Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-Based Computations through Principled Speculation

Zhijia Zhao Bo Wu Xipeng Shen
College of William and Mary, Williamsburg, VA, USA
{zzhao,bwu,xshen}@cs.wm.edu

Abstract

Finite-State Machine (FSM) applications are important for many domains. But FSM computation is inherently sequential, making such applications notoriously difficult to parallelize. Most prior methods address the problem through speculations on simple heuristics, offering limited applicability and inconsistent speedups.

This paper provides some principled understanding of FSM parallelization, and offers the first disciplined way to exploit application-specific information to inform speculations for parallelization. Through a series of rigorous analysis, it presents a probabilistic model that captures the relations between speculative executions and the properties of the target FSM and its inputs. With the formulation, it proposes two model-based speculation schemes that automatically customize themselves with the suitable configurations to maximize the parallelization benefits. This rigorous treatment yields near-linear speedup on applications that state-of-the-art techniques can barely accelerate.

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

General Terms Languages, Performance

Keywords FSM, Speculative Parallelization, Lookback, DFA, Multicore, Partial Commit

1. Introduction

Parallelization is key to the computing efficiency and scalability of modern applications. In the spectrum of parallelism, at the most challenging end lies the category of Finite-State Machine (FSM) applications, which are also known as "embarrassingly sequential" applications [6].

In these applications, the core computation can be formulated as an abstract machine with a finite number of possible states. Transitions among the states follow some predefined mechanism that can be represented with a state-transition graph. Each node in the graph stands for a state and each transition edge is labeled with the symbol that triggers that transition. Figure 1 (a) shows the state-transition graph for a pattern-matching FSM, along with an example input to it.

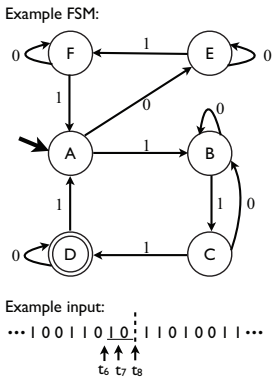
To check whether a string matches the pattern, the FSM starts with the initial state (state A) and processes the input character one after one. At each input character, the FSM moves to a state specified by the state-transition graph. Its arrival at state D indicates the recognition of a string that matches the pattern; such states are called *acceptance states*.

The special difficulty for parallelizing FSM applications is as suggested by its nickname: They are inherently sequential. Dependences exist between every two steps of their computations. Consider the string matching example in Figure 1 (a). 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 (a), and let the threads process the segments concurrently, one segment per thread. The difficulty is in determining the correct state for the second thread to start with. It should equal the state at which the FSM ends when the first thread finishes processing the first segment. Such dependences connect all threads into a dependence chain, preventing concurrent executions of any two threads.

For the extreme difficulty, parallelizing general FSM applications has been lying beyond the reach of existing techniques. The problem, however, is hard to circumvent any longer, partially thanks to the increasing importance of handheld applications; FSM is the backbone of many of them. Take web browsers as an example. FSM-like computations form the core of many activities inside a browser, ranging from lexing, to parsing, syntax-directed translation, image

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541989



(a) Example FSM

State s		A	B	C	D	E	F
$P(s)(\%)$		14.2	28.9	13.9	13.9	14.7	14.4
Expected Merging Length $L_M(x,y)$	A	0	101.00	89.61	59.71	39.12	59.71
	B	101.00	0	16.45	81.99	66.94	81.99
	C	89.61	16.45	0	70.60	51.99	70.60
	D	59.71	81.99	70.60	0	69.91	1.79
	E	39.12	66.94	51.99	69.91	0	69.91
	F	59.71	81.99	70.60	1.79	69.91	0
$p(R=0\|s)$ (%)		51.69	51.85	50.72	50.65	50.10	49.10
$p(R=1\|s)$ (%)		48.31	48.15	49.28	49.35	49.903	50.90

(b) Attributes of the FSM

Figure 1. An FSM for pattern matching and its attributes. In graph (a), each circle 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.

decoding. As prior research shows, even without counting image decoding, such computations could take about 40% of the loading time of many web pages [1]. Besides browsers, most applications on handheld devices use visual or audio media and hence involve media encoding and decoding—which both are typical FSM computations. For its appearance on the critical path of the many applications, improving FSM performance is vital for the response time and hence users experience on handheld devices. At the same time, FSM is essential to many other domains. It consumes most time in pattern matching [6], XML validation [26], front end of a compiler [4], compression and decompression [20], model checking, network intrusion detection [17], and many other important applications. According to Amdahl's Law, without parallelizing FSM operations, it is infeasible for these applications to achieve sustained performance improvement on modern machines, no matter how well other parts of these applications are parallelized.

2. Overview

This section describes the state of the art in parallelizing FSM computations, with an important concept, *lookback*, explained. It then points out their limitations and gives an overview of this work.

State of the Art Among various forms of FSM, Deterministic Finite Automaton (DFA) has been the focus in prior studies, thanks to its broad usage and its capability to approximate other forms of state machines (e.g., Context-Free Grammars with a limited levels of self-embedding recursions [9].) We hence focus our discussion on such FSMs.

A classic approach to parallelizing FSM computations is through variations of the parallel prefix sum algorithm [22]. The idea is to treat each character in the vocabulary of an FSM as a function. FSM computations can then become a series of associative operations of these functions, which can be done in the manner of parallel prefix sum. The method increases the total computation by a factor of $\log N$ and incurs $O(N * |S|)$ space overhead, where N is the length of the input, and $|S|$ is the size of the FSM state set. So the method is beneficial only when the number of processors is greater than $\log N$ and the FSM has a small state set[1].

Recent studies [18, 26] have attempted to address the problem through speculation. As aforementioned, the key difficulty for parallelizing FSM applications is to determine the start state for a thread. The basic idea of these studies is to guess that state. Letting a thread, say T_7, guess the correct FSM state for it to start processing segment S_7 is equivalent to guessing at which state the FSM stops when thread T_6 finishes processing the preceding segment S_6. A random guess is subject to large errors. Previous studies [18, 26] have found it helpful to do a **lookback**—that is, thread T_7 runs the FSM on a number of ending symbols (called a *suffix*) of the preceding segment S_6, and uses the ending state as its speculated start state.

For instance, in Figure 1, a lookback (from state A) by the second thread on the suffix "1 0" stops at state B; the thread will then start processing its segment from state B. Lookback helps speculation by offering some context. The context may not completely determine the actual start state, but is often helpful to avoid some impossible states. In our example, the lookback can safely avoid picking state A as the start state because the FSM structure determines that no state can transit to state A on the end of the suffix, "0".

Lookback-based speculative parallelization has been the central technique of all state-of-the-art FSM parallelizations [18, 26]. As Figure 2 shows, on an 8-core Intel Xeon E5620 system, the approach [18, 26] yields almost ideal speedups on the Huffman decoding "huff" and XML lexing programs "lexing". However, its performance is inconsistent. On the other five programs in Figure 2, it produces

[1] The original paper [22] proposes to represent each function with a boolean matrix, which incurs even a higher time and space complexity.

speedups less than two. One of the programs, *div*, even runs slower than its sequential execution.

The primary reason for the inconsistent performance is the lack of rigor in existing designs of speculation, reflected in multiple aspects. For instance, the length of the suffix to examine by a lookback directly affects the parallelization benefits. A longer suffix exposes more context, but at the same time incurs more overhead. Previous studies [18, 26] select it by simply trying several lengths in profiling runs, while leaving the vast remaining space unexplored. Another example is the state used for starting a lookback. Previous studies always use the initial state of the FSM (state A in Figure 1 (a)) for lookbacks. It could seriously limit lookback benefits. For the example in Figure 1 (a), if the lookback starts from state D rather than A, it would end at the correct state, state E. A further example is that all prior studies have used the ending state of a lookback as the speculated start state for processing the next segment. Although seeming an intuitive decision, does it always maximize the parallelization benefits? If not, is it ever possible to efficiently find a state that does?

Answering these open questions, or more generally, creating a rigorous design requires some comprehensive understanding of the relationship between speculative parallelization and the target FSM and its inputs. It demands models that are able to capture the effects of various speculative parallelizations. Without them, it is hard to determine the design that best fits a given FSM problem.

Meeting these demands involves many challenges. Both FSMs and their inputs are of various size, structure, and complexity. How to characterize them and capture their features that are critical for speculative parallelization? How to formulate the effects of lookback? How to quantify the likelihood for a state to be the true state? How to select the best state after a lookback? And how to formulate the overall benefits of a speculative parallelization with the effects of its different components integrated together? All these questions are important for achieving a principled understanding of speculative parallelization, but they all remain open.

Overview of This Work The goal of this work is to present a rigorous approach to parallelization of FSM computations. Our solution comes from the observation that the likelihoods for a state to be the actual start state at a speculation point are usually non-uniform: Some states in an FSM may be more often to be visited than others, and more importantly, the likelihoods vary from one FSM to another and from one context (or input suffix) to another. *The principle of our approach is to match the design of a speculation scheme with the properties of the target FSM and input.*

To that end, we propose a set of techniques, organized into five boxes in Figure 3. Specifically, we introduce three novel abstractions (Box ①) to effectively characterize the stochastic properties of an FSM. With the abstractions, we build up a probabilistic performance model to quantify the

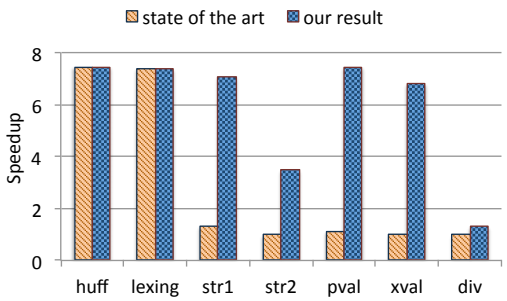


Figure 2. The speedups brought by the state-of-the-art speculation scheme [26] are limited on some complex FSM applications. The results are for 8 threads running on an 8-core machine. Details are shown in Section 7.

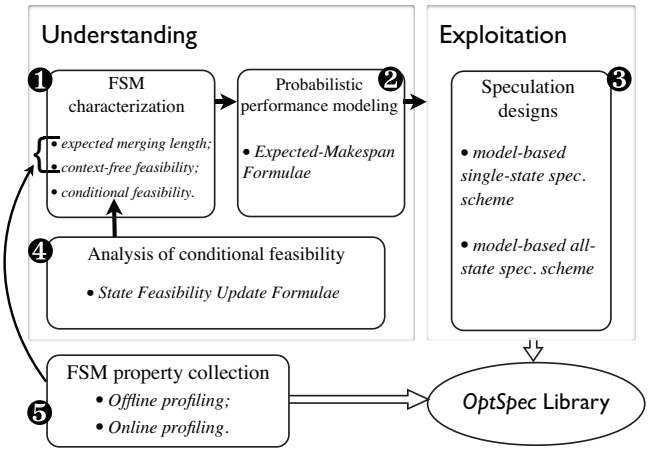


Figure 3. Overview of this work.

expected performance of a speculatively parallel execution (Box ②.) The model unifies the considerations of lookback overhead, misspeculation penalty, and parallelization benefits into a single formulae. Based on the probabilistic performance formulation, we develop two model-based speculation schemes (Box ③), which automatically customize themselves to suit the probabilistic properties of an FSM and its inputs. For practical deployment, we integrate the models into a library named *OptSpec* with a simple API. An important challenge in characterizing an FSM is to capture how its structure influences the effects of a lookback on a speculation, for which, through a formal analysis, we uncover the connections between state transitions and the probability for a speculation to succeed (Box ④.) In addition, as part of the *OptSpec* library construction, we explore the attainment of the FSM properties through both online and offline profiling (Box ⑤.)

The benefits brought by the rigorous treatment are significant. It boosts the parallelization speedups by more than a factor of four over the state of the art for most programs as shown in Figure 2. It yields near optimum performance on five programs, and reverses the slowdown on *div* to a 31%

speedup. The unprecedented level of speedup challenges the common perception of FSM being "embarrassingly sequential", showing that they are inherently sequential *but very parallelizable*.

Contributions This work makes several contributions:

- To the best of our knowledge, this work provides the first principled understanding of speculative parallelization of FSM computations, and gives the first rigorous analysis of it.
- It offers the first probabilistic model of lookback and its influence on speculative parallelization, and produces the first probabilistic performance model for speculative FSM parallelization.
- The two stochastic model-based speculation schemes, for the first time, enable an automatic match between speculative parallelization and the properties of FSM and its inputs.
- It yields near optimal speedups on FSMs that the state-of-the-art techniques can barely accelerate.
- It sheds insights on the importance of adding rigor into heuristic-based speculative parallelizations, and gives new understanding to the potential of parallelizing "embarrassingly sequential" applications.

A Running Example As most of our explanations will draw on the example in Figure 1, we provide some more information about it. The FSM was deliberately made simple for illustration purpose. The table in Figure 1 (b) presents some statistical attributes of the FSM, obtained by running the FSM on a typical input consisting of a string of 0 and 1. The second row ($P(s)$) shows the frequency of each of the states reached in the FSM execution. The second section of the table shows the expected merging length of the states. For instance, the second number in the third row of the table shows that if the FSM processes an input segment in two ways—one starts from state A and the other starts from state B—on average (across various input segments), the two runs don't reach the same state until finishing processing 101 characters. This length section in the table is symmetric because the expected merging length is apparently a symmetric metric. (Section 6 will describe how the lengths can be measured efficiently in practice.) The bottom two rows in the table show the frequencies in which 0 or 1 follows a particular state of the FSM. For instance, the first number in the bottom row shows that during the execution, in 48.31% of time when the FSM is at state A, the next input character is 1. These attributes will be used in our following discussions.

Paper Organizations In the following, we explain each component in Figure 3. We first present the components for enhancing the understanding of FSM properties and their connections with speculative executions (Sections 3 and 4), and then describe the two speculative schemes and the OptSpec library (Sections 5 and 6.) For the nature of rigorous analysis, some formalism and mathematical inferences are hard to avoid in the following description, for which, we create some figures and examples to assist understanding. To make the presentation especially easy to follow, we also include all the important notations in Table 1.

Table 1. Notations

Notation	Description
N	FSM input length
T	number of threads
S, V	the state set and vocabulary of an FSM
$s \xrightarrow{c} r$	state s transits to state r after reading c
$P(s)$	initial feasibility of s
$P^v(s)$	state feasibility of s after a lookback on suffix v
S_k	the feasible state set after a k-long lookback
C_t	cost of a state transition
C_p, C_w	cost of a probability update, and workload
λ	C_p/C_t
β	workload parameter, ie., C_w/C_t
$\omega(l)$	l-long lookback overhead
$\chi(v, s)$	reexecution time when s is speculation state after a lookback on suffix v
$L_M(s, r)$	expected merging length between states s and r
$L_M^v(s)$	expected merging length between s and all possible real states after a lookback on suffix v
ES	the expectation make-span
$ES(l)$	ES when looking back length is l
r_i	the real state at the ith time point
L_i, R_i	the contexts before and after the ith time point

3. Probabilistic Analysis of FSM Speculation

When analyzing the benefits of an FSM speculation scheme, it is important to take a probabilistic perspective: A speculative execution is inherently stochastic. The result of a speculation may be a success or failure, depending on what will happen in the future.

This section presents a probabilistic formulation for modeling the expected benefits of a speculative parallelization of FSM. The formulation is fundamental as it enables a systematic assessment of various designs of speculation, and hence paves the path for creating an effective design.

3.1 Essence of Lookback

As lookback is a key operation in FSM speculation, to build up the performance model, we have to understand the essence of lookback. To that end, we introduce a term *feasibility*:

Definition 1. *For a speculation point, the* feasibility *of a state s is the probability for s to be the correct state at that point.*

Without consideration of contexts, statistically, the feasibility of a state s is the same at every speculation point (although the feasibilities of different states may differ), approximately equaling the frequency for the FSM to visit that

state in its executions. We call these probabilities *initial feasibilities* or *context-free feasibilities*, denoted with $P(s)$, as the second row of Figure 1 (b) illustrates. We call the feasibilities after an input string v *conditional feasibilities*, denoted with $P^v(s)$.

A straightforward way to estimate the conditional feasibility, $P^v(s)$ or equivalently P(real state=s | left string=v), is to count the frequency for s to appear after a string v in profiling runs. But because the value space of v grows exponentially with its length, the approach is generally infeasible.

A key insight exploited in this study is that lookback is essentially a process that tries to use context (i.e., a suffix, v) to improve the knowledge about feasibilities. It implicitly exploits the property that the conditional state feasibilities, to a certain degree, are dictated by the inter-state relations specified by the FSM. For instance, processing a suffix ending with "0" cannot stop at states A or C in Figure 1 (a). By running the FSM on the suffix, lookback essentially employs the state transitions specified by the FSM to help focus the estimation of the conditional probabilities, and prune impossible states for speculation.

3.2 Formulation of Performance Expectation

With the essence of lookback understood, we are ready to build up a performance model for lookback-based speculative parallelization of FSM. We use make-span for performance. The make-span of an execution (either sequential or parallel) is the time elapsed from the start to the end of the execution. The *expected make-span* is the statistical mean of the make-spans of all executions of an application on various inputs of a given length, denoted as ES.

Specifically, our goal in this section is to come up with a set of formulae that can answer the following question: Given an FSM and a speculation scheme to use, what is the expected make-span of the speculative execution on an arbitrary input of a given length? Here, we use $\mathbb{S}$ to represent a speculation scheme, which indicates the lookback length l to use and the state to take as the speculation at each speculation point.

Having such a formulae is fundamental as it allows a systematic examination of the design space of FSM speculations.

The make-span of a thread in a lookback-based speculative execution is the sum of three components: its lookback overhead, the time for processing its own workload, and the reprocessing time if the speculation fails, as shown in Figure 4. We discuss the calculation of each as follows.

1) Lookback Overhead Lookback overhead depends on lookback length L. We denote the overhead with $\omega(L)$. The basic operations during a lookback are the transitions (and associated probability update) from one state to another on the suffix.

2) Workload Processing Time One step in the workload processing by an FSM includes a state transition, and often some additional operations to consume the results produced by the FSM state transition. In an XML-based database constructor, for example, once an object is recognized by its FSM, it is stored into a relational database. We use C_w to represent the average time consumed by such an operation. Sometimes, the operations are buffered until the end of the FSM processing, in which case, C_w equals the time to do the buffering. If we use C_t to represent the time consumed by one state transition, the time taken by one step of the processing is $(C_t + C_w)$. Let N be the length of the entire input, T be the number of threads. An input segment is hence N/T long. The processing time for an input segment is $(C_t + C_w) \cdot N/T$. Both C_t and C_w can be easily measured through profiling. Let β equal C_w/C_t. The processing time for an input segment is $(1 + \beta) \cdot C_t \cdot N/T$. We call β the *workload parameter*.

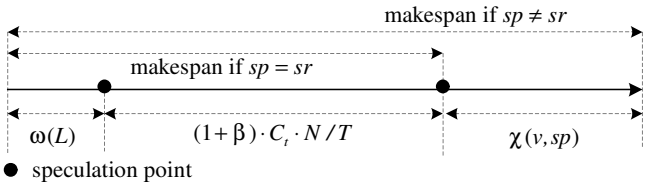


Figure 4. The make-spans of a thread, in cases of correct speculation ($s_p = s_r$) and wrong speculation ($s_p \neq s_r$), where, s_p and s_r are the speculation and real states respectively.

3) Reexecution Time Upon a failed speculation, the data segment needs to be reprocessed from the real state, sr. However, often not the entire data segment needs to be reprocessed because even though the speculation state sp differs from sr, state transitions starting from them tend to converge gradually. For example, when the FSM in Figure 1 sees string "0 0 1 1 0", no matter it starts with state B or C, after processing the first three characters "0 0 1", it always reaches state C. We call the number of state transitions needed before two states converge *the merging length* of the two states, illustrated by the second section in Figure 1 (b).

Typically, reexecution is needed only for the data processed before sp and sr converge. Apparently the merging length depends on input strings and what the real state sr is. Recall that our goal is to compute the statistical expectation of make-span. So it is natural to use the statistical expectation of the merging length across all inputs and all possible true states, denoted as $L_M(sp)$.

Suppose after a lookback on a suffix v, the feasible states set (i.e., the set of states whose feasibilities are positive) is S_v and feasibilities are $\{P^v(s)|s \in S_v\}$. The expected merging length, $L_M^v(sp)$ is computed as follows:

$$L_M^v(sp) = \sum_{s_i \in S_v} L_M(sp, s_i) \cdot P^v(s_i), \qquad (1)$$

where, $L_M(sp, s_i)$ is the statistical expectation of the merging length of sp and s_i on all possible inputs. To understand the formula, one only needs to notice that $L_M(sp, s_i)$ is the reexecution time needed if s_i turns out to be the real state, while $P^v(s_i)$ is the probability for that case to happen.

As the actual reexecution length cannot exceed the length of the segment (N/T), $min\{L_M^v(sp), N/T\}$ is the expected reexecution length for a given speculation sp. Because a reexecution needs to reprocess the workload besides conducting state transitions, the expected reexecution time for a thread is

$$\chi(v, sp) = min\{L_M^v(sp), N/T\} \cdot (1 + \beta) \cdot C_t. \quad (2)$$

Putting All Together The sum of the three components gives the make-span of a thread. Without loss of generality, assume that all threads start at the same time. For the make-span of the entire execution, it may be tempting to think that it equals the maximum of the make-spans of all threads. It is incorrect because all reexecutions have to happen in serial: A thread does not know the real state until all the prior threads have completed their needed reexecutions[2] The correct way to compute the expected make-span of the execution, for a given $\mathbb{S}$, is as follows:

$$ES(\mathbb{S}) = \omega(l) + N/T \cdot (1 + \beta) \cdot C_t + \sum_{i=2}^{T} \chi(v, sp(i)). \quad (3)$$

where, l is the length of the suffix v, and $sp(i)$ is the speculated state of thread i, specified in $\mathbb{S}$. The three components on the right side of the formula respectively correspond to the overhead of one lookback, the time to process one input segment, and the reexecution time of all threads (other than the first as it needs no reexecution). We call this formulae, along with its assistant formulas 1 and 2, the *ES Formula*.

Example We now show how the ES Formula applies to the example DFA in Figure 1. Suppose that our goal here is to compute the expected make-span in the following case: The second thread looks back at 2 characters. If by the end of the lookback, the thread picks state A as its speculated start state, some part of the second chunk of input may have to be reprocessed as A may not be the real start state r. The length of that part is the expected merging length between A and r, denoted as $L_M(r, A)$. The third row of the table in Figure 1 (b) gives all the lengths. The real state r could be any of the seven states, but the examination of the suffix "1 0" helps refine the probabilities. As explained earlier, the refined probabilities are denoted as $P^v(s)$, meaning the probabilities for the real state to equal state s ($s = A, B, \cdots, F$) following suffix v (i.e., $p(r = s|v =$"1 0")). So if we use $L_M^v(A)$ to represent the statistical expectation of the merging length between A and all possible real start states after v, according to Equation 1, $L_M^v(A)$ can be computed as follows:

$L_M^v(A)$=$\sum_{s \in S} P^v(s) L_M(s, A)$

The computation of $P^v(s)$ (i.e., $p(r = s|v =$"1 0")) will be explained in the next section. Here, we list their values: $P^v(s)$=0, 0.42, 0, 0.14, 0.29, 0.15 ($s = A, B, \cdots, F$). The third row of the table in Figure 1(b) gives all the values of $L_M(s, A)$. Together, they give us the follows:

$L_M^v(A)$=0.42*101+0.14*59.71+0.29*39.12+0.15*59.71 =71.

From Formula 2, we know that in this case, the expected reexecution time is

$\chi($"1 0", A$) = min\{L_M^{\text{"10"}}(A), N/T\} \cdot (1 + \beta) \cdot C_t$.

Assuming $N = 400$, $T = 2$, $\beta = 1$, $C_t = 1$, we get $\chi($"1 0", $A) = 142$. The look-back overhead is $2 \cdot C_t = 2$. The time to process the second chunk of input is $N/T \cdot (1 + \beta) \cdot C_t = 200 \cdot 2 \cdot 1 = 400$. So the expected make-span in this case (i.e., when the second thread looks back by two characters and picks A as the speculated start state) is $ES(A) = 2 + 400 + 142 = 544$. In the same way, we can compute the expected make-span of the second thread when it picks any other state as the start state: $ES(s) = 488, 487, 512, 499, 512$ ($s = B, \cdots, F$). State C is hence the best to pick as it minimizes the make-span. In the same vein, we can compute the minimum make-span when some other length of lookback is used. The results can help select the best lookback length (further elaborated in Section 5).

Discussion The ES Formula allows us to compute the expected performance of an arbitrary speculation scheme. It is fundamental for finding a suitable speculation scheme for an FSM. All parameters in the formula—l, $sp(i)$, N, T, β, C_t, $L_M(sp, s_i)$—are given by the FSM or $\mathbb{S}$ or can be measured easily (shown in Section 6) from the FSM, except for the conditional feasibilities $P^v(s_i)$ that appears in Formula 1. We next show how to compute $P^v(s_i)$ from state transitions.

4. Computing Conditional Feasibilities

Recall that conditional feasibility $P^v(s_i)$ is the probability for s_i to be the correct state following a lookback on suffix v. A key insight used in our design is that $P^v(s_i)$ is essentially a refinement of the context-free feasibility, $P(s_i)$, with the influence of the suffix considered. Given that suffixes cast their influence by dictating the FSM state transitions in an execution, the key to computing $P^v(s_i)$ is hence to find out the connections between state transitions and conditional feasibilities.

For convenience, we introduce several notations:

r_i: the real state of the FSM at time point t_i.
L_i: the string processed before the time point t_i.
R_i: the string processed after the time point t_i.
S: the entire set of states in an FSM.

[2] Theoretically speaking, reexecutions can be speculatively parallelized as well. But it adds more complexity.

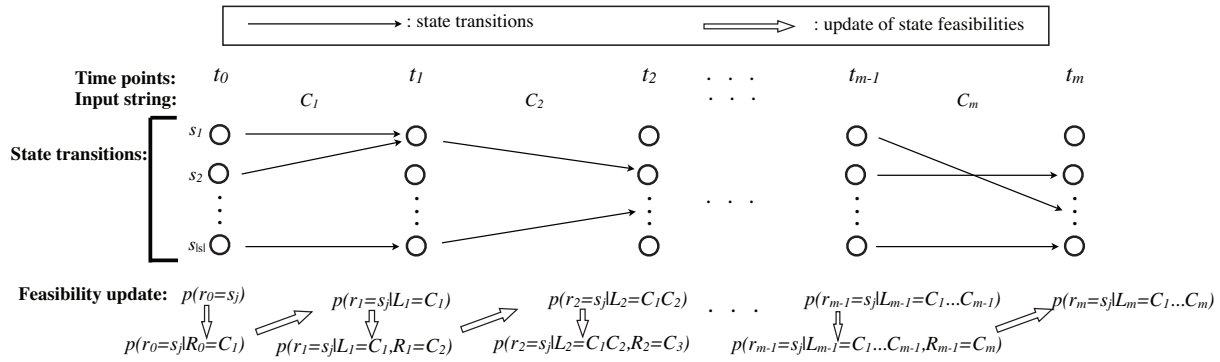


Figure 5. Gradual refinement of conditional feasibilities along with state transitions. The state transitions graph in the middle shows all possible transitions allowed by the FSM on the input characters.

Our analysis centers on the following observation: State transitions essentially lead to an incremental propagation of conditional feasibilities, with the conditions enriched gradually.

We will use Figure 5 to assist the explanation. The graph in the middle of the figure illustrates all possible state transitions upon a string $v = C_1C_2 \cdots C_m$. Our goal is to compute the conditional feasibility of each state after the m stages of state transitions on the string. It is essentially the conditional probability $p(r_m = s_j | L_m = C_1C_2 \cdots C_m)$—that is, the probability for s_j to be the true state at time t_m given that the segment processed before that point equals $C_1C_2 \cdots C_m$ ($j = 1, 2, \cdots, |S|$).

The calculation starts with the context-free feasibilities of all the states, $P(s_j)$, which is the $p(r_0 = s_j)$ shown in the leftmost column in Figure 5. Context-free feasibilities are easily obtainable through profiling (Section 6); they are considered as given. As the input characters are added to the condition of the feasibilities one after one, initial probabilities $p(r_0 = s_j)$ ($j = 1, 2, \cdots, |S|$) are gradually enriched to the conditional feasibilities $p(r_m = s_j | L_m = C_1C_2 \cdots C_m)$.

Intuition Let us examine the first stage of state transitions to gain some intuition. At this stage, we aim at putting the first input character C_1 into the condition of the feasibilities. In another word, we try to compute $p(r_1 = s_j | L_1 = C_1)$ ($j = 1, 2, \cdots, |S|$.) We solve it by decomposing the computation into two steps. The first step uses $p(r_0 = s_j)$ to compute $p(r_0 = s_j | R_0 = C_1)$—that is, the state feasibilities when the upcoming character is C_1 at time t_0. The second step computes $p(r_1 = s_j | L_1 = C_1)$ from $p(r_0 = s_j | R_0 = C_1)$. The first step is a simple application of the standard Bayes' Theorem. It is easy to understand. We describe it later in this section.

We explain the second step here. This step exploits state transitions encoded in the FSM. In the transition graph in the middle of Figure 5, both and only states s_1 and s_2 transit to s_1 from t_0 to t_1 upon the input character C_1. Therefore, for s_1 to be the real state at time t_1, either s_1 or s_2 must be the real state at time t_0. Hence, the feasibility of s_1 at time t_1 with $L_1 = C_1$ as the condition equals the sum of the feasibilities of s_1 and s_2 at time t_0 with C_1 as the upcoming character, that is, $p(r_1 = s_1 | L_1 = C_1) = p(r_0 = s_1 | R_0 = C_1) + p(r_0 = s_2 | R_0 = C_1)$.

These two steps of context enrichment are called *inner-stage update* and *inter-stage update* respectively, corresponding to the downward arrow and right upward arrow from time t_0 to t_1 in the bottom graph of Figure 5. With all $P(r_1 = s_j | L_1 = C_1)$ ($s_j \in S$) computed, we can add the second character C_2 into the condition in the same manner. Continuously doing this leads to the ultimate goal, $p(r_m = s_j | L_m = C_1C_2 \cdots C_m)$.

General Form The formulae in Figure 6 express the two types of feasibility update. We call them *Feasibility Formulae*. The inner-stage formula captures the feasibility changes when the upcoming character C_i is considered, given that all the conditional feasibilities at time t_{i-1}, $p(r_{i-1} = s_j | L_{i-1} = C_{1 \cdots (i-1)})$, have been computed. The first line of the inner-stage formula comes directly from the Bayes' Theorem. The second line comes from a simple inference on the fact that $\sum_{s_j \in S} p(r_{i-1} = s_j | L_{i-1} = C_{1 \cdots (i-1)}, R_{i-1} = C_i) = 1$. The inter-stage formula computes the conditional feasibilities at time t_i based on the results of the inner-stage update. Its rationale is the same as the intuition given by the example in the previous paragraph. The computation results of the inter-stage update are then used by the inner-stage update (as they appear on the righthand side of the inner-stage formula) of the next stage. In this manner, these two kinds of update go hand in hand, leading to the final conditional feasibilities.

As the righthand side of the inner-stage update equation shows, using the formulae needs context-free feasibilities and conditional probabilities $p(R_{i-1} = C_i | r_{i-1} = s_j, L_{i-1} = C_{1 \cdots (i-1)})$ ($s_j \in S$.) Context-free feasibilities are easy to obtain through profiling, but conditional ones are hard: There are too many variations of the condition to profile. However, notice that even though the string before t_{i-1}, L_{i-1}, has influence on the probabilities of which

Inner-stage update of feasibilities:

$$p(r_{i-1}=s_j|L_{i-1}=C_{1\cdots(i-1)},R_{i-1}=C_i) = \frac{p(R_{i-1}=C_i|r_{i-1}=s_j,L_{i-1}=C_{1\cdots(i-1)})\cdot p(r_{i-1}=s_j|L_{i-1}=C_{1\cdots(i-1)})\cdot p(L_{i-1}=C_{1\cdots(i-1)})}{p(L_{i-1}=C_{1\cdots(i-1)},R_{i-1}=C_i)}$$
$$= \frac{p(R_{i-1}=C_i|r_{i-1}=s_j,L_{i-1}=C_{1\cdots(i-1)})\cdot p(r_{i-1}=s_j|L_{i-1}=C_{1\cdots(i-1)})}{\sum_{s\in S} p(R_{i-1}=C_i|r_{i-1}=s,L_{i-1}=C_{1\cdots(i-1)})\cdot p(r_{i-1}=s|L_{i-1}=C_{1\cdots(i-1)})}$$

Inter-stage update of feasibilities:

$$p(r_i=s_j|L_i=L_{i-1}C_i) = \sum_{\substack{s\in S\\ s\xrightarrow{C_i}s_j}} p(r_{i-1}=s|L_{i-1}=C_{1\cdots(i-1)},R_{i-1}=C_i).$$

Figure 6. Formulae for inner-stage and inter-stage update of state feasibilities, where, $C_{1\cdots(i-1)}$ stands for $C_1C_2\cdots C_{i-1}$, and $\{s|s\in S; s\xrightarrow{C_i} s_j\}$ contains all the states that can transit to s_j on input character C_i from time t_{i-1} to t_i.

character to appear next, the influence is largely throttled when the real state at t_{i-1}, r_{i-1}, is given: As a result of L_{i-1}, r_{i-1} already captures most of its influence. Therefore, $p(R_{i-1}=C_i|r_{i-1}=s_j)$ is used as a replacement of $p(R_{i-1}=C_i|r_{i-1}=s_j,L_{i-1}=C_{1\cdots(i-1)})$. The probability $p(R_i=C|r_{i-1}=s_j)$ ($C\in V$; V is the FSM vocabulary) can be obtained through profiling as Section 6 will show. With that replacement, the inner-stage update of $p(r_{i-1}=s_j|L_{i-1}=C_{1\cdots(i-1)},R_{i-1}=C_i)$ becomes

$$\frac{p(R_{i-1}=C_i|r_{i-1}=s_j)\cdot p(r_{i-1}=s_j|L_{i-1}=C_{1\cdots(i-1)})}{\sum_{s\in S} p(R_{i-1}=C_i|r_{i-1}=s)\cdot p(r_{i-1}=s|L_{i-1}=C_{1\cdots(i-1)})}. \quad (4)$$

Example We now show how the formulae can be used to compute the conditional feasibility of $p(r=B|v=$"1 0") for the example FSM in Figure 1. We decompose the computation into four steps so that the inner-stage and inter-stage updates of the probabilities can be seen clearly.

Step 1: The calculation starts with using initial probabilities $p(r=s)$ ($s=A,B,\cdots,F$) to compute the conditional probabilities when the upcoming character is "1"—that is, $p(r=s|R=$"1"). This step corresponds to the point t_6 in Figure 1 (a). When s=A, for instance, the conditional probability is computed as follows:

$$p(r=A|R=\text{"1"}) = \frac{p(R=\text{"1"}|r=A)\cdot p(r=A)}{p(R=\text{"1"})}.$$

The components of the numerator are attributes of the DFA given in the table in Figure 1. The denominator equals $\sum_{s\in S} p(R=\text{"1"}|s)\cdot p(s)$ and hence can also be computed from the table. The results of this step are as follows:

$p(r=s|R=$"1") $= 0.14, 0.28, 0.14, 0.14, 0.15, 0.15$ ($s=A,B,\cdots,F$).

Step 2: We are now ready to compute the conditional probability when the left character is "1": $p(r=s|L=$"1"), which corresponds to the point t_7 in Figure 1 (a). When s=A, for instance, it is computed as follows:

$$p(r=A|L=\text{"1"}) = \sum_{s\in S, s\xrightarrow{1}A} p(r=s|R=\text{"1"})$$
$$= p(r=D|R=\text{"1"})+p(r=F|R=\text{"1"})$$
$$= 0.29.$$

The results from this step are as follows:

$p(r=s|L=$"1") $= 0.29, 0.14, 0.28, 0.14, 0, 0.15$ ($s=A,B,\cdots,F$).

Step 3: We now add the second lookback character into the condition to compute the probabilities $p(r=s|L=$"1", $R=$"0"). This step still corresponds to the point t_7 in Figure 1 (a). When s=A, for instance, the probability is computed as follows based on Formula 4:

$$p(r=A|L=\text{"1"},R=\text{"0"}) = \frac{p(R=\text{"0"}|r=A)\cdot p(r=A|L=\text{"1"})}{\sum_{s\in S} p(R=\text{"0"}|r=s)\cdot p(r=s|L=\text{"1"})}$$

The results from this step are as follows:

$p(r=s|L=$"1", $R=$"0") $=$ 0.295, 0.143, 0.279, 0.139, 0, 0.145 ($s=A,B,\cdots,F$).

Step 4: We are now ready to compute the conditional feasibilities, $p(r=s|L=$"1 0"). When s=A, for instance, it is computed as follows:

$$p(r=A|L=\text{"1 0"}) = \sum_{s\in S, s\xrightarrow{0}A} p(r=s|L=\text{"1"},R=\text{"0"}).$$

As there are no state transiting to A through "0", the probability is 0. When s=B, the conditional feasibility, $p(r=B|L=$"1 0") equals $p(r=B|L=\text{"1"},R=\text{"0"})+p(r=C|L=\text{"1"},R=\text{"0"})=0.422$.

Discussion With the Feasibility Formulae computing the conditional feasibilities, the ES Formulae is finally complete for modeling the expected performance of an FSM on a speculation scheme. It paves the way for a rigorous design of FSM parallelization. With it, some intuitive designs manifest

their problems immediately. For instance, at a speculation point, choosing the state that is most likely to be the true state (i.e., with the largest $P^v(.)$) may not be the best strategy. As Equation 1 shows, the reexecution length, when sp is used for speculation, is a weighted sum of all feasibilities $P^v(s_i)$, with weights equaling the expected merging length, $L_M(sp, s_i)$ ($s_i \in S_l$). Hence, the most plausible state may result in a long reexecution for certain values of $L_M(sp, s_i)$. We next describe how the performance model helps find the best configurations for some speculation schemes.

5. Towards Optimal Designs

In this section, we first discuss the major dimensions in designing speculative parallelization of FSM. We then demonstrate how the described formulations help appropriately configure speculation schemes.

5.1 Design Dimensions

There are three main dimensions in configuring a lookback-based speculation scheme for FSM computations. The first is lookback length, which has some mixed effects: A long lookback may help reduce misspeculation by exploiting more context, but it meanwhile increases lookback overhead.

The second dimension is the set of states for starting a lookback. All previous speculation schemes use the default initial state of the FSM as the start state for lookback, which restraints the lookback benefit. As we will show, a larger start state set tends to yield a better speculation result. The main design questions in this dimension are how large the set should be, and which states the set should contain.

The third design dimension is the selection of lookback results for speculation. When the start state set of a lookback includes more than one state, the FSM executions from each of them will reach a state by the end of the lookback. For example, if we use states C and E as the start states for lookback for the FSM in Figure 1, on a suffix "1 0", the two lookbacks will end up at states D and F respectively. Choosing the best lookback ending state for speculation is the core question in this dimension.

These three dimensions interrelate with one another. For instance, optimal lookback lengths depend on what start states the lookback uses. Designs in all these dimensions together determine the quality of a speculation scheme. But it is difficult to compute the optimal values for all three dimensions at the same time.

In this section, we take the following strategy. We fix the configuration of the second dimension (i.e., the set of start states for lookback), and try to find the appropriate configurations for all other dimensions. In particular, we concentrate on two configurations of the second dimension. One uses the complete state set S as the lookback start state set, the other uses a single state (adaptively determined) as the lookback start state set. Our analysis will demonstrate how the formalization described in the previous sections makes it possible to configure the two speculation schemes effectively. After that, we briefly discuss some other possible configurations of the second dimension.

5.2 Speculation through All-State Lookback

In this scheme, the lookback uses the complete state set as the start state set—that is, during the lookback, each thread other than the first processes a suffix for $|S|$ times, each time starting with a different state. The key design questions are how to determine suitable lookback lengths and how to select a state for speculation. To minimize make-span, the first step in the design is to instantiate the form of make-span given by the ES Formulae (Equations 1,2,3). To do so, we need to calculate lookback overhead $\omega(l)$ and expected reexecution time. We start with $\omega(l)$.

Lookback Overhead $\omega(l)$ Because of the use of all states when a lookback starts, it is easy to see that the total number of state transitions throughout a lookback is $\sum_{k=0}^{l} |S_k|$, where, S_k is the set of feasible states after k stages of state transitions since the start of a lookback. In all-state lookback, there is an update to the feasibility of a state after every state transition in a lookback. Suppose the cost of a state transition is C_t, and the cost of a feasibility update is C_p. Then the overhead of an l-long lookback is

$$\omega(l) = (\sum_{k=0}^{l} |S_k|) \cdot (C_t + C_p). \tag{5}$$

Suppose $C_p = \lambda \cdot C_t$, then we have

$$\omega(l) = (\sum_{k=0}^{l} |S_k|) \cdot (1 + \lambda) \cdot C_t. \tag{6}$$

Both C_t and λ can be easily measured (Section 6.)

Selecting the Speculation State In this all-state lookback scheme, after a lookback, there are typically multiple ending states. Which is selected for speculation determines the expected reexecution time. Our selection algorithm is as follows. With state feasibilities computed using the technique given in Section 4, for a given l, it is easy to use Equation 1 to compute the expected merging length, $L_M^v(sp)$, between every sp and the real state—that is, the expected reexecution length when sp is selected for speculation. The best speculation state can then be selected: It is the one that minimizes $L_M^v(sp)$ (hence the make-span.) We use s^* to represent such a state. Based on Equation 2, the minimal reexecution cost can be computed as follows:

$$\chi(v, s^*) = min\{L_M^v(s^*), N/T\} \cdot (C_t + C_w). \tag{7}$$

Determining Lookback Length The selection of the appropriate lookback length is based on the expectation of make-span (i.e., Equation 3.) The first two components of the make-span are easy to compute. The third component is

the sum of all threads' reexecution overhead, which is unavailable before the execution finishes. It can be approximated by running l-long lookback on a number of typical suffixes and then using Equation 7 to compute the reexecution overhead of each. Let $\overline{\chi(l, s^*)}$ be the average. The expectation of make-span using l-long lookback can be calculated as follows:

$$ES(l) = \omega(l) + N/T \cdot (C_t + C_w) + (T-1) \cdot \overline{\chi(l, s^*)}. \quad (8)$$

A brute-force way to obtain the best lookback length is to use equation 8 to compute the $ES(l)$ for all values of l and then find the minimum. It is unappealing because of the need for collecting $P^l(s)$ for all l and the corresponding overhead.

We use curve fitting to circumvent the problem. Curve fitting is applied to the first and third components of Equation 8 individually. These two are the only components relevant to l in the formula. Fitting them individually is easier than fitting their summation because their summation is often not monotonic while the two components are individually: The lookback overhead $\omega(l)$ increases as l increases, and the expected reexecution cost $(T-1) \cdot \overline{\chi(l, s^*)}$ decreases as l increases. The monotonicity simplifies curve fitting.

The implementation of the curve fitting is in a standard way. Using many suffixes, it first obtains a number of samples of $\omega(l)$ and $(T-1) \cdot \overline{\chi(l, s^*)}$ at some sample values of l ($l = 2^i, i = 0, 1, \cdots, K$). It then uses a set of functions of i to fit the points, and finds the functions producing the least mean square errors for $\omega(l)$ and $(T-1) \cdot \overline{\chi(l, s^*)}$ respectively. The best value of l is then directly computed as the value that minimizes the sum of the two functions. Please refer to our technical report [35] for details.

With the techniques described in this sub-section, we can configure an all-state lookback-based speculation scheme to best meet the probabilistic properties of the FSM and inputs. The implementation, including the needed profiling, is detailed in Section 6.

5.3 Speculation through Single-State Lookback

All prior FSM speculation methods use a single state to start lookback. We show that the probabilistic analysis can also help such single-state schemes.

In the prior schemes [18, 26], the execution by a thread (except the first) starts with a lookback using the default initial state as the start state. After that, it uses the ending state of the lookback as the start state to process the input segment assigned to it. Reprocessing is done upon a misspeculation.

We now show how the scheme can be enhanced through probabilistic models. We start with its make-span. If we use l_b and l_x to represent the lookback length and expected reexecution length respectively, we can rewrite the ES Formula (Equation 3) to

$$ES(l_b) = l_b \cdot (1+\lambda) \cdot C_t + N/T \cdot (1+\beta) \cdot C_t + (T-1) \cdot l_x \cdot (1+\beta) \cdot C_t. \quad (9)$$

Let s'_d represent the start state of a lookback. The ES Formula can be simplified with the following lemma:

Lemma 1. *For single-state speculative executions, if $L^0_M(s'_d) > l_b$, then $l_x = L^0_M(s'_d) - l_b$, otherwise, $l_x = 0$.*

In the lemma, $L^0_M(s'_d)$ is the expected merging length between state s'_d and all other states without looking back. The lemma is proved in our technical report [35].

Putting l_x values from Lemma 1 into Equation 9, the make-span equation is simplified, from which, we get the following theorem:

Theorem 1. *For single-state speculative execution ($T \geq 2$ and $\beta \geq \lambda$), the best lookback length equals $L^0_M(s'_d)$, and the expected make-span equals $L^0_M(s'_d) \cdot (1+\lambda) \cdot C_t + N/T \cdot (1+\beta) \cdot C_t$, where s'_d is the lookback start state.*

The theorem is proved in our technical report [35].

All parameters in the theorem, including $L^0_M(s)$ ($s \in S$), can be obtained through profiling (Section 6.) Based on this theorem, one can easily compute the minimum expected make-span $min\_em(s)$ for each s. The suitable state to use for lookback is just the one whose $min\_em(s)$ is the smallest; its corresponding best lookback length is the overall best choice of lookback length. This gives the configuration that minimizes the expectation of make-span. The theorem has two conditions: $T \geq 2$ and $\beta \geq \lambda$. The first says that there are more than one thread in the FSM computation, and the second says that the time overhead of a probability update is no greater than the average time overhead in workload processing upon a FSM state transition. Both hold in a typical parallel FSM execution.

5.4 Other Configurations

Besides the all-state and single-state lookback schemes, the configurations can also use a subset of S for lookback. The appropriate designs can be obtained in a manner similar to the all-state case. One complexity is that the use of a subset of all states leaves some state transitions unexamined during the lookback. Some approximations may have to be used as remedy when computing conditional state feasibilities. Details are out of the scope of this paper.

6. Implementation and Library Development

The implementations of the two speculation schemes both consist of a profiler and a controller. The controller runs online. By feeding information collected by the profilers to the analytic models described in the previous section, it configures the speculation schemes (e.g., lookback length, start states, selection of speculation states) on the fly to suite the properties of the FSM and inputs.

The profiler collects data needed by the analytic models. The single-state scheme requires the following data: context-free state feasibilities $P(s)$ ($s \in S$), expected merging length between every pair of states $L_M(s, r)$ ($s, r \in S$) (for computing $L^0_M(s)$), overhead parameters λ and β, the number of

threads T, and the length of the input string N. The actual values of all these parameters may vary across FSM as well as input strings. The all-state scheme needs the following additional data: $p(R_i = C|r_{i-1} = s_j)$ ($C \in V$) for inner-stage probabilities update, and the values of $L_M^l(s^*)$ and $\omega(l)$ at 17 sampled values of l (2^k, $k = 0, 1, \cdots, 16$) for finding the suitable lookback length through curve fitting.

The profiler can run either online or offline. We explain the online case first. The online profiler has an adaptive switch. It first collects the values of T, N, S, λ, and β, with negligible overhead. It then uses these values to estimate the time needed to collect the remaining parameters, based on their computational complexities. If the overhead is larger than 10% of the single-thread workload processing time, it falls back to the default simple heuristic-based parallelization. Otherwise, it collects the other parameters as follows.

The collection of all $P(s)$ and $p(R_i = C|r_{i-1} = s_j)$ is through a sequential execution of the FSM on the first 2% input. As the execution is normal rather than speculated, the results are used as part of the final output of the FSM. As side products of the execution, the two kinds of probabilities are estimated based on their occurring frequencies in the execution. The overhead of this step is small.

The step to collect all the expected state merging length, $L_M(s, r)$, is quadratic to $|S|$. It is the most likely cause of the shutdown of the online profiling. The collection runs the FSM on an l-long segment of the input string $|S|$ times, with a different state in S used as the start state each time. Meanwhile, during each process of the string segment, the FSM is reset to the start state after processing every l/5 input symbols. It ensures that the start state is visited by at least 5 times during the process. The state sequence in each run is recorded. After all the $|S|$ runs finish, the comparison between every two sequences gives at least 5 merging lengths of the two corresponding states (say s and r.) The average is used for $L_M(s, r)$. The whole collection process runs in parallel across different states. In our experiments, l is set to 1.6 million or the length of the training input if it is less than 1.6 million. Choosing 1.6 million is because it is greater than 5 times of largest merging length in our measurements. Such a length also ensures that with 99% confidence, the distribution of the characters in the training input is no more than 0.0011 off (in terms of the proportion of each character) that in the testing input [30]. If two states have not merged by 100,000 state transitions, their L_M is set to ∞.

When online profiling is not affordable, offline profiling is always an option. A shortcoming of offline profiling is the input sensitivity issue. But in many uses of FSM applications, the same FSM runs on many similar inputs again and again, for example, an XML validator that deals with a large collection of XML files from similar sources. Furthermore, most input-sensitive parameters (e.g., T, N, $|S|$, $P(s)$, $p(R_i = C|r_{i-1} = s_j)$) can still be collected during runtime as they consume little overhead.

Table 2. Benchmarks

Name	Description	$\|S\|$	$L_M(s,r)$	$P(s)$	L^*	Input
huff	Huffman Decoding	46	4∼25	0∼0.21	23	209MB
lexing	XML Lexing	3	1.0∼6.8	0.06∼0.5	2	76MB
str1	String Pattern Search	496	10∼41K	0∼0.037	362	70MB
str2	String Pattern Search	131	2.98∼∞	0∼0.063	724	70MB
pval	Pattern Validation	28	0∼∞	0∼0.50	0	96MB
xval	XML Validation	742	∞	0∼0.054	229	57MB
div	Unary Divisibility	7	∞	0.143	0	97MB

The space overhead of data collection is $max(|S|^2, |S| \cdot |V|)$, negligible for all the tested FSM executions (V for vocabulary.)

To make the model-based speculative schemes easy to use, we develop a library named *OptSpec* which integrates the all-state and single-state speculative schemes and the online and offline profiling procedures together. It is implemented in C and POSIX Threads, detailed in our technical report [35].

7. Evaluation

We evaluate the proposed techniques on seven FSMs listed in Table 2. They are developed based on the literatures in the web XML processing community (e.g., *lexing* and *xval* [34]), mathematics (e.g., *div* [5]), classical Huffman decoding (*huff* [20]), and string pattern matchings (*str1*, *pval*, and *str2* [2].) FSM computations take majority (mostly over 90%) of their execution time. They are selected for their wide usage in practice, and the spectrum of statistic features and complexities they exhibit as the right columns in Table 2 show. The features shown are those mostly related with the difficulty for speculative parallelization. The third column shows the ranges of state merging lengths (averaged over 100 runs.) The infinities (∞) indicate that some pairs of states in that FSM never converge. The $P(s)$ column shows the ranges of context-free state feasibilities. An FSM with flat distribution of state feasibilities, such as *div* and *xval*, is usually hard to speculate. The L^* column shows the lookback length that our approach finds for the all-state scheme ($\beta = 50$.) The rightmost column shows the size of the testing inputs. We collected inputs mostly from some public sources. For example, the input to pval, str1 and str2 are some novels; the input to huff is a 209MB pre-encoded text file; the input to lexing is a large XML file containing the information on the students in some college. We used the first about 2% of the collected data set as the training input.

Our experiments run on a dual-socket quad-core machine equipped with Intel Xeon E5620 processors. The machine runs Linux 2.6.22 and has GCC 4.4.1 as the compiler with "-O3" optimization flag. All timing results reported are the average of 10 repetitive runs with all runtime cost included.

For each benchmark, we compare the results from the following speculative executions:

heuris: Previous scheme [26].

heuris+: Our simple extension to previous scheme [26].

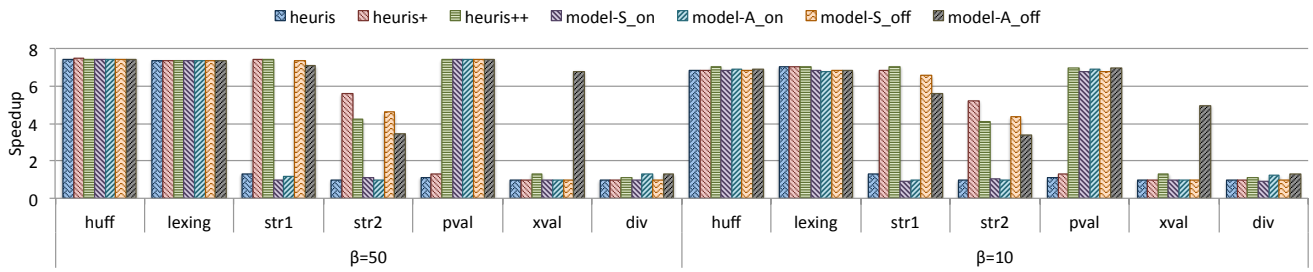


Figure 7. The overall speedup when 8 threads used.

heuris++: Our further extension to previous scheme [26].
model-S_on: Our single-state scheme with online profiling.
model-A_on: Our all-state scheme with online profiling.
model-S_off: Our single-state scheme with offline profiling.
model-A_off: Our all-state scheme with offline profiling.

The *heuris* shows the performance from the state-of-the-art scheme described in recent work [26]. It has lookback and other recent techniques incorporated, but relies on simple heuristics and is not adaptive to FSM properties or input strings. As the previous work offers no systematic solutions for finding the suitable lookback length, we implement the scheme with three lookback lengths, 32, 128, 512, that are used by the previous study [26] and use the best performance for *heuris*.

To examine the value of the insights and techniques described in this work, we develop six extra versions of speculative parallelization, which exhibit a spectrum of complexity and generality.

The *heuris+* version is our simplest extension to *heuris*. It leverages one of the insights in Section 3.2: Upon a failed speculation, often only the first part of the data segment needs to be reprocessed as state transitions starting from the wrong speculation state and the real state may converge. At a failed speculation, the reprocessing of this version stops at the convergence. This partial reprocessing has been used before, but only for some special DFA [20].

The *heuris++* version extends the *heuris+* version by using the state with the largest initial feasibility $P(s)$ as the start state for lookback. Similar to *heuris*, for these two extended versions, we try the three lookback lengths and report the best results.

The other four versions are based on the full model presented in this paper, with either online or offline profiling.

Figure 7 reports the overall speedups compared with the sequential performance when 8 threads are used. Results on 4 threads are similar.

As executions of an FSM may have different workload parameters (β) in different uses of the FSM, we report the results upon two different β values, 10 and 50. Figure 8 reports the influence of input size on the performance of model-A_off with β=10, where, the "medium" size is the same as the testing input in Table 2, and the "small" and "large" sizes are five times smaller and larger than the "medium".

Results The speedups differ between FSMs. The following two properties of an FSM are especially critical:

(1) *Probability distribution*: How biased the state probabilities are determines the difficulty for speculating the right state. If there is an extremely popular state, simple speculations would suffice as long as it picks that popular state. But if the distribution is flat, finding the right state would rely more on effective exploitations of contexts and probabilistic analysis.

(2) *Merging length*: How fast two states merge determines the cost of a misspeculation. If all states merge quickly, a misspeculation causes only a small segment of input to be reprocessed, and hence, a simple method may work fine even if it makes lots of wrong speculations.

In our experiments, *huff* and *lexing* have much skewed probability distributions and short merging lengths. All methods work well on them. As the FSM gets more challenging, those versions start showing disparity in the speedups. The *heuris* shows less than 20% speedups on all remaining five benchmarks, partial reexecution helps *heuris+* achieve 5-7X speedups on *str1* and *str2*, and *heuris++* gives more than 7X speedup on one more FSM, *pval* by exploiting the unconditional feasibilities in lookback. The *model-A_off* version gives significant speedups on all FSMs except for the most challenging one, *div*, demonstrating the generality brought by the principled speculation on the full model. The online model-based methods are beneficial to small FSMs only; the overhead of online profiling prevents them from taking effect on large FSMs.

Overall, the results show that simple capitalization of partial reexecution and state unconditional feasibilities can significantly improve the effectiveness of speculative speculations, making it suffice for most FSMs. But the full model-based method has the greatest generality, and may serve for very complex FSMs.

We further examine each individual program to provide a more detailed analysis.

a) huff and lexing. The program, *huff*, is a Huffman decoding tool. The input is a 209MB pre-encoded text file. The program, *lexing*, is an XML lexing tool, whose FSM

contains only three states. Its testing input is a 76MB XML file. We include the two programs because they are used in prior studies [20, 26]. They turn out to be the only programs, on which, the previous technique shows speedups comparable to the other extended methods. The observed speedups agree with the results reported previously [26]. An examination of the two FSM shows that they have one or two very popular states. As a result, all lookbacks lead to those states, yielding 100% speculation accuracy, and large speedups.

b) str1, str2. These two programs are both for string pattern searching. The pattern for *str1* is $(.^*l.^*i.^*k.^*e)^6)|(.^*a.^*p.^*p.^*l.^*e)^5$; the pattern for *str2* is $((.^+,.^+ \setminus .)^4|(.^+,)^4|(.^+ \setminus .)^4)^3$. The "." in the patterns is for any character, "\." for the period, and superscripts for repetitions. They are selected to represent some complex cases in string pattern matching. The FSM of *str2* has some states that never converge, but most do. The ad hoc lookback in the *heuris* version gives almost entirely wrong speculation states. However, because most states in the FSMs have a short merging length, partial reexecution is sufficient to exploit the parallelism. The online model-based versions are shut down automatically for the required large profiling overhead. The offline model-based versions provide comparable speedups with *heuris+*.

c) pval. The program, *pval*, validates a binary string pattern, $111([01]^*00[01]^*)^{10}111$, where the superscript "10" means that the pattern in the parentheses repeats for 10 times. The speculation accuracy of the *heuris* method drops to 0–27%. In contrast, the all-state methods keep most prediction accuracies higher than 70%. Coupled with the minimization of reexecution time, they give the near linear speedups shown in Figure 7. Similar speedups are obtained by *heuris++*, indicating that exploiting unconditional feasibility and partial reexecution is sufficient for this FSM.

d) xval. The program, *xval*, checks the validity of an XML file. It has a more complex FSM than the previous five, including 742 states to implement an simplified Schema validation algorithm [34]. It does both lexical and syntactic validations for XML files containing up to five levels of nested tags on college personnel dataset. For this complex FSM, our online methods automatically fall back to the basic speculative scheme. Our *method-A_off* method produces 5.7 times of speedup, while none of the other methods gives any noticeable speedup.

On *str1*, *str2*, *pval* and *xval*, the *heuris* method is subject to near zero speculation accuracy, while the probabilistic models boost the accuracy to about 50%. Moreover, as the time breakdown shows (Figure 9), the model-based speculation selects the state that has a small penalty of misspeculation. The majority of the speculative execution is hence still valid (except *div*), yielding the much larger speedups.

e) div. This program checks whether an input binary string is 7 divisible. The FSM is a classical solution to the problem from the mathematic community [5]. Structure-

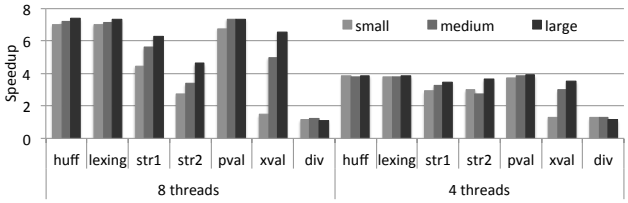


Figure 8. "Model-A off" on different input sizes.

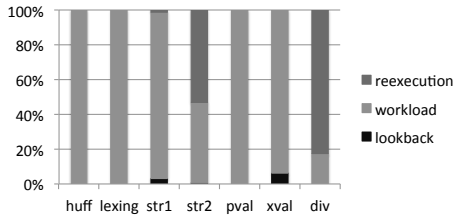


Figure 9. Time breakdown of model-A_off with β=50. Notations: "lookback" for lookback overhead, "workload" for workload processing time, "reexecution" for reexecution time.

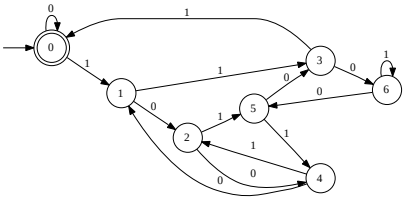


Figure 10. The FSM of *div*.

wise, it is simple, containing only seven states, shown in Figure 10.

However, it is extremely challenging for speculation. The seven states have exactly the same state feasibilities, and any two states never merge regardless of input. Consequently, making speculation is both difficult and risky—a wrong speculation leads to completely useless execution by a thread. All of the four model-based approaches select 0 as the lookback length, achieving 14.4% and 28.8% speculation accuracies and 1.06-1.31X speedups. The heuristic method yields 0 speculation accuracy but pays 1–5% overhead.

The offline profiling of the FSMs took less than a minute for most programs and about 10 minutes for *xval* for its many states. The cost could be further reduced through a more efficient implementation. But as these programs often serve as frequently used utilities for many inputs, the one-time training process is acceptable in many practical scenarios.

Summary of Results The results lead to the following conclusions:

(1) State probability distribution and merging length primarily decide the difficulty for speculative parallelization.

(2) The basic heuristic method works only on FSMs with a highly skewed state distribution. Extending it with partial reexecution and unconditional feasibilities improves it significantly for FSMs with short merging lengths.

(3) The all-state model-based speculation, when used with offline profiling, has the greatest generality, leading to near linear speedups for most FSMs.

(4) The online version of model-based speculations is effective when the FSM is not large. Compared to other methods, it is the only method that can be applied on the fly with no need for offline profiling, which makes it potentially more resilient to input sensitivity issues.

(5) FSMs with uniform state probabilities and infinite merging lengths have little potential for speculative parallelization. However, if such an FSM contains only a few states, each input segment could be processed from all states in parallel. This method however, increases the amount of computation by a factor of $|S|$, and is hence not scalable nor energy efficient.

8. Related Work

Program parallelization has drawn explorations from language design (e.g. Cilk [15], X10 [11]), to hardware support (e.g., TLS [16, 29]) and programming models (e.g., STM [3, 10]). For lack of space, we concentrate on work closely related with FSM and software speculation.

Some studies try to parallelize some specific FSM applications. Jones and others, for instance, focus on a browser's front-end [18]. They introduce lookback (called overlap) for enhancing speculation accuracy, but did not study how to design the scheme to maximize the benefits. Klein and Wiseman [20] have designed a parallel JPEG decoder, which explores parallel Huffman decoding. Luchaup and others [25] have used hot state prediction in a pattern matching FSM to identify intrusions. Other examples include speculative parsing [19] and speculative simulated annealing [32]. These studies shed important insights into parallelizing FSM applications. But they all rely on simple heuristics rather than a systematic exploration of the design space.

There have been some studies in implementing parallel Non-deterministic Finite Automata (NFA) [36]. Unlike other types of FSM, the non-determinism in NFA inherently exposes a large amount of parallelism. 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 [7, 8] by exploiting some special properties of the target language or parsing algorithm (e.g., LR parsing in Fischer's seminal work [14]). The second tries to decompose the input [24], and can often leverage more parallelism than the first approach. They typically use a sequential prescan to partition data at appropriate places. Prescan is sequential and can benefit from the parallelization proposed in this work. The prescan-based data decomposition is often subject to load imbalance because the cutting points can only be the boundaries of certain constructs. Some work tries to allow even data partition by leveraging speculation for parallel parsing [33]. Similar to many prior speculative parallelizations, they are also based on heuristics and can potentially benefit from the rigorous analysis proposed in this work.

There are some efforts on speculatively parallelizing applications beyond a specific domain. Prabhu and others [26] proposed two new language constructs to simplify programmers' job in using speculation schemes to parallelize applications. Some other work has used software speculation to selectively parallelize programs with dynamic, uncertain parallelism, either at the level of processes [12] or threads [13, 28, 31]. 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 [23]. Kulkarni and others have showed the usage of abstraction to find parallelism in some irregular applications [21]. The pre-computation used by Quinones and others for speculative threading [27] 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.

9. Conclusion

This paper introduces formal analysis into speculative parallelization by formulating FSM speculative executions and the connections between the design of speculation schemes and the characteristics of FSM and their inputs. It deepens the understanding to speculative execution of FSM computations with a series of theoretical findings, including the essence and effects of lookback for speculation, the connections between state transitions and conditional feasibilities, and the relationship between partial committing and overall running times. It provides a set of model-based speculation schemes, with suitable configurations automatically determined. Experiments demonstrate that the new techniques outperform the state of the art by a factor of four on most programs, showing that "embarrassingly sequential" applications are in fact quite parallelizable. The insights, especially the importance of rigor and how to achieve it, could potentially benefit speculative parallelization of programs beyond FSM.

Acknowledgment

We thank Ras Bodik, Thomas Dillig, Isil Dillig, and Weizhen Mao for their feedback. This material is based upon work supported by DOE Early Career Award, IBM CAS Fellowship, and the National Science Foundation under Grant No. 0811791, 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 or IBM.

References

[1] R. Bodik. Browsing web 3.0 on 3.0 watts: Why browsers will be parallel and implications for education. Invited talk at The 3rd Workshop on Software Tools for MultiCore Systems, April, 2008.

[2] Regular expression. http://www.regular-expressions.info.

[3] 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.

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

[5] B. Alexeev. Minimal DFA for testing divisibility. *Journal of Computer and System Sciences*, 69(2), 2004.

[6] 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.

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

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

[9] H.C. Bunt and A. Nijholt. *Advances in probabilistic and other parsing technologies*. Kluwer Academic Publishers, 2000. Chapter 12.

[10] 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.

[11] 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.

[12] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior-oriented parallelization. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] B. Kaplan. Speculative parsing patch. In bugzilla.mozilla.org, 2009.

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

[21] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L.P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[22] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.

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

[24] Wei Lu, Kenneth Chiu, and Yinfei 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.

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

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

[27] C.G. 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 *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2005.

[28] 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.

[29] 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.

[30] Steven K. Thompson. Sample size for estimating multinomial proportions. *The American Statistician*, 1987.

[31] 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.

[32] 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.

[33] Zhiqiang Yu, Yu Wu, Qi Zhang and Jianhui Li. A hybrid parallel processing for XML parsing and schema validation. In *Balisage: The Markup Conference 2008*.

[34] Y. Zhang, Y. Pan, and K. Chiu. Speculative p-DFAs for parallel XML parsing. In *Proceedings of the International Conference on High Performance Computing*, 2009.

[35] Z. Zhao, B. Wu, and X. Shen. Probabilistic analysis for optimal speculation of finite-state machine applications. Technical Report WM-CS-2014-01, College of William and Mary, 2014.

[36] 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.