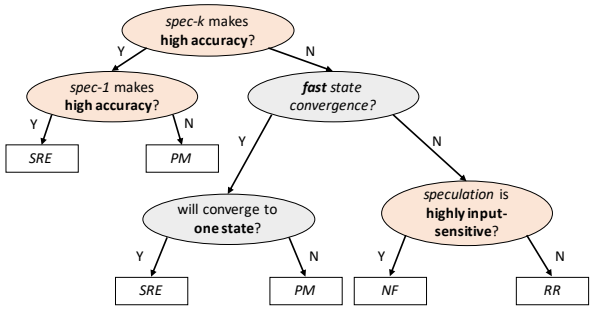

Fig. 6: Decision tree used for scheme selection.

checking may downgrade the benefits from speculative recovery. In Section V, we will examine this with experiments.

D. Parallel Scheme Selection

Guided by the analysis in Section III-C, we build up a decision tree as the scheme selector in *GSpecPal*, as shown in Figure 6. It briefly summarizes the two key factors used for selecting the parallel scheme on GPUs, i.e., the quality of speculation (the orange nodes) and FSM convergence property (the gray nodes). However, considering the cost of precisely capturing speculation characteristics and FSM properties, the decision tree used in *GSpecPal* is in coarse-grained level and it is a simplified version of the model introduced in [21]. Basically, for a given FSM, we run the predictor on a piece of training input and collect the speculation accuracy. To confirm whether the speculation is highly input-sensitive, we examine the similarity of speculation results over different portions of the training input. We also perform a lightweight state convergence profiling, by counting the number of unique states after running 10 steps of transitions starting from all states.

V. Evaluation

In this section, we evaluate the effectiveness of *GSpecPal* on a set of real-world benchmarks from *ANMLZoo* [35] and its improved version *AutomataZoo* [36].

A. Methodology

The evaluation of *GSpecPal* includes all four speculation-based parallel schemes as well as the scheme selection. Note that, *PM* and *SRE* are proposed by prior work [19] and [21], respectively. We integrate them into *GSpecPal* framework, and the implementations are based on our best understanding of their work. And we consider *PM* with *spec-4* as the baseline for comparison.

Experimental Setup. We conduct all experiments on a system consisting of two Intel 3.0GHz Xeon Gold 6248R processors and an Nvidia GeForce RTX 3090 GPU (Ampere architecture) [37]. The GPU device contains 82 streaming multiprocessors, each consisting of 128 cores and 100KB of shared memory, and is equipped with 24GB of global memory. The system runs CentOS Linux release 7.9.2009 (Core). All GPU codes are compiled with CUDA 11.2 toolkit and NVCC V11.1.105 using the highest optimization level. We report the GPU kernel time collected by using CUDA events. The timing results reported are the average of five repetitive runs. We do not report 95% confidence interval of the average as the variation is not significant. In fact, we observed that the experimental results are consistent across different runs, i.e., the variance in execution time was around 1%. For a fair comparison with prior work [19], [21], the I/O time and data structure preparation time are not included since prior work does not focus on optimizing them. These costs are expected to be amortized as inputs or FSMs are repeatedly used and memory technologies like NVLink and unified memory develops.

B. Benchmarks

ANMLZoo and *AutomataZoo* are two diverse automata processing benchmark suites consisting of multiple real-world FSM-based applications from different domains. In this paper, we collect FSMs from 3 applications, including a widely used network intrusion detection system (NIDS) *Snort*, an open source virus-detection tool *ClamAV*, and *PowerEN*, a regular expression benchmark suite originally developed by IBM. Each application contains thousands of Perl-compatible regular expressions, so similar to prior work [24], we compile these regular expressions to DFAs using *RE2* [38], an open-source regular expression library. However, instead of compiling an individual regular expression, each FSM in our evaluation is generated from a disjunction of multiple randomly selected regular expressions. We finally produce 12 FSMs for each application. Table II shows characteristics of these FSMs.

There are twenty 10MB inputs provided for each FSM: the inputs to *Snort* FSMs are network traffic traces collected from a Linux server with *tcpdump*; the inputs to *ClamAV* are concatenations of the binary executables from a Linux machine; and the input trace files to *PowerEN* are released from IBM [39]. A 1MB trace is randomly selected from each group of inputs (i.e., 0.5%) and used for collecting the FSM properties offline. These offline profiling results are reported from the second column to the last column in Table II.

C. Results

Effect of Register Usage. Since the number of registers used for VR_i^{others} (where speculative recovery results from other threads are stored) may greatly affect the performance of verification and recovery, as discussed in Section IV-C, we first investigate its best setting. Figure 7 shows the normalized execution time of applying parallel scheme *RR* on different groups of FSMs with using various number of registers for VR_i^{others}. Two groups of benchmarks (*Snort* and *clamAV*) achieve the best performance when the number of registers used is 16. When running on FSMs from *PowerEN*, the best number of registers used is 18, but the performance lost in using 16 registers is less than 1%. As we continue to increase the number of registers, the execution time slightly increases on all benchmarks. This implies that the ground truth in a speculation usually appears in the top 16 states in QS_i, which is also observed in [23]. In the following, we empirically use 16 registers for VR_i^{others} in *RR* and *NF*.

TABLE II: Benchmarks

Source	#States		accuracy(*spec-1*)		accuracy(*spec-4*)		#DFAs with *highly input-sensitive spec.*	#uniqStates(10 trans.)		Profiling Time (seconds)
	range	mean	range	mean	range	mean		range	mean	
Snort	[423, 42k]	10k	[0, 100%]	23%	[0, 100%]	38%	3	[1.8, 27.2]	10.7	0.6
ClamAV	[541, 8k]	3k	[0, 100%]	16%	[0, 100%]	39%	5	[2, 24]	9.7	0.6
PowerEN	[109, 1501]	650	[0, 85%]	29%	[0, 85%]	30%	6	[3, 32]	12.3	0.6

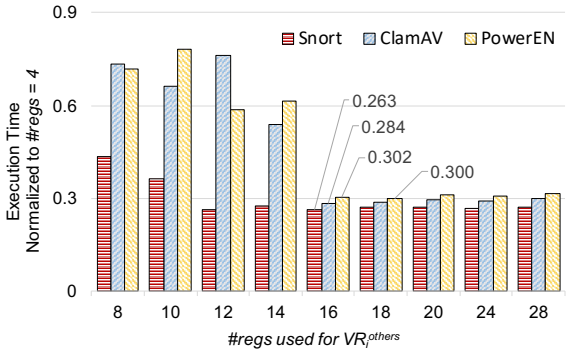

Fig. 7: Performance sensitivity to the number of registers used for VR_i^{others} (the parallel scheme evaluated is *RR*; the number of registers used for VR_i^{end} is 16).

Overall Comparison. Figure 8 reports the speedup of all three speculative recovery schemes and the proposed selector, compared with *PM*, on three benchmark groups. On average, *RR* achieves 6.25× speedup and *NF* achieves 6.76× among all 36 FSMs. However, these two aggressive speculative recovery schemes yield inconsistent speedups across various FSMs, ranging from 0.11× to 18×. The reason is that the parallelization performance depends on the speculation and FSM properties. On some benchmarks (*Snort1-2*, *ClamAV1-3* , and *PowerEn1-2*), *PM* yields the best performance among all parallel schemes in *GSpecPal*. This implies that the enumerative speculation (*spec-k*) used in *PM* is easier to cover the correct start state and thus recovery is generally unnecessary for these benchmarks. It can be observed that on *Snort3-4* and *ClamAV4-5*, *SRE* performs better than other parallel schemes (up to 20×) while *RR* and *NF* also perform well in general. This demonstrates the benefits of utilizing end states from predecessor chunks to trigger the earlier recovery. For the rest benchmarks, the performance improvement from the two heuristics is significant, with an average speedup of 8.11× and 8.87×, respectively.

As for the scheme selector, by feeding the decision tree with collected properties shown in Table II, it can pick the best parallel scheme for 29 out of 36 cases (i.e., 80.6% accuracy). The reasons of selecting the sub-optimal parallel scheme for the rest 7 FSMs are (1) the selection model in fact is built in a coarse-grained level and (2) only a tiny portion of testing inputs is used for offline profiling. Though it cannot perfectly predict the optimal scheme of running a given FSM on GPUs, it won't cause significant performance lost (only 3% on average) comparing to the ideal selection. The selector can obtain a 7.2× speedup on average.

Analysis of Improvement. We observe that the DFA Transformation introduced in Section IV-B can bring a 15% performance improvement on average, but the detailed evaluation is skipped due to space limitation. To have a better understanding of the improvement from speculative recovery, we further examine the speculation accuracy and the number of active threads during recovery, as shown in Table III. Note that the first four columns in Table III report the runtime speculation accuracy, which is defined as the frequency of the matches occurring in verification. It represents the quality of state prediction as well as the benefits from speculative recovery. The extremely high speculation accuracy of *PM* running on *Snort1-2* confirms the benefits of utilizing *spec-k*. For *Snort3-4*, *SRE*, *RR* and *NF* all reach significant high speculation accuracy, which aligns with their similar numbers of active threads during recovery (as shown in the last three columns). On the rest FSMs, *RR* and *NF* boost the accuracy because the number of threads activated during recovery is one to two orders of magnitude higher than the ones in *PM* and *SRE*.

TABLE III: Runtime speculation accuracy and the average number of threads activated during recovery for *Snort* DFAs running in different schemes.

Snort	Accuracy (%)				Average #Active Threads			
	PM	SRE	RR	NF	PM	SRE	RR	NF
1	99.7	0.2	97.4	94.0	1	1	164.9	229.6
2	100.0	0.2	98.8	94.1	0	1	175.8	235.7
3	95.0	99.9	99.9	99.9	1	51	63.0	63.0
4	97.9	99.9	99.9	99.9	1	31	51.0	51.0
5	98.6	8.2	99.9	99.9	1	15	113.0	113.0
6	0.1	0.2	94.7	93.7	1	1	154.1	222.3
7	0.1	0.2	94.6	93.7	1	1	160.0	222.3
8	0.1	0.2	92.4	93.2	1	1	158.1	209.4
9	0.1	0.2	94.2	93.6	1	2	168.1	235.2
10	0.5	0.2	95.3	93.7	1	2	174.2	237.3
11	0.1	0.2	96.3	93.9	1	1	164.4	227.0
12	0.1	0.2	95.6	93.8	1	1	172.5	224.1

We also investigate how the high thread utilization in *RR* and *NF* affects the recovery performance. The recovery execution time per chunk, which is normalized to the one in *SRE*, is reported in Figure 9 (the 12 DFAs are randomly selected from 3 groups). We observe that the costs of recovery per chunk in *RR* and *NF* are generally higher than the ones in *SRE* because of the resource contention. However, we also found that *NF* with a larger number of threads activated during recovery performs more efficiently than *RR* on all selected FSMs. This is because a lot of threads in *NF* run on the same divided chunk during recovery, which reduces thread divergence and improves data locality.

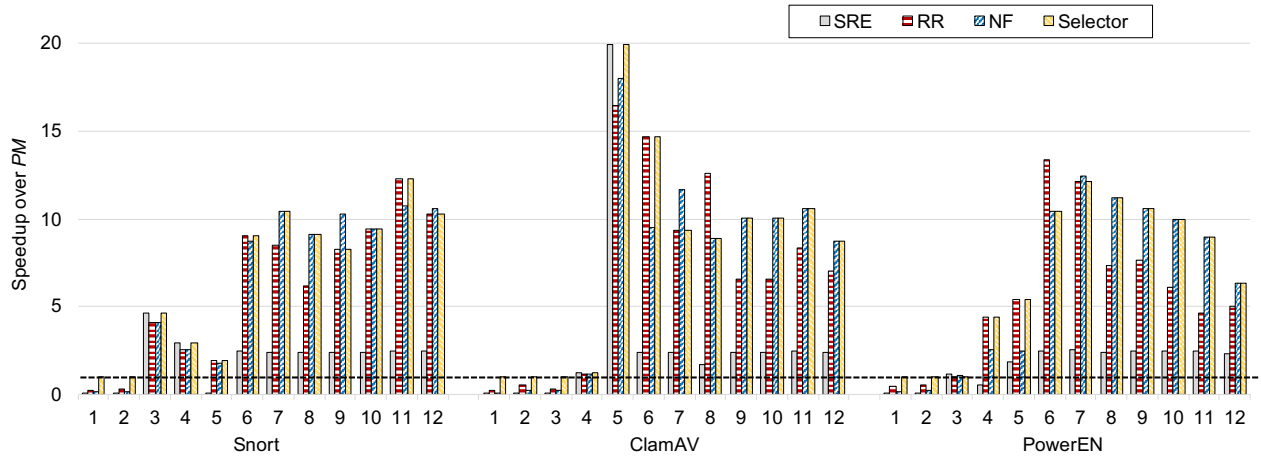

Fig. 8: Performance improvements of *SRE*, *RR*, *NF* and the scheme automatically selected by *GSpecPal*.

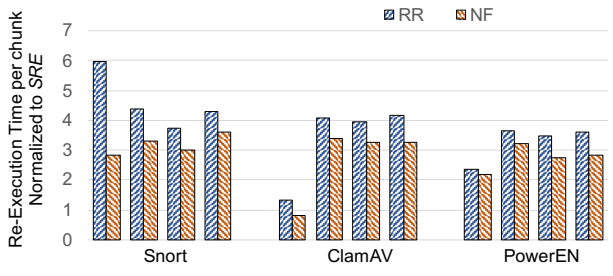

Fig. 9: Effect of higher thread utilization during recovery.

VI. Related Work

This section summarizes the related work into three categories: FSM-based applications, parallelization on DFAs, and GPU accelerations.

FSM-based Applications. FSMs form the backbone of a variety of data processing routines. They are built for deep packet inspection, which requires packet payloads matching against a large set of patterns [7], [8]. FSMs are also used in motifs searching, an important application in bioinformatics, and run across multiple nucleic acid sequences in order to find approximately conserved sub-sequences [3], [4]. Applications in natural language processing, such as part-of-speech tagging, also involve heavy usages of FSMs [5], [6].

Parallelization on DFAs. To break the data dependencies of "embarrassingly sequential" DFA computations, the current efforts mainly fall into two directions: speculative and enumerative parallelization. Zhao *et al.* [26] first propose principled speculation which configures speculations automatically based on pre-profiling. Later in [28], an on-the-fly principled speculation is proposed for reducing the overhead of pre-processing. There are a series of studies [11], [30], [40] on speculative FSM parallelization, which consist of granularity, scalability and energy efficiency analysis. On the other hand, Pan *et al.* [1] start to use enumeration for parallelization. Except for a specific application, Mytkowicz *et al.* [23] propose a general data-parallel approach based on the parallel prefix-sums algorithm. To reduce the overhead from enumeration, Jiang *et al.* [20] then propose speculative enumeration. Except for the *SRE* design, [21] also proposes path fusion to improve the performance of enumerative parallelization.

GPU Accelerations. There has been a significant amount of work on accelerating applications on GPUs. Here we only introduce research closely related to our work. Cascarano *et al.* [16] propose the first GPU-based NFA engine and then Liu *et al.* [7] analyze the bottleneck of NFA computations on GPUs and develop an engine that optimizes the irregular memory access and low thread utilization. Nourian *et al.* [12] further propose compiler supports for optimizing memory access.

VII. Conclusions

This work targets the principal limitation in the existing latency-sensitive designs of speculative FSM parallelization on GPUs. To address the performance bottleneck caused by sequential verification and recovery, this work explores efficient speculative recovery. By breaking the one-to-one mapping between threads and divided chunks, the speculative recovery design can enable parallel verification and recovery, and improve the GPU thread utilization. To leverage the proposed design, this work develops *GSpecPal*, a latency-sensitive FSM parallelization framework which integrates four parallelization schemes and a scheme selector. Experiments show that *GSpecPal* outperforms the state-of-the-art by up to 20×, demonstrating the effectiveness of speculative recovery.

VIII. Acknowledgments

We thank all anonymous reviewers for their constructive comments. This material is based upon work supported by the National Science Foundation (NSF) under 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] A. Sergiyenko, M. Orlova, and O. Molchanov, "Hardware/software co-design for xml-document processing," in *International Conference on Computer Science, Engineering and Education Applications*. Springer, 2020, pp. 373–383.

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

[4] C. Bo, V. Dang, E. Sadredini, and K. Skadron, "Searching for potential grna off-target sites for crispr/cas9 using automata processing across different platforms," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 737–748.

[5] 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.

[6] 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.

[7] C. Liu and J. Wu, "Fast deep packet inspection with a dual finite automata," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 310–321, 2011.

[8] M. Avalle, F. Risso, and R. Sisto, "Scalable algorithms for nfa multi-striding and nfa-based deep packet inspection on gpus," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1704–1717, 2015.

[9] 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.

[10] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy, J. Wadden, M. Stan, and K. Skadron, "An overview of micron's automata processor," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2016, pp. 1–3.

[11] A. Subramaniyan and R. Das, "Parallel automata processor," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 600–612.

[12] M. Nourian, H. Wu, and M. Becchi, "A compiler framework for fixed-topology non-deterministic finite automata on simd platforms," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 507–516.

[13] S. Mittal, "A survey on applications and architectural-optimizations of micron's automata processor," *Journal of Systems Architecture*, vol. 98, pp. 135–164, 2019.

[14] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *International workshop on recent advances in intrusion detection*. Springer, 2008, pp. 116–134.

[15] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2009, pp. 265–283.

[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] 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.

[18] 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.

[19] 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 Programming*, 2020, pp. 160–172.

[20] P. Jiang, Y. Xia, and G. Agrawal, "Combining simd and many/multi-core parallelism for finite-state machines with enumerative speculation," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 3, pp. 1–26, 2020.

[21] 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, pp. 887–901.

[22] J. E. Hopcroft and J. D. Ullman, *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[23] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 529–542.

[24] Y. Zhuo, J. Cheng, Q. Luo, J. Zhai, Y. Wang, Z. Luan, and X. Qian, "Cse: Parallel finite state machines with convergence set enumeration," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 29–41.

[25] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[26] 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.

[27] 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.

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

[29] P. Prabhu, G. Ramalingam, and K. Vaswani, "Safe programmable speculative parallelism," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 50–61.

[30] 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.

[31] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 228–241.

[32] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-centric execution of speculative parallel programs," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[33] V. A. Ying, M. C. Jeffrey, and D. Sanchez, "T4: Compiling sequential code for effective speculative parallelization in hardware," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 159–172.

[34] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Chapter: Scheduling introduction," *Operating Systems: Three Easy Pieces; Arpaci-Dusseau Books: WI, USA*, 2014.

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

[36] 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.

[37] NVIDIA, "Geforce rtx 3090," https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090, 2020.

[38] Re2. [Online]. Available: https://github.com/google/re2

[39] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel, "Hardware acceleration in the ibm poweren processor: Architecture and performance," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 389–400.

[40] 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.