On-the-Fly Principled Speculation for FSM Parallelization

Zhijia Zhao
Computer Science Department
College of William and Mary, USA
zzhao@cs.wm.edu

Xipeng Shen
Computer Science Department
North Carolina State University, USA
xshen5@ncsu.edu

Abstract

Finite State Machine (FSM) is the backbone of an important class of applications in many domains. Its parallelization has been extremely difficult due to inherent strong dependences in the computation. Recently, principled speculation shows good promise to solve the problem. However, the reliance on offline training makes the approach inconvenient to adopt and hard to apply to many practical FSM applications, which often deal with a large variety of inputs different from training inputs.

This work presents an assembly of techniques that completely remove the needs for offline training. The techniques include a set of theoretical results on inherent properties of FSMs, and two newly designed dynamic optimizations for efficient FSM characterization. The new techniques, for the first time, make principle speculation applicable on the fly, and enables swift, automatic configuration of speculative parallelizations to best suit a given FSM and its current input. They eliminate the fundamental barrier for practical adoption of principle speculation for FSM parallelization. Experiments show that the new techniques give significantly higher speedups for some difficult FSM applications in the presence of input changes.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors

General Terms Languages, Performance

Keywords Finite State Machine, FSM, DFA, Speculative Parallelization, Multicore, Online Profiling

1. Introduction

Finite State Machine (FSM) is the backbone of many important applications in various domains. It consumes most time in pattern matching [3], XML validation [23], front end of a compiler [2], compression and decompression [16], model checking, network intrusion detection [13], and many other important applications. The performance of these applications is often critical for improving user experienced responsiveness, supporting large-scaled network traffics, or meeting the quality of service provided by commercial servers. Given that processors are gaining more parallelism rather than higher frequency, it is hence essential to parallelize FSM computations to achieve sustained performance improvement.

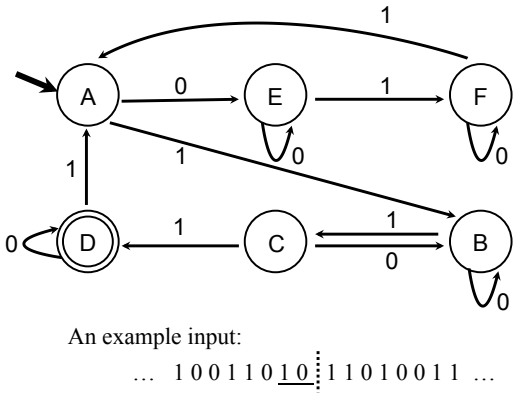

Figure 1. An FSM for string pattern matching and an example input string to it. Each circle on the graph represents an FSM state. State A is the initial state (marked by the extra incoming edge), and state D is an acceptance state. The symbols on the edges indicate conditions for state transitions.

However, FSM applications are challenging to parallelize —so challenging that they are known as "embarrassingly essential" applications [3]. Dependences exist between every two steps of the computations of an FSM application. Consider the string matching example in Figure 1. On a machine with two computing units, a natural way to parallelize the pattern matching is to evenly divide the input string, S, into two segments as illustrated by the broken vertical line in Figure 1, and let the threads process the segments concurrently, one segment per thread. The difficulty is in determining the correct start state to use by the second thread. It should equal the state at which the FSM ends when the first thread finishes

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.

ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694369

processing the first segment. In general, such dependences connect all threads into a dependence chain, preventing concurrent executions of any two threads.

Driven by the importance and challenges of FSM parallelization, recent years have seen an increasing research efforts in enabling high performance parallel FSM executions [22, 31]. A traditional way to parallelize FSM is through prefix-sum parallelization or its variations [18]. A recent study [22] shows that a careful implementation of the method on vector units on modern machines can produce large speedup. However, as that study also acknowledges, the approach is fundamentally subject to scalability issues: The number of threads that conduct useless computations increase linearly with the number of states.

An orthogonal approach is speculative parallelization [14, 23, 31]. The basic idea is to take a guess of the starting state for every thread (other than the first) such that all threads can process their chunks of inputs concurrently. The part of inputs that are processed incorrectly by a thread due to the speculation errors will be reprocessed after the correct ending state of the previous thread (i.e., the correct starting state of this thread) becomes known.

The quality of state speculation is apparently critical for the performance. It depends on a number of design choices, including where to draw hints for speculation, how to use those hints, what state to take as the starting state to execute the FSM speculatively, and so on. A recent work [31] presents a rigorous approach to finding the design that best suits a FSM and its inputs. The approach is called *principled speculation*, which employs a probabilistic make-span model to formulate the profits of a speculation, through which, speculations can be customized to the probabilistic properties of the FSM and hence maximize the overall performance of its parallel executions.

Although the method outperforms previous ad-hoc designs significantly, it is based on offline training. Before using the speculative parallelization, users in most cases have to conduct some tedious, time-consuming training runs of the FSM to collect some probabilistic properties of the states of the FSM and its inputs. The offline training imposes much burden on users. More importantly, it limits the applicability of principled speculation to cases in which the real inputs to the FSM are similar to the training inputs. Otherwise, the offline collected probabilistic properties fail to match with the real inputs, throttling the effectiveness of principled speculation significantly (up to 7x in our experiments shown in Section 6). In practice, many FSM applications need to deal with various kinds of inputs. A compression tool may be used to compress images, videos, texts, and other types of data; a compiler may need to scan or parse programs of various sizes, complexity, and styles; a network intrusion detector may have to go through all kinds of data packages go through the network. Therefore, the lack of cross-input adaptivity forms a fundamental barrier for practical deployment of principled speculation.

This paper addresses the problem by, for the first time, enabling on-the-fly principled speculation for FSM. It proposes a novel static FSM property analysis and two new dynamic optimizations for collecting statistical properties of an FSM. By lowering the cost in collecting expected convergent lengths by orders of magnitude, it makes the principled speculation able to get deployed on the fly, and hence fundamentally removes the input sensitivity issue faced by the state-of-the-art design. With the solution, an FSM application can automatically equip the speculative parallelization with design configurations that best suite the properties of the FSM and its current input. The entire process happens during production runs of an FSM, requiring no offline training or user intervention. Experiments show that the technique reduces the time needed to collect the statistical properties of an FSM by tens to thousands of times, and improve the parallel performance of seven FSMs by up to 7x (1.5x on average).

To our best knowledge, this work is the first that makes on-the-fly principled speculation possible. By removing a fundamental limitation of the basic principled speculation, it eliminates the major barrier for practical adoption of principled speculation, and hence opens new opportunities for maximizing the performance of real-world FSM applications that deal with data in an increasing variety.

2. Background and Problem

In this section, we first present the background of principled parallelization, and then explain the key barrier for its practical deployment.

Principled Speculation At the center of principled speculative parallelization is a make-span formula, which offers the statistical expectation of the make-span of a speculatively parallelized execution of an FSM. Here, *make-span* means end to end execution time, including all speculation overhead and reexecutions upon speculation errors. An important feature of the method is that the make-span formula is based on the statistical properties of the given FSM. So if the statistical properties of an FSM are known, from the formula, the method can analytically figure out how good a design of speculative parallelization is for that FSM. Through the method, the authors of the previous work [31] have shown the feasibility to automatically customize the design of speculative parallelization based on the properties of a given FSM such that its parallel performance can be maximized.

Barrier for Practical Deployment A challenge for practical deployment of the method is in how to obtain the statistical properties of an FSM. These properties include the following:

- *size*: the number of states in the FSM
- *state feasibilities*: the probability of each of its states to get reached in an execution. For instance, if the FSM is at state A in 10% time of an execution, the state feasibility of state A is 10%.
- *character probabilities*: the probability for the next input character to be a particular character while the FSM is at a given state. For instance, if 20% of time, the next character to the FSM is "a" while the current state of the FSM is B, the character probability is $P(a|B) = 0.2$.
- *expected convergent lengths*: Consider two copies of an FSM that start processing the same input string independently from two states, s_i and s_j. If the two FSM copies move into the same state after they finish processing l-character of the input string but not before that, we say that the two states s_i and s_j converge into the same state, and their convergent length is l. For instance, no matter whether the FSM in Figure 1 starts from state C or E, it always moves into the same state A after processing string "11". Therefore, the convergent length between C and E on that input is 2. For a long input string, two states may have different convergent lengths at different sections of the input. The expected convergent length of the two states is the average (or statistical expectation) of all those possible convergent lengths, denoted as $L_M(s_i, s_j)$.

Except *size*, all the other properties are decided by not only the FSM, but also its input. Prior work has shown that *state feasibilities* and *character probabilities* can be easily obtained through lightweight online profiling of the execution of the FSM. But *expected convergent lengths* cannot for most FSMs due to the large cost of collecting convergent lengths.

The large cost comes from three reasons. First, since $L_M(s_i, s_j) = L_M(s_j, s_i)$ and $L_M(s_i, s_i) = 0$, there are $N \cdot (N-1)/2$ pairs of states needed to profile for an FSM with N states. Second, for each pair of states, it usually requires a number of samples to get a reliable average value. Third, to get one sample for a pair of states (s_i, s_j), it takes $(L_M(s_i, s_j) \cdot 2)$ state transitions. Suppose the average number of transitions among all state pairs is L. (If some pairs of states never converge on the given input, a large number is used, denoted as $MAXLEN$.) Let the number of needed samples of convergent lengths per pair of states is $SAMPLE$ ($SAMPLE = 10$ in prior work). The complexity for collecting all expected convergent lengths for an FSM is $O(N^2 \cdot SAMPLE \cdot L)$. Algorithm 1 shows the default algorithm used for profiling the state convergent length.

The actual cost of this default profiling algorithm could range from several seconds to tens of minutes (details in Section 6). As that is comparable or even longer than the parallel execution time of many FSM applications, doing it online is often not affordable. In the prior work, it is affordable on only two out of seven benchmarks [31].

Algorithm 1 Default Convergent Length Profiling

```
1:  for each pair of states (s_i, s_j) do
2:      result[s_i][s_j].sum = 0
3:      result[s_i][s_j].cnt = 0
4:      while result[s_i][s_j].cnt < SAMPLE do
5:          l = 0
6:          s_a = s_i, s_b = s_j
7:          while (s_a ≠ s_b || l < MAXLEN) do
8:              c = read()
9:              s_a = transit(s_a, c), s_b = transit(s_b, c)
10:         end
11:         result[s_i][s_j].sum += l
12:         result[s_i][s_j].cnt++
13:     end
14:     print result[s_i][s_j].sum / SAMPLE
15: end
```

For that reason, convergent length profiling has been mostly done offline on some training runs of an FSM. It is acceptable if the inputs in the production runs of the FSM are similar to the training inputs. But that condition often do not hold for many real-world FSM applications, which often need to process a variety of inputs. As data variety rapidly increases with the trends towards "big data", the issue is a critical barrier for practical usage of the principled speculative parallelization for FSM.

3. Overview of Solutions

The solution developed in this work addresses the issue by completely eliminating the need for offline training. By lowering the cost in collecting expected convergent lengths by orders of magnitude, it makes the principled speculation able to get deployed on the fly, and hence fundamentally removes the input sensitivity issue faced by the state-of-the-art design.

The removal of the high collection cost is through a synergy between a novel static FSM property analysis and two new dynamic optimizations. With this solution, the speculative parallelization runs in this way. At the beginning of a production run, the FSM application calls our static FSM analyzer (through an inserted library function call), which examines the FSM and infers some inherent properties that can hold regardless of what inputs are used. Then, it invokes a lightweight online profiling of the FSM on a small portion of the current input. The profiling, guided by the properties from the static analyzer and facilitated with two new dynamic optimizations, goes orders of magnitude faster than the previous FSM profilings. After obtaining all the statistical properties of the FSM, the FSM application automatically equips the speculative parallelization with the best suit-

able design configurations accordingly, processes the input in parallel, and returns the output to the user.

The next two sections explain the static analysis and the two dynamic optimizations respectively.

4. Static Analysis on FSMs

The objective of the static analysis is to infer some inherent properties of an FSM, which may guide the online FSM profiling. Specifically, our static FSM analyzer infers two properties of an FSM as follows:

- *FSM convergence:* whether there is any pair of states in the FSM that are ever possible to converge.
- *Minimal convergent lengths:* what is the minimal convergent length of each pair of states in the FSM.

The two properties both relate with convergent length collection of an FSM; the first helps avoid convergent length profiling at a coarse grain, while the second helps the avoidance at a finer grain. If the first property says that no two states in the FSM converge, the convergent length collection can be safely skipped entirely; otherwise, if the second property says that the minimal convergent length of a certain pair of states is infinity, the collection can simply avoid profiling on that pair of states.

The static analysis is based on a series of theoretical results developed in this work. This section presents them. For the theoretical nature, the presentation contains some formalism, which we find indispensable for the rigorous inference. However, we try to ease the understanding efforts through examples and graphs. As Deterministic Finite Automaton (DFA) is the most common form of FSM, it has been the focus of recent studies and also this current study on FSM parallelization. The following of this section uses the term DFA rather than FSM to be specific.

4.1 Preliminaries

Before explaining how the static analysis infers the two properties of FSM, we first introduce some concepts and notations to be used in the follow-up explanation. Some concepts have been mentioned in earlier sections but in an informal way; for rigor and completeness, we give their formal definitions here as well.

A *deterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite state set called *states*, Σ is a finite set called the *alphabet*, $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the *set of accept states*. The members of the alphabet are called *symbols*. A *string* over an alphabet is a finite sequence of symbols from that alphabet. We use α, β, or γ to denote strings. The set of all possible strings over Σ is denoted as Σ^*. If α is a string over Σ, the *length* of α, written $|\alpha|$, is the number of symbols that it contains. The string of length zero is called the *empty string*, denoted as ϵ. If α has length n, we can write $\alpha = a_1 a_2 \cdots a_n$, where $a_i \in \Sigma$. If there exists a transition function from state s and symbol a to state s' in δ, we say s transits to s' on symbol a, denoted as $s \xrightarrow{a} s'$.

For convenience, we introduce the following terms and notations.

Definition 1 (State Combination)**.** *A k-state combination of $\mathcal{M}$ is a k-element subset of the states of a DFA, denoted as $c^{(k)}$. The set of all k-state combinations of a DFA is called k-state combination set, denoted as $C_Q^{(k)}$. Clearly, $C_Q^{(1)} = Q$, where Q is the state set of the DFA.*

As $c^{(1)}$ contains only one state, we sometimes use it to also refer to the state it contains.

Definition 2 (State Combination Transition)**.** *Let $c^{(k)}$ be a k-state combination of a DFA $\mathcal{M}$ and $c^{(k')}$ be a k'-state combination of $\mathcal{M}$, we say $c^{(k)}$ transits to $c^{(k')}$ on input symbol a, if and only if $c^{(k')} = \{s' | s \xrightarrow{a} s', \forall s, s \in c^{(k)}\}$, denoted as $c^{(k)} \xrightarrow{a} c^{(k')}$.*

Note that if $c^{(k)} \xrightarrow{a} c^{(k')}$, k' must be no greater than k, guaranteed by the deterministic transitions in the DFA. Actually, k' could be smaller than k in the cases multiple states in $c^{(k)}$ transit to a single state on symbol a.

Definition 3 (Convergence)**.** *Let $c^{(k)}$ be a k-state combination of a DFA, if there is a string α such that $c^{(k)} \xrightarrow{\alpha} c^{(1)}$, then we say $c^{(k)}$ converges on string α, or $c^{(k)}$ is* convergent*. The state in $c^{(1)}$ is called convergence state.*

Since $c^{(1)} \xrightarrow{\epsilon} c^{(1)}$, we consider every $c^{(1)}$ is convergent. According to Definition 3, it is obvious that if a k-state combination $c^{(k)}$ converges on α, then it converges on any string with α as the prefix, that is $c^{(k)} \xrightarrow{\alpha\beta} c^{(1)}$, where β is any string in Σ^*.

Definition 4 (Convergent Length)**.** *Let $c^{(k)}$ be a k-state combination of a DFA that converges on a string α, then the convergent prefix of α is the shortest prefix of α that $c^{(k)}$ can converge on, denoted as α_c. The length of α_c is called the convergent length of $c^{(k)}$ on α, denoted as $L_\alpha(c^{(k)})$.*

It is obvious that $L_\alpha(c^{(k)}) \geq 1$ for any string $\alpha \in \Sigma^*$, when $k > 1$.

Definition 5 (Minimal Convergent Length)**.** *Let $c^{(k)}$ be a k-state combination of a DFA, the minimal convergent length of $c^{(k)}$ is the minimum of convergent lengths on all possible strings in Σ^*, denoted as $L_{min}(c^{(k)})$.*

If $c^{(k)}$ converges on no strings in Σ^*, we say that its minimal convergent length is infinite, denoted as $L_{min}(c^{(k)}) = \infty$.

Example We illustrate the concepts with the DFA in Figure 1. Let's consider the 2-state combination $\{A, B\}$. When reading symbol "1", it transits to another 2-state combination $\{B, C\}$; then reading symbol "0" would let it transit to a 1-state combination $\{B\}$, hence the 2-state combination $\{A, B\}$ is convergent, and it converges on string "10".

In the example, since the 2-state combination $\{A, B\}$ converges on string "10", but does not converge on the prefix "1", its convergent length on string "10" is 2.

The minimal convergent length for the 2-state combination $\{A, B\}$ is 2, since it converges on string "10", but converge on neither "1" nor "0".

4.2 Property I: FSM Convergence

We now present a theorem that allows quick determination of the first property, *FSM convergence*—that is, to decide whether there is any pair of states in the FSM that are ever possible to converge.

We first introduce the following lemma:

Lemma 1. *Let $c^{(k)}$ and $c^{(k')}$ be two state combinations of a DFA, $c^{(k)} \supseteq c^{(k')}$, if there is a string α such that $c^{(k)}$ converges on α, then $c^{(k')}$ must also converge on α.*

The correctness of the lemma is obvious. On the other hand, even if every $c^{(k-1)} \subset c^{(k)}$ is convergent, $c^{(k)}$ may be not.

Based on the lemma, we have the following theorem:

Theorem 1. *Given a DFA $\mathcal{M}$, if and only if none of its states has two or more incoming edges that carry the same symbol, no state combinations of $\mathcal{M}$ are convergent except single-state combinations.*

Proof. It is easy to see that the condition is necessary for the conclusion to hold: If the condition is not met, the sources of the two edges must converge on the common symbol. To prove that the condition is sufficient, we assume that under that condition, there is still a k-state combination $c^{(k)}$ that converges on a string α, where $1 < k \leq |Q|$. That means there is a 1-state combination $c^{(1)}$ such that $c^{(k)} \xrightarrow{\alpha} c^{(1)}$. Then according to Lemma 1, there is a 2-state combination $c^{(2)} \subseteq c^{(k)}$ such that $c^{(2)} \xrightarrow{\alpha} c^{(1)}$. Suppose the two states in $c^{(2)}$ are s_1 and s_2, and the convergence prefix of α is $\alpha_c = a_1 a_2 \cdots a_l$, then we must have $s_1 \xrightarrow{a_1 a_2 \cdots a_{l-1}} s_1' \xrightarrow{a_l} c^1$, $s_2 \xrightarrow{a_1 a_2 \cdots a_{l-1}} s_2' \xrightarrow{a_l} c^1$. According to Definition 4, $s_1' \neq s_2'$. Thus, state c^1 is the state with two transitions from s_1' and s_2' leading to it, both on symbol a_l, which is contradicts the assumption. □

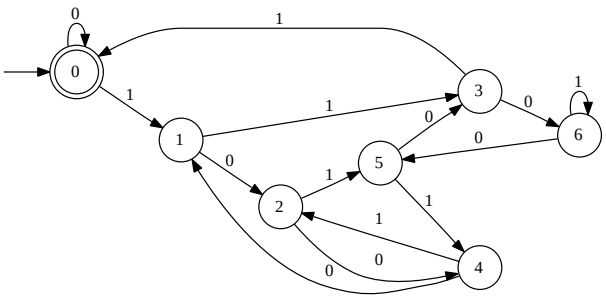

Figure 2. DFA for Testing Divisibility by Seven

Discussion. Theorem 1 tells us that to find out whether states in an FSM are going to ever converge, one only needs to check the incoming edges of every state in the DFA. The time complexity is $O(|Q| \cdot |\Sigma|)$ since the algorithm only needs to examine each edge once. It may significantly cut cost in some existing DFA parallelizations. For example, Figure 2 shows a DFA used for testing whether a binary number is seven-divisible. Previously, for speculative parallelization, people empirically collect the convergent lengths between every pair of its states by running it on many inputs [31]. With Theorem 1, all these collection work can be safely saved as it can immediately tell that no two states of the DFA are convergent.

4.3 Property II: Minimal Convergent Lengths

As aforementioned, directly collecting convergent lengths is time consuming. Even worse, when a combination does not merge, the empirical approach is difficult to tell no matter how long the profiling runs.

In this section, we give an in-depth study on convergent length, especially, on the computations of the minimal convergent lengths of state combinations of a DFA. We prove that it can be computed in polynomial time for a given k by providing a concrete algorithm with $O(k \cdot |Q|^k \cdot |\Sigma|)$ time complexity. The algorithms leverage the relationships of convergent length among different state combinations. They are applicable to all kinds of DFA, including those whose convergent lengths of some or all state combinations are infinity.

The minimal convergent length of a state combination reflects how fast these states could converge. Our algorithm for computing minimal convergent lengths is based on the following lemma.

Lemma 2. *Let $c^{(k)}$ be a k-state combination of a DFA, and $c^{(k')}$ be the state combination that $c^{(k)}$ transits to on symbol a, $a \in \Sigma$, then*

$$L_{min}(c^{(k)}) \leq L_{min}(c^{(k')}) + 1.$$

Now we describe the algorithm for computing the minimal convergent length of every k-state combinations, $L_{min}(c^{(k)})$ for given k, $1 \leq k \leq |Q|$.

Step 1. Check whether the condition of Theorem 1 is met. If so, set all $L_{min}(c^{(k)}) = \infty$ and terminate; otherwise, continue.

Step 2. Construct a graph based on the following rules:

1. For each minimal convergent length of $c^{(i)}$, where $1 \leq i \leq k$, create a node;
2. For each k-state combination $c^{(k)}$, if there is a symbol a, such that $c^{(k)} \xrightarrow{a} c^{(i)}$, where $c^{(i)} \in C_Q^{(i)}$ and $2 \leq i \leq k$, create an edge from $c^{(i)}$ to $c^{(k)}$;
3. Iterate state combination set from $C_Q^{(k-1)}$ to $C_Q^{(2)}$: If an i-state combination $c^{(i)}$, $2 \leq i < k$ has at least

one outgoing edge and there exists symbol a, such that $c^{(i)} \xrightarrow{a} c^{(i')}$, create an edge from $c^{(i')}$ to $c^{(i)}$;

Figure 3 illustrates such a graph. The nodes are aligned by layers, each layer corresponds to a state combination set, with the 1-state combination set on the top. Based on the discussion in Section 4.1, edges may exist among nodes in the same layer or from a higher layer to a lower layer, but not the other way.

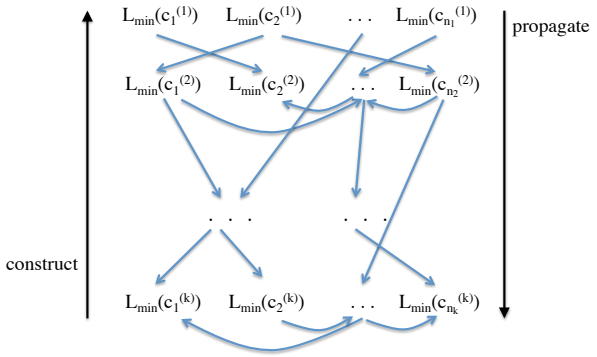

Figure 3. Minimal Convergent Length Propagation Graph

Step 3. Compute the minimal convergent lengths through propagation:

1. Label all nodes in the graph with `UNSET`;
2. Set nodes $L_{min}(c^{(1)}) = 0$, $c^{(1)} \in C_Q^{(1)}$ and label them with `SET`;
3. Propagate the minimal convergent lengths from top layer to bottom layer: If there is an edge from $L_{min}(c^{(i)})$ to $L_{min}(c^{i'})$ and $L_{min}(c^{i'})$ is `UNSET`, then set $L_{min}(c^{i'}) = L_{min}(c^{(i)}) + 1$;
4. Check if any $L_{min}(c^{(k)})$ is `UNSET`, if so, set $L_{min}(c^{(k)}) = \infty$.

With this solution, we can compute the minimal convergent lengths for all k-state combinations in $C_Q^{(k)}$, for any given k, $1 \leq k \leq |Q|$.

Theorem 2. *Let $c^{(k)}$ be a k-state combination in a DFA, if the minimal convergent length of $c^{(k)}$ is finite, then it is bounded by $\sum_{i=2}^{k} |C_Q^{(i)}|$.*

Proof. On its convergent prefix, $c^{(k)}$ cannot transit to a combination more than once. Otherwise, removing the part of the prefix between the two visits to the state would still make $c^{(k)}$ converge. There are only $\sum_{i=1}^{k} |C_Q^{(i)}|$ unique combinations a transit could reach, and for $c^{(k)}$ to converge, it only needs to transit to one c^1. Therefore, the length of its convergent prefix can be no greater than $\sum_{i=2}^{k} |C_Q^{(i)}|$. □

Corollary 1. *Given a DFA, if the minimal convergent length of all the states, that is, $c^{|Q|}$ is finite, then it is bounded by $2^{|Q|} - |Q| - 1$.*

Theorem 3. *The time complexity of the minimal convergent length algorithm is $O(k \cdot |Q|^k \cdot |\Sigma|)$.*

Proof. The complexity mainly comes from two parts: graph construction in *Step 2* and minimal convergent length propagation in *Step 3*. The number of nodes in the graph is $\sum_{i=2}^{k} |C_Q^{(i)}|$, which is $O(|Q|^k)$. Each node has at most $|\Sigma|$ edges, with each corresponding to one unique input character. The construction of an edge coming out of a node $c^{(i)}$ requires i checks to determine the target combination, with each check figuring out what target state one state in $c^{(i)}$ connects to on the input character in the original DFA. So the graph construction time complexity is bounded by $O(k \cdot |Q|^k \cdot |\Sigma|)$. Given that the propagation time cost is bounded by the number of edges, the whole algorithm time complexity is $O(k \cdot |Q|^k \cdot |\Sigma|)$. □

Algorithm 3 in appendix shows the pseudocode of this algorithm.

Discussion. The minimal convergent length of a state combination provides several insights to help reduce the cost and penalty in the speculation-centered DFA parallelization. For example, if one state has very long minimal convergent length with every other state, then it would be quite risky to select it as the predicted state, since a wrong prediction in this case would cause a very large penalty. For the principled speculation in particular, the static inferences of the minimal convergent lengths can help avoid spending time in collecting the empirical convergent length between two states if the minimal convergent length between them is known to be infinity. Since empirical profiling of such cases would not find out that they can never converge, it usually goes through the maximum length of the training inputs before it gives up. The savings by the minimal convergent length inference hence can be substantial.

Inference of the two properties can avoid the profiling on some if not all combinations of states for an FSM in the collection of expected convergent lengths. Next we introduce two dynamic optimizations that take place during the profiling of the remaining state combinations. They further reduce the profiling cost substantially.

5. Dynamic Profiling Optimizations

We introduce two dynamic optimizations to further accelerate the collection of FSM convergent lengths. They are both built on some simple observations. However, when used together, they cut the overhead by up to thousands of times on our experimented FSMs.

To help analyze the benefits of the optimizations, we first introduce a metric, called *transition reduction ratio*.

Definition 6. Transition Reduction Ratio (TRR) *is the ratio of state transitions between the optimized profiling and the default profiling.*

Apparently, the inverse of TRR reflects the speedup that the optimization can bring: *Speedup = 1/TRR*.

5.1 Optimization I: IR Reuse

The first method is called *Intermediate Results Reuse*, or *IR Reuse* for short. The basic idea is simple: when profiling the convergent length between a pair of states s_i and s_j on training input I_{train} starting at position 0, the intermediate transition states could be also considered as the state pairs on the same training input, but with different starting positions. For instance, suppose that the FSM in Figure 1, when running on an input "01..." from state pair (A, C), converges after l characters. one can infer a convergent length sample for state pair (E, B) as $l-1$ because the FSM reaches those two states after processing the first input character. In the same vein, we get a sample for (F, D) as $l-2$, and so on. In total, the IR reuse helps produce l samples of convergent lengths for the FSM through the one profiling on a single pair of states.

We show the implementation of the idea together with the second optimizations later in this section.

Complexity Analysis *IR Reuse* optimization helps reduce the S factor in the complexity of the default algorithm. Since the reuse of intermediate state transitions can potentially contribute to the sample collection of some other state pairs, some sample runs for those state pairs could be eliminated. The actual reduction of S depends on the FSM, training input and the profiling order of state pairs.

The TRR of IR reuse can be expressed symbolically as the following:

$$TRR(\text{IR Reuse}) = \frac{O(N^2 \cdot S' \cdot L)}{O(N^2 \cdot SAMPLE \cdot L)} = O(\frac{S'}{SAMPLE}) \quad (1)$$

where S' is the average number of samples after IR reuse is applied. Since the lower bound of the IR reuse optimized profiling is $O(N^2 \cdot L)$, we have $O(1/SAMPLE) \leq TRR(\text{IR reuse}) \leq 1$.

5.2 Optimization II: Early Stop

The second optimization method is called *Early Stop*. As the name implies, it terminates state transitions before state converge actually happen. The rational of this method is based on the conditional correlations among state pairs. Still use the example in Figure 1. When it is profiling state pair (A, C) on an input "01...", it notices that the FSM will reach states E and B respectively after processing the first character. So if the expected convergent length of (E, B) is already known, then it would stop after processing the first character, and infer the convergent length as $L_M(E, B) + 1$. We show the implementation of the idea in the next section.

Complexity Analysis *Early stop* helps reduce the L factor in the complexity of the default profiling algorithm. Since early stop terminates the profiling of a sample before it ends, the averaged number of transitions for one sample (i.e., L) could decrease. Similar to IR reuse, the actual reduction depends on the FSM, training input and the state pair profiling order.

The TRR of early stop can be expressed symbolically as the following:

$$TRR(\text{Early Stop}) = \frac{O(N^2 \cdot SAMPLE \cdot L')}{O(N^2 \cdot SAMPLE \cdot L)} = O(\frac{L'}{L}) \quad (2)$$

where L' is the average number of transitions for profiling one sample when early stop is applied. It is straightforward to get the similar conclusion as IR reuse, that is, $O(1/L) \leq TRR(\text{Early Stop}) \leq 1$, assuming $L' \leq L$.

5.3 Compound Effects

It is important to note that when the two optimizations are used together, they manifest a compound effect, which dramatically magnifies the benefits by the optimizations. On one hand, by reducing the transition length (L) for profiling one sample, early stop also reduces the cost of "reusing" the intermediate results. On the other hand, by reusing the intermediate results, much more samples are generated within a short period, hence more state pairs obtain their convergent length quickly. This greatly increases the chance of early stop happening, thus further reduces the transition length L. Such an interplay results in more and more powerful optimizations as profiling continues.

Algorithm 2 shows how the two optimizations are implemented together. Line 14 and Lines 20 to 25 correspond to the IR reuse optimization; line 9 to line 11 and line 16 to line 17 correspond to the early stop optimization. In actual implementation, an assistance table, `ready[N][N]`, is created to indicate whether a state pair already has enough samples. With it, the checks of `result[`s_a`][`s_b`].cnt` $\geq$ `SAMPLE` could be more a more efficient table lookup. Since both IR reuse and early stop need these checks, and they occurs at every iteration, the benefit could be substantial.

Complexity Analysis In terms of complexity, both the factors S and L can be reduced drastically. The TRR becomes:

$$TRR(\text{IR Reuse + Early Stop}) = O(\frac{S' \cdot L'}{SAMPLE \cdot L}). \quad (3)$$

Thanks to the compound effects, our experiments show that on typical FSMs, $S' \ll SAMPLE$ and $L' \ll L$. As to the TRR range, we have $O(1/SAMPLE \cdot L) \leq TRR(\text{IR Reuse + Early Stop}) \leq 1$.

6. Evaluation

We evaluate our techniques using the benchmark suite used in a recent study [31] but with more inputs added. Table 1 lists their basic information, including the number of states and input sizes for training and testing. They come from a variety of communities: XML processing (*lexing, xval*), Decompression (*huff*), Pattern Searching and Validation(*str1,*

Algorithm 2 IR Reuse + Early Stop

```
1:  for each pair of states (s_i, s_j) do
2:      result[s_i][s_j].sum = 0
3:      result[s_i][s_j].cnt = 0
4:      while result[s_i][s_j].cnt < SAMPLE do
5:          l = 0
6:          s_a = s_i, s_b = s_j
7:          flag = false
8:          while (s_a ≠ s_b || l < MAXLEN) do
9:              /* Early stop transitions */
10:             if result[s_a][s_b].cnt ≥ SAMPLE then
11:                 flag = true, break
12:             c = read()
13:             s_a = transit(s_a, c), s_b = transit(s_b, c)
14:             states_a[l] = s_a, states_b[l] = s_b
15:         end
16:         if flag then
17:             l = result[s_a][s_b].sum/result[s_a][s_b].cnt
18:         result[s_i][s_j].sum += l
19:         result[s_i][s_j].cnt++
20:         /* Reuse intermediate transitions */
21:         for (i = 0; i ¡ l; i++) do
22:             s_a = states_a[i], s_b = states_b[i]
23:             result[s_a][s_b].sum += l - i
24:             result[s_a][s_b].cnt++
25:         end
26:     end
27:     print result[s_i][s_j].sum / SAMPLE
28: end
```

str2, pval) and Mathematics (*div*). They also have a spectrum of complexities, ranging from 3 to more than 700 states. This range covers the sizes of commonly used FSMs that we have examined, which echoes the observations from prior studies [22, 31]. The default benchmark suite comes with two inputs per program, a small one and a large one. The small one is the first 2% of the large one and was used for training in the previous work [31]. To test the capability of the FSM parallelizations in dealing with different inputs, we added one new small input to each program. The inputs were collected from some public sources. For example, the input to pval is a segment randomly extracted from a novel; the input to huff is a 1.6MB pre-encoded text file; the input to lexing is a 1.7MB segment of an XML file. We kept these inputs of a size similar to the default small inputs in the benchmark suite for a direct comparison. Also for the comparison, we use the default large input as the testing input in all our experiments. The new small input is used for the offline training of the previous principled speculative parallelization to expose its sensitivity to inputs. Our on-the-fly method needs no training inputs.

Our implementation is built on the *SpecOpt* library created by Zhao and others [31] in C language. We mainly replaced its profiling component with the new one supported by our static analyzer and dynamic optimizations. The interface remains the same. In this way, the existing applications using *SpecOpt* can transparently upgrade to our new library. GCC 4.8 is used for compiling the library. The optimization level is set to O3. The results are collected on a server equipped with Intel Xeon E5-4650L CPU (8 physical cores).

Table 1. Benchmarks

Name	Description	States	Train Input	Test Input
lexing	XML Lexing	3	1.6MB	76MB
huff	Huffman Decoding	46	1.6MB	209MB
pval	Pattern Validation	28	1.7MB	96MB
str1	String Pattern Search 1	496	1.5MB	70MB
str2	String Pattern Search 2	131	1.5MB	70MB
xval	XML Validation	742	1.7MB	170MB
div	Unary Divisibility	7	1.7MB	97MB

Results: Static Analysis Table 2 shows the results of the static analysis. The second column lists the overall number of state pairs in each of those programs, which ranges from 3 to 274,911. The FSM convergence property analysis swiftly recognizes that no state pairs of *div* converge on any inputs. The minimum convergent length analysis further recognizes two other benchmarks *pval* and *xval* respectively contain 24 and one state pairs that have infinite minimal convergent length. The fourth column of the table lists the range of the minimal convergent lengths of the FSM state pairs that the analysis finds out. The rightmost column reports the time taken by the static analyses. Generally the more state pairs there are, the longer the time is. But it is not absolute; the time also depends on the structure of the FSM. For instance, *xval* has twice as many as the state pairs *str1* has, but the analysis takes only about half of the time. Overall, the static analysis takes negligible overhead for those benchmarks.

Table 2. Static Analysis Results

	total state pairs	pairs w/ infinite len	min converg length	exec. time
lexing	3	0	0–1	$<$ 1ms
huff	1035	0	0–6	1ms
pval	378	24	0–∞	1ms
str1	122760	0	0–42	139ms
str2	8515	0	0–44	19ms
xval	274911	1	0–∞	62ms
div	21	21	∞–∞	$<$ 1ms

Results: Dynamic Optimizations Table 3 shows the FSM profiling cost reduction by the proposed dynamic optimizations. The profiling times in the "optimized" column are the ones when all the proposed dynamic optimizations in Section 5 are applied after the static analysis. Comparing to the default profiling, the speedup goes from tens of times

to thousands of times. The largest speedup is 6381x, shown on benchmark *str1*. The FSM has 496 states that have much more connections among them than many other FSMs in the suite. The many connections offer more chances for both IR reuses and early stops.

Table 3. Profiling Time

	default (s)	optimized (s)	speedup
lexing	0.50	0.016	31X
huff	1.03	0.033	31X
pval	4.54	0.023	195X
str1	1353.5	0.212	6381X
str2	16.99	0.069	247X
xval	148	2.272	65X
div	1.84	0.019	98X

The dynamic optimizations, especially the early stop, use expected convergent lengths in the inference, which could introduce some differences in the profiling results compared to the results produced by the default offline profiling. Table 4 reports the convergent length profiles of the default and optimized profiling respectively. It shows that the results are quite close in most cases. The most significant differences show up on *lexing* and *xval*. The former has only three states and hence is sensitive to changes; the latter has a complicated FSM with the largest number of states but relatively few connections. Even with the differences, the method still shows good help to FSM parallel performance as shown next.

Table 4. Average Convergent Length

	default	optimized
lexing	2.9	5.9
huff	19.2	21.3
pval	17002.5	18466.6
str1	77997.9	89139.2
str2	66973.0	69068.0
xval	699.3	342.4
div	85714.3	85714.3

Results: On-the-fly Speculation With the enhancement from static analysis and dynamic optimization, the profiling costs are greatly reduced, making on-the-fly speculation possible. To examine its potential, we evaluate three method:

- **offline**: Speculation with the default offline profiling. Profiling cost is not counted in the overall time.
- **online (naive)**: Speculation with online profiling but without the static analysis or dynamic optimizations. Profiling cost is included.
- **online (optimized)**: Speculation with online profiling equipped with both static and dynamic optimizations. Profiling cost is included.

Table 5. Speedups over sequential executions

	offline	online (naive)	online (optimized)
lexing	6.47X	4.46X	6.60X
huff	6.76X	4.05X	6.68X
pval	0.94X	1.53X	6.49X
str1	3.29X	0.01X	3.76X
str2	3.79X	0.39X	3.98X
xval	1.52X	0.12X	3.50X
div	1.26X	1.00X	1.26X
geomean	2.68X	0.49X	4.08X

Table 5 reports the speedups produced by the three parallelization methods over the performance of the sequential execution of the programs. The offline profiling-based approach gives good speedups on *lexing* and *huff*. These two programs have been shown before to be the simplest to parallelize; even simple heuristic-based speculation can already work well on them [23, 31]. They feature very short convergent lengths among all state pairs. Hence, even though the offline profiling-based approach does not lead to accurate speculation results, the FSMs still converge to the correct states quickly and enjoys good speedups. But for the other programs, the input sensitivity causes the approach some substantial degrees of loss in the parallelization benefits. Compared to the offline approach, the optimized online approach gives substantially higher speedups on four out of the five challenging benchmarks. Overall, on average it brings 1.4x extra speedups over the offline method, demonstrating the promise of the on-the-fly speculative parallelization of FSM enabled by the new techniques.

The comparison between the optimized online approach and the naive online approach highlights the importance of the static and dynamic optimizations introduced in this work. Although the naive online profiling method is also able to adapt to input changes, it suffers from the large profiling overhead. With it, most of the parallelized FSMs run even slower than their sequential counterpart.

The program *div* is an extreme case. As none of its state pairs converge, it is not input sensitive in terms of the suitable configurations of the speculation. Our static analysis recognizes this property and avoids any online profiling. It achieves the same 1.26x speedup as the offline approach does. In contrast, without such analysis, the profiling overhead of the naive online approach completely cancels the parallelization benefit.

7. Related Work

There have been many efforts spent on program parallelization. The efforts are from various angles, from language design (e.g. Cilk [11], X10 [7]), to hardware support (e.g., TLS [12, 27]) and programming models (e.g., STM [1, 6]). In section, we focus the discussion on the studies that closely relate with FSM and software speculation.

Parallelization of FSM To parallelize FSM, a traditional way is through prefix-sum parallelization or its variations [18]. A recent study [22] shows that a careful implementation of the method on vector units on modern machines can produce large speedup. The approach is however subject to large FSMs as the number of threads that conduct useless computations increase linearly with the number of states.

There have been some studies on parallelizing some specific FSM applications. An example is the work by Jones and others on parallelizing a browser's front-end [14]. They introduce the concept of lookback (called overlap) for enhancing speculation accuracy. Other examples include the parallelization of JPEG decoder by Klein and Wiseman [16], hot state prediction in a pattern matching FSM to identify intrusions by Luchaup and others [21], speculative parsing [15], and speculative simulated annealing [30]. There have been many efforts in parallel parsing. They can be roughly classified into two categories. The first tries to decompose the grammar among threads [4, 5] by exploiting some special properties of the target language or parsing algorithm (e.g., LR parsing in Fischer's seminal work [10]). The second tries to decompose the input [20], and can often leverage more parallelism than the first approach. All these prior studies employ simple heuristics for speculation. Zhao and others [31] introduce the concept of principled speculation, which is the first rigorous approach to speculative parallelization. There have been some studies in implementing parallel Non-deterministic Finite Automata (NFA) [32]. Unlike other types of FSM, the non-determinism in NFA inherently exposes a large amount of parallelism and is hence easier to parallelize.

Speculative Parallelization Beyond a specific domain, there are also a number of studies in speculative parallelization. Prabhu and others [23] proposed two new language constructs to simplify programmers' job in using speculation schemes. There are some software frameworks developed to speculatively parallelize programs with dynamic, uncertain parallelism, either at the level of processes [8] or threads [9, 25, 28]. They are mainly based on simple heuristics exposed in program runtime behaviors (e.g., speculation success rate). Llanos and others use probabilities of a dependence violation to guide loop scheduling of randomized incremental algorithms in the context of speculative parallelization [19]. Kulkarni and others have showed the usage of abstraction to find parallelism in some irregular applications [17]. The pre-computation used by Quinones and others for speculative threading [24] shares the spirit with lookback in exploiting some part of the program execution for speculation. They construct no rigorous speculation models, but relies on subset of instructions to resolve dependences.

FSM Static Analysis Finite state machine or finite state automaton, as the abstract machine of computing, has been studied for a long time. It is one of the oldest topics in theoretical computer science, and has formed its own theory, automata theory. Our discussion here focuses on work closely related to this study. One of them is from the system testing community. FSMs are used there for testing and discovering the properties of given systems [26, 29]. An important concept is synchronizing string, which is the input string that can lead a set of states transiting to a common state. The length of the shortest synchronizing string is similar to minimal convergent length. However, research on synchronizing strings emphasizes on finding synchronizing strings, while our static analysis emphasizes the connections with convergent length profiling for speculation, and practical algorithms for leveraging the connections.

8. Conclusion

This paper presents a two-fold solution to remove a key barrier that has been preventing practical deployment of principled speculation for FSM parallelization. The solution is a synergy of a series of theoretical results regarding the inherent properties of FSMs and two dynamic optimizations on effectively reusing state profiling results. Through static analysis, it first examines the FSM and infers its inherent properties on state convergence, with which, the profiling space could be safely pruned. Second, by exploring the convergence correlations among state pairs, it further reduces the cost at runtime via IR reuse and early stop. The synergy and the compound effects of the two dynamic optimizations save up to thousands of times of profiling overhead. Together, they yield the first approach to enabling on-the-fly deployment of principled speculation, which demonstrates substantial improvement in speeding up FSM computations.

Acknowledgments

We thank Weizhen Mao for her feedback to this work. This material is based upon work supported by DOE Early Career Award, and the National Science Foundation grant No. 1320796 and CAREER Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or DOE.

A. Pseudocode of Minimal Convergent Length Algorithm

Algorithm 3 Minimal Convergence Length Computation

```
1: L_min ← −1 /*L_min is N-D min convergence len. matrix*/
2: l ← 0 /*Value of minimal convergence length*/
3: diagonal(L_min) ← l
4: l ++
5: while true do/*Iteratively compute new values*/
6:     converge ← true /*Any new value in last iteration*/
7:     for e in uppertrangular(L_min) do /*e is an N-D vector*/
8:         if L_min(e) = −1 then
9:             for c in Σ do
10:                e --c--> e' /*States transit on character c*/
11:                e'' ← sort(e') /*To ascending order*/
12:                if L_min(e'') = −1 and L_min(e'') < l then
13:                    L_min(e) ← l
14:                    converge ← false
15:                    break for-loop c
16:    if converge = true then
17:        for e in uppertrangular(L_min) do
18:            if L_min(e) = −1 then /*States don't converge*/
19:                L_min(e) = ∞
20:        break while-loop
21: end while
```

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2008.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-18, University of California at Berkele, 2006.

[4] F. Baccelli and T. Fleury. On parsing arithmetic expressions in a multiprocessing environment. *Acta Inf.*, 17:287–310, 1982.

[5] F. Baccelli and P. Mussi. An asynchronous parallel interpreter for arithmetic expressions and its evaluation. *IEEE Trans. Computers*, 35(3):245–256, 1986.

[6] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming langauges. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[8] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior-oriented parallelization. In *PLDI*, 2007.

[9] M. Feng, R. Gupta, and Y. Hu. Spicec: Scalable parallelism via implicit copying and explicit commit. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, 2011.

[10] C. N. Fischer. *On Parsing Context Free Languages in Parallel Environments*. PhD thesis, Cornell University, 1975.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[12] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 1993.

[13] K. Ingham, A. Somayaji, J. Burge, and S. Forrest. Learning dfa representations of http for protecting web applications. *Computer Networks*, 51(5):1239–1255, Apr. 2007.

[14] C. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the web browser. In *HotPar*, 2009.

[15] B. Kaplan. Speculative parsing path. `http://bugzilla.mozilla.org`.

[16] S. Klein and Y. Wiseman. Parallel huffman decoding with applications to jpeg files. *Jounal of Computing*, 46(5), 2003.

[17] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[18] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, Oct. 1980. ISSN 0004-5411.

[19] D. Llanos, D. Orden, and B. Palop. New scheduling strategies for randomized incremental algorithms in the context of speculative parallelization. *IEEE Transactions on Computers*, 2007.

[20] W. Lu, K. Chiu, and Y. Pan. A parallel approach to xml parsing. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID '06, pages 223–230, 2006.

[21] D. Luchaup, R. Smith, C. Estan, and S. Jha. Multi-byte regular expression matching with speculation. In *RAID*, 2009.

[22] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *ASPLOS '14: Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*. ACM Press, 2014.

[23] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2010.

[24] C. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzalez, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI*, 2005.

[25] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multithreaded transactions. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, 2010.

[26] S. Sandberg. Homing and synchronizing sequences. *Model-based testing of reactive systems*, 2005.

[27] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[28] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the International Symposium on Microarchitecture*, 2008.

[29] M. V. Volkov. Synchronizing automata and the ern conjecture. *Lecture Notes in Computer Science*, pages 11–27, 2008.

[30] E. Witte, R. Chamberlain, and M. Franklin. Parallel simulated annealing using speculative computation. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):483–494, 1991.

[31] Z. Zhao, B. Wu, and X. Shen. Challenging the "embarrassingly sequential": Parallelizing finite state machine-based computations through principled speculation. In *ASPLOS '14: Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*. ACM Press, 2014.

[32] 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 *PPoPP '12: Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–140, 2009.