William & Mary
CHARTERED 1693

W&M ScholarWorks

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

2015

Enabling Parallel Execution via Principled Speculation.

Zhijia Zhao
College of William and Mary

Follow this and additional works at: https://scholarworks.wm.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation

Zhao, Zhijia, "Enabling Parallel Execution via Principled Speculation." (2015). *Dissertations, Theses, and Masters Projects.* Paper 1593092101.
https://dx.doi.org/doi:10.21220/m2-qgv5-0x24

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Enabling Parallel Execution via Principled Speculation

Zhijia Zhao

Kaiyuan, Liaoning, China

Master of Science, Harbin Institute of Technology, 2009
Bachelor of Science, Harbin Institute of Technology, 2007

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
August 2015

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Zhijia Zhao

Approved by the Committee, July 2015

Committee Chair
Associate Professor Xipeng Shen, Computer Science
The College of William and Mary

Professor Weizhen Mao, Computer Science
The College of William and Mary

Assistant Professor Kun Sun, Computer Science
The College of William and Mary

Assistant Professor Xu Liu, Computer Science
The College of William and Mary

Dr. Youfeng Wu
Intel

ABSTRACT

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 known as "embarrassingly sequential" applications. As suggested by its nickname: They are inherently sequential, and hence especially difficulty to parallelize.

This dissertation presents three important classes of computations in this category and a series of model-based techniques to overcome their parallelization challenges. They are: 1)Finite-State Machine (FSM)-based computations, which can be formulated as an abstract machine with a finite number of possible states and transitions among the states; 2)Just-in-Time (JIT) compilation, a key component in compilers for managed programming languages, such as Java and JavaScript; and 3)HTML parsing, the step that transforms unstructured byte streams into tree-like structures in web browsers. Though coming from different domains, they share the same parallelization difficulty – carrying prevalent dependencies with their computations: The next state in every FSM transition depends on earlier state(s); Each function to compile by JIT compiler depends on previous executed functions; The handling of each byte or token by an HTML parser depends on what it has observed.

For the extreme difficulty, parallelizing these applications has been lying beyond the reach of existing techniques. This dissertation presents how to overcome the difficulties through a principled approach – the first disciplined way of speculative parallelization. The dissertation additionally covers some other recent progress and exiting opportunities in program parallelization and optimizations. Together, they demonstrate the large potential of the principled approach for advancing the state of the art of code parallelization and optimizations. They represent a new direction to narrow the gap between modern computing hardware and the computing efficiency of software applications.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

My thanks go to my advisor, Dr. Xipeng Shen, without whom this would not be possible. His intelligence, support, patience, and encouragement are throughout my graduate studies, deeply inspiring my future research.

I appreciate the help from Dr. Weizhen Mao, who is such a warm-hearted professor and is always willing to help me every time I turned to her, Dr. Youfeng Wu from Intel, who has been such a wonderful collaborator and provided numerous valuable suggestions on my research, Dr. Michael Bebenita from Mozilla, who mentored me and help cultivate my interests in practical programming, and Dr. Sriram Krishnamoorthy, who offered me the precious opportunity to visit PNNL and greatly inspired me on my future research.

Thanks Prof. Kun Sun and Prof. Xu Liu who later join my dissertation committee for their strong supports.

I want to give my great thanks to my teammates who I have been worked with and who are also my great friends, Bo Wu, Mingzhou Zhou, Yufei Ding, Qi Zhu, Jianhua Sun and Lin Ning.

Above all, thanks to my wife Lifen Zhang whose support and dedication cannot be expressed in words.

To my parents for allowing me to study abroad in pursuit of my dreams;
To my wife Lifen and my son Evan for offering so many joys every day;
To my coming child who will join this wonderful world in this fall!

LIST OF TABLES

LIST OF FIGURES

1 Introduction

1.1 Motivation

The increasing of CPU frequency has reached a limit since 2004 due to "the Power Wall". In the late 1990's and early 2000's, the clock rate was driven by adding more transistors to a smaller chip. Meanwhile, according to the power dissipation relation, the dynamic power consumption increased linearly. When it goes beyond the capacity of inexpensive cooling techniques, a barrier is reached. To overcome this barrier, industry has turned to put several or even many conventional cores into a single die, creating multicore and manycore processors. Following this direction, the hardware will embrace more and more parallelism. However, on the other hand, has the software been ready for running on the increasingly parallel hardware?

The answer depends on the parallelisms that the software can expose. If we consider a line representing the parallelization difficulty spectrum. On one end lie the programs that require little or no effort to be divided into a number of smaller tasks, which can be executed in parallel naturally, such as rendering in computer graphics, matrix multiplication in scientific computing. On the other end lie the programs that are extremely hard to parallelize due to the strong inherent data dependencies through out the executions of these programs. Examples include Huffman decoding, pattern matching, parsing and dynamic compilation.

Without conquering the parallelization challenge, people may have to sacrifice efficiency and turn to other alternatives which may enable better scalability, for example, substituting Huffman coding with another easy-to-parallelize compression algorithm. We believe this is the last option, since it incurs additional costs and suffers long-term low-efficiency. To avoid this, we instead challenge parallelization difficulties and let more software applications achieve better scalability, such that they could better serve the increasing needs for software efficiency.

The key to this challenge is effectively breaking the dependencies that are inherently carried by these computations. Previously, researchers have found that *speculative parallelization* show some promise. As illustrated in Figure 1.1, the basic idea of speculative parallelization is using

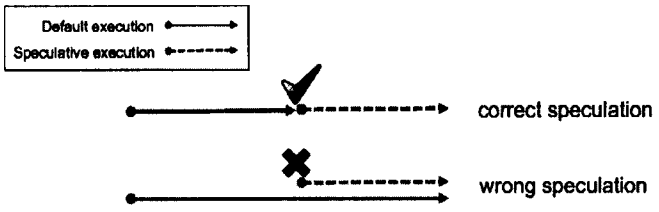


Figure 1.1: Illustration of Speculative Parallelization
With prediction, speculative execution can start from a "future" point. If prediction succeeds, it may yield speedup; Otherwise, it goes back to default sequential execution.

prediction to "guess" the data that will be used, so that certain computations relying on them can be initiated or executed earlier, exposing more computation parallelism. However, the prediction in general turned out to be very hard [70, 98]. A common insight is that it could help the speculation if one draws on some kind of domain knowledge. However, we found that the previous work mainly used some simple ad hoc ways to leverage the domain knowledge, leaving some design dimensions unexplored. Without systematical exploration, they either perform badly, or suffer inconsistent performance gains. Even worse, they might slowdown the applications.

In this research, we introduce *principled speculative parallelization* which adds rigor to the speculative design to realize the full potential of speculative parallelization. More precisely, we give its definition as below:

Definition 1. *Principled speculative parallelization is a software speculative parallelization paradigm that employs rigrous probabilistic models to characterize the inherent relation between speculation and parallelization benefits, hence is able to provide statistically optimal configuration for any given speculative arallelization design.*

To verify this idea, we have looked into a spectrum of computations that play critical roles in today's applications, yet very challenging to parallelize. They are finite state machine (FSM)-based computations, HTML parsing, and Just-in-Time (JIT) Compilation. These computations have been either widely used cross various domains or serving as key components in modern software (e.g., web browsers and runtime compilers). But none of them are easy to parallelize. The former is recognized as a typical "Embarrassingly Sequential" dwarf by the parallel computing community [20]; while the latter two remain sequential since they have been proposed. So far, we haven't seen any systematic explorations to make them run efficiently in parallel. Through the deep study on these three types of challenging cases, we obtained some deep insights and established principled speculative parallelization as a novel and effective paradigm for automatic software parallelizations.

In the following of this dissertation, my research will be grouped into four pieces of work and

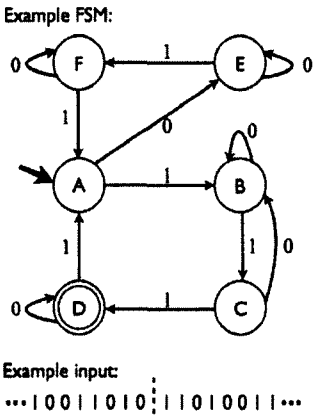


Figure 1.2: Example FSM
Each circle represents an FSM state. State A is the initial state (marked by the extra incoming edge), and state D is an acceptance state. The symbols on the edges indicate conditions for state transitions.

each of them will be first briefly introduced in this section, and detailed in one of the later sections in the same order.

1.2 Principled Speculation for FSM Parallelization

Finite State Machine (FSM) is the backbone of many important applications in various domains. It consumes most time in pattern matching [19], XML validation [90], front end of a compiler [12], compression and decompression [62], model checking, network intrusion detection [54], 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.

However, FSM applications are challenging to parallelize —so challenging that they are known as "embarrassingly essential" applications [19]. Dependences exist between every two steps of the computations of an FSM application. Consider the string matching example, as shown in Figure 1.2. 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, 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 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 [58, 83, 90]. A traditional way to parallelize FSM is through prefix-sum parallelization or its variations [65]. A recent study [83] shows that a careful implementation of the method on vector units on modern machines can produce large speedup.

An orthogonal approach is speculative parallelization [58, 90, 119]. 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. In our work [119], we present 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. As far as we know, this is the first work that employs rigorous model for speculative parallelization.

1.3 On-the-Fly Principled Speculation for FSM Parallelization

Although our principled 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 5.5). 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.

Our research [117] addresses this 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.

1.4 HPar: Enabling Parallel HTML Parsing

Research has long shown that reducing application response time is one of the most important variables for user satisfaction on handheld devices [52, 85]. Multiple studies by industry have echoed the finding in the context of web browsers: Google and Microsoft reported that a 200ms increase in page load latency resulted in "strong negative impacts", and that delays of under 500ms "impact business metrics" [44, 63, 76]. One approach to reducing response time that has drawn lots of recent attentions is parallelization of web browser activities, which range from the creation of web page layout [24, 78], lexing [59], Javascript translation and execution [55], to web page loading [106], XML parsing [73, 115], and web page decomposing [76]. Industry (e.g., Qualcomm and Mozilla) has began designing new browser architectures [34,82] to leverage these results and to exploit task-level parallelism.

However, despite all these efforts, an important part of web browsers, HTML parsing, remains sequential. The second column in Table 1.1 reports the loading time of a number of popular web pages on Firefox 18 on a Nexus 7 tablet. The third column shows the portion taken by HTML parsing, measured through a profiler named *Speed Tracer* (on Chrome 25 on a Macbook Pro as no such tool was found on Android 4.2 for Nexus 7). On average, the parsing takes 11.1% of

Table 1.1: Web Page Loading Time and Percentages of HTML Parsing.

benchmarks	page loading time (ms)	HTML parsing portion		
		Speedup on non-parsing		
		1X	2X	4X
youtube	2357	8.5%	15.7%	27.1%
bbc	3081	5.6%	10.6%	19.2%
linkedin	7950	16.4%	28.2%	44.0%
yahoo	3962	6.3%	11.9%	21.2%
amazon	2476	19.8%	33.1%	49.7%
qq	6198	10.1%	18.3%	31.0%
twitter	3520	11.9%	21.3%	35.1%
taobao	10891	14.1%	24.7%	39.6%
wikipedia	4771	10.4%	18.8%	31.7%
facebook	4209	17.1%	29.2%	45.2%
geomean	4397	11.1%	19.9%	32.9%

The right two columns show the percentages when the non-parsing parts of the loading are sped up by a factor of two or four through parallelization.

the total loading time. According to Amdahl's law, any sequential part will become more critical as the other parts become parallel. As the rightmost two columns in Table 1.1 show, when the other activities are sped up by two or four times, the portion of the sequential parsing would increase to 20% or 33% respectively. Although it is possible that parsing speed may increase with new innovations in parsing algorithms and HTML design, being sequential inherently limits its enhancement and scalability on modern parallel machines. As web pages become larger and more complex, the issue is getting more serious.

The goal of this work is to remove this arguably final major obstacle for realizing a fully parallel web browser.

HTML parallel parsing is an open problem. The difficulties come from the special complexities in the HTML definition. In this paper, we focus on the latest definition, HTML5. Unlike most other languages, HTML5 is not defined by clean formal grammars, but some specifications with many ad-hoc rules. An HTML document often contains scripts written in other languages (e.g., Javascript), whose executions during the parsing of the HTML document could add new content into the document. Furthermore, the official HTML organizations have imposed some detailed specifications on HTML5 parsing algorithms. As Section 4.1 will show, these complex specifications introduce some inherent dependences into the various parsing operations, making parallel parsing even more difficult. As a result of all these special complexities, no parallel parsing techniques have been successfully applied to HTML parsing.

This paper presents the first effort to remove the obstacles for parallel HTML parsing. It describes a set of novel techniques and parsing algorithms for taming all those complexities. Specifically, this paper makes the following major contributions.

1. It presents the first systematic analysis that uncovers the special complexities in parallel HTML parsing.

2. It proposes a set of novel solutions to enable speculative parsing to address the complexities, and to circumvent various data dependences in the parsing.

3. It constructs the first two parallel HTML5 parsers, one using pipelining parallelism, the other using data-level parallelism.

4. It evaluates the parsers on a set of real-world workloads. The results show significant speedups (up to 2.4x, 1.73x on average), demonstrating the effectiveness of the (data-level) parallel parser.

5. It reveals a set of novel insights in constructing parallel HTML parsers:

   (a) Despite being inherently sequential, with a systematic treatment and carefully designed speculation, HTML parsing is possible to run efficiently in parallel.

   (b) It is difficult for pipelined parsing to generate large speedups for HTML parsing.

   (c) The scalability of data-level parallel parsing tends to grow as the web page size increases. As web pages continue to get larger [10], more speedups are expected on future web pages.

1.5 Call Sequence Prediction and Parallel JIT Compilation

Languages with a managed environment—such as JAVA, Javascript, C#—become increasingly popular. Programs in these languages often have a large number of functions, and feature many dynamic properties. For them, knowing the upcoming sequence of function calls in a run can be helpful. For example, a feature in these languages is dynamic function loading: Some classes or functions are loaded from local disks or remote servers during an execution [71]. The loading takes time. With the upcoming call sequence known, the delay can be largely hidden through prefetching. As another example, the runtime system supporting those languages, especially on embedded systems, often uses a small chunk of memory (called code cache) to store the generated native code for reuse. Knowing the upcoming call sequence can enhance the code cache usage substantially [49]. It can also help the runtime system decide when to invoke JIT to compile which function and at which optimization levels [42], and so on. The benefits may go beyond the runtime of managed languages. Co-design virtual machines [53], for instance, use

runtime Binary Code Translation to reconcile disparity between conventional ISA and native ISA. Its runtime translation also uses JIT, sharing similar opportunities.

Call sequence prediction is to provide such knowledge through prediction. It is challenging for the large scope of prediction. The state of the art is yet preliminary. Most of them have concentrated on exploiting statistical patterns in call history [17, 68, 84], and predicting the next one call rather than a sequence of calls. This limited prediction scope does not well suit the many needs of runtime systems. Even worse, as the scope enlarges, the regularity diminishes, forming a main barrier for existing prediction techniques.

In this paper, we present a new way to enable call sequence prediction. It centers on an effective exploitation of program structures. The rationale is that *program structures inherently define some constraints on function calling relations, which often cast some deciding effects on function call sequences.* Conceptually, the key of this approach is in developing an expressive model of the relations among function calls to effectively capture those constraints. To facilitate runtime call sequence prediction, the model must distinguish call sites, capture calling contexts, incorporate the influence of branches and loops, and finally accommodate the various complexities in programs and language implementations (e.g., function dynamic dispatch, function inlining, code coverage variations across inputs). Existing models—such as call graphs, call trees, and calling context trees [16]—meet some but not all these requirements.

We present *Probabilistic Calling Automata* (PCA), a new program representation that uses extended Deterministic Finite Automata (DFA) to capture both the inherent ensuing relations among functions, and the probabilistic nature of execution paths caused by branches, loops, and dynamic dispatch. A PCA is composed of a number of augmented state machines, with each encoding the control flows related function calls in a function. The PCA features a return stack and a shadow stack for efficiently maintain calling contexts, an α-stack to handle complexities brought by exceptions and unknown function calls, and the concept of v-nodes and candidate tables for addressing calling ambiguities caused by polymorphism, function pointers, and dynamic dispatch. Serving as a unified representation for function calls, PCA incorporates static program structures with profiling information, supports easy runtime state tracking, and tolerates various complexities in practical deployment.

After presenting the definition, properties, construction and usage of PCA in Section 5.1 and Section 5.2, we discuss the insufficiencies of existing program representations in Section 5.3, introduce some metrics for call sequence prediction in Section 5.4, and then describe an empirical comparison between PCA-based predictors and the extensions of three alternative methods, respectively based on Calling Context Trees and statistical patterns. Experiments show that PCA-

based predictor achieves 89% on average in a basic accuracy metric, 20–50% higher than that of the other predictors. Through parallel JIT compilation, we demonstrate that a simple usage of the PCA-based prediction can lead to performance improvement by up to 32% (15% on average).

Overall, this work makes the following contributions:

- It introduces PCA, a novel representation of function ensuing relations in a program that captures the influence cast by control flows and calling contexts.
- It shows how PCA can be used to enable effective call sequence prediction with design choices and usage study, as well as a systematic comparison with alternatives.
- It provides a set of metrics for measuring the quality of a call sequence prediction at various levels. They may meet the needs of different uses of the prediction.
- Finally, this work, for the first time, demonstrates the feasibility and benefit of accurate call sequence prediction, which opens up new opportunities for dynamic optimizations in various layers of the execution stack.

2 Principled Speculation for FSM Parallelization

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.

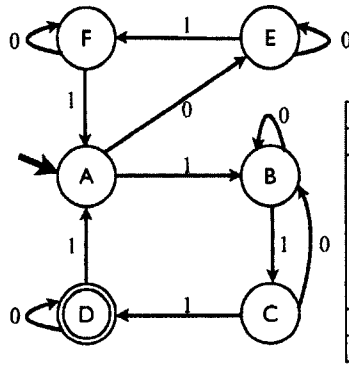
This work provides some principled understanding of FSM parallelization, and offers the first disciplined way to exploit application-specific information to inform speculations for parallelization. Through a series of rigorous analysis, it presents a probabilistic model that captures the relations between 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.

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 [19].

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 2.1 (a) shows the state-transition graph for a pattern-matching FSM, along with an example input to it.

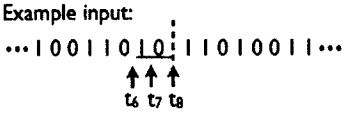
To check whether a string matches the pattern, the FSM starts with the initial state (state A) and processes the input character one after one. At each input character, the FSM moves to a state specified by the state-transition graph. Its arrival at state D indicates the recognition of a string that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

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

Example input:



(a) Example FSM

(b) Attributes of the FSM

Figure 2.1: 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.

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 2.1 (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 2.1 (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 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 [19], XML validation [90], front end of a compiler [12], compression and

decompression [62], model checking, network intrusion detection [54], 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.

2.1 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 [30].) We hence focus our discussion on such FSMs.

A classic approach to parallelizing FSM computations is through variations of the parallel prefix sum algorithm [65]. 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 [58, 90] 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 [58, 90] 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 2.1, 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

[1] The original paper [65] proposes to represent each function with a boolean matrix, which incurs even a higher time and space complexity.

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 [58, 90]. As Figure 2.2 shows, on an 8-core Intel Xeon E5620 system, the approach [58, 90] 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 2.2, it produces 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 [58, 90] 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 2.1 (a)) for lookbacks. It could seriously limit lookback benefits. For the example in Figure 2.1 (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.

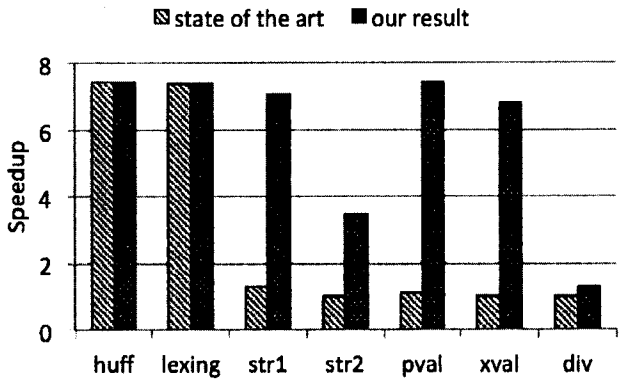


Figure 2.2: Speedups Comparison between the state-of-the-art and ours.
The speedups brought by the state-of-the-art speculation scheme [90] are limited on some complex FSM applications. The results are for 8 threads running on an 8-core machine. Details are shown in Section 2.6.

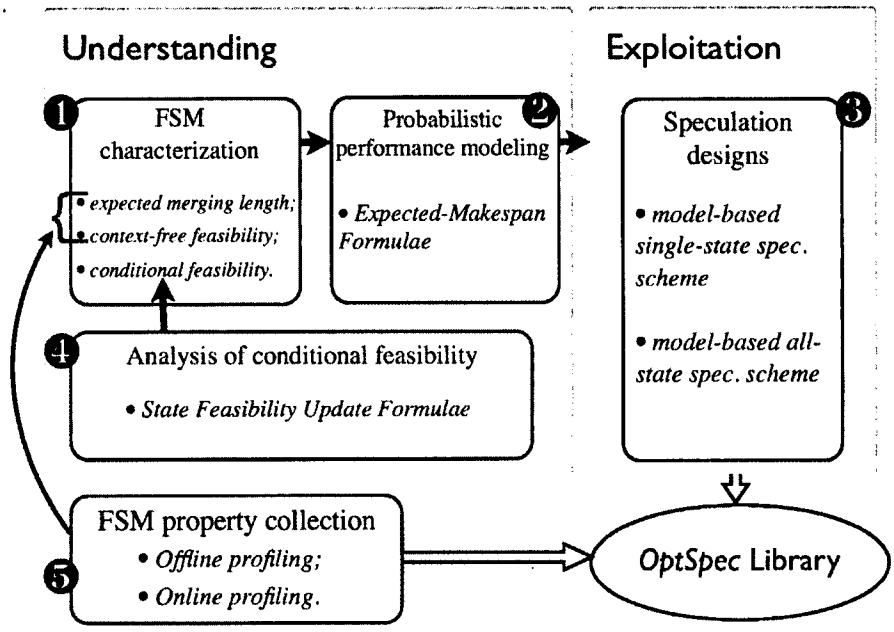


Figure 2.3: Overview of this work.

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 2.3. 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 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 2.2. 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 2.1, we provide some more information about it. The FSM was deliberately made simple for illustration purpose. The table in Figure 2.1 (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 2.5 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.

Chapter Organizations In the following, we explain each component in Figure 2.3. We first present the components for enhancing the understanding of FSM properties and their connections with speculative executions (Sections 2.2 and 2.3), and then describe the two speculative schemes and the OptSpec library (Sections 2.4 and 2.5.) 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 2.1.

Table 2.1: 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

2.2 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.

2.2.1 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 2. *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 2.1 (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 2.1 (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.

2.2.2 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 2.4. 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.

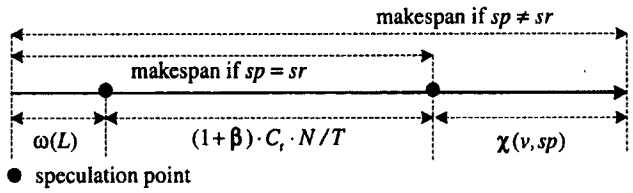


Figure 2.4: The make-span of a thread.
In cases of correct speculation ($s_p = s_r$) and wrong speculation ($s_p \neq s_r$), where, s_p and s_r are the speculation and real states respectively.

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*.

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 2.1 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 2.1 (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:

$$L_M^v(sp) = \sum_{s_i \in S_v} L_M(sp, s_i) \cdot P^v(s_i), \tag{2.1}$$

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

$$\chi(v, sp) = min\{L_M^v(sp), N/T\} \cdot (1 + \beta) \cdot C_t. \tag{2.2}$$

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:

$$ES(\mathbb{S}) = \omega(l) + N/T \cdot (1 + \beta) \cdot C_t + \sum_{i=2}^{T} \chi(v, sp(i)). \tag{2.3}$$

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 2.1 and 2.2, the *ES Formula*.

Example We now show how the ES Formula applies to the example DFA in Figure 2.1. 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 2.1 (b) gives all the

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

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 2.1, $L_M^v(A)$ 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 2.1(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 2.2, we know that in this case, the expected reexecution time is

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

Assuming $N = 400, T = 2, \beta = 1, C_t = 1$, we get χ("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 2.4).

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 2.5) from the FSM, except for the conditional feasibilities $P^v(s_i)$ that appears in Formula 2.1. We next show how to compute $P^v(s_i)$ from state transitions.

2.3 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

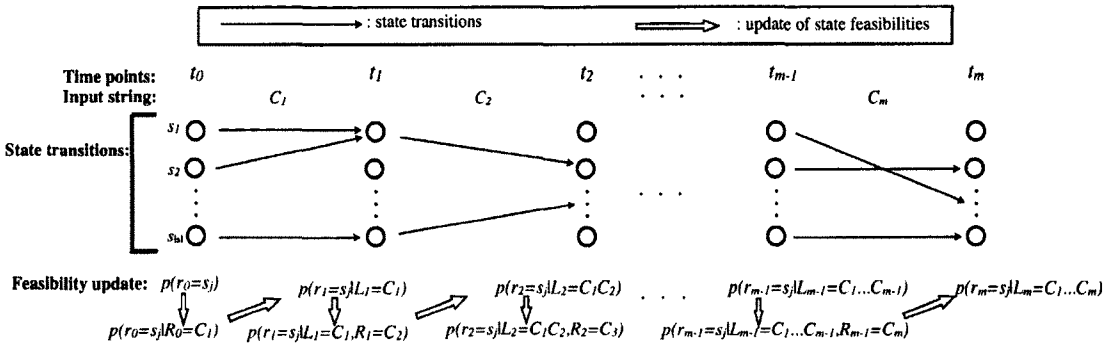


Figure 2.5: Computing conditional feasibilities.
Gradual refinement of conditional feasibilities along with state transitions. The state transitions graph in the middle shows all possible transitions allowed by the FSM on the input characters.

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.

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 2.5 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 2.5. Context-free feasibilities are easily obtainable through profiling (Section 2.5); 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 2.5, 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 2.5. 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 2.6 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 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},

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:

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

Figure 2.6: Formulae for inner-stage and inter-stage update of state feasibilities.
$C_{1\cdots(i-1)}$ 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.

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 2.5 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

$$\frac{p(R_{i-1} = C_i | r_{i-1} = s_j) \cdot p(r_{i-1} = s_j | L_{i-1} = C_{1\cdots(i-1)})}{\sum_{s\in S} p(R_{i-1} = C_i | r_{i-1} = s) \cdot p(r_{i-1} = s | L_{i-1} = C_{1\cdots(i-1)})}. \quad (2.4)$$

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 2.1. 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 2.1 (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 2.1. The denominator equals

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

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

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 2.1 (a). When s=A, for instance, it is computed as follows:

$$\begin{aligned} p(r = A|L = \text{“1”}) &= \sum_{s \in S, s \xrightarrow{1} A} p(r = s|R = \text{“1”}) \\ &= p(r = D|R = \text{“1”}) + p(r = F|R = \text{“1”}) \\ &= 0.29. \end{aligned}$$

The results from this step are as follows:

$$p(r = s|L = \text{“1”}) = 0.29, 0.14, 0.28, 0.14, 0, 0.15 \ (s = A, B, \cdots, F).$$

Step 3: We now add the second lookback character into the condition to compute the probabilities $p(r = s|L = \text{“1”}, R = \text{“0”})$. This step still corresponds to the point t_7 in Figure 2.1 (a). When s=A, for instance, the probability is computed as follows based on Formula 2.4:

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

The results from this step are as follows:

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

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,

$$p(r = B|L = \text{“10”}) = p(r = B|L = \text{“1”}, R = \text{“0”}) + p(r = C|L = \text{“1”}, R = \text{“0”}) = 0.422.$$

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 2.1 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.

2.4 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.

2.4.1 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 2.1, 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.

2.4.2 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 2.1,2.2,2.3). 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

$$\omega(l) = (\sum_{k=0}^{l} |S_k|) \cdot (C_t + C_p). \tag{2.5}$$

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

$$\omega(l) = (\sum_{k=0}^{l} |S_k|) \cdot (1 + \lambda) \cdot C_t. \tag{2.6}$$

Both C_t and λ can be easily measured (Section 2.5.)

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 2.3, for a given l, it is easy to use Equation 2.1 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 2.2, the minimal reexecution cost can be computed as follows:

$$\chi(v, s^*) = min\{L_M^v(s^*), N/T\} \cdot (C_t + C_w). \tag{2.7}$$

Determining Lookback Length The selection of the appropriate lookback length is based on the expectation of make-span (i.e., Equation 2.3.) 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 2.7 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:

$$ES(l) = \omega(l) + N/T \cdot (C_t + C_w) + (T-1) \cdot \overline{\chi(l, s^*)}. \tag{2.8}$$

A brute-force way to obtain the best lookback length is to use equation 2.8 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 2.8 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 [120] 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 2.5.

2.4.3 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 [58,90], 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 2.3) to

$$ES(l_b) = l_b \cdot (1+\lambda) \cdot C_t + N/T \cdot (1+\beta) \cdot C_t + (T-1) \cdot l_x \cdot (1+\beta) \cdot C_t. \tag{2.9}$$

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 [120].

Putting l_x values from Lemma 1 into Equation 2.9, 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 [120].

All parameters in the theorem, including $L^0_M(s)$ ($s \in S$), can be obtained through profiling (Section 2.5.) 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.

2.4.4 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 work.

2.5 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 $L^l_M(s^*)$ 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 [102]. 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.

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 [120].

Table 2.2: 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

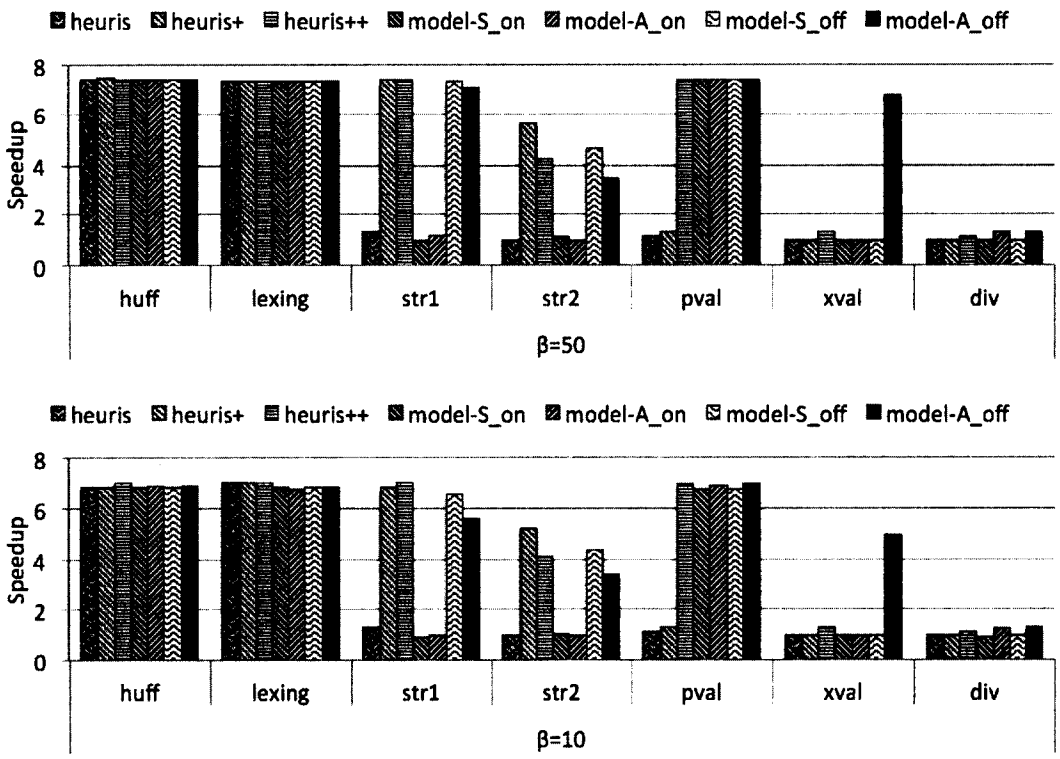


Figure 2.7: The overall speedup when 8 threads used.

2.6 Evaluation

We evaluate the proposed techniques on seven FSMs listed in Table 2.2. They are developed based on the literatures in the web XML processing community (e.g., *lexing* and *xval* [116]), mathematics (e.g., *div* [14]), classical Huffman decoding (*huff* [62]), and string pattern matchings (*str1*, *pval*, and *str2* [6].) 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 2.2 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 [90].

heuris+: Our simple extension to previous scheme [90].

heuris++: Our further extension to previous scheme [90].

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 [90]. 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 [90] 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 2.2.2: 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 [62].

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 work, with either online or offline profiling.

Figure 2.7 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 2.8 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 2.2, 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 [62, 90]. 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 [90]. 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 2.7. 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

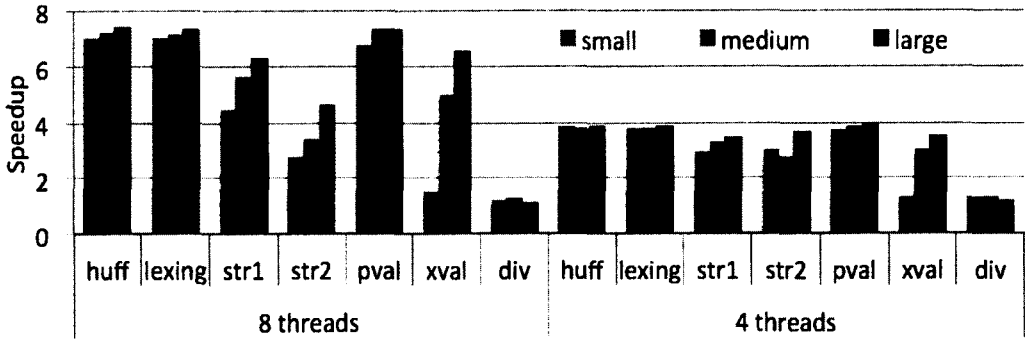


Figure 2.8: "Model-A off" on different input sizes.

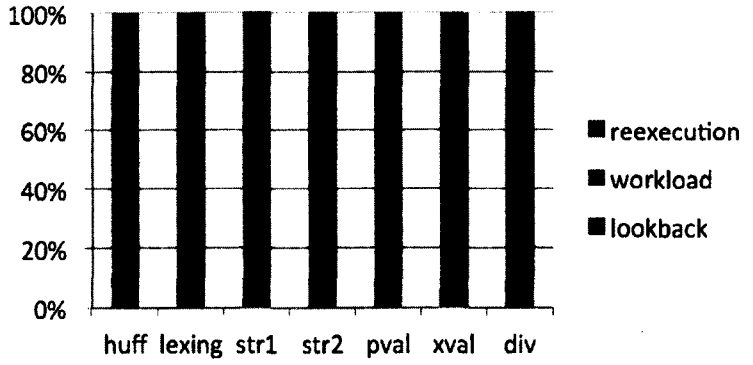


Figure 2.9: Time breakdown of model-A_off with β=50.
Notations: "lookback" for lookback overhead, "workload" for workload processing time, "reexecution" for reexecution time.

than the previous five, including 742 states to implement an simplified Schema validation algorithm [116]. 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 2.9), 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 [14]. Structure-wise, it is simple, containing only seven states, shown in Figure 3.2.

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

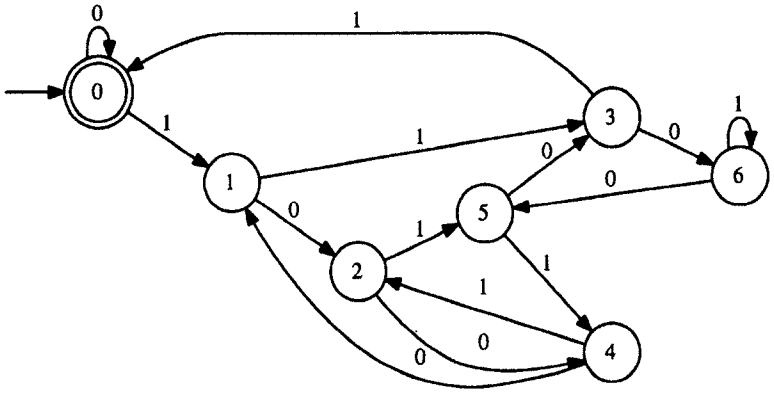


Figure 2.10: The FSM of *div*.

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.

2.7 Related Work

Program parallelization has drawn explorations from language design (e.g. Cilk [48], X10 [36]), to hardware support (e.g., TLS [51, 101]) and programming models (e.g., STM [11, 33]). 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 [58]. 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 [62] have designed a parallel JPEG decoder, which explores parallel Huffman decoding. Luchaup and others [74] have used hot state prediction in a pattern matching FSM to identify intrusions. Other examples include speculative parsing [60] and speculative simulated annealing [109]. 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) [121]. 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 [21, 22] by exploiting some special properties of the target language or parsing algorithm (e.g., LR parsing in Fischer's seminal work [47]). The second tries to decompose the input [73], 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 [115]. 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 [90] 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 [40] or threads [45, 94, 103]. 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 [72]. Kulkarni and others have showed the usage of abstrac-

tion to find parallelism in some irregular applications [64]. The pre-computation used by Quinones and others for speculative threading [93] 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.

2.8 Summary

This work 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.

3 On-the-Fly Pincipled Speculation for FSM Parallelization

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.

3.1 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

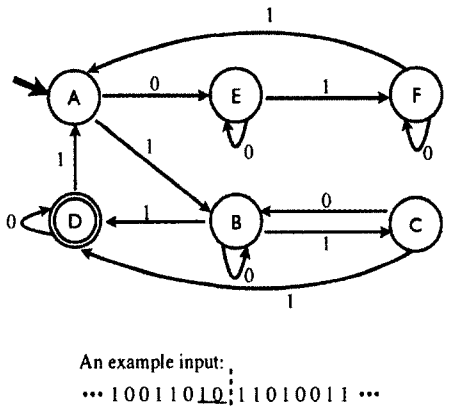


Figure 3.1: An FSM for string pattern matching and an example input string to it. Each circle on the graph represents an FSM state. State A is the initial state (marked by the extra incoming edge), and state D is an acceptance state. The symbols on the edges indicate conditions for state transitions.

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 [119] 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 3.1 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.

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
```

The actual cost of this default profiling algorithm could range from several seconds to tens of minutes (details in Section 3.5). 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 [119].

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.

3.2 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 suitable 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.

3.3 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.

3.3.1 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 3 (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 4 (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 5 (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 5, 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 6 (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 7 (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 3.1. 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".

3.3.2 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 2. *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 2. *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 2, 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 6, $s_1' \neq s_2'$. 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. □

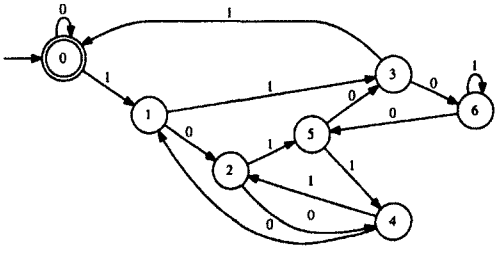


Figure 3.2: DFA for Testing Divisibility by Seven

Discussion. Theorem 2 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 3.2 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 [119]. With Theorem 2, all these collection work can be safely saved as it can immediately tell that no two states of the DFA are convergent.

3.3.3 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 3. *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 2 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 3.3 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 3.3.1, edges may exist among nodes in the same layer or from a higher layer to a lower layer, but not the other way.

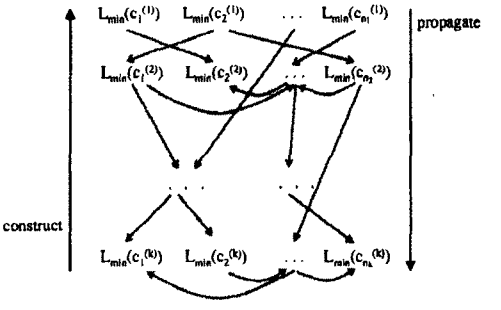


Figure 3.3: Minimal Convergent Length Propagation Graph

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 3. *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 4. *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 2 shows the pseduocode of this algorithm.

Algorithm 2 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
```

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.

3.4 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 8. 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.*

3.4.1 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 3.1, 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:

$$TRR(\text{IR Reuse}) = \frac{O(N^2 \cdot S' \cdot L)}{O(N^2 \cdot SAMPLE \cdot L)} = O(\frac{S'}{SAMPLE}) \tag{3.1}$$

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 $O(1/SAMPLE) \leq TRR(\text{IR reuse}) \leq 1$.

3.4.2 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 3.1. 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:

$$TRR(\text{Early Stop}) = \frac{O(N^2 \cdot SAMPLE \cdot L')}{O(N^2 \cdot SAMPLE \cdot L)} = O(\frac{L'}{L}) \tag{3.2}$$

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 $L' \leq L$.

3.4.3 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 3 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 and 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:

$$TRR(\text{IR Reuse + Early Stop}) = O(\frac{S' \cdot L'}{SAMPLE \cdot L}). \tag{3.3}$$

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 $O(1/SAMPLE \cdot L) \leq TRR(\text{IR Reuse + Early Stop}) \leq 1$.

3.5 Evaluation

We evaluate our techniques using the benchmark suite used in a recent study [119] but with more inputs added. Table 5.1 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, 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

Algorithm 3 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
```

from prior studies [83, 119]. 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 [119]. 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 [119] 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

Table 3.1: 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

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).

Results: Static Analysis Table 3.2 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 3.2: 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 3.3 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 3.4 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 3.3: 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 3.4 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 3.4: 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 3.5: 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 3.5 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 [90, 119]. 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.

3.6 Related Work

There have been many efforts spent on program parallelization. The efforts are from various angles, from language design (e.g. Cilk [48], X10 [36]), to hardware support (e.g., TLS [51, 101]) and programming models (e.g., STM [11, 33]). 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 [65]. A recent study [83] 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 [58]. They introduce the concept of lookback (called overlap) for enhancing speculation accuracy. Other examples include the parallelization of JPEG decoder by Klein and Wiseman [62], hot state prediction in a pattern matching FSM to identify intrusions by Luchaup and others [74], speculative parsing [60], and speculative simulated annealing [109]. 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 [21, 22] by exploiting some special properties of the target language or parsing algorithm (e.g., LR parsing in Fischer's seminal work [47]). The second tries to decompose the input [73], and can often leverage more parallelism than the first approach. All these prior studies employ simple heuristics for speculation. Zhao and others [119] 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) [121]. 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 [90] 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 [40] or threads [45, 94, 103]. 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 [72]. Kulkarni and others have showed the usage of abstraction to find parallelism in some irregular applications [64]. The pre-computation used by Quinones and others for speculative threading [93] 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 [96,105]. 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.

3.7 Summary

This chapter 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.

4 HPar: Enabling Parallel HTML Parsing

Parallelizing HTML parsing is challenging due to the complexities of HTML documents and the inherent dependences in its parsing algorithm. As a result, despite numerous studies in parallel parsing, HTML parsing remains sequential today. It forms one of the final barriers for fully parallelizing browser operations to minimize the browser's response time—an important variable for user experiences, especially on portable devices. This paper provides a comprehensive analysis on the special complexities of parallel HTML parsing, and presents a systematic exploration in overcoming those difficulties through specially designed speculative parallelizations. This work develops, to the best of our knowledge, the first pipelining and data-level parallel HTML parsers. The data-level parallel parser, named *HPar*, achieves up to 2.4x speedup on quad-core devices. This work demonstrates the feasibility of efficient, parallel HTML parsing for the first time, and offers a set of novel insights for parallel HTML parsing.

4.1 Challenges of Parallel HTML5 Parsing

In this section, we first provide some background on HTML5, and then describe five classes of special challenges it imposes on parallel parsing.

4.1.1 Background on HTML5 and Its Model of Parsing

HTML5 is the latest version of the HTML standard, with enhanced support for multimedia and complex web applications. Different from all previous versions, Web Hypertext Application Technology Working Group (WHATWG) and World Wide Web Consortium (W3C) provide some concrete specifications on the model of HTML5 parsing [2] (hereafter referred as *HTML5 spec*). Previously, the HTML parsers developed by different companies interpreted HTML documents differently, causing inconsistent behaviors among browsers. For instance, IE and Opera read `<foo<bar>` as one tag `foo<bar`, while Firefox and Chrome read it as two tags, `foo` and `bar`. For that reason, the HTML5 spec includes specifications on the parsing. According to HTML5

tests [7], major modern web browsers, such as Chrome 23, Safari 5 and Firefox 17, fully support HTML5 parsing rules.

Figure 4.1 shows the standard HTML5 parsing model [2], which contains two stages: *tokenization* and *tree construction.* An input byte stream that comes from network first flows into the tokenization stage, where a *tokenizer* parses it into tokens. Upon the recognition of a token by the tokenizer, a *tree builder* immediately consumes it and organizes the parsing results in a Document Object Model (DOM) tree. The HTML5 spec uses finite state machines to track the progress of both the tokenization and the tree construction. The states for the tokenization are called the *tokenization state*, and the states for the tree construction are called the *insertion modes.* After parsing, the DOM tree will be passed to the layout engine for web page rendering. When the tree builder finds some executable scripts (e.g., Javascript code) embedded in the HTML document, the scripts will be executed immediately. The execution may generate some new HTML strings, which also need to be parsed by the HTML parser. Hence the path from the Script Engine to the input stream.

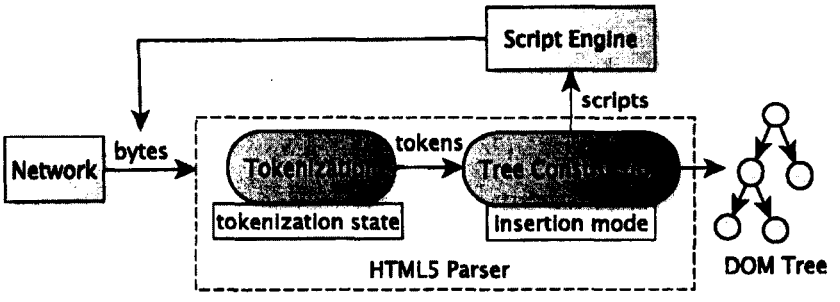


Figure 4.1: HTML5 parsing model

4.1.2 Special Challenges for Parallel Parsing

The standard HTML5 parsing model [2] designed by W3C and WHATWG consists of complex state machines. The state machines for tokenization, for instance, contains 68 states, and more than 250 transitions among those states. The complex parsing model just reflects the tip of the iceberg for the development of a parallel HTML parser. There are more inherent complexities preventing the HTML parser running in parallel. In this part, we use the example in Figure 4.2 to explain these complexities.

Informal language. Unlike many other languages, HTML5 is not defined with some clean, formal grammars, [1] but a collection of ad-hoc rules and specifications. Most previous work on parallel

[1] We are aware of some Antlr grammars for earlier HTML versions, but they do not conform to HTML5 spec.

parsers [13, 21, 22, 47, 75, 100] has targeted formally defined languages (e.g., in a Context-Free Grammar) and leveraged the features of the grammars for parallelization. The ad-hoc definition of HTML5 makes these techniques hard to be applied to its parsing.

Error-correcting. As many existing HTML documents contain syntax errors, web browsers have been tolerating such errors with some kind of automatic error correction. Now, this feature is formally integrated into the official HTML5 parsing model. As a result, HTML5 parsers typically resolve some common errors to their best knowledge. When validating it with W3C's HTML5 conformance checker, the example in Figure 4.2 contains three syntax errors: missing `title` element, bad nested tags between `<b>` and `<i>` in line 8, and a tag name typo `</dt>` in line 30. Unlike the conformance checker, HTML5 parsers typically do not report errors to end users, but try to fix them. For instance, they would ignore the typo tag `</dt>` since no start tag `<dt>` was met before. When the next `<td>` is read, they would assume a `</td>` is missing and automatically insert it into the token flow. This forgiveness is essential for working with real-world HTML documents, but complicates parallel parsing. For instance, suppose the HTML document in Figure 4.2 is split into two parts and assigned to two threads to parse in parallel. When the second thread encounters `</dt>`, it does not know whether this unmatched tag is an error or has its partner tag `<dt>` in the first part of the document that this thread does not see. The parallel parsing algorithm has to correctly handle such ambiguities.

Embedded languages. It is common for an HTML5 document to contain some embedded code written in other scripting languages (e.g., CSS, Javascript, MathML and SVG.) Theoretically, there is no restriction on what language can be used, as long as the embedded codes are marked by the `<script>` tag. In Figure 4.2, a CSS segment is embedded at lines 4 and 5, while a Javascript segment is between lines 13 and 18. These embedded languages bring their own syntaxes into HTML5 documents, complicating the parsing. Previous work on parallel parsing has not considered such complexities.

Dynamic inputs. Another complexity coming with the embedded scripts is that they may alter the input HTML5 document. As shown in line 18 of Figure 4.2, the script `document.write()` inserts extra tags into the HTML5 input stream. The inserted tags may even introduce new errors to the HTML document, which will not be detected before the script is executed. In our example, an `alt` attribute in the tag `<img>` is required by HTML5 spec, but is missing there. This feature introduces hazards or dynamic dependences into the parallel parsing of HTML documents, because the changes the scripts make to the input document (e.g., generating a new tag) may affect how the later part of the document should be parsed.

```
<!DOCTYPE html>
<html>
<style type="text/css">
 i {color:red;}
 li {color:blue;}
</style>
<body>
  <b><i> My List </b></i>
  <ol>
    <li> random picture
    </br>
    <script type="text/javascript">
    var time = new Date();
    if(time.getSeconds()<30)
        var image = "a.jpg";
    else
        var image = "b.jpg";
    document.write("<img src="+image+">");
    </script>
    <li> a table
    <table border="1">
      <thead>
        <tr>
          <th>Month</th>
          <th>Savings</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>January</td>
          <td><i>$100</i></td>
        <!—<tr>
          <td>February</td>
          <td><i>$80</i></td>
        </tr>—>
      </tbody>
    </table>
  </ol>
</body>
</html>
```

Figure 4.2: An HTML5 Example

Strong dependences. As the next section will show, due to the unique features of HTML, cyclic dependences exist between tokenization and tree construction, as well as among the processing of different parts of the HTML document, making neither pipelining nor data-level parallelization directly applicable.

Because of these five special complexities, previous techniques developed for parallel parsing other languages have not been applied to HTML successfully. In this work, we address these complexities through a carefully designed speculative parallelization, which exploits the special features of HTML as well as some insights drawn from the statistical properties of real-world HTML files. Our explorations cover both pipelining and data-level parallelism. We next present each of them.

4.2 Speculative Pipelining

Pipelining is a parallelization technique in which the execution process is divided into stages, and the data go through the stages one by one, so that the stages can run in parallel. This technique

has been used in various layers of computer systems for performance. In this section, we present a lightweight pipelining algorithm for parallel HTML5 parsing.

To form a pipeline, one must first decouple the original task into multiple stages. A data element goes through all the stages. The processing of a data element in a stage depends on the earlier stages, but different stages may process different data elements concurrently. So a pipeline has the potential to produce a speedup up to the number of stages. But the actual speedup is often lower than that due to the workload distribution among stages and synchronizations between threads [104].

The HTML5 spec suggests that HTML parsing can be naturally decoupled into two stages: tokenization and tree construction. Our manual analysis of some typical HTML5 parsers (such as jsoup [5] and Validator.nu [9]) shows that these two stages have the least number of dependent data structures between them and have a clear interaction interface. Finer-grained ways to partition the parsing are possible. But the complex interactions would result in more complicated code and tremendous difficulty for later maintenance and extension. We hence focus our implementation on this two-stage pipeline.

Even for this natural two-stage partition, directly applying the basic pipelining scheme to them cannot expose any parallelism. We next analyze two special challenges for pipelining HTML5 parsing, and then explain how our speculative pipelining solves those problems.

4.2.1 Pipelining Hazards

Data dependences that prevent a pipeline stage from processing the next item before its following pipeline stages finish processing the current item are called *pipelining hazards*. In the HTML5 parsing model, there are a series of such hazards that restrain the tokenizer and tree builder from running in parallel.

Tokenization State The HTML5 spec requires that the tokenizer cannot start processing new data elements before the tree builder finishes consuming the newly recognized token. The reason for this requirement is that the tree builder may modify the tokenization state during its consumption of the newly recognized token. If the tokenizer continues to generate the following tokens regardless of the tree builder's status, it may start from a wrong tokenization state, leading to tokenization errors.

Figure 4.3 shows such an instance happening when the example in Figure 4.2 is parsed. In a sequential parsing, when the tree builder sees the `start style` tag, it expects that the following token must be from CSS code and hence correctly switches the tokenization state from

DATA to Rawtext. But in a pipelined parsing, if the tokenizer does not wait for the tree builder, it would tokenize the following input elements before the tokenization state gets updated, and hence produce the wrong tokenization results as Figure 4.3 shows.

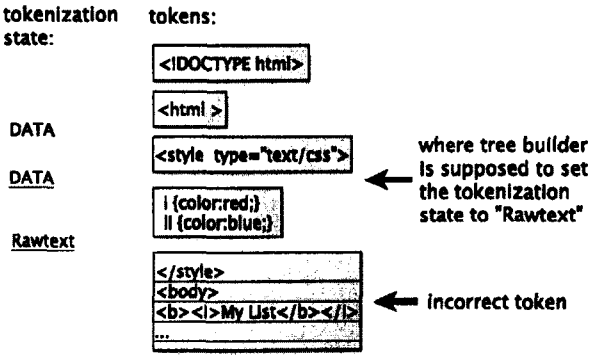


Figure 4.3: A failure of naive pipelined HTML parser.
An example showing that the naive pipelined HTML5 parser emits incorrect tokens, marked with underlines.

Based on the HTML5 spec, we identify 10 cases, listed in Table 4.1, when the tree builder may change the tokenization state during its processing of tokens. The HTML5 spec puts the responsibility of updating the tokenization state on the tree builder because the new state depends on both the new token and the insertion mode of the tree builder. For instance, as the first row of Table 4.1 shows, when the tree builder receives a token <title>, it would switch the tokenization state to RCDATA only when the insertion mode of the tree builder is InHead (which means that the parser is parsing the region in the scope of head). [2] The tokenization state hence causes cyclic dependences between the tokenizer and tree builder.

Table 4.1: Pipelining hazards from tokenization state

Insertion Mode	Next Token	Switch-to State
in head	<title>	RCDATA
in head	<noframes>	RAWTEXT
in head	<style>	RAWTEXT
in head	<script>	script data
in body	<xmp>	RAWTEXT
in body	<iframe>	RAWTEXT
in body	<noembed>	RAWTEXT
in body	<noscript>	RAWTEXT
in body	<plaintext>	PLAINTEXT
in body	<textarea>	RCDATA

Self-Closing Acknowledged Flag A more obscure pipelining hazard in HTML5 spec is the *self-closing acknowledged flag*, a global variable in the tokenizer. In HTML, a set of tags are predefined to be self-closing tags, meaning that they need no separate end tag. The new line

[2] The insertion mode and next token are not the only conditions that cause the tokenization state to switch; *script flag*, for instance, can also trigger it.

tag
 is such an example. A tag with "/" at the end is considered as being written in a self-closing form; a predefined self-closing tag should be written in such a form, but does not have to—another evidence of the flexibility of HTML5 spec.

The self-closing acknowledged flag is used in the parser to assist the check on whether a tag written in a self-closing form (i.e., with "/" as its ending symbol) indeed belongs to the predefined set of self-closing tags. According to the HTML5 spec, the tokenizer and tree builder must cooperate to do the check. The global variable self-closing acknowledged flag is set to true initially. When the tokenizer recognizes a token, if the ending symbol of it is "/", it pessimistically sets the flag to false, otherwise, it keeps the self-closing acknowledged flag unchanged. Then, when the tree builder receives this start tag, it will check if this tag is in the predefined self-closing tag set. If so, it sets the self-closing acknowledged flag to true. Next, right before the tokenizer attempts to process the next symbol in the input, it checks the self-closing acknowledged flag, and records an error if that flag is false, indicating that the token just processed is not a self-closing tag but is mistakenly written in a self-closing form in the input. So, in a naive parallel processing, if the tokenizer continues to generate the next token before the tree builder finishes processing the current token, it would miss the self-closing error that the tree builder would find when processing the current token, leading to wrong parsing results on the following tokens.

The cyclic dependences between the tokenization and tree construction serialize the two stages. As a result, a basic pipelining scheme can exploit no parallelism in the HTML parsing.

4.2.2 Speculative Pipelining

To overcome the challenges, an attempting option is to remove all the dependences by redesigning the state machines in the tokenizer and tree builder. But the cyclic dependences between them are inherent rather than side products of the state machines. For instance, one may let the tokenizer rather than tree builder get in charge of the tokenization state update. But that does not address the problem because in either design, the update depends on the information from both the tokenizer (the next token) and the tree builder (the insertion mode). The same happens to the self-closing acknowledged flag update. Even if some dramatic changes to the parser might go around these issues, they would likely seriously violate the HTML5 spec.

We instead address the challenges by developing a *speculative pipelining* scheme. Speculative pipelining is a pipelining technique that employs speculation to break the cyclic dependences between stages. Its basic structure contains three steps: (1) predict the value of the data items that cause the dependences; (2) with the predicted values, process the tasks in pipeline specu-

latively; (3) when the dependent data is produced, verify the prediction, and if the prediction is wrong, trace back and reprocess the data from the correct state (called *rollback*).

Our design is based on some empirical observations. We conduct a systematic measurement on how often the tree builder alters the tokenization state. Our measurement on the top 1000 websites [8] shows that it happens in less than 0.01% of all the operations of the tree builder. It suggests that if the tokenizer optimistically assumes that its state won't be altered by the tree builder, there is less than 0.01% probability that it has to rollback. Moreover, we find that among the 117 HTML5 tags [3] , only 18.8% of them are self-closing, suggesting that speculation may be sufficient to overcome the self-closing acknowledged flag hazard as well.

Based on these observations, we develop a speculative pipelining framework for HTML5 parsing, as shown in Figure 4.4. This framework contains four main components: a predictor that predicts the context (including the tokenization state and self-closing acknowledged flag) for tokenization; a hazard detector that detects the pipelining hazards; a rollback function that recovers the context from misspeculation; and a token buffer for passing the tokens from the tokenizer to the tree builder. We explain each of the components as follows.

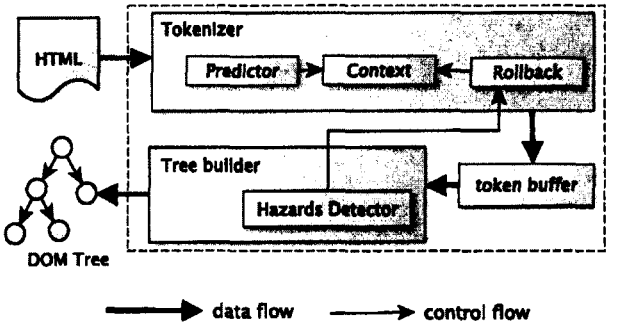


Figure 4.4: The speculative pipelining framework for HTML5 parsing.

Predictor Basically, there are two ways to predict the context: One is optimistic prediction, in which we expect that no hazards will happen. So the context after emitting the prior token is considered as the current context. The other is pessimistic prediction, that is, we do not expect the prior context to be a good guess, but use an advanced predictor (e.g., a predictor that exploits conditional probabilities based on partial contexts [118]) to gain good speculation accuracy. There is a classic tradeoff between prediction overhead and prediction accuracy. In this work, we choose the optimistic prediction because of its simplicity and the low probabilities of the hazards. In Section 4.4, we will show that the design provides sufficient accuracy.

Hazard Detector To detect pipelining hazards, we use a flag, *hazard flag*, to indicate whether

```
class Snapshot {
    TokenizerState state;
    boolean acknowledgeSelfClosingFlag;
    Token.StartTag lastStartTag;
    ...
}
abstract class Token {
    TokenType type;
    int index;
    Snapshot snapshot; /* new */
    ...
}
abstract class TreeBuilder {
    CharacterReader reader;
    Tokenizer tokenizer;
    protected Document doc;
    ...
    private Token previousToken; /* new */
    ...
}
```

Figure 4.5: Data structures for rollback

some pipelining hazards have occurred. For example, when the tree builder reads a start tag whose name is "plaintext", it switches tokenization state to PLAINTEXT. Right after this, it sets the hazard flag to true. The pipelining hazards mentioned in Section 4.2.1 scatter over the entire tree construction algorithm. To cover them, we look into the HTML5 spec and identify all sites in the algorithm where a pipelining hazard may happen. For each site, we insert a corresponding detection instruction. In this way, the detector is fully integrated into the tree builder, and is automatically triggered by any pipelining hazard.

Rollback When a pipeline hazard occurs, the tokenizer may run into some incorrect states. To effectively rollback from the incorrect status, two questions need to be answered. The first is which place the tokenizer should rollback to? This place should be safe and should be as late as possible to minimize the rollback overhead. Our solution meets both criteria by moving the tokenizer back to a state where the tree builder just finishes consuming the token that causes the hazard. The second question is how to recover the status at the safe point. Our solution is to add a *Snapshot* data structure that encapsulates the data needed for recovering and a *previousToken* variable which contains the needed snapshot, as shown in Figure 4.5.

Our solution introduces a hazard flag. The hazard detector sets the flag once it finds some hazards. The tokenizer checks the flag every time before it starts to produce the next token; so usually no more than one token is produced between the time the flag is set and the time the tokenizer sees it. Once the tokenizer sees that the flag is set, it invokes the rollback, and then resets the flag. The overhead of rollback mainly consists of the flag checking and recovery of the tokenization state.

Token buffer. The token buffer plays the "pipe" role in this pipelining scheme. Its efficiency is important for the pipelining performance. We initially employ a non-blocking concurrent queue,

which implements an efficient "wait-free" algorithm [80].

In our experiments, we empirically found that the tokenizer and tree builder generally have similar rates in terms of token buffer operations, i.e. dequeue and enqueue. It suggests that a small buffer may be sufficient for performance. We tried a spectrum of buffer sizes. As Section 4.4.2 will show, the smallest buffer size, 1, gives similar performance as other sizes. It is hence adopted in our design. At a pipeline hazard, the tokens in the queue have to be discarded.

4.3 Speculative Data-Level Parallelization

Data-level parallelization is orthogonal to pipeline parallelization (they could be used together). It partitions the input into a number of chunks and distributes them to processors for parallel processing. Applying it to HTML5 parsing faces the five complexities discussed in Section 4.1.2 . This section presents our solution, a speculative scheme that exploits the special features of HTML and statistical properties of real-world HTML files.

To ease the explanation, we start with a simpler problem: parallelizing the parsing of documents in LIST, a language that we made up with much simpler features than HTML. The discussion offers some important intuitions for parallelizing HTML5 parsers. We then describe how we capitalize on those insights for developing our data-level parallel HTML5 parser.

4.3.1 Insights From a Simplified Parsing Problem

The Problem of Parallel LIST Parsing

Earlier HTML specifications are based on the Standard Generalized Markup Language (SGML). These versions of HTML can be defined by Document Type Definition (DTD). LIST is an instance of such a DTD-defined languages. Figure 4.6 gives the definition of LIST and one legal input. To explore data-level parallelism, our goal is to design a parallel parsing algorithm with three major functions: $partition()$ for partitioning the input into smaller units, $parallelParsing()$ for running a number of parsers on the smaller units, and $merge()$ for merging parsing results into a complete one.

Algorithm Design Space

At the core of designing a data-level parallel parsing algorithm for LIST are the designs of the partitioning function $partition()$ and merging function $merge()$. Based on different emphasis put on the two functions, there are two directions: being either *partitioning-oriented* or *merging-oriented.*

```
<lists>   ::= <list> <lists>
            | <empty>

<list>    ::= <list> <items> </list>

<items>   ::= <item> <items>
            | <empty>

<item>    ::= <item> <data> </item>
            | <list>

<data>    ::= [a-zA-Z ]*
```

```
<list>
  <item>pen</item>
  <list>
    <item>laptop</item>
    <item>tablet</item>
  </list>
  <list>
    <item>cartoon</item>
    <list>
      <item>Jane Eyre</item>
      <item>Oliver Twist</item>
    </list>
  </list>
</list>
```

Figure 4.6: The LIST language and its example input.

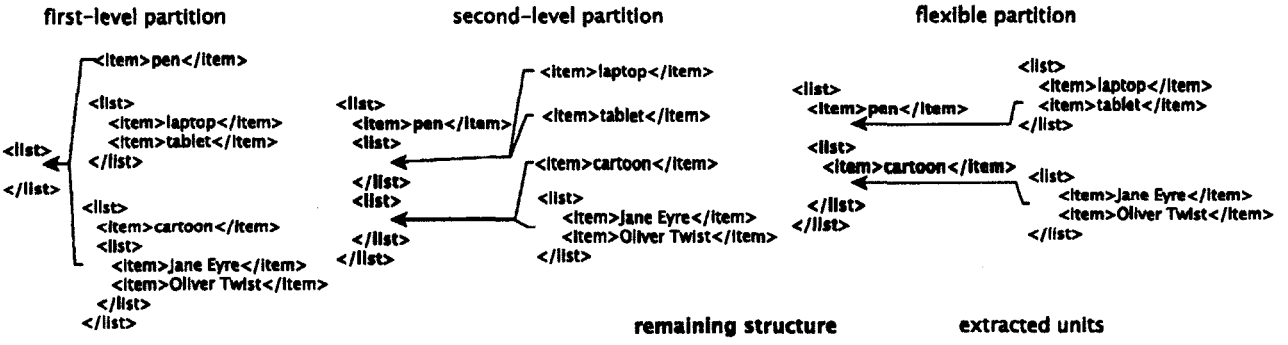


Figure 4.7: Partitioning strategies in the partitioning-oriented scheme.
Partitioning results of the input in Figure 4.6 through three different partitioning strategies in the partitioning-oriented scheme (bold font for remaining structures and normal font for extracted units).

Partitioning-Oriented Scheme. In this scheme, the partitioning function $partition()$ is carefully designed to take advantage of the structure of input. Before regular parsing, the $partition()$ makes a quick parse of the input, called $preparse()$, to extract the input units from the original input by traversing the input up to a certain number of levels of structure. The extraction result is called *high-level structure* (level zero is the highest). Depending on the number of levels that $preparse()$ considers, there are two partitioning strategies: fixed-level partition, and flexible partition, as shown in Figure 4.7 .

The fixed-level partition extracts input units from some fixed number of levels of the input structure, while the flexible partition extracts an input unit if and only if its size exceeds a given threshold. Which one is better depends on the input structure. A good partition should meet two criteria: First, the extracted input units can be evenly grouped and distributed to a given number of parsers; second, the size of the high-level structure should be small as the extraction step is sequential.

After $parallelParsing()$, each parser outputs one or more small DOM trees, one for each input

unit. The parser that parses the high-level structure produces a special DOM tree, which offers the guide for merging the small DOM trees into a complete one. In this scheme, the $merge()$ function is straightforward to implement.

Merging-Oriented Scheme. The merging-oriented scheme follows an opposite design principle. In this scheme, the $partition()$ is simple, just cutting the input into chunks (almost) evenly. Note that the cut may break some pairs of tags. To handle this special kind of unmatched tags, the parsers record them during $parallelParsing()$, and resolve them during the merging stage.

In this work, we choose the merging-oriented scheme over the partitioning-oriented scheme for two reasons:

1. To take advantage of the structure of input, the partitioning-oriented scheme requires a $preparse()$ step, which sequentially scans the entire input. The time complexity is $O(n)$, where n is the number of total tokens[3]. In contrast, the merging-oriented scheme has no such requirement. Even though it has a more complex merging process, the merging cost is only $O(log_2 n)$ as proved later in this section.

2. Compared to the even cut of input in the merging-oriented scheme, the partitioning-oriented scheme is subject to load imbalance. When the high-level structure becomes the bottleneck, it may further worsen the issue.

Tree Merging-Based Parallel Parsing Algorithm (Temppa)

In this subsection, we introduce Tree Merging-Based Parallel Parsing Algorithm (*Temppa*) for LIST. Temppa is a merging-oriented scheme. Figure 4.9 shows the core of this algorithm. We will use the example in Figure 4.6 for our explanation.

At first, the input is partitioned into chunks. The partition tries to be even in size and meanwhile avoid breaking tag names. The example input in Figure 4.6 is partitioned into two chunks at the point right after "cartoon" at line 8. Then, for each chunk, the $parse()$ function creates a DOM tree, as shown as the left two trees in Figure 4.8. Each non-leaf node in a DOM tree represents a construct in the input LIST document. The node hence usually corresponds to the start tag and end tag of the construct in the input document. An example is the leftmost "item" node in Figure 4.8, whose start and end tags are the first "<item>" and "</item>" in the input shown in Figure 4.6.

During parsing, the $parse()$ function uses a stack to store incomplete tag tokens and pops them off when finding their matching tags. Some nodes in the tree remain incomplete to the end.

[3]Parallelizing the $preparse$ may help reduce the overhead, but only modestly due to the I/O-bound property of $preparse$ [115].

That happens when the input partition separates a pair of tags into two chunks (or errors in the input). For instance, the "item" node right above "cartoon" in the first tree is an incomplete node without an end tag (called a *end-missing node*), while the leftmost "item" node in the second tree is incomplete for missing a start tag (called a *start-missing node*). Other incomplete nodes in the two trees correspond to the lists enclosing the broken item. Note that all tokens with start tag missing are put as immediate children of the root and are always attached as the rightmost node (line 21 in Figure 4.9); this helps merging efficiency as discussed next.

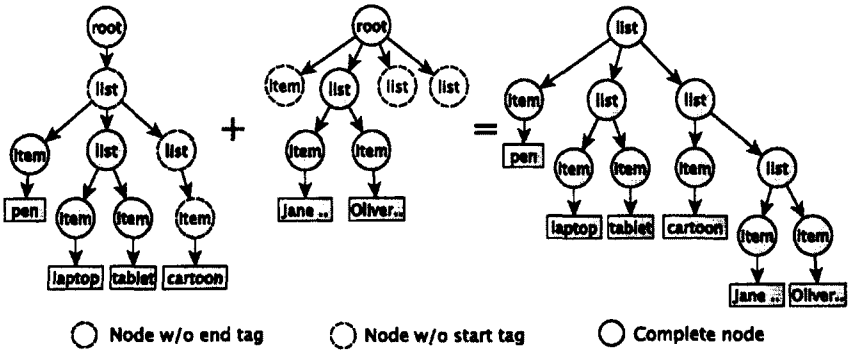


Figure 4.8: An example of *Temppa* algorithm.
The DOM trees produced and merged by the *Temppa* algorithm on the input shown in Figure 4.6. The input is partitioned right after the word "cartoon". Note that *root* is an assistant node and is removed after tree merging.

The $merge()$ function merges every tree into the first. The algorithm is designed by leveraging the following important property of the DOM trees:

Property 1. Single-path property: *For a LIST document with no missing tags, a DOM tree from Temppa can have at most one path that contains end-missing nodes.*

A path in a DOM tree here refers to a branch spanning from the root node to a leaf node. For instance, the branch from "root" to "cartoon" in Figure 4.8 is a path that contains three end-missing nodes. We call a path with end-missing nodes *an end-missing path*. The single-path property claims that one DOM tree cannot have more than one end-missing path.

This property can be proved easily. Suppose there are two such paths, and $node_i$ and $node_j$ are two different end-missing nodes in them respectively. Without loss of generality, assume that the start tag of $node_i$ appears before the start tag of $node_j$ in this chunk of the LIST document. There are only three possible cases regarding the end tag of $node_i$: It appears before the start tag of $node_j$, after the end tag of $node_j$, or in the between. The first case is impossible in our setting because $node_i$ would be a complete token as both its start and end tags appear in this chunk of document. The second case is not possible either because $node_j$ would be a child of $node_i$ (and hence appears on the same path) as its corresponding construct would be embedded

in that of $node_i$. The third case is impossible for a legal LIST document because it would violate the hierarchical structure of LIST documents. (If it does appear in an illegal LIST file, according to the autocorrection rules of HTML, the parser would create an end tag for $node_j$ before the end tag of $node_i$ and ignore the later appearance of the end tag of $node_i$. That would make $node_j$ a child of $node_i$.)

With that property, merging $tree_i$ to $tree_0$ just needs to check the end-missing path in $tree_0$ from bottom to top and change a node's status to complete if its end tag is found in $tree_i$. The algorithm is shown in Figure 4.9. Here we highlight several features designed for efficiency. First, during the check, the algorithm only needs to examine the immediate children of the root in $tree_i$ (line 40 in Figure 4.9) because the $parse()$ function puts all start-missing nodes at that level. Second, thanks to the way $parse()$ attaches nodes into a DOM tree, the left-to-right order of the start-missing nodes aligns exactly with the bottom-to-top order of the nodes in the end-missing path; both correspond to the inner-to-outer nesting levels of the broken tokens. Two trees are connected at line 48 in Figure 4.9. In our example, the top-most list node of the second tree is attached as a child of the rightmost list node at the third level of the first tree. After merging, the root node of the tree is removed.

Let n be the number of tokens the input document contains. The height of the complete DOM tree is $O(log_2 n)$, hence the complexity of the merging.

Temppa cannot directly apply to HTML5 due to the special complexities and dependences in HTML5 parsing. We next present how we address these complexities by integrating carefully designed speculation into Temppa.

4.3.2 HPar: Breaking Data Dependences with Mixed Speculation

In this part, we describe *HPar* (*H* for "HTML"; *Par* for "Parallel Parsing"), our data-level parallel HTML parser. *HPar* uses the merging-oriented parallel parsing scheme. As known, data dependence is the single important barrier for parallelization. The various types of complexities of HTML5, as listed in Section 4.1.2, are ultimately reflected by their impact on complicating data dependences in parsing. Our description will hence focus on how HPar breaks the many special data dependences through a set of effective speculations; the treatment of the many complexities of HTML5 is described along the way.

As Section 4.2 has mentioned, the dependences in HTML5 parsing can be categorized into tokenization dependences and tree construction dependences. Although they may interact with each other, we describe our solutions to them separately for clarity.

```
1  // run by each parsing thread
2  Node parse() {
3    stack = new Stack<Node>();
4    current = root;
5    while(true)
6      token = tokenize();
7      switch(token.type)
8        case startTag:
9          //Node(parent, data, isComplete)
10         node = new Node(current,token,false);
11         current.add(node);
12         current = node;
13         stack.push(node);
14         break;
15       case endTag :
16         if(stack.top == token)
17           current.isComplete = true;
18           current = stack.pop();
19         else
20           node = new Node(current,token,false);
21           root.add(node);
22         break;
23       default :
24         node = new Node(current,token,false);
25         current.add(node);
26   return root;
27 }
28
29 // runs on main thread
30 Node merge() {
31   for(i = 1; i < numThreads; i++)
32     root[0] = mergeTwo(root[0], root[i]);
33   return root[0];
34 }
35
36 // merge two incomplete trees
37 Node mergeTwo(Node root0, Node rooti) {
38   endMissingNode =
39           getLowestEndMissingNode(root0);
40   for(child in rooti.children)
41     if(child.StartMissing == true)
42       if(endMissingNode.name == child.name)
43         endMissingNode.EndMissing = false;
44         endMissingNode = endMissingNode.parent;
45       else
46         error("mismatched tags");
47     else
48       endMissingNode.add(child);
49   return root0;
50 }
```

Figure 4.9: The core of the *Temppa* algorithm.

Tokenization Dependences

Suppose an HTML file has been partitioned into chunks. To let an HTML5 parser process an HTML chunk, it needs to know the *starting tokenization state*. The real starting tokenization state depends on the parsing results of previous chunks. So before parallel parsing of HTML chunks, only the parser for the first chunk is sure of the real starting state, according to HTML5 spec, the DATA state; all the other parsers have to speculate their values. To speculate starting tokenization states, we employ a series of techniques, and combine them to achieve a high speculation accuracy.

a. Smart Cutting. A simple but efficient way to improve the accuracy is to select a good cutting point. The policy used in this work and some previous papers [115], is to always cut just

before (or after) a tag. It starts from an even partitioning point and looks forward until it finds the left angle bracket of a tag and uses that point as a cutting point. This strategy guarantees that no tags are broken, and limits the prediction targets to these restricted scenarios. As only a subset of tokenization states are feasible in these scenarios, the strategy simplifies speculation. It may cause some load imbalance, but by at most one token; the effect is typically negligible.

b. Frequency-Based Speculation. A straightforward way to get a statistically good candidate is to choose the state with the highest probability. We profiled the top 1000 popular websites [8], and collected the tokenization state distribution. In this distribution, the `DATA` state has the highest probability, 12.3%. In another word, choosing the `DATA` state as the starting tokenization state can achieve a 12.3% accuracy. However, when we apply the *smart cutting* to restrict the scenarios to predict, the `DATA` state appears to be the true state in 92.3% of the cases. So the *smart cutting* and this frequency-based speculation together offer a simple but effective way to achieve a 92.3% speculation accuracy.

c. Speculation with Partial Context. The previous two techniques explore static characteristics to help speculation. Although they work in most cases, they cannot handle cases when a comment or script is broken by the cutting. Although these two cases do not happen often in the parsing of most documents, they appear to be critical for some files where comments or scripts are used intensively. We use *partial context* to address these issues. The method exploits the conditional probabilities of partial context—which here refers to the prefix of HTML chunks. We explain the two cases as follows.

c.1) Handling Broken Comments. Broken comments happen when different parts of a comment are partitioned into different chunks. For example, when an HTML chunk ending with "`<!--<h2>heading`" is being parsed, its following input elements "`</h2>--><h1>`" are being concurrently parsed. The second parse would mistakenly consider the actually commented-out tag "`</h2>`" as a normal tag. The cause of this error is that the second parse starts from the default tokenization state `DATA`, while the real state should be `Comment`. To solve this problem, we introduce two new token types: `StartComment` and `EndComment`, so that the broken comments can be recognized individually and merged later. The construction of the `StartComment` is simply triggered by an encounter of "`<!--`", while an `EndComment` token is built when "`-->`" is encountered and this current HTML chunk contains no matching starting comment tag. The parser would include all the text from the starting of the chunk to this end comment tag into that `EndComment` token. Note that these two token types are intermediate representations. After merging, they will be replaced by `Comment` nodes.

The above technique has risks and benefits. As it solves the broken comments problem,

but also brings some side effect. For example, when parsing an HTML tutorial page, a parser thread would mistakenly consider the normal text segment "<p>-->" in text "<p>--> ends an HTML comment" as an EndComment. To solve this dilemma, we simultaneously maintain two alternative parsing results: normal interpretation and EndComment interpretation. Which one is correct depends on the parsing results of previous chunks. Specifically, if the prior chunk has a StartComment as its rightmost incomplete node, then EndComment interpretation is correct and used; otherwise, the normal interpretation is used. This approach safely avoids rolling back in either case.

c.2) Handling Broken Scripts. Another interesting case involves broken scripts. When a parser thread starts in the middle of a pair of script tags, it may not realize this until it reads a </script> token. During the merging of two incomplete trees, if the first tree has an incomplete script node without an end tag, while the first child of the second tree's root is an incomplete script node without a start tag, these two nodes will be merged to form a complete script node. But this is not the end of the story. Due to the broken script tags, the script cannot be executed until it is complete, hence a delay for its execution results to take effect. As the scripts may modify the HTML input and hence the DOM tree, the delay may cause side-effects: changing the application scope of the script. As shown in Figure 4.10, the script in the piece of HTML is supposed to change only the first two h2 tags, however, since its execution is after parallel HTML parsing, it changes the third h2 tag as well. The key to solving this issue is in remembering the application scope of scripts by inserting some anchors into the DOM tree. Each anchor indicates the end of the scope for a broken script to take effect. After the execution of a merged complete script, if it alters the HTML document, the parser reparses the affected chunks and remerges them into the tree. All anchors are removed after the executions of all scripts.

```
<h2>head 2.1</h2>
<h2>head 2.2</h2>
<script type="text/javascript">
var el = document.getElementsByTagName('h2');
for(var i=0; i<el.length; i++) {
   el[i].innerHTML='changed head';
}
</script>
<h2>head 2.3</h2>
```

changed head changed head head 2.3	changed head changed head changed head
correct display	**wrong display**

Figure 4.10: Script scope issue.
An example that shows the application scope of script may be changed by the delay of its execution.

c.3) Handling Other Broken Tags. There are some other cases that cause the speculation to fail. For example, as shown in Figure 4.3, when a pair of `style` tags are broken, the `DATA` will not be the true tokenization state. This can be easily fixed by cutting just before a tag, instead of after a tag, assuming the symbol < does not appear within `style` tags. Other cases, such as broken `svg` and `math` tags, are handled similarly.

Tree Construction Dependences

Besides tokenization dependences, we also need to solve the tree construction dependences. For a parser thread to process an HTML chunk, it needs to know the *starting insertion mode*. In this subsection, we show that the dependences can also be addressed through mixed speculations.

a. Before Speculation. Autocorrection is one of the key features in HTML parsing. This feature makes the HTML syntax more flexible and tolerant to some errors, but also makes parallelization difficult. For instance, the Temppa algorithm is supposed to attach the token of an unpaired end tag to the *root* node. But autocorrection may instead regard the tag as "errors" and let the parser ignore it. We adopt a simple solution that delays this type of autocorrections until the merging stage. The parser always attaches unpaired end tags to the tree, and removes them from the tree if it finds no matching start tags for them at the tree merging time.

The other type of errors handled by autocorrection is missing end tags (e.g., missing "`</li>`" in "`<li> item one. <li> item two.`"). By default, an HTML parser is supposed to autocorrect such errors by inserting some end tags according to some rules in the HTML5 specification. What end tags to insert depends on the current *insertion mode* and the top of an *open stack* that records all start tags not yet matched with end tags. In HPar, the *insertion mode* is determined through speculation as detailed next. As to the open stack, before starting parsing a chunk, a parser determines the top of the open stack by finding the last tag in the suffix of the previous chunk. If it is a start tag, the parser puts it to the top of the open stack, otherwise, it leaves the stack empty. During the parse, it uses the speculated insertion mode and the open stack to autocorrect missing end tags. If the autocorrection hits the bottom of the open stack, the parser conservatively assumes that the open stack does not contain enough information for the autocorrection, aborts, and waits for the previous parser to reparse this chunk. Our study shows that most missing end tags are "`</li>`", which were always fixed by the speculation scheme. No autocorrection-triggered abort is observed in our experiments.

b. Profile-Based Speculation. Similar to the tokenization state, we also profiled the distribution of insertion mode based on top 1000 popular websites [8]. We find that the `InBody` mode is the dominating mode, with a 51.3% probability. However, directly using `InBody` as the starting

insertion mode can cause some side effects, because of missing `html`, `head`, and `body` elements. For example, when a pair of *head* tags is broken, if the `InBody` mode is used, the parser that processes the second piece would mistakenly consider that all tags it encounters are within the scope of `body`, and attach the corresponding elements to `body`. To avoid this, we instead use the `Initial` mode as the starting state for the parsing of each HTML chunk such that the elements `html`, `head`, and `body` will be automatically inserted. And right after those insertions, the insertion mode will automatically switch into the most probable state, the `InBody` mode.

c. Speculation with Partial Context. Partial contexts can also help the speculation of insertion mode. We explain it by drawing on broken `table` tags as an example.

In the HTML5 specification, there are a set of insertion modes designed for parsing table-related tokens, including `InTable`, `InTableText`, `InRow`, and others. This set of insertion modes are important: They cover more than 38% of the insertion modes during the parse of the top 1000 websites.

When a cutting point is within a pair of `table` tags, the speculation with `InBody` as the starting insertion mode will fail. The true insertion mode depends on the position of the cutting point within the pair of `table` tags. So we profiled the relations between the relative positions of cutting point and the insertion modes, and found an interesting result: In the situations where `table` tags are broken, the correct starting insertion mode strongly correlates with the first table-related token in the current HTML chunk, as summarized in Table 4.2. We integrate the table into the parallel parser to help the speculation.

Table 4.2: Insertion Mode Prediction

Table-related tokens	Starting Insertion Mode
</td>	InCell
</tr>	InTableText
</th>	InCell
</thead>	InTableText
</tbody>	InTableText
</tfoot>	InTableText
...	...

For example, suppose a parsing thread starts in the middle of a pair of `table` tags. At first, it does not realize this, and starts with the `InBody` mode. After processing a number of tokens, it reads a token `</td>` without previously reading any start tag `<td>`. It realizes that earlier speculation fails. Based on Table 4.2, it picks the insertion mode `InCell` as the new speculation of starting insertion mode, and reparses its HTML chunk.

Even with all the techniques for effective speculation, mistakes can still happen in some cases.

Table 4.3: Benchmark size and speculation accuracy

website		youtube	bbc	linkedin	yahoo	amazon	qq	twitter	taobao	wikipedia	facebook	average
size (KB)		120	152	208	308	320	332	396	476	504	708	352
acc (%)	pipeline	99.9	99.9	99.9	99.9	99.9	99.9	100	99.9	99.9	99.9	99.9
	data-level	100	89.9	66.7	66.7	100	33.3	100	88.9	85.7	77.8	77.5

We treat them in an eager manner: As soon as the parser realizes that the speculation is wrong (by, for instance, encountering an end tag that is inconsistent with the assumed insertion mode), it immediately picks a more plausible speculation state and starts to reparse the chunk of input. The refinement of the speculation state can leverage everything the parser has observed so far. However, in our exploration, we found that the most useful info is the tag or token the parser encounters right before it realizes the error. For instance, if the tag is <meta>, it would speculate that the beginning of the chunk is more likely to be part of a `head` tag pair (than a `body` tag pair). The refined speculation may still need to be rolled back if it is mistaken. In the worst case, the mistake could not get fixed until the finish of the parsing of all previous chunks of input, when the correct starting states for parsing this chunk become explicit. But our experiments show that in practice, the refinement is quite effective in fixing speculation errors, as shown by the results in the next Section.

4.4 Evaluation

We implemented both the pipelining and data-level parallelization (HPar) in *jsoup* [5], an open-source standalone HTML5 parser that implements the *HTML5 spec*. This actively maintained open-source parser is written in Java. It is the most popular open-source standalone HTML parser in the community [4].

Benchmarks. We use real HTML pages as our benchmarks. They are downloaded from popular websites [8], including *youtube.com*, *bbc.co.uk*, *linkedin.com*, *yahoo.com*, *amazon.com*, *qq.com*, *taobao.com*, *wikipedia.org*, and *facebook.com*. From each of them, we collect one page that contains some substantial content[5]. These pages are mostly the first page showing up after logging into the web site. Their sizes range from 120KB to 708KB (352KB on average) as shown in Table 4.3.

Platforms. We use three types of devices with different degrees of portability: a Linux server, a laptop, and a tablet, as listed in Table 4.4.

Methodology. To measure the steady-state performance, we repeated each run for ten times after a warm-up run on the Java runtime. We observed some variations among the running

[4]Its popularity is reflected by the large amount of discussion on it in www.stackoverflow.com.
[5]The whole set of webpages are accessible at http://www.cs.wm.edu/caps/projects/hpar/.

Table 4.4: Evaluation platforms

	MacBook Pro	Nexus 7	Linux Server
CPU	Intel Core i7	Nvidia Tegra 3	Intel Xeon E7
#cores	4 cores	4 cores	10 cores
Last-level shared cache	8MB	1MB	20MB
OS	OS X 10.7	Android 4.1	openSUSE 12.1
JVM	HotSpot Server	Dalvik	OpenJDK Sever

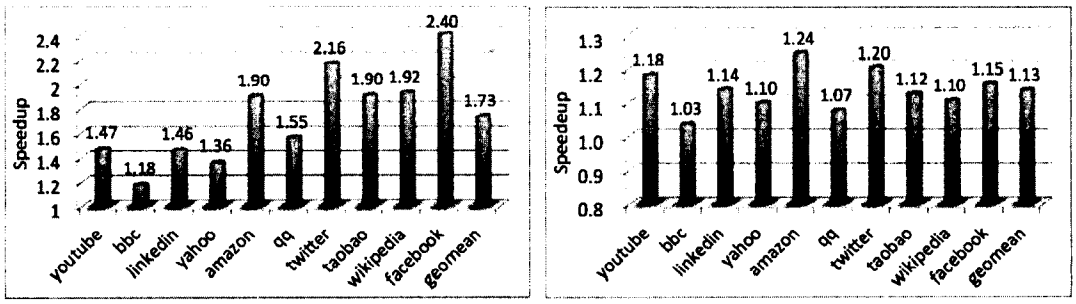
times, especially on the portable devices due to the more prominent influence of its many system activities. We used the shortest time for all versions (including the baseline sequential version) in their repetitive runs. The reason for such a policy is that our focus is on the evaluation of the effectiveness of the parallelization techniques. In these runs, the interference from system-level noises is minimal; the time can hence better reflect the actual effects of the parallelization techniques. All speedups are for the parsers' execution time.

4.4.1 Speculative Data-Level Parallelization

Figure 4.11(a) reports the speedups of our speculative data-level parallel parser, HPar, on MacBook Pro. The baseline is the default sequential parser from *jsoup*. The rightmost bar reports the geometric mean of the results in all benchmarks.

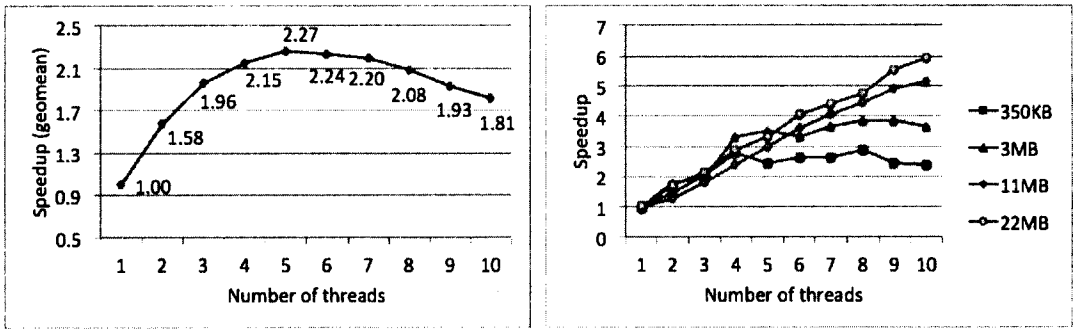
On the MacBook Pro with four cores, HPar achieves speedups as high as 2.4x. Its average speedup is 1.73x. Our analysis shows that the sublinear speedup is due to three main resources. The first is the tree merging step. As a sequential step, it takes about 5% the overall parse time in a eight-thread case on Macbook Pro (and 25% on Nexus 7 with four threads). The second is the overhead in the parallelization, including thread creation and communication, and the rollback overhead when the speculation fails. Our speculation success rate is 78% on average, shown in Table 4.3. The benchmark qq has the largest speculation error because the a majority of the cutting points happen to fall into Javascript code embedded in it. Wrong speculations cause some part of input to be reparsed, which generates some but not much overhead because the parser realizes and fixes the speculation error quite early such that the reparsed segments are on average only 4.6% of a chunk of input. Yahoo is an exception. Two chunks of it are entirely Javascript code and the speculation errors were not realized until the end. But Javascript sections do not need to be reparsed by HTML5 parsers. So the errors did not incur much overhead. The third source of overhead is the contention in system resources (e.g., shared cache, memory controllers, etc.). As the first two sources typically do not increase much when the input size increases, we expect that when the input becomes larger, the speedup would increase as well. It is confirmed by the scalability study described next.

Scalability Study Due to the limited number of cores on current portable devices, we conduct the scalability study on a dedicated server equipped with ten cores as listed in Table 4.4. Figure 4.12(a) shows the average speedup of the ten benchmarks as the number of cores grows. The peak speedup appears when five cores are used. An analysis of the individual benchmarks shows that the *facebook* benchmark is the one that has the best scalability, with the speedup up to 2.93x. Considering that *facebook* is the largest benchmark of all, we speculate that the benchmark size could be an important factor for scalability. To confirm it, we create a series of web pages of a spectrum of sizes by duplicating some web pages multiple times. Figure 4.12(b) reports the speedups of HPar on these web pages when different numbers of cores are used. The scalability increases as the input grows, confirming our speculation. As the trend is that web pages are becoming increasingly complex and large, the results suggest that even larger benefits of HPar may be expected for future web pages.



(a) MacBook Pro (b) Nexus 7

Figure 4.11: HPar speedup.



(a) Speedup scalability (b) Impact of input size on scalability

Figure 4.12: Scalability of HPar.

The speedup on the Nexus 7 tablet is reported in Figure 4.11(b). It is relatively modest: up to 1.24x with an average of 1.13x. System resource contention in the parallel runs is the main reason for the speedup to be lower than on the MacBook Pro. To confirm it, we create an artificial scenario that has full parallelism and little parallelization overhead by running four standalone copies of the

sequential parser on the device. We see 2.3x slowdown compared to the performance of the sequential parser when it runs alone. The larger impact of resource contention is due to the much more limited L1 and L2 cache and memory bandwidth on Nexus 7 than on MacBook Pro (e.g., 1MB v.s. 8MB of L2 cache). As handheld devices are evolving rapidly, we expect that the speedup will get closer to that on MacBook Pro as these devices become more powerful in the near future.

4.4.2 Speculative Pipelining

We have also measured the benefits of the speculative pipelined HTML5 parser. We found that its benefits are much less than HPar. Figure 4.13 shows the speedup (or slowdown when the value is lower than 1) on the ten benchmarks. The baseline is also the default sequential parser from *jsoup*. There are three bars for each benchmark. The first one is the speedup on the MacBook Pro. There is no speedup but an average 8% slowdown. Load balance is not a main issue as the two stages of the pipeline appear to run for a similar amount of time for most benchmarks. Speculation error is not an issue either as shown in Figure 4.3. Our analysis show two main reasons for the slowdown. First, the pipeline has only two stages and hence has limited parallelism. Second, there is some overhead for threads creation and synchronization and scheduling, among which, the main overhead turns out to be the data transfer from the tokenization thread to the tree builder thread. We tried to change the transfer to a batch fashion rather than in each token, but observed no performance benefits. We also tried a spectrum of token buffer sizes to measure its impact on the performance. Figure 4.14 shows the geometric means of the parsing times of all the web pages when different buffer sizes are used. There is no clear correlation between the parsing time and the buffer size, offering the supporting evidence for using 1-token buffer in our design.

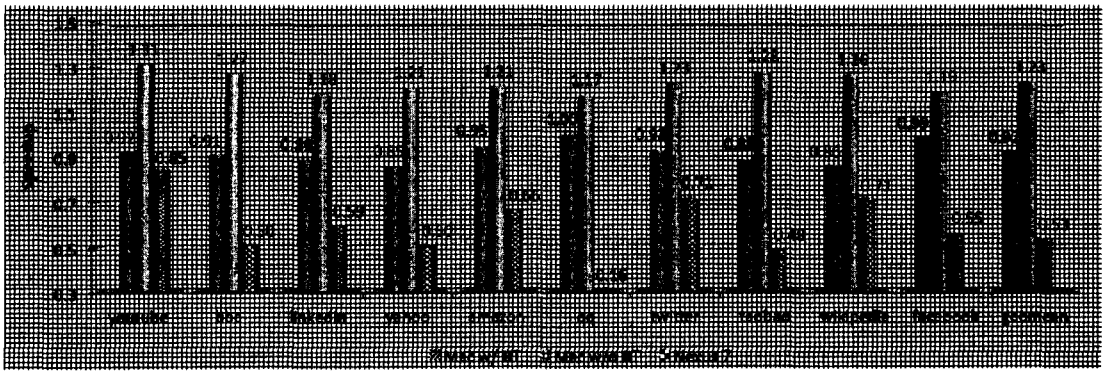


Figure 4.13: Speculative pipelining speedup

To further analyze the tradeoff between time savings and overhead, on the same platform, we

conduct another set of experiments. The Java runtime on the laptop allows two modes of code translation, through either an interpreter or the adaptive Just-in-Time (JIT) compiler. By default, the JIT is enabled as it usually provides better code and hence higher performance (as done in the experiment described in the previous paragraph.) However, in this experiment, we deliberately disable the JIT to see the changes. The second bar in Figure 4.13 for each benchmark shows the speedups of the pipelined parser compared to the sequential one when JIT is disabled for both. The pipelined parser shows an average 1.29x speedup. Such a dramatic change is because without JIT compilation, the parser runs for a much longer time. So the memory accessing overhead becomes less critical. In another word, the parsing becomes more CPU bound, and hence the savings by the parallelization outweigh the overhead.

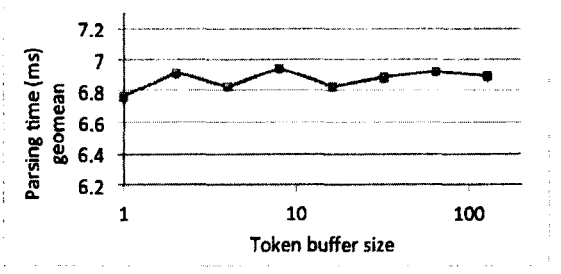


Figure 4.14: Performance impact of token buffer size.

The comparison between these two scenarios is mainly to confirm that the pipelined parser works properly, and to help understand the tradeoffs between its savings and overhead. For practice, the results suggest that the pipelined HTML parser is not beneficial on the Macbook Pro—despite the speedups when JIT is disabled, the fastest version is still the default sequential parser with JIT enabled.

The third bar of each benchmark shows the speedup on the tablet Nexus 7. The Java runtime is Dalvik with JIT. As the memory bandwidth of the tablet is even smaller than the Macbook Pro, it is expected to see even lower performance of the pipelined parser, given the memory-bound property of the program. The results confirm the expectation: greater slowdown are shown in the experiments.

Overall, the results indicate that the pipelining parallelization is not a viable way to develop parallel parsers for HTML.

4.5 Potential Benefits Beyond Parsing

The benefits of parallel HTML5 parsing go beyond parsing itself. It may open up new optimization opportunities for modern web browsers. Here we list two examples.

Parallel Object Downloading. HTML documents usually contain various objects, such as images, stylesheets, Javascript code. These objects are usually not downloaded until the parser reaches the links to those objects in the HTML document. In HPar, the parsing threads start at multiple points in an HTML document; these objects can hence be detected earlier and downloaded speculatively in parallel. A complementary approach to enabling early downloading is prescan [34], which scans and parses the input at a high level. It has been a sequential process, suffering substantial overhead on large webpages. The techniques of HPar may be combined with prescan—such as, creating a parallel prescan—to minimize the latency.

Parallel Javascript Execution. In our current implementation, the Javascript code in a webpage is executed in the default manner—typically sequentially. But as multiple pieces of Javascript code can now be recognized in parallel and objects embedded in different parts of a webpage could be downloaded concurrently, it is possible for several pieces of Javascript code in a webpage to be launched in parallel. There are some complexities in this optimization though. A piece of Javascript code may depend on the parsing threads, or another piece of Javascript code. Hence, to capitalize on these opportunities, one must not ignore these subtle issues.

4.6 Related Work

4.6.1 Parallel Browsers

Recent years have seen some increasing interest in parallelizing web browsers. Web browsers include many kinds of activities. Prior efforts have concentrated on creation of web page layout [24, 78], lexing [59], Javascript translation and execution [55], web page loading [106], XML parsing [73, 115], and web page decomposing [76]. Industry (e.g., Qualcomm and Mozilla) has began designing new browser architectures [34, 82] to leverage these results or better exploit task-level parallelism. These studies have not focused on parallelizing HTML parsing, an important component for the response time of a browser. Integration of HPar into a browser is ongoing work.

4.6.2 Parallel Parsing

Many studies have been devoted to parallel parsing between 1970's and 1990's. They fall into two classes: parallel parse of programming languages or natural languages. For programming languages, arithmetic expressions and bracket matching are mostly studied and evaluated. A pipeline is used between lexing and parsing [92], or even among lexing, parsing and semantic

processing [25]. For parsing itself, the parallelism can come from either grammar decomposition or input decomposition [13]. Fischer's seminal work [47] proposes a parallel parsing algorithm based on LR parsing. Given a starting point of the string, the algorithm uses a number of sequential parsers for every possible parsing state, and maintains a parsing stack for each parser. Prefix-sum algorithm is also applied to parallel parsing problems to compute the effects of a series of stack transitions [100]. These approaches suffer from scalability issues for the rapid growth of the possible states and transitions of the stack as the vocabulary and inputs increase. In grammar decomposition, the grammar is decomposed such that each processor is in charge of a part of the grammar [21, 22]. The decomposition exploits limited parallelism compared to input data decomposition. Luttighuis leveraged parallel bracket matching algorithms for parallel parse of some special subclasses of context free language [75]. In natural language processing, many efforts have been put to CYK [61, 113, 114] and Earley [43] parsing algorithms. A recent work has examined how to synthesize a parallel schedule of structured traversals over trees, offering some help for parallel CSS processing [79]. These previous studies have laid the foundation for this work. However, most of them have focused on languages defined with clean formal grammars. They have not been able to be applied to HTML successfully, as Section 4.1 has discussed. Natural languages are also defined in an ad-hoc manner. But they have a whole set of different complexities from HTML. The parallel parsing techniques developed for natural languages have a high time complexity, unsuitable for programming languages.

Parallel XML Parsing. The two basic ways of data-level parallel parsing—the partition-oriented and the merging-oriented— have been applied for XML parallel parsing [73, 86, 99, 115]. The work by Lu and others, for example, first extracts the high-level structure of XML documents through a quick prescan [73], and parses each part of the document in parallel. The work by Wu and others [115] cuts XML documents into chunks, parses them and merges the result together. HTML5 parsing has many special complexities compared to XML parsing. In fact, all the major complexities listed in Section 4.1.2 do not present in typical XML documents, and hence have been ignored by previous studies in XML parallel parsing. For example, none of them handles error-correcting, embedding of other languages, and executable scripts that modify the document it resides in. As we have discussed in Section 4.3.2, these complexities cause complicated effects on the data dependences in HTML5 parsing, and hence the many difficulties for parallelization. They prompt our development of the speculative techniques described in this work. In contrast, simple methods have been used in prior XML parallel parsing studies. The work by Wu and others [115] simply cuts input right after a right angle bracket and let every thread parse their chunk from initial state. They do not consider the failing cases, in which the cutting point is actually

inside a string of comment, in the middle of a piece of Javascript code, or after an erroneous ending tag. They do not have the sophisticated speculation designs or error handling schemes as we present. Nor do they handle modifications of the document by the scripts it contains. It is exactly these special complexities that have been the key barriers for HTML parallel parsing. Effectively handling them to enable HTML parallel parsing is the distinctive contribution of this current work.

There are some prior studies on using context to help parse languages that allows embedding of domain-specific languages [112]. They are for enhancing sequential parsers rather than parallelization.

To the best of our knowledge, this work is the first study that systematically investigates the special complexities in parallel HTML parsing; by developing a set of speculation-centered solutions customized to HTML parsing, it creates HPar, the first practical parallel parser for HTML.

4.6.3 Speculative Execution

A number of studies have explored speculative execution techniques for solving challenging parallelization problems, at both the levels of programming languages and compilers [41,46,91,93,95] and architecture designs [39, 77]. To simplify programmers' task, Prabhu and others proposed two new language constructs to parallelize applications with speculation [91]. Some work uses software speculation to parallelize certain regions of the applications. This can be implemented either at process level [41] or thread level [46,95]. These automatic or semi-automatic approaches for general application parallelization have their limitations. They usually focus on loops, and the dependences that they can deal with are relatively simple. A recent study introduces rigorous analysis to DFA speculation [118].

4.7 Summary

This work presents the first systematic study in taming the complexities of HTML5 and developing speculation-centered techniques to create parallel HTML parsers. The outcome includes a set of insights on effectively parallelizing HTML parsers, and HPar, the first practical HTML parallel parser that yields up to 2.4X speedup (1.73X on average) on quad-core devices. This study breaks a challenging barrier for fully parallelizing web browsers, improves the efficiency of a critical component in modern web browsers, and opens up new opportunities for browser optimizations.

5 Call Sequence Prediction and Parallel JIT Compilation

Predicting a sequence of upcoming function calls is important for optimizing programs written in modern managed languages (e.g., Java, Javascript, C#.) Existing function call predictions are mainly built on statistical patterns, suitable for predicting a single call but not a sequence of calls. This paper presents a new way to enable call sequence prediction, which exploits program structures through Probabilistic Calling Automata (PCA), a new program representation that captures both the inherent ensuing relations among function calls, and the probabilistic nature of execution paths. It shows that PCA-based prediction outperforms existing predictions, yielding substantial speedup when being applied to guide Just-In-Time compilation. By enabling accurate, efficient call sequence prediction for the first time, PCA-based predictors open up many new opportunities for dynamic program optimizations.

5.1 Problem Definition and Design Considerations

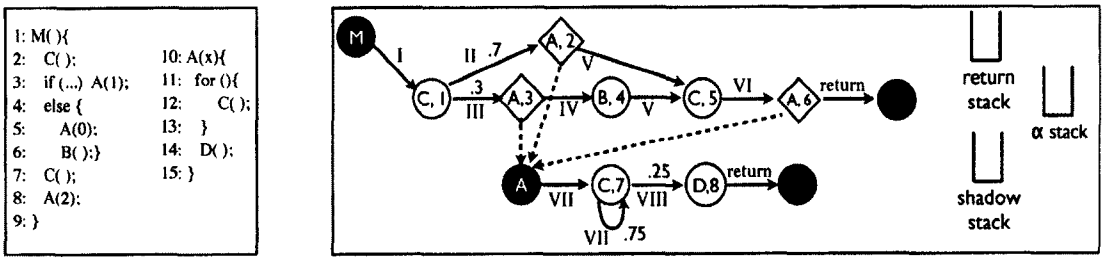
As the problem has not been systematically explored before, we first provide a formal definition as follows.

5.1.1 Definition of Call Sequence Prediction

Definition 9. *A* **function call sequence** *of a program is a sequence of the IDs of the functions in the order of their invocations in an execution time window.*

Definition 10. Call Sequence Prediction: *For a given execution of program P, function call sequence prediction at a time point t is to predict the function call sequence of P in the time window that immediately follows t.*

The time window is called *prediction window.* Its length is usually in logical time (e.g., the number of function calls), and may be fixed or vary. Depending on the windows' length, the prediction



(a) "PCAExample" program

(b) PCA

Figure 5.1: An example program named "PCAExample" and its PCA.

may happen many times during an execution of a program. For a multithreading execution, the prediction can be at the whole program level including all threads, or at the level of one or several specific threads.

Function call sequences are largely dictated by program structures. A primary goal of this work is to examine how to leverage program structures for call sequence prediction. Conceptually, the problem is to develop a representation that is effective in capturing the relevant constraints on call sequences coded in a program.

5.1.2 Design Considerations

Ensuing Relations vs. Calling Relations There are some commonly used inter-procedural program representations, such as call graphs, call trees and calling context trees. They primarily represent *calling relations* among functions. But a call sequence is about which call follows which—and hence an embodiment of a series of *ensuing relations*.

It is important to note the differences between these two kinds of relations. Calling relations are about what functions would be called **by** what functions, while ensuing relations are about what function would be called **right after** what function calls. Calling relations affect ensuing relations. Knowing Y as one of the callees of X, for instance, suggests that Y will be, with some uncertainty in the presence of branches, called after a call to X. But when that call will happen is not coded in the calling relation: It could be immediate, several or hundreds of calls later. An example is the call of "D()" by "A()" in the *PCAExample* code in Figure 5.1 (a). For the loop before "D", there could be zero or thousands of "C" being called before "D" is called. Conversely, two adjacent function calls in a call sequence, say "V W", does not entail that W must be called by V: That call to W could be made by V, its caller, or any of its calling ancestors— respectively illustrated by "A C" in lines 10 and 12, "C D" in lines 12 and 14, and "D B" in lines 14 and 6 in the PCAExample.

Four Basic Properties to Consider So the first consideration in our design of program representation is that it must capture ensuing relations of function calls. Ensuing relations naturally relate with program control flows (e.g., branches, loops), and often differ from one call site to another and from one calling context to another. So the representation should also consider these factors. In addition, to be used in runtime call sequence prediction, the representation should be reasonable in size, resilient to program complexities, and applicable to most executions of a program. We put these considerations together as follows, and call them the *four basic properties*:

- *Ensuing relations*: capturing ensuing relations among function calls;
- *Discriminating:* discerning different control flows, call sites, and calling contexts[1];
- *Generality:* being resilient to program complexities, such as ambiguous calling targets in the presence of virtual functions, function dynamic dispatch, and so on;
- *Scalability:* having a bounded and acceptable size, regardless of the program execution length.

Existing representations were designed mainly for program analysis rather than call sequence prediction. They center around calling relations, and fall short in some of the four basic properties (detailed in Section 5.3). Because of their insufficiency, we propose the following design of PCA.

5.2 Probabilistic Calling Automaton (PCA)

Intuition PCA is in an augmented form of finite state automata. Before defining it formally (in Section 5.2.6), we first offer some intuitive explanations. We choose automata as the basic form for their natural fit for expressing ensuing relations. For instance, Figure 5.2 shows an automaton for code "A( ); B( ); A( ); D( );". The four nodes represent four stages of an execution of the code. The DFA can easily track the execution through state transitions: Upon the first call of the function "A", it moves to state 1, and then to state 2 after "B" is invoked, and so on. The structure of the DFA reflects the ensuing relations of function calls imposed by the control flow in the function—in Figure 5.2, a constraint is that "A D" but not any other sequences immediately follows the call of "B". With this DFA, call sequence prediction becomes a simple walk over the DFA. For instance, suppose the DFA is now at node 1. To predict the remaining call sequence, we can simply walk along the DFA from node 1 and output the functions on the edges we bypass ("B A D" in this example.)

[1] Here, calling context refers to the sequence of functions on the current call stack. A more precise context also includes parameter values, which further complicates the problem. It is out of the scope of this work.

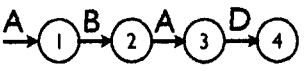


Figure 5.2: A DFA for code "`A(); B(); A(); D();`".

This example is simple, but conveys the basic idea of PCA: Incorporating constraints defined by program code into a finite automaton and converting call sequence prediction into a walk over the automaton. For the idea to work, there are many challenges, some from program structures (e.g., branches, loops), some from language implementations (e.g., function dynamic dispatching), some from compiler transformations (e.g., function inlining and outlining). PCA addresses these challenges through a careful design.

To make the explanation easy to follow, we first draw on an example (*PCAExample*) rather than formalism to explain our PCA design and how it addresses various complexities for runtime call sequence prediction. After that, we provide a formal, rigorous definition of PCA, along with the algorithm to construct it automatically.

5.2.1 Structure of PCA

A PCA consists of a number of finite state automata, and three types of stacks. There is one automaton for each non-leaf function in the program. (A function is a *leaf* function if its code contains no function calls; it is *non-leaf* otherwise.)

Nodes Each node in a PCA automaton corresponds to one call site in the function. If the invoked function is non-leaf, we call the node a *diamond*, otherwise, a *circle*. Each node carries a label, written as ⟨FunctionID, CallSiteID⟩, where "FunctionID" and "CallSiteID" are the ID of the function invoked at that call site and the ID of the call site itself. (A unique ID is assigned to every function and every call site.) A diamond carries an extra field, recording the address of the entry point of the automaton of the function called at the call site represented by the diamond. This field allows smooth transitions among automata of different functions.

When the "FunctionID" at a call site is either non-unique or unknown at the PCA construction time, "*" is used for that field of the node. Such a node is called a v-*node* (v stands for virtual function). A v-*node* can be either a diamond or circle. The abstraction of v-*nodes* is important for treating ambiguous function calls as Section 5.2.4 will show.

An automaton has a single entry node, and a single terminal node. They correspond to no call site, just indicating the entry and exit points of the automaton respectively.

Edges Edges in a PCA represent the ensuing relations among the function call sites contained in a function. There is a directed edge from node A to node B in an automaton if after A's call site (i.e., the call site represented by node A) is reached in an execution, node B's call site could be reached before any other call site in that automaton is reached. Note that some call sites in other automata could be reached between them. An example is the call of "A" on line 5 and the call of "B" on line 6 in Figure 5.1 (a). The latter immediately follows the former and hence there should be an edge between their nodes, despite that the callees of function "A" are reached between the calls to "A" and "B".

Each edge carries a label and a weight. The label is the ID of the sink node's call site. It gives conveniences to tracking program state transitions in a call site discriminative manner as we will see later. The weight is the probability for the sink to follow the source in the program's executions. An edge flowing into a terminal node can have only "return" or "exit" as the label, indicating the exit of the function.

Stacks There are three stacks, associated with the entire PCA of a program. They are the *return stack*, *shadow stack*, and α-*stack*. The first two are designed to provide discrimination of calling contexts, explained in Section 5.2.2. The third helps handle unexpected function calls for practical deployment of PCA, explained in Section 5.2.4.

Example Figure 5.1 (b) shows the PCA of the PCAExample code in Figure 5.1 (a). The top part shows the automaton of function "M". It contains three diamonds, all representing calls of the non-leaf function "A" at the bottom. The three dotted lines are not PCA edges, but illustrations of the three diamonds' references to the entry of A's automaton. Entry and terminal nodes are shown as disks. Each node has its ID labeled. For instance, the diamond on top has a label "A,2", meaning that this call site ID is "2" and the call is to function "A". The edge from node "C,1" to "A,2" has label II as it is the ID number of the call site represented by the sink node. Its weight ".7" indicates that 70% instances of the call site 1 are immediately followed by a call made at call site 2 in function "M". Weights equaling 1 are not shown for readability. Theoretically speaking, the call site ID needs to be labeled only on either the edge or in the sink node. Having the label at both places is for conveniences.

5.2.2 Basic Usage for Tracking and Prediction

The design of PCA makes it handy for efficient tracking the state of a program execution and predicting its upcoming function calls.

Tracking Execution State To track the execution state of a program through PCA, we just need to let PCA transit to its next state upon every function call in an execution. An exit or return prompts a transit to its terminal state. When reaching a diamond node, the transition immediately moves to the entry node of the corresponding automaton. For instance, a call at line 3 of PCAExample makes the PCA move from state "C,1" to "A,2", and then immediately to the entry node of the automaton of function "A". State transitions when a PCA reaches a terminal state are facilitated by the *return stack*. When the PCA transits from a diamond node to the entry of another PCA, the address of the diamond node is pushed into the *return stack*. When the PCA reaches the terminal state, it pops the diamond node address out of the *return stack* and jumps to that address immediately.

The runtime tracking requires some code to be instrumented at function call sites. To minimize the overhead, the inserted code only puts the ID of the function and call site (or a predefined numerical ID for "return") into a buffer. At the beginning of a prediction, the buffer is consumed to bring the PCA status to date.

Predicting Call Sequences Call sequence prediction by PCA is a quick walk over the PCA while outputting the IDs of the functions in the passed nodes. The *return* and *shadow* stacks make it possible for the prediction to discriminate call sites and calling contexts. When a sequence prediction starts, the shadow stack gets a copy of the content of the return stack to attain the current program state to work with. When a sequence prediction finishes, the shadow stack is emptied.

Figure 5.3 illustrates how the PCA in Figure 5.1 (b) supports the prediction process of PCAExample. The gray color indicates the time when the stacks are inactive. When the program starts, both stacks are empty and the PCA is at the starting state, state M. After line 3 (C and A are called), the PCA moves to state "A" and the diamond node "A,2" is pushed into the return stack. After a call to C on line 12, the PCA gets to state "C,7". It is assumed that the runtime now starts a call sequence prediction. The shadow stack attains a copy of the content of the return stack and becomes active, while the return stack pauses its operations. The predictor starts walking on the PCA from the current state, state "C,7", which has two outgoing edges. Suppose that the predictor takes the backedge (which carries a call to "C") three times before taking the edge (carrying a call to "D") towards node "D,8". That walk yields the predicted sequence "C C C D". As node "D,8" leads to a terminal node, the shadow stack pops out node "A,2" and the prediction walk immediately jumps to that node. It is assumed that the walk then takes the edge to node "C,5" and then to node "A,6" and outputs "C A" as the prediction. It then gets to node "A" again

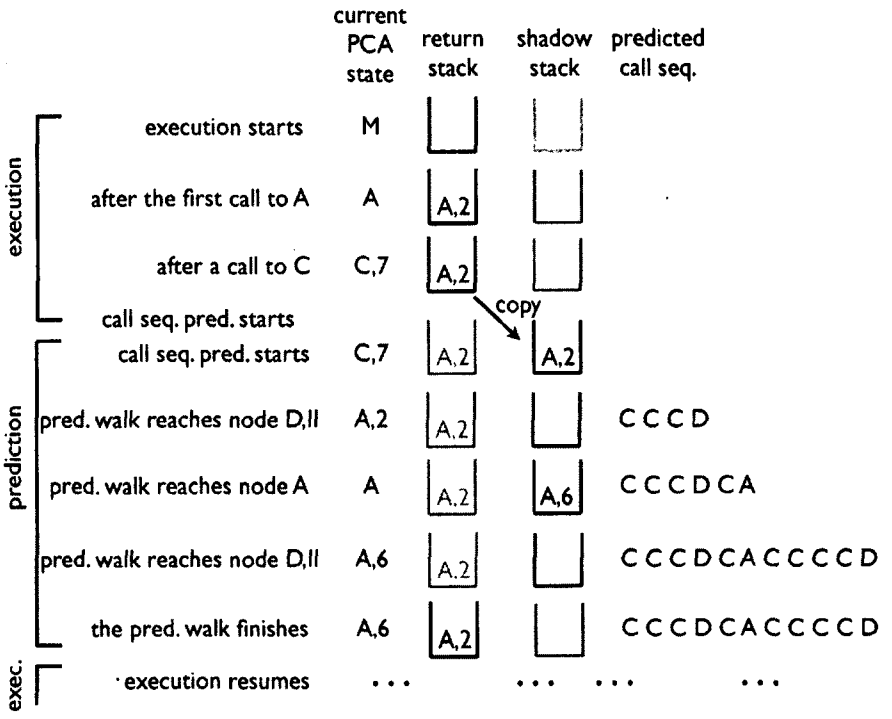


Figure 5.3: Illustration of how the PCA in Figure 5.1 (b) supports call sequence prediction. It assumes that the prediction starts after execution sees the call sequence "M C A C" and the goal is to predict all remaining function calls.

and continues the prediction. When the prediction finishes, the shadow stack is emptied and the return stack becomes active again.

The example touches one type of ambiguity in PCA: A node has more than one outgoing edge, as exemplified by nodes "C,1" and "C,7". We call this *edge ambiguity*. Edge weights provide probabilistic clues on resolving the ambiguity. We experiment with two policies for exploiting the hints. The first is the *maximum likelihood (ML) approach*, which always selects the edge with the largest weight. The second is *random walk*, which chooses an edge with a probability equaling the weight of that edge. For a node with k outgoing edges, the approach works like throwing a k-sided biased dice, the biases of which equal the edge weights.

The ML approach seems to be subject to loops: A backedge with a high probability may trap the predictor into the loop[2]. However, when using PCA for call sequence prediction, the runtime queries the PCA occasionally. Hence, even though PCA might predict a seemingly-infinite loop, continued execution of the real program generally results in escaping the loop. A subsequent PCA query would then ask about execution following the loop. In practice, it outperforms random walk in most cases as Section 5.5 will show.

[2]If the edge weights get appropriately updated across iterations, ML may not face such a problem.

/* a is an array of Animal that has a virtual function "voice()"; * class Animal has subclasses Cat, Dog, and Sheep.*/

```
1: for (i=0;i<N;i++){
2:    F();
3:    a[i].voice();
4:    G();
5: }
```

Figure 5.4: An example of dynamic dispatch for polymorphism

5.2.3 Challenges for Practical Deployment

The aforementioned basic usage of PCA for prediction has two implicit assumptions:

1. *Known-ID condition:* The PCA construction can completely determine the ID of the function to be invoked at every call site.

2. *Completeness condition:* The PCA captures all possible and correct ensuing relations among function calls of a program.

The two conditions ensure that all call sequences occurring in an execution would be expected (and hence processable) by the PCA. However, in many practical cases, the two conditions do not hold due to the complexities in language implementation, compiler optimizations, and PCA construction process. We will base our discussion mainly on a managed programming language (e.g., Java). Other types of languages (e.g., C/C++) share some of those complexities.

Function Dynamic Dispatch The *known-ID condition* does not always hold in the presence of function dynamic dispatch, with which feature, what function is called at a call site may remain unknown until the call actually happens. It often relates with polymorphism. For instance, suppose Cat, Dog, and Sheep are all subclasses of Animal, and they all have their own implementation of the virtual function "voice()" in Animal. The call to "a[i].voice()" at line 3 in Figure 5.4 may actually invoke the "voice()" function of any of the three classes, depending on which subclass $a[i]$ is. Another common cause of dynamic dispatch is function pointers, whose values may not be precisely determined at compile time in a C program. No matter what the implementation is, a common property of dynamic dispatch is that the exact function to be invoked at a call site sometimes cannot be determined until the call happens. As an analogy to the *edge ambiguity* mentioned earlier, this issue can be regarded as *node ambiguity*. It is embodied by v-nodes in a PCA, the labels of which have "*" as the FunctionID.

Compilation Complexities As Section 5.2.6 will show, PCA construction usually happens through some training runs with the help of compilers. In a managed environment, the compilation is through a JIT compiler, and typically happens in every run. The compilation may differ in different runs, causing different ensuing relations among function calls, and hence the violation of the *completeness condition*. For instance, function inlining replaces a call site with the code of the callee, while function outlining forms new functions in the binary code. So different inlining and outlining decisions in different runs could lead to different sets of call sites and ensuing relations.

Furthermore, training runs and production runs may have a different coverage of the code. Some functions invoked in a production run may have never been encountered by the JIT compiler in training runs, and hence may not appear in the constructed PCA. The training process could aggressively apply JIT to all possibly invoked functions, no matter whether they are invoked in the training runs. However, due to ambiguity in calling targets, it could end up including too many irrelevant functions (e.g., an entire library excluding library calls to JNI, which are not JITed).

Exception Handling Exception handling causes violations to both conditions. In Java, it is usually implemented with static exception tables, which, similar to function pointers, cause fuzziness in function calling targets. At the same time, some exceptions (e.g., division by zero) are not checked. Similar to signal handlers in C code, there may be no explicit calls to those handlers in the code, forming violations to the *completeness condition*.

Moreover, sometimes users may not be concerned of all functions. They, for instance, may not be interested in the invocations of functions in the Java Runtime but only those in the application. The bottom line is that some kind of resilience to the incompleteness of PCA and node ambiguity would be necessary for a practical deployment of PCA.

5.2.4 Solutions through v-Node and α-Stack

Features of v-nodes help address the issues related to the *known-ID condition*. Each v-node is equipped with a *candidates table*. Every entry in the table indicates the possibility for that call site to be an invocation of a particular function. A threshold K is used to control the size of the table. Only the top K most likely candidates appear in the table. Figure 5.5 shows the PCA for a variant of the "M" function in our PCAExample, in which, the call to "A" at line 3 is replaced with a function pointer whose most likely calling targets are functions "A" and "B". Besides them, there is another 15% chance for the target to be some other functions. The probabilities of candidate targets are obtained through offline profiling, but adjustable at runtime as explained later.

During sequence prediction, the candidate table is used for speculating on the ID of the func-

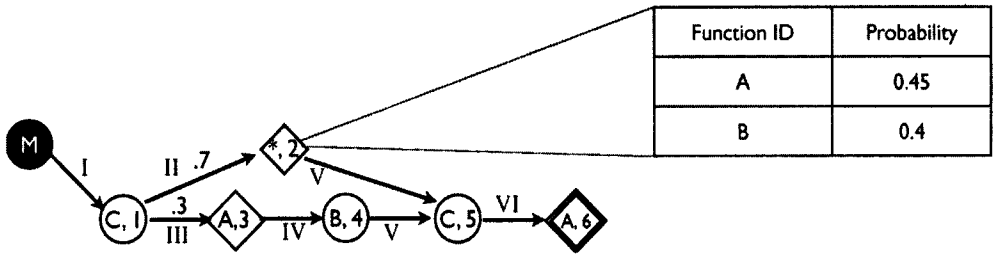


Figure 5.5: Example PCA with v-nodes Used.

tion to be invoked at the corresponding call site. The speculation employs the same methods as in resolving edge ambiguity (i.e., the ML or random walk method). The speculation happens every time when the prediction-oriented PCA walk reaches an v-node.

The issues on *completeness condition* are addressed through a combination of dynamic PCA evolvement and α-stack. The dynamic evolvement is done at JIT time. Our examination shows that function inlining and outlining are the major reasons for violations of the *completeness condition*. The dynamic adjustment for inlining and outlining is straightforward. Upon a function inlining, the JIT replaces the node of that call site with the automaton of the inlined function; upon a function outlining, the JIT creates an automaton for the newly formed PCA, assigns an ID to the new call site, and updates the automaton of the parent PCA accordingly. As outlining happens rarely, negligible overhead was seen on the runtime PCA construction. The edge weights of the newly created PCA are initiated with some values determined by the compiler (e.g., a policy common in compiler construction is to put 0.9 for backedges and 0.5 for normal two-way branches [111]).

As an option, during runtime, edge weights can be refined with the runtime observations through weighted average (i.e., new weight = old weight*r + new observations*(1-r), $0.5 > r > 0$) with the decay rate r set by the user. Such an adjustment can be applied to other existing edges as well. (Our experiments did not use this runtime refinement.)

The α-stack addresses the issue of incomplete PCA (i.e. some functions do not have automata built). Initially the α-stack is empty and inactive. At an invocation of a function that has no automata built, the ID of the function is pushed into the α-stack, and the α-stack becomes active. While the α-stack is active, the ID of an invoked function is automatically pushed into the stack, regardless of whether the function has PCA; the top of the stack pops out at each function return. The PCA stalls while the α-stack is active. It resumes state transitions as soon as the α-stack becomes inactive when it turns empty. For example, suppose that the PCA is now in state "C,1" of Figure 5.1 (b) and some unexpected function "X" is then invoked. Assume that "X" calls "Y" and "Y" calls "A". Neither "X" nor "Y" has automaton built. The PCA would stay at node "C,1" until "X"

returns. It then resumes state transition according to the PCA.

Essentially, the α-stack makes operations on PCA skip functions that do not have automata, as well as the functions directly or indirectly invoked by them. Such a design offers a simple way to deal with unexpected calls. A more sophisticated design is to skip only functions that have no automata (e.g., "X" and "Y" but not "A" in our example). It is potentially doable, but adds much complexity: It has to deal with broken chains of states. For instance, when "A" returns, it is unclear which state the PCA should return to.

Additional complexities include native function calls and tail call optimizations. Native code is ignored. Optimized tail calls become jump instructions and hence are not tracked or predicted.

5.2.5 Properties

We now examine how PCA embodies the four basic properties listed in Section 5.1.

(1) *Ensuing relations.* PCA is centered on ensuing relations. A transition edge represents what function call follows (rather than invokes) another call. For example, in Figure 5.1 (b), "C,5"→"A,6" represents that after the finish of "C" on line 7 in Figure 5.1 (a) (represented by node "C,5"), the next function call must be a call to "A" at line 8 (represented by node "A,6"), despite that "C" never calls "A" in the program.

(2) *Discriminating.* The structure of PCA encodes both branches and loops. Its edge weights facilitate the resolution of ambiguities caused by control flows. With node and edge labels carrying call site IDs, PCA naturally distinguishes different call sites. The *return stack* and *shadow stack* add calling contexts to PCA. For example, suppose the PCA is now at state "C,7" in Figure 5.1 (b). The two stacks help the prediction automatically tell whether the call of "A" was from node "A,2", "A,3", or "A,6" when the PCA walk returns from node "D,8", and hence produce different prediction results. In addition, the PCA structure allows an even deeper level of discrimination: Instead of defining an edge weight as a probability given the source node, one could employ conditional probabilities as edge weights, with the top k levels of the shadow stack as the conditions. In this way, they could further discriminate call sites and calling contexts. Such a model may increase the size of the PCA; we leave it to future study.

(3) *Generality.* The design of v-node and α-stack, along with runtime PCA evolvement, make PCA resilient to various complexities in the language implementation, compilation, and other aspects.

(4) *Scalability.* Unlike some other representations (e.g., dynamic call tree), the size of a PCA is bounded by the number of call sites in a program, independent of the length of an execution.

Section 5.5 reports the size on some programs.

Besides those, the modular structure of PCA gives good compatibility. If the body of a function is changed, except edge weights and that function's automaton, the structure of the program's PCA needs no change.

5.2.6 Formal Definition of PCA and Its Construction

We now give a formal definition of PCA as follows:

Definition A PCA $\mathcal{P}$ is a tuple

$\langle A_1, A_2, \cdots, A_k, \Sigma, \Gamma, \Lambda, r, s, \alpha \rangle$, where:

A_i: a finite state machine Σ: input alphabet

Γ: stack alphabet Λ: exit alphabet

r: the return stack s: the shadow stack

α: the α-stack

State machine A_i in a PCA is a tuple

$\langle N_i, s_i, f_i, e_i, \delta_i, P, D_i, M_i \rangle$, where:

N_i: the set of nodes.

s_i: a unique entry node. $s_i \in N_i$.

f_i: a unique terminal node. $f_i \in N_i$.

e_i: a unique error node. $e_i \in N_i$.

δ_i: transition relation over N_i.

P: transition probability over δ_i.

D_i: the set of diamonds, $D_i \subset N_i$.

M_i: a mapping function from D_i to s_j, $j \neq i$.

Transition relation δ_i is a finite set of rules such that: ① for every state $S \in (N_i - f_i)$ and an input symbol $a \in \Sigma$, there is a unique rule of the form $\delta_i(S, a) \rightarrow T$, where $T \in (N_i - f_i - s_i)$; ② for any input symbol $a \in \Lambda$, there is at least one state $S \in (N_i - f_i)$ such that $\delta_i(S, a) \rightarrow f_i$. Recursion is allowed; a self-recursive call corresponds to a diamond that carries a reference to the entry node of its own automaton. Transition probability P is a function which to every rule $\delta_i(S, a) \rightarrow T$ assigns its probability $P(\delta_i(S, a) \rightarrow T) \in (0, 1]$ so that for any given $S \in N_i$, we have

$$\sum_{T \in N_i : \exists a, \delta_i(S,a) \rightarrow T} P(\delta_i(S, a) \rightarrow T) = 1.$$

There is a special transition: $d \rightarrow M_i(d)$, $\forall d \in D_i$, which happens every time when d is

reached; with the transition, d is pushed into the return stack, r. Another special transition is: $f_i \rightarrow \tau(r)$ (where $\tau(r)$ is the top of the stack r), which happens every time when f_i is reached; meanwhile, r pops its top off the stack. The shadow stack s gets a copy of r when call sequence prediction starts. During and only during the prediction, s plays the role of r in dictating the state transitions. Like many conventional automata, there is an implicit error state e associated with a PCA; $\forall x \in N_i$, $x \rightarrow e$ on any unexpected input. Along with such a transition, x is pushed into the α-stack. While the PCA is at e, every input belonging to the exit alphabet Λ makes the α-stack pop, while all other inputs are pushed into the α-stack. When the length of the stack becomes two, an encounter of input $a \in \Lambda$ prompts the transition $e \rightarrow \tau(\alpha)$ besides making the α-stack pops. After the transition, the α-stack pops again to turn empty.

Construction The construction of PCA involves two main steps: The first builds up the PCA structure during compilation; the second trains the PCA by adding weights through profiling. Algorithm 4 outlines the procedures.

Algorithm 4 PCA Construction

```
1: /* Building PCA Structure */
2: G[F] = control flow graph of function F;
3: P={};
4: for each function f do
5:     If G[f] contains function calls then
6:         R = buildRegExp(G[f]);
7:         R' = cleanUp(R);
8:         A = regExp2DFA(R');
9:         createCandidateTables(A);
10:        P.add(A);
11: connectAutomata(P);
12:
13: addWeights(P); /* Profiling for Weights */
```

For the modularity of PCA, the first step can happen on each function individually. Not all instructions in a function are relevant to function calls. A compiler goes through the code, ignores irrelevant parts, and converts the rest into an automaton. Conceptually, in this step, the compiler derives a skeleton of the control flow graph, where, all statements but function calls and branches are removed, (leaving some empty basic blocks), while the edges remain. The compiler derives a regular expression from the skeleton graph. The vocabulary of the regular expression consists of β, representing an empty block in the skeleton graph, a terminal variable and a non-terminal variable for each function in the program. The β helps encode the logic of empty blocks into regular expression. The non-terminal variable represents a call to the function. The terminal variable represents the entry of the function, which is always the first symbol in the regular expression of the function. Branches are represented with the "|" operator, while loops (or backedges) are represented with the "*" or "+" operator.

$$
\begin{aligned}
\mathsf{M} &\rightarrow \mathsf{m}\, C_1\, (A_2 |\, (A_3\, B_4))\, C_5\, A_6 \\
\mathsf{A} &\rightarrow \mathsf{a}\, C_7{}^{+}\, D_8 \\
\mathsf{B} &\rightarrow \mathsf{b} \\
\mathsf{C} &\rightarrow \mathsf{c} \\
\mathsf{D} &\rightarrow \mathsf{d}
\end{aligned}
$$

Figure 5.6: The FCG of the program in Figure 5.1 (a). Every letter represents a function. An upper-case letter is a non-terminal variable, and a lower-case letter is a terminal variable, representing the prologue of a function represented with the corresponding upper-case letter, and a subscript represents a call site ID.

Next, the compiler simplifies the regular expression in a standard way, which removes all βs as well. Figure 5.6 shows the regular expressions of our PCAExample. We refer to such a set of regular expressions as the function calling grammar (FCG) of the program. As a whole, an FCG is a Context Free Grammar (CFG). The simple form directly leads to PCA through standard algorithms of regular expression-to-automaton conversion. Another advantage of using FCG as the intermediate form is that the conversion algorithms, by default, minimize the generated automaton and hence the overall size of the PCA. The candidate tables are then built for each v-node in the DFA.

The final step adds weights to the edges in the PCA. It uses profiling executions of the program to do so. During a profiling run, the PCA runs along with it by updating its state upon each function call. A profiler records the number of times an edge is visited if the out-degree of the source is greater than one. The weight of an edge is then used to calculate the weights on those edges. It puts in the probabilities for the entries in candidate tables in the same manner. An edge that has not been encountered in the profiling runs is assigned with an extremely small weight for the completeness of the PCA.

What profiling mechanism to use is orthogonal to the proposal of PCA. Besides offline profiling, there are many other techniques for efficient online sampling [27,37,57] or cross-run accumulation of samples [69]. They could all be used for PCA construction, depending on the usage scenario.

5.3 Comparisons to Existing Representations

Before this work, there are a variety of program representations relevant to program function calls. In this section, we examine four most commonly studied ones, qualitatively showing that they are ill fit for call sequence prediction for not meeting some of the *four basic properties*. Section 5.5 will complement the comparison with some quantitative evidences.

Among existing models of program function calls, the most influential are static and dynamic call graphs, dynamic call trees, and calling context trees (CCTs). We use Figure 5.7 to review

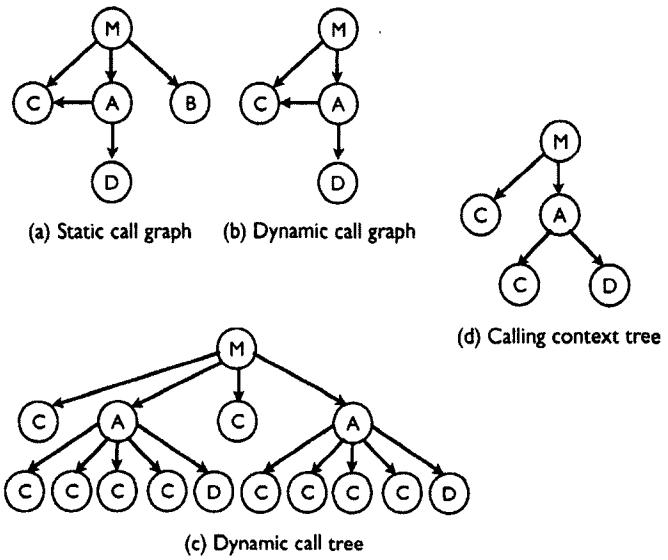


Figure 5.7: Four other representations of function calls in executions of "PCAExample" in Figure 5.1 (a). In the program, no functions except "M" and "A" contain function calls. In the considered execution, the "if" branch is taken.

them briefly. In a static call graph (Figure 5.7 (a)), each function has a unique node no matter at how many call sites it is invoked, and there is an edge directed from function "M" to function "A" if it is possible for "M" to call "A". A dynamic call graph (Figure 5.7 (b)) has the same structure, except that it is built through a profiling run and there is an edge between two nodes only if that invocation actually takes place in that run. A dynamic call tree (Figure 5.7 (c)) also comes from a profiling run. It adds calling context information, with each node representing a function invocation, and the path to it from the root representing its calling context. A CCT [16] is similar to a dynamic call tree except that it uses a single node to represent all calls to a function that have the same calling contexts. In Figure 5.7 (d) for instance, all the "C" nodes under "A" in Figure 5.7 (c) are folded into one.

All four representations are designed for program analysis rather than call sequence prediction. They have some variations. We analyze their properties with their basic forms first, and discuss their extensions later. Specifically, we examine them against the *four basic properties*, which qualitatively reveals their limitations for call sequence prediction.

- *Ensuing relations.* The four representations are all centered on calling relations rather than ensuing relations. For example, Figures 5.7 (a) (b) (c) and (d) all indicate that both "C" and "D" are possible callees of "A", but none encodes the relation that a call to "D" must follow calls to "C" if those calls are made by "A"[3]. The lack of ensuing relations makes them

[3]Nodes in a dynamic call tree by default have no specific orders. If extended with a time order, the tree may capture some ensuing relations.

fundamentally ill fit for call sequence prediction.

- *Discriminating.* **Control flows**: None of the four representations encodes branches or loops. The static call graph in Figure 5.7 (a), for instance, fails to show that function "A" is invoked at both branches and "B" is not. The other three representations, on the other hand, completely miss the branch that contains "B". Moreover, none of the representations expresses that "C" is called inside a loop (the dynamic call tree in Figures 5.7 (c) shows four consecutive calls to "C" in "A" but leaves it unclear whether they are caused by a loop or four different call sites of "C".) Missing control flows hinders these representations for call sequence prediction. For example, the control flows tell us that if and only if the second call sites of "A" is reached, "B" will be called immediately after "A" finishes. None of the four representations captures that constraint. **Calling contexts**: Dynamic call tree and CCT both maintain calling contexts. But static and dynamic call graphs do not. In Figures 5.7 (a) and (b), for instance, all calls to "C" are aggregated into a single node, despite that they differ in their calling contexts. **Call sites**: None of the representations except dynamic call trees offers a full discrimination of call sites. For instance, the two sites of calls to "C" in "M" are folded into a single node in Figures 5.7 (a) (b) (d). They hence fail to encode that different call sequences could follow the two calls.

- *Generality.* Dynamic call graphs, call trees, and CCT all contain only the invocations made in some training execution(s) rather than the complete calling relations in the program. Some functions (e.g., "B") absent from them may be called in other runs. It is possible to append these newly encountered calls to these graphs or trees at runtime. But there are no machinery in these representations to overcome the incompleteness and the ambiguity (e.g., by dynamic dispatch) for call sequence prediction.

- *Scalability.* Static and dynamic call graphs are bounded by the number of unique functions in the program. CCT is bounded by the number of distinct calling contexts. They all have reasonable scalability, although sometimes a CCT could be orders of magnitude larger than the program itself. A dynamic call tree, on the other hand, may contain as many nodes as the number of function invocations in a run, often too large for practical usage.

Overall, in their basic forms, the four representations all miss some of the basic properties. They have some variations, the extra features of which may alleviate some issues, but cannot address their inherent limitations. For example, in a call graph with labeled edges, a caller may have multiple calling edges connecting to a callee, with each edge corresponding to a distinct call

site. Similarly, CCT can be made call site-aware as well if different call sites of a function are represented with different nodes, even if they have the same calling context [97]. However, these variations do not change the inherent nature of these representations of centering around calling rather than ensuing relations. Neither do they address the issues on control flows or generality.

Consequently, these representations cannot well capture the relevant constraints defined by the program. In Figure 5.7, for example, none of them reflects the constraint that either "B" or "C" but not any other functions will follow the first invocation of "D". Neither do they reflect that if "A" has been invoked twice by "M" and the current execution point is inside "D", there will be definitely *no other function calls by the end of the execution.*

The qualitative analysis reveals the high-level limitations of these representations for call sequence prediction; Section 5.5 confirms them through some quantitative comparisons with PCA.

5.4 Metrics for Call Sequence Prediction

We find no prior definition of metrics for assessing a call sequence prediction. We introduce three levels of metrics, which are of different strictness, suitable for different uses of the prediction results.

Let Q and $\hat{Q}$ be the true and predicted call sequences, and U and $\hat{U}$ be the set of unique functions in Q and $\hat{Q}$. The three levels of metrics are as follows.

- *Set-level:* It quantifies the closeness between U and $\hat{U}$. We introduce the following notations: $TP = |U \cap \hat{U}|$, $TN = |\overline{U} \cap \overline{\hat{U}}|$, $FP = |\hat{U} - U|$, $FN = |\overline{\hat{U}} - \overline{U}|$; $\overline{U}$ and $\overline{\hat{U}}$ are the set of functions in the entire program that do not appear in U or $\hat{U}$ respectively. ("T" for true, "F" for false, "N" for negative, "P" for positive.) Following information retrieval theory [50], we use two common metrics: **recall**=$TP/|U|$; **precision**=$TP/|\hat{U}|$. They respectively measure how much the true set is uncovered and how precise the prediction set is. To integrate them into a single metric, we borrow the concept of Matthews correlation coefficient (MCC) [89], which takes into account true and false positives and negatives and is generally regarded as a balanced measure. It is defined as
  MCC=$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$.
  MCC has a value range [-1, 1]. We normalize it to [0, 1] as follows: **Set accuracy=(MCC + 1)/2**.

- *Frequency-level:* Let n_f and $\hat{n_f}$ be the numbers of times the function f appears in Q and $\hat{Q}$ respectively. The **frequency accuracy** of $\hat{Q}$ is

$1 - average_{f \in U \cup \hat{U}}(|n_f - \hat{n_f}|/max(n_f, \hat{n_f}))$.

- *Sequence-level:* Let e be the minimum number of atomic editing operations (insertion, deletion, or replacement of a single token in $\hat{Q}$) needed to change $\hat{Q}$ into Q. The **sequence accuracy** of $\hat{Q}$ is $1 - e/max(|Q|, |\hat{Q}|)$. Let Q^* be the sequence of the functions in U ordered in their first occurrences in Q, and $\hat{Q}^*$ be the counterpart for $\hat{Q}$. The **first-occ sequence accuracy** is the *sequence accuracy* of $\hat{Q}^*$ regarding Q^*.

The usage of "max" in the frequency accuracy and sequence accuracy ensures that the accuracy is in the range of 0 and 100%. For instance, the e in sequence accuracy must be no greater than $max(|Q|, |\hat{Q}|)$ since a naive way to generate Q from $\hat{Q}$ is to replace every token in $\hat{Q}$ with the corresponding one in Q and its number of operations is $max(|Q|, |\hat{Q}|)$.

As an example, assume that the true sequence is "A A B C B D", while the predicted sequence is "A A A B E F", and there are 10 unique functions in the whole program. The measures are as follows: TP=2, TN=4, FP=2, FN=2, recall=0.5, precision=0.5, set accuracy=0.58, frequency accuracy=0.19, sequence accuracy=0.33, first-occ sequence accuracy=0.5.

Set-level measures are the most relaxed among all. They ignore the order and frequency of function calls in the sequences. First-occ sequence accuracy is slightly stronger by considering the order of the first-time occurrences of the functions in Q. They are useful when the prediction is for guiding early compilation or prefetching.

Frequency accuracy reflects how well the prediction captures the hotness of the functions in Q. It is useful for hotness-based optimizations.

Sequence accuracy is the most strict on the difference between two sequences. The usage of atomic editing operations in the definition avoids some misleading effects of alternative definitions. For instance, Hamming distance—which does pair-wise comparison at token level—is sensitive to local differences and cannot precisely measure the similarity of two sequences. For example, Q is "A B C D" while $\hat{Q}$ is "E A B C", accuracy based on Hamming distance is 0, even though the two sequences share a large subsequence. The definition on atomic operations is not subject to the problem. Computing the needed minimum number of operations can be challenging, but some existing tools (e.g., the Linux utility "diff") can be used as the ruler.

5.5 Evaluation

For evaluation, we concentrate on the following questions:

- Can PCA enable accurate call sequence predictions? What is the time and space cost?

- Is the enabled prediction useful?

For the first question, we design a set of experiments to measure the call sequence prediction accuracies and overhead; for the second question, we apply the prediction results to help JIT decide when to compile which methods for reducing response time. It would be ideal to assess these results in the context of existing techniques. But it is difficult as there are no existing work directly on call sequence prediction. To circumvent the difficulty, we implement three other call sequence predictors by extending most relevant existing techniques.

5.5.1 Three Alternatives

In Machine Learning, there is a problem called discrete sequence prediction [66], but its prediction target is still just the next symbol in a sequence. To put our results into a context, we implement two representatives of such methods and extend them for call sequence prediction.

Alternative-1: The first is called *Pattern method*, an extension from the single-call predictor by Lee and others [35, 68]. It is based on Markov model. Through a Machine Learning engine, it derives statistical patterns by examining all the $K+1$-long subsequences of a training sequence, based on which, its predictor looks at the K most recent function calls to predict which function will be called next. The authors showed the usage of the prediction for detecting OS security issues.

Alternative-2: The second is called *TDAG* method, which also exploits frequent subsequences but in a more sophisticated manner through a classical Machine Learning method called Markov Tree [108]. It uses a tree to store frequent subsequences of various lengths and maintains confidence for each tree node. With the tree, it intelligently picks the best subsequence (frequent enough with strong predictive capability) for each prediction. To avoid tree size explosion, it adds some constraints on the nodes and height of the tree [66]. In our implementation, we adopt the same parameter values as in the previous publication.

Both methods were originally designed for predicting only the next symbol. We expand the prediction target naturally to a sequence of calls. The training process remains the same as in the previous work. At a prediction time, the extended methods gives prediction of the next symbol, $\hat{s_{t+1}}$, based on the previous k-symbol sequence $(s_{t-k+1}, s_{t-k+2}, \cdots, s_t)$ in their default manner, and then in the same manner, gives prediction $\hat{s_{t+2}}$ by regarding the sequence $(s_{t-k+2}, s_{t-k+3}, \cdots, \hat{s_{t+1}})$ as the most recent k-symbol sequence. Other symbols in the time window are predicted likewise. A comparison to these methods helps reveal the benefits of PCA's capitalization of program inherent constraints.

Alternative-3: Although we are not aware of previous usage of the other representations listed in Section 5.3 for call sequence prediction, they can be adapted to do so in a manner similar to our PCA. We implement such a predictor on CCT, the most sophisticated representation of all of them. It is called *CCT-based predictor.* There are two extensions. First, we add an edge from every node to each of its immediate parents (callers), representing the transition happening when the current function returns. Second, we use profiling to add probabilities to all the edges in the extended CCT in the same way as in PCA construction. As a program executes, each function call triggers one move on the CCT. At prediction time, the predictor walks on the CCT based on the directions of its edges, and outputs as the predicted sequence the functions corresponding to the nodes it encounters. For a node with multiple outgoing edges, we also experiment with both the ML and random walk approaches.

A comparison to CCT-based predictor helps quantitatively assess the benefits of PCA for its better treatment to control flows, calling contexts and call sites.

5.5.2 Methodology

All experiments happen on a machine equipped with dual-socket quad-core Intel Xeon E5310 processors that run Linux 2.6.22; the heap size ("-Xmx") is 512MB for all. We use Jikes RVM [4] (v3.1.2), an open-source Java Virtual Machine, as our basic framework. We modify its JIT to derive the FCG from a function's bytecode, and to collect calling sequence for training the edge weights on a PCA and a CCT. The Jikes RVM runs with the default JIT (including both baseline and optimizing compilation and inlining) unless noted otherwise.

We use the Dacapo (2006) benchmark suite [26]. (The latest version of Dacapo does not work well with Jikes RVM [56].) Two programs, *chart* and *jython,* were left out because they fail to run on the Jikes RVM-based profiler. Table 5.1 shows the benchmarks, their lines of code, the numbers of unique calling contexts, and sequence lengths(i.e., the total numbers of calls a program makes in a run) on the *small* and *default* inputs coming with the benchmark suite. In our experiment, we use *small* runs for training and *default* runs for testing. On most programs, the two runs differ substantially in both the length of call sequences, as shown in Table 5.1. All executions involve a few JNI calls. As Java uses dynamic dispatch, Table 5.2 reports the size distribution of the candidate sets of function calls. For all programs except for *bloat,* the call sites with larger than 4 candidate set are less than 5%. We use ten as the upper bound of the candidate table size.

Our evaluation concentrates on the startup phase of program executions. Here, the startup

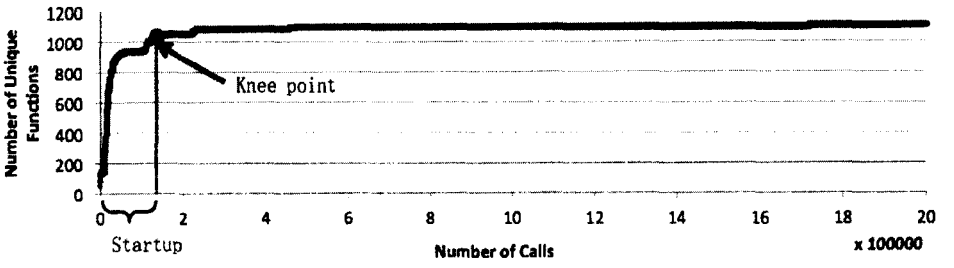


Figure 5.8: The cumulative compilation curve of benchmark *bloat* and its knee point.

phase refers to the beginning part of a program execution, by the end of which, a major portion of the methods that the whole execution needs have been compiled. Quantitatively, we determine the startup phase by finding the *knee point* on the cumulative compilation curve of an execution. Figure 5.8 illustrates the concept by depicting the curve of an execution of benchmark *bloat*. Formally, a knee point on a smooth ascending convex curve is defined as the point where the radius of curvature is a local minimum. The cumulative compilation curve of a program execution are often not smooth, but its trend (i.e., when local noises are smoothed out) is typically so. In our experiments, we draw all the cumulative compilation curves of all executions and manually find the knee points through visual examination of the trend of the curves. We observe that the knee points of all of the program executions appear before the 700,000th function call in their executions. For simplicity, we take the first 700,000 function calls as the approximated startup phases of all the programs in our evaluation[4]. In all those executions, a majority of method compilations happen in those startup phases.

For many applications, the end of the startup phase roughly corresponds to the time when the application finishes initialization and becomes ready to interact with users. The length of the phase, therefore, critically determines the responsiveness of the launches of such applications. It is especially so for utility programs, which, unlike server programs, are often utility tools that do not have a long-running execution, but whose responsiveness is important for user experience. For them, compilation could take a substantial portion of its execution time, especially during the startup stage of their executions. In our experiments of the replay runs of the Dacapo benchmarks, we observe that method compilations take 7~96% (65% on average) of their startup times.

All reported timing results are average of ten repetitive measurements. Each reported accuracy number of a benchmark is computed by averaging the prediction accuracy of all its prediction windows. In all experiments, a prediction window is in the unit of the number of function calls. If the prediction window size is 20, after a program starts, the predictor is triggered after every 20

[4]Because the optimizations based on our prediction, as shown in Section 5.5.4, save more time in the startup phase than in the stable-running phase due to the more compilations in the startup phase, the true speedups for the startup phase could be higher than the reported due to the over approximation of startup phases.

Table 5.1: Benchmark Information

Program	# code lines	# unique call. contexts ($\times 10^3$)	Seq. length ($\times 10^6$) *small*	*default*
antlr	32263	1006	7	490
bloat	73563	1980	9	6276
eclipse	1903219	4816	18	1267
fop	88846	175	3	44
luindex	8570	374	10	740
lusearch	12709	6	9	1439
pmd	49331	8043	6	2727
xalan	243516	163	33	10084

Table 5.2: Size of Candidate Sets

Program	size distribution				
	1	2	3	4	≥ 5
antlr	73%	10%	9%	4%	4%
bloat	49%	15%	8%	3%	25%
eclipse	65%	22%	8%	2%	3%
fop	65%	18%	8%	8%	0%
luindex	57%	26%	15%	1%	1%
lusearch	51%	31%	6%	9%	3%
pmd	70%	18%	5%	3%	4%
xalan	72%	15%	5%	3%	4%

function calls to output the prediction of what the next 20 function calls will be.

5.5.3 Accuracy

Table 5.3 shows the comparison among the four predictors on all six metrics. In the setting, the prediction window length is 20, and the maximum likelihood is employed for both the PCA and CCT predictors. (Other settings are shown later.) The rightmost column shows the geometrical mean.

PCA results are consistently better than the other predictors, with about 20% higher set accuracy, 40-50% higher frequency accuracy, about 40% higher first occurrence sequence accuracy, and 30-56% higher whole sequence accuracy. As the metrics become stricter, the accuracies of all methods except PCA drop sharply to no greater than 30% on average. The PCA results also show some considerable drop, but it still keeps the accuracy on half of the benchmarks higher than 70% on all the metrics. There are some quite challenging programs. For example, the program *eclipse*, for its large number of functions and complex control flows, causes the CCT, TDAG and Pattern methods to get near zero frequency and sequence accuracies and less than 62% set accuracy. The PCA does not get very high frequency and sequence accuracies either, but it manages to still achieve a 79% set accuracy.

It is important to note the connections between prediction errors and the usefulness of the

Table 5.3: Prediction Accuracy (window size=20)

		antlr	bloat	eclipse	fop	luin.	luse.	pmd	xalan	mean
Set accuracy	PCA	0.94	0.96	0.79	0.89	0.90	0.86	0.91	0.92	0.89
	CCT	0.67	0.77	0.58	0.65	0.79	0.72	0.62	0.65	0.68
	TDAG	0.65	0.79	0.62	0.65	0.78	0.69	0.60	0.65	0.68
	Pattern	0.79	0.87	0.51	0.82	0.51	0.86	0.69	0.79	0.72
Set recall	PCA	0.92	0.96	0.65	0.84	0.95	0.96	0.85	0.92	0.87
	CCT	0.69	0.91	0.31	0.66	0.89	0.89	0.52	0.64	0.66
	TDAG	0.65	0.90	0.50	0.77	0.94	0.84	0.50	0.74	0.71
	Pattern	0.63	0.79	0.02	0.66	0.04	0.78	0.40	0.59	0.29
Set prec	PCA	0.87	0.91	0.56	0.75	0.71	0.58	0.81	0.81	0.74
	CCT	0.18	0.34	0.09	0.15	0.40	0.23	0.12	0.17	0.19
	TDAG	0.15	0.42	0.14	0.13	0.35	0.19	0.10	0.15	0.18
	Pattern	0.53	0.70	0.01	0.63	0.04	0.68	0.37	0.57	0.25
Frequency accuracy	PCA	0.78	0.87	0.36	0.66	0.52	0.45	0.74	0.77	0.62
	CCT	0.07	0.13	0.05	0.08	0.26	0.12	0.06	0.12	0.10
	TDAG	0.05	0.11	0.05	0.05	0.22	0.08	0.05	0.05	0.07
	Pattern	0.44	0.62	0.01	0.49	0.03	0.55	0.27	0.43	0.20
1st occ. Sequence accuracy	PCA	0.81	0.88	0.37	0.66	0.62	0.55	0.73	0.77	0.65
	CCT	0.17	0.34	0.08	0.15	0.38	0.22	0.11	0.17	0.18
	TDAG	0.15	0.40	0.13	0.09	0.35	0.18	0.10	0.11	0.16
	Pattern	0.75	0.88	0.00	0.80	0.28	0.92	0.51	0.69	0.20
Full Sequence accuracy	PCA	0.77	0.85	0.26	0.63	0.56	0.45	0.70	0.74	0.59
	CCT	0.01	0.04	0.00	0.05	0.36	0.11	0.02	0.10	0.03
	TDAG	0.00	0.00	0.00	0.00	0.35	0.04	0.00	0.00	0.01
	Pattern	0.73	0.86	0.00	0.79	0.38	0.89	0.51	0.68	0.28

Table 5.4: Size and Training Time

Program	Size (MB)				Training Time (sec)			
	PCA	CCT	TDAG	Pattern	PCA	CCT	TDAG	Pattern
antlr	0.55	0.96	0.02	1.7	21	15	1019	325
bloat	0.56	1.77	0.03	83	21	21	1316	6735
eclipse	1.33	1.95	0.11	15	68	39	2710	4385
fop	0.43	0.69	0.05	9.7	19	7	359	1884
luindex	0.23	0.07	0.01	0.92	23	24	1510	145
lusearch	0.20	0.02	0.01	1.1	24	22	1335	175
pmd	0.49	0.73	0.03	51	6	3	78	3724
xalan	0.55	0.24	0.02	46	11	6	477	3146

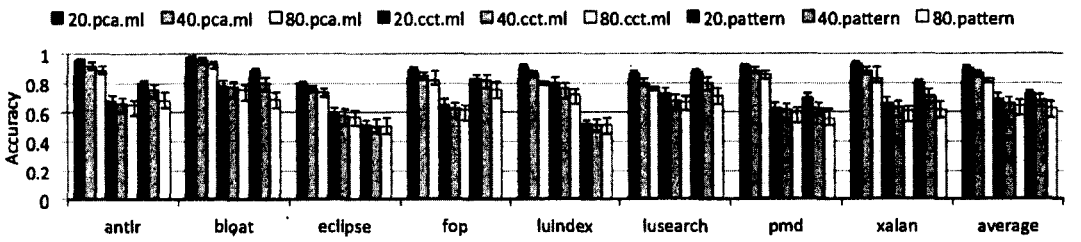


Figure 5.9: Comparison of Set Accuracy among different prediction window sizes.

prediction. It is generally true that a more accurate prediction may give a larger benefit for program optimizations. However, many program optimizations have a certain degree of tolerance of prediction errors. For instance, when predicted call sequences are used to trigger function prefetching from remote servers, a 80% means that 20% of the prefetched functions may not be useful. The prefetching of them may waste some bandwidth and energy. But the prefetching of the 80% useful functions may still shorten the execution time of the program substantially and considerably outweigh the loss by the 20%. In the next subsection, we will see that the 79% set accuracy on *eclipse*, for example, yields up to a 10% speedup when the prediction is applied to code cache management.

Another observation is that the CCT-based approach is overall no better than the Pattern-based approach in terms of prediction accuracies. It indicates that although capitalization of program structure can be beneficial for call sequence prediction, how to capitalize it and using what representation to encode the structure are critical: The lack of support in CCT for various levels of contexts leaves its capitalization of program structures ineffective.

Figure 5.9 gives a more detailed report. (TDAG performs the worst and is hence omitted for lack of space.) As the prediction scope increases, the difficulty for prediction increases. All three methods show a certain degree of reduction in accuracy. On two programs with some frequently occurring call sequence patterns (*fop* and *lusearch*), the Pattern method performs well, yielding set accuracies close to those from the PCA method. But across all window sizes, PCA maintains an average accuracy higher than 80%, about a 20% edge over the other methods.

Another dimension of comparison is between the Random Walk and Maximum Likelihood. From Figure 5.10, we can see their influence on the PCA method in terms of three types of accuracies. For most programs, Maximum Likelihood gives higher accuracies. An exception is *lusearch*, which has a number of loops with a low loop trip-count. Being able to get out of the loop early, Random Walk helps the prediction. But overall, the average accuracies show that their influence on PCA and CCT does not differ much.

Besides accuracy, we have examined the size, training time, and prediction time of the four

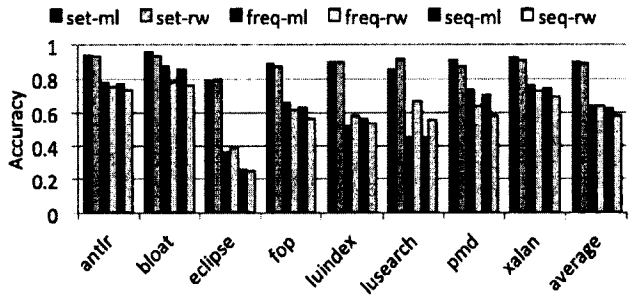


Figure 5.10: Comparison between maximum likelihood (ml) and random walk (rw). (window size=20)

methods. As Table 5.4 shows, the pattern-based predictor can be much larger than the other three predictors, when there are many different subsequences (e.g., *bloat*, *pmd*, and *xalan*.) The TDAG method successfully reduces the size of the predictor through its constrained tree structure (but fails in enhancing the prediction accuracy.) The training time of both Pattern and TDAG are several orders of magnitude longer than the other two predictors. PCA predictors are slightly larger than CCT predictors; both are quick to train. The time taken to perform a prediction is independent of benchmarks. The PCA and CCT predictors take $32\mu s$ and $8\mu s$ to predict a 40-call sequence respectively, negligible compared to the time needed to compile the functions by JIT. By contrast, the TDAG predictors take $827\mu s$ on average, caused by the Markov tree searching at each call prediction. The time overhead of state tracking is marginal, no more than 2% for the programs.

5.5.4 Uses

The PCA-based call sequence prediction may benefit many uses, such as guiding the replacement policy in code cache to reduce cache misses [49], enabling better prefetching to enhance instruction cache performance [84], and helping preload remote classes in mobile computing.

In this work, we experiment with parallel JIT compilation. Parallel JIT creates multiple threads to compile functions. By default, it compiles a method only after the method gets called. With the prediction of upcoming method calls, the compilation of a method could happen earlier, enabling better overlap between execution of the program and compilations of to-be-invoked methods. The overlap can help prevent some (part of) compilations from appearing on the critical path of the program execution. It is especially beneficial for speeding up the startup phase of a program.

In our experiment, we implement a prototype of parallel JIT on JikesRVM. For parallel JIT to work well, there are two aspects. The first is to determine the appropriate optimization level to use for the target function, the other is to decide the good time to compile the function. There

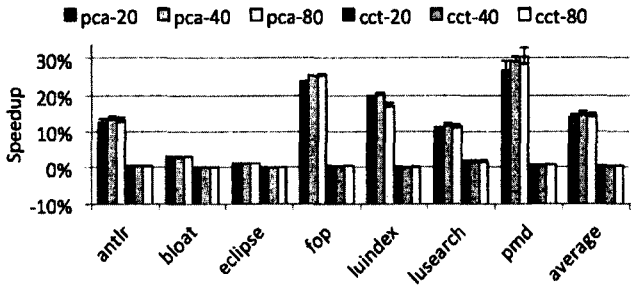


Figure 5.11: Speedup when call sequence prediction is used for parallel JIT compilation (Two compilation threads are used).

are many studies on predicting the optimization levels [18]. The focus of our experiment is on the compilation timing aspect. So to avoid the distractions of the other factor, we use the advice files produced by JikesRVM for all experiments. The files record the appropriate optimization level for each method based on its importance.

In our experiments, after each prediction window, the JIT invokes the predictor to get the predicted call sequence in the next time window. It then creates compilation events for the methods in the predicted sequence that have not been compiled before, and puts those events into the compilation event queue in JikesRVM. Compilation threads automatically dequeue the events and conduct the compilation.

The number of compilation threads we tested ranges from two to seven. We see diminishing gains from parallel JIT when the number is greater than two. As two is the most cost efficient, Figure 5.11 reports the speedup in that setting. We chose CCT method as the representative of alternatives to PCA for its relative ease to use and having a similar or higher prediction accuracy and prediction speed than others.

The baseline in Figure 5.11 is the performance when the default replay mode is enabled, which uses the same compilation levels as in the advice files but uses no prediction of call sequences. Given that most studied programs are utility programs, their responsiveness rather than steady-state performance is what often matters. The performance is based on the end-to-end wall-clock time of the startup phase of an execution.

Call sequence prediction not only increases compilation parallelism, but also enables better overlapping between execution and compilation. The PCA-based predictions, in all three window sizes, lead to more than 20% speedups on three programs, and an average around 15% on all seven programs ("xalan" fails working in the default replay mode). In contrast, the CCT-based prediction gives only slight speedup on lusearch and pmd. It is due to its low prediction precision: an average of 19% versus the 74% of PCA-based approach as Table 5.3 shows. In consequence,

many useless functions are compiled, which delays the compilation of those useful ones.

Two other observations are worth mentioning. First, a larger prediction window does not always deliver higher speedup. It is because on large windows, the prediction, although finding more useful methods to compile, could enqueue more methods that won't be used in the near future. Second, some programs (e.g., bloat) that have high prediction precision and accuracy do not show large speedups. It is because the speedup also depends on how much the compilation time weight in the overall running time. If it is small, the entire potential of parallel JIT is small.

5.6 Related Work

5.6.1 Program Representations

Besides the work mentioned in Section 5.3, some other studies also relate with calling contexts. Program summary graphs by Callahan [32], for instance, use nodes for formal and true parameters of functions and edges for their bindings. By showing the flow of values across procedures, the graphs facilitate inter-procedural data flow analysis. The *probabilistic calling context* by Bond and McKinley [28] offers an efficient way to collect and represent calling contexts. Later work proposes other ways to encode calling contexts precisely [110]. Alur and others [15] analyzed Recursive State Machines for representing recursive procedural calls in the context of system verification. As Section 5.1 discusses, calling context is only one of the necessary conditions for call sequence prediction. Without capturing control flows, call sites discrimination, and *ensuing* relations among calls, calling contexts alone do not suffice for call sequence prediction. Moreover, these representations provide no machinery—such as the v-nodes, α-stack in PCA—to overcome the various ambiguities (e.g., by dynamic dispatch) for call sequence prediction.

Some previous studies aim at finding hot code or data streams [38, 67]. Similar to the pattern method implemented in Section 5.5, these methods centered on statistical patterns of sequences, and hence suffer from the diminishing regularity as prediction scope increases. Moreover, predicting cold call sequences and dealing with local variations (e.g., caused by branches) are essential for our call sequence prediction and its usage for startup time reduction.

A previous study uses DFA to record traces found in a binary translation process [88]. It starts from traces of function calls and builds automata based on their patterns. Another study that uses DFA is to construct object usage models [107]. For each abstract object, it builds an automaton with some places in the code as nodes and function calls related to that object as edge labels. Neither of the two studies is for predicting function call sequences; the first is for compressing

traces and the second is for detecting anomalies in object usage. Consequently, their designs are not suitable for call sequence predictions. First, they are at the level of either traces or objects, rather than the whole program. When the scope goes to the whole program level with potentially infinite recursions, it becomes more complex than the pure automata can model. Second, they do not have any of the three stacks in PCA. The stacks adds more expressiveness to automata. More importantly, the stacks, along with unique call site IDs, inject into PCA the capability to discriminate different call sites and calling contexts in the prediction. In addition, their designs give no systematic treatment to ambiguous or unexpected function calls.

The probabilities associated with the v-nodes were inspired by some prior work on virtual function target prediction [23]. There are many other works trying to predict program behaviors beyond function calls, such as function returning values [87], load value prediction [31]. They center on leveraging statistical patterns rather than constraints through program representations.

5.6.2 Stochastic Models

In time series related domains, lots of data analysis has been based on probabilistic state machines (e.g., weighted automata [81], probabilistic pushdown automata [29]), or other stochastic models (e.g., Markov Model, Markov Tree). PCA can be regarded as an augmented form of probabilistic state machines that is specially customized for leveraging constraints coded in programs and for accommodating their various complexities, reflected by its design of the three types of stacks, diamond and v-nodes, and the edge and node labels. These features make PCA more effective in predicting function call sequences, as exemplified by the comparison with Markov Trees in the evaluation.

5.7 Summary

In this paper, we have presented the first systematic study in exploiting program defined constraints to enable function call sequence prediction. We have introduced PCA, a new program representation that captures both the inherent calling relations among functions, and the probabilistic nature of execution paths determined by conditional branches and loops. Experiments show that the new approach can produce more accurate call sequence predictions than alternatives. As a fundamental representation of function calling relations, PCA may open up many new opportunities for optimizing the performance of modern virtual machines and beyond.

6 Conclusion

Modern computing hardware development exposes more and more parallelism to software applications. To harness the increasing hardware parallelism, it is critical that programs can be executed in parallel. However, many programs exhibit inherent dependencies, which fundamentally prevent their parallel executions.

To fill this gap, this dissertation shows the promise of principled speculative parallelization. It offers a systematic and rigorous treatment to the speculation — the key to such parallelization scheme. Its power is demonstrated by parallelizing three important types of computations: finite-state machine, a classic mathematical computation model; HTML parsing, a necessary component in web browsers; and just-in-time compilation, which is commonly seen in many modern compilers.

Parallelizing FSM-based Computation FSM executions naturally introduce dependencies among state transitions, making it extremely hard to parallelize. This work formulates FSM speculative executions and the connections between the speculation schemes design and the characteristics of FSM and their inputs. It provides deep understandings to speculative executions of FSM computations with a series of theoretical findings. It offers 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 this class of computations are in fact quite parallelizable.

However, the proposed techniques require time-consuming offline profiling, making it hard to be adopted online. To address this, we present a two-fold solution to remove a key barrier in the offline profiling. The solution utilizes both static analysis and dynamic optimization techniques to mininimze the profiling cost. Results show that this solution can save up to thousands of times of profiling cost. Finally, through these two phases of study, we provide the first on-the-fly principled speculation for FSM parallelizttion.

Parallelizing HTML Parsing Parsing HTML is a key step in the front end of modern web browsers. However, it has been implemented in a sequential way due to the various complexities with HTML files. This work presents the first systematic study in taming the complexities and developing speculation-centered techniques to create parallel HTML parsers. The outcome includes a set of insights on effectively parallelizing HTML parsers, and HPar, the first practical HTML parallel parser that yields up to 2.4X speedup on quad-core devices. This study breaks a challenging barrier for fully parallelizing web browsers and opens up new opportunities for browser optimizations.

Call Sequence Prediction and Parallel JIT Compilation Runtime compilation has been adopted by many today's popular programming languages, such as Java, C#, JavaScript, and Python. However, runtime compilation puts extra costs on the critical path of program execution, especially during program startup phase. In this work, we provide a remedy based on a novel technique, PCA, that enables function call sequence prediction. PCA captures both the inherent calling relations among functions, and the probabilistic nature of execution paths determined by conditional branches and loops. Experiments show that the new approach can produce more accurate call sequence predictions than alternatives, and make parallel runtime compilation possible.

Through the deep study on the three types of computations, we have developed a set of novel techniques, including two model-based speculation schemes FSM parallelization, a hybrid optimization framework for efficient online FSM profiling, PCA-based call sequence prediction for parallel JIT compilation and correlation-based speculation for parallel HTML parsing. They together form the essence of principled speculative parallelization, advancing the state of the art in software parallelization from being ad hoc to being rigorous.

A question remaining open is how to generalize the principled speculative parallelization paradigm from the several types of computations into general sequential programs. There are some challenges to address. For instance, in our application of the principled speculative parallelizations, we have been drawing on some special properties of FSM, HTML grammar, and Just-In-Time compilations. For general programs, such domain knowledge may not be available or hard to extract. It yet remains to explore how to automatically discover some relevant properties from a computation, or somehow remove the reliance of the speculation on such domain knowledge by, for instance, creating a more general model of speculative executions.

Bibliography

[1] 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] HTML Living Standard : Section 12.2 Parsing HTML Documents. http://www.whatwg.org/specs/web-apps/current-work/multipage/parsing.html.

[3] HTML reference. http://www.w3schools.com/tags.

[4] Jikes rvm. `http://jikesrvm.org`.

[5] Jsoup: Java HTML Parser. http://jsoup.org.

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

[7] The HTML5 test. http://html5test.com/.

[8] Top 1000 websites. http://www.alexa.com/topsites.

[9] Validator.nu. http://about.validator.nu/.

[10] Web pages getting bloated here is why. http://royal.pingdom.com/2011/11/21/web-pages-getting-bloated-here-is-why/.

[11] 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.

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

[13] H. Alblas, R. op den Akker, P. O. Luttighuis, and K. Sikkel. A bibliography on parallel parsing. *SIGPLAN Not.*, 29(1):54–65, Jan. 1994.

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

[15] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, July 2005.

[16] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 1997.

[17] M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems*, 21(4), 2003.

[18] M. Arnold, A. Welc, and V. Rajan. Improving virtual machine performance using a cross-run profile repository. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 297–311, 2005.

[19] 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.

[20] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

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

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

[23] D. F. Bacon and P. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[24] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum. Towards parallelizing the layout engine of Firefox. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, 2010.

[25] J. L. Baer and C. S. Ellis. Model, design, and evaluation of a compiler for a parallel processing environment. *IEEE Trans. Softw. Eng.*, 3(6):394–405, Nov. 1977.

[26] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2006.

[27] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[28] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

[29] T. Brzdil, J. Esparza, S. Kiefer, and A. Kucera. Analyzing probabilistic pushdown automata. *Formal Methods in System Design*, 43(2):124–163, 2013.

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

[31] M. Burtscher and B. G. Zorn. Prediction outcome history-based confidence estimation for load value prediction. *Journal of Instruction-Level Parallelism*, 1999.

[32] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *PLDI*, 1988.

[33] 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.

[34] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. Zoomm: a parallel web browser engine for multicore mobile devices. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 271–280, New York, NY, USA, 2013. ACM.

[35] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), 2009.

[36] 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.

[37] W. Chen, S. Bhansali, T. M. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of PLDI*, pages 332–340, 2006.

[38] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[39] C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proc. of the 8th Int. Symp, on High Performance Computer Architectures*, 2002.

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

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

[42] Y. Ding, M. Zhou, Z. Zhao, S. Eisenstat, and X. Shen. Finding the limit: Examining the potential and complexity of compilation scheduling for jit-based runtime systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 607–622, New York, NY, USA, 2014. ACM.

[43] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970.

[44] J. B. Eric Schurman. Performance related changes and their user impact. http://velocityconf.com/velocity2009.

[45] 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.

[46] 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.

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

[48] 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.

[49] A. Guha, K. Hazelwood, and M. L. Soffa. Balancing memory and performance through selective flushing of software code caches. In *CASES*, 2010.

[50] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[51] 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.

[52] J. Hoxmeier and C. Dicesare. System response time and user satisfaction: An experimental study of browser-based applications. In *Proc. of the Association of Information Systems Americas Conference*, 2000.

[53] S. Hu and J. E. Smith. Reducing startup time in co-designed virtual machines. In *Proceedings of the International Symposium on Computer Architecture*, 2006.

[54] 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.

[55] Intel Corporation. River Trail. `https://github.com/RiverTrail/RiverTrail`.

[56] Jikes rvm project and status. `http://http://jikesrvm.org/Project+Status`.

[57] G. Jin, A. V. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2010.

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

[59] C. G. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodík. Parallelizing the web browser. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, 2009.

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

[61] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, 1965.

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

[63] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *IEEE Computer*, 40(9):103–105, 2007.

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

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

[66] P. Laird and R. Saul. Discrete sequence prediction and its applications. *Machine Learning*, 15:43–68, 1994.

[67] J. R. Larus. Whole program paths. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[68] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[69] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[70] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *MICRO '96: Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–237, 1996.

[71] B. Livshits and E. Kiciman. Doloto: code splitting for network-bound web 2.0 applications. In *Symp. on the Foundations of Software Engineering*, 2008.

[72] 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.

[73] 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.

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

[75] P. Luttighuis. *Parallel Parsing of Regular Right-part Grammars.* Memoranda informatica. 1989.

[76] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 2–2, 2012.

[77] P. Marcuello and A. Gonzlez. Thread-spawning schemes for speculative multithreaded architectures. In *Proc. of the 8th Int. Symp, on High Performance Computer Architectures*, 2002.

[78] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, 2010.

[79] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik. Parallel schedule synthesis for attribute grammars. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 187–196, 2013.

[80] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, 1996.

[81] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:269–311, 1997.

[82] Mozilla Corporation. Servo. https://github.com/mozilla/servo.

[83] 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.

[84] P. Nagpurkar, H. Cain, M. Serrano, J. Choi, and C. Krintz. Call-chain software instruction prefetching in j2ee server applications. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[85] F. Nah. Study on tolerable waiting time: How long are web users willing to wait? *Behavior and Information Technology*, 23(3):153–163, 2004.

[86] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. A static load-balancing scheme for parallel xml parsing on multicore cpus. In *Proc. of the 7th International Symposium on Cluster Computing and the Grid (CCGRID)*, 2007.

[87] C. J. Pickett, C. Verbrugge, and A. Kielstra. Adaptive software return value prediction. Technical Report 1, McGill University, 2009.

[88] J. Porto, G. Araujo, E. Borin, and Y. Wu. Trace execution automata in dynamic binary translation. In *Proceedings of 3rd Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2010.

[89] D. Powers. Evaluation: From precision, recall and f-factor to roc, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, pages 37–63, 2007.

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

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

[92] V. Pronina and A. Chudin. Syntax analysis implementation in an associative parallel processor. *Automation and Remote Control*, 36(8):1303–308, 1975.

[93] 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.

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

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

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

[97] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schoeberl, and M. Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the java virtual machine. In *The 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[98] Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO '97: Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–258, 1997.

[99] B. Shah, P. Rao, B. Moon, and M. Rajagopalan. A data parallel algorithm for xml dom parsing. *Database and XML Technologies, Lecture Notes in Computer Science*, 5679, 2009.

[100] D. B. Skillicorn and D. T. Barnard. Parallel parsing on the connection machine. *Inf. Process. Lett.*, 31(3):111–117, May 1989.

[101] 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.

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

[103] 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.

[104] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, 2007.

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

[106] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed? In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 31–40, 2012.

[107] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Symp. on the Foundations of Software Engineering*, 2007.

[108] R. William. Dynamic history predictive compression. *Information Systems*, 13(1):129–140, 1988.

[109] 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.

[110] W.M.Summer, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *Proc. of the International Conf. on Software Engineering*, 2010.

[111] Y. Wu and J. Larus. Static branch frequency and program profile analysis. In *Proceedings of the International Symposium on Microarchitecture*, 1994.

[112] E. Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Proceedings of the Intl. Conference on Generative Programming and Component Engineering.*

[113] Y. Yi, C.-Y. Lai, S. Petrov, and K. Keutzer. Efficient parallel CKY parsing on GPUs. In *Proceedings of the 12th International Conference on Parsing Technologies*, IWPT '11, pages 175–185, 2011.

[114] D. H. Younger. Context-free language processing in time n3. In *Proceedings of the 7th Annual Symposium on Switching and Automata Theory*, pages 7–20, 1966.

[115] Z. Y. Yu Wu, Qi Zhang and J. Li. A hybrid parallel processing for xml parsing and schema validation. In *Balisage: The Markup Conference 2008.*

[116] 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.

[117] Z. Zhao and X. Shen. On-the-fly principled speculation for fsm parallelization. In *ASPLOS '15: Proceedings of the 20th International Conference on Architecture Support for Programming Languages and Operating Systems*, 2015.

[118] Z. Zhao, B. Wu, and X. Shen. Speculative parallelization needs rigor: Probabilistic analysis for optimal speculation of finite state machine applications. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2012.

[119] 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.

[120] 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.

[121] 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.