Speculative Parallelization Needs Rigor *

—Probabilistic Analysis for Optimal Speculation of Finite-State Machine Applications

Zhijia Zhao
College of William and Mary
zzhao@cs.wm.edu

Abstract

Software-based speculative parallelization has shown effectiveness in parallelizing certain applications. Prior techniques have mainly relied on simple exploitation of heuristics for speculation. In this work, we introduce probabilistic analysis into the design of speculation schemes. In particular, by tackling applications that are based on Finite State Machine (FSM) which have the most prevalent dependences among all programs, we show that the obstacles for effective speculation can be much better handled with rigor. We develop a probabilistic model to formulate the relations between speculative executions and the properties of the target computation and inputs. Based on the formulation, we propose two model-based speculation schemes that automatically customize themselves with the best configurations for a given FSM and its inputs. The new technique produces substantial speedup over the state of the art.

1. Introduction

Parallelization is key to computing efficiency and scalability. Many programs, however, are hard to parallelize for their data dependences. A typical solution is to circumvent the dependences through speculation. Prior speculative parallelizations have mainly relied on simple use of heuristics. In this work, we explore rigorous analysis for tapping into the full potential.

Particularly, we concentrate on applications that are based on Finite-State Machine (FSM)—when it comes to the prevalence of dependences, FSM is truly unbeatable and gains a title "embarrassingly sequential application" [5]. An FSM is an abstract machine with a finite number of possible states. State transitions are often dictated by a state-transition graph, with each node for a state and each transition edge labeled with the symbol that triggers that transition. An example is the FSM in Figure 1 for matching regular expression $([01]^*00[01]^*11[01]^*)^{10}$, where 10 is repetition times.

Dependences exist between every two steps of an FSM computation. They effectively chain through the computation, making parallelization difficult. Consider the string matching example in Figure 1. On a machine with p computing units, a natural way to parallelize it is to evenly divide the input string, S, into p segments and let p threads process them simultaneously, one segment each. The difficulty is in the determination of the starting state for a thread except the first one. For thread 2 to process its segment, it has to know which of the 21 states it should use to start its process. That state should be the state the FSM is in when thread 1 finishes processing the first segment. Such dependences inherently prevent concurrent execution of any two threads. The difficulty motivates our focus on FSM applications: Solutions for parallelizing such applications will naturally shed insights on circumvention of dependences in general, hence provide new understanding to the overall problem of speculative parallelization.

Meanwhile, tackling FSM applications is important for some immediate usage. FSM computation is the core of many important applications in various domains, including lexing in web browsers, intrusion detection in networking security, Huffman decoding in multimedia processing, string pattern matching in document processing, model checking for software and hardware testing [5, 14, 16, 19]. Effective parallelization is critical for their scalability and responsiveness, especially as they run on low-frequency portable devices that are embracing increasing parallelism.

State of the Art Among various forms of FSM, Deterministic Finite Automaton (DFA) has been the focus in prior studies, thanks to its common usage and capability to approximate other forms of automaton (e.g. pushdown automaton [6].) We hence focus our discussion on such FSMs.

As seen, the key difficulty for parallelizing FSM applications is to determine the starting state for each thread. Researchers have attempted to circumvent the difficulty by speculation—that is, letting a thread T_i guess the correct starting state to process segment S_i, i.e. the ending state after processing the preceding segment S_{i-1}. A random guess is subject to large errors. Researchers have found it helpful to do a **lookback**—that is, thread T_i runs the FSM on a number of ending symbols (called a *suffix*) of the preceding segment S_{i-1}, and uses the ending state as the speculated starting state for processing its own segment S_i. The lookback helps speculation by offering some context. For instance, for the FSM in Figure 1, if the suffix of S_{i-1} is "11", a lookback will always finish at an even-numbered state. It is easy to see that the correct ending state on that segment S_{i-1} can only be an even-numbered state since transitions to all the odd-numbered states require "0" as the trigger. The lookback hence successfully prevents the speculation from taking these impossible states.

Lookback-based speculative parallelization has been the central technique of all state-of-the-art FSM parallelizations [14, 19]. As Figure 2 shows, on an 8-core Intel system, the approach [14, 19] yields almost ideal speedups on the Huffman decoding and XML lexing. However, its performance is inconsistent. On the other five programs , it produces speedups less than two. One of the programs, *div*, even runs slower than its sequential execution.

The main reason for the inconsistent performance is the lack of rigor in the designs of speculation, reflected in multiple dimensions. The *first* is in the length of the suffix to examine by a lookback. A longer suffix exposes more context, but at the same time incurs more overhead. Previous studies [14, 19] select it by trying several lengths in profiling runs, rather than systematically examining the full spectrum of possible lengths. The *second* dimension is in the selection of the state for starting a lookback. Previous studies use the initial state of the FSM for all lookbacks, which limits lookback benefits. For instance, suppose the correct starting state for thread 2 to process segment S_2 is state 12 in the FSM in Figure 1. A lookback by that thread on suffix "00" would end at state 2 if it starts from the initial state, state 0, causing a large disparity from the correct state. The *third* dimension is in the usage of lookback results. All prior studies have used the ending state of a lookback as the speculated starting state for processing the next segment, which may not be the best choice. Without rigorous ways to treat these factors and dimensions in the design, existing parallelizations are vulnerable to FSM complexities, yielding the inconsistent speedups.

* This work is supervised by Xipeng Shen

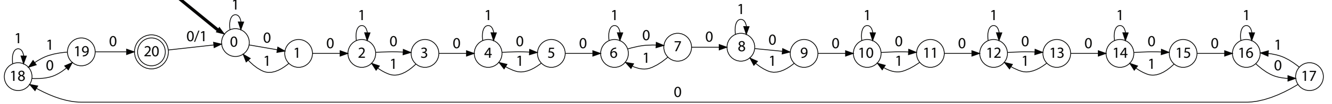


Figure 1. An FSM for matching regular expression $([01]^*00[01]^*11[01]^*)^{10}$. Each circle represents an FSM state. State 0 is the initial state (marked by the extra incoming edge), and state 20 is an acceptance state. The symbols on the edges indicate conditions for state transitions.

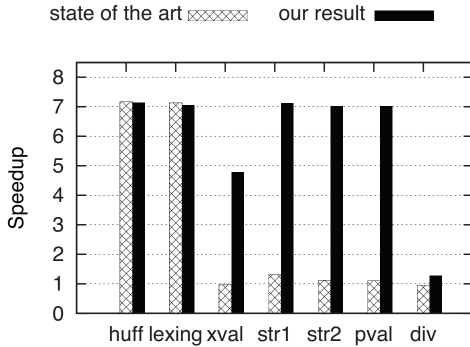


Figure 2. The speedups brought by the state-of-the-art speculation scheme [19] are limited on some complex FSM applications. The results are for 8 threads running on a 8-core machine (Section 5).

Overview of This Work In this work, we conduct a four-fold analysis into the nature of lookback-based speculative parallelizations. At the core of the analysis is a stochastic perpective on the speculative parallelization. The answers to its design questions should be adaptive to the *probabilistic* properties of the target FSM.

First, we reveal that lookback is essentially a process that converts context-free (no-suffix) state feasibilities into conditional feasibilities with the suffix gradually put into consideration. We develop a model of *state feasibility propagation* to formulate the effects and generate the conditional feasibilities (Section 2.1.)

Second, we introduce the concept of *expected merging length* to help model the penalty of a speculation. An execution based on a wrong starting state may not be completely useless. Consider a string segment "00100010" to the FSM in Figure 1, if the true starting state is state 4 but the speculation is state 5. It is easy to see that although the processing of the first "001" is wrong, after that, the execution ends at state 6 where the correct execution would end. So, the process of the remaining string "00010" will be correct. The penalty of the wrong speculation is reprocessing the first three characters. Expected merging length characterizes the statistical expectation of the number of state transitions needed for two states to converge. Together with state feasibility, it enables the computation of the expected risk of a speculation, making optimal usage of lookback results possible (Section 2.2.)

Third, we integrate the expected speculation risk and lookback overhead into a unified model for quantifying the statistical expectation of the overall performance of a speculative execution (Section 2.2.) It lays the foundation for analytically selecting the best lookback length and starting state(s) for lookback.

Finally, based on the set of probabilistic models, we develop two model-based speculation schemes, single-state and all-state lookback schemes, which automatically customize themselves with the configurations that best suite the probabilistic properties of an FSM and its inputs so that the parallelization benefits can be maximized (Section 3.) Both online and offilne profilings are considered. For practical deployment, we integrate all the techniques into a library with a simple API (Section 4.)

The benefits brought by the systematic analysis are significant. It improves the speculation accuracy and minimizes the penalty of wrong speculations. As shown in Figure 2, our technique boosts the speedups by more than a factor of four over the state of the art. It yields near optimum performance on five programs, and reverses the slowdown on *div* to a 31% speedup (Section 5.)

The value of this work goes beyond the parallelization of FSM applications, in two aspects. First, what this work produces is essentially a rigorous approach to circumventing dependence chains such that the parallelization benefits can be maximized. It is potentially applicable to dependence chains in many kinds of applications. Second, the simple way of using heuristics is not unique to FSM parallelization. Most speculative parallelizations beyond FSM share the same manner. For instance, function-level speculative executions use some recent history as the clue to predict the return value of a function invocation. There is no rigorous analysis on the best history length and other relevant factors [18]. This work may shed insights to these speculative parallelizations.

2. Probabilistic Analysis of FSM Speculation

This section presents a probabilistic formulation for modeling the expectation of the benefits from an FSM speculative parallelization. The formulation enables assessment of different designs of speculative parallelization, and hence lays the foundation for creating optimal speculative schemes as later sections will show.

2.1 Essence of Lookback

Lookback, the key operation for speculation accuracy, is essentially a process that tries to use the context (i.e., a suffix, $C_1 C_2 \cdots C_l$) to improve the knowledge about the chance for a state to be the correct state at a given speculation point. For convenience, we introduce the term *feasibility* as follows:

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

Without consideration of contexts, statistically, the feasibility of a state s at every speculation point is the same, approximately equaling the frequency for the FSM to visit that state in normal executions. We call these probabilities *initial feasibilities* or *context-free feasibilities*, denoted with $P^0(s)$. Correspondingly, we call the feasibilities after an l-long input string *conditional feasibilities*, denoted with $P^l(s)$ ($l > 0$).

A straightforward way to estimate the conditional feasibility, $P^l(s)$ or equivalently P(real state=s | left string=$C_1 C_2 \cdots C_l$), is to count the frequency for s to appear after a string $C_1 C_2 \cdots C_l$ in profiling runs. But because the value space of $C_1 C_2 \cdots C_l$ grows exponentially with l, the approach is generally infeasible.

Our approach to estimate the conditional feasibility centers on the following observation: State transitions essentially lead to an increamental propagation of conditional feasiblities, with contexts enriched gradually, as Figure 3 illustrates. For each state transition, it inludes two feasibility updates, named inner-stage and inter-stage updates, shown as the downward arrows and upright arrows respectively in Figure 3. We leave their derivations in a technical report [25] for lack of space.

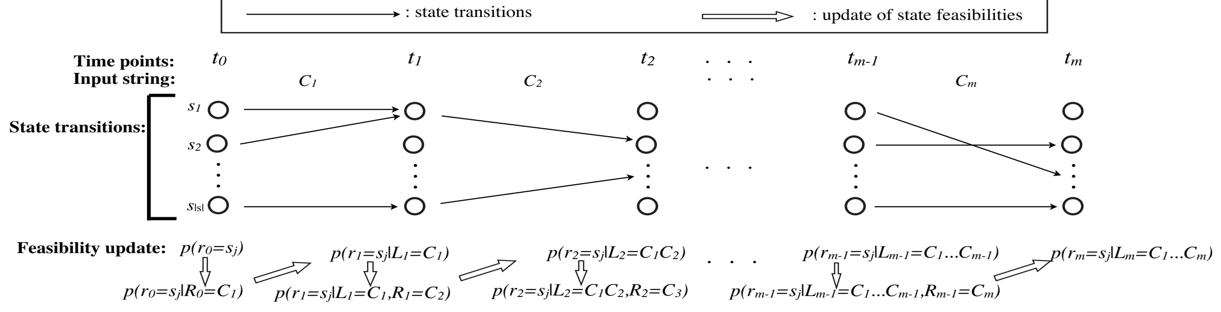


Figure 3. Gradual refinement of conditional feasibilities along with state transitions. The state transitions graph in the middle shows all possible transitions allowed by the FSM on the input characters.

Lookback exploits the property that the conditional state feasibilities, to a certain degree, are dictated by the state transitions specified by the FSM. It can quantitatively update each state's feasibility, and even prune impossible states, i.e. the states with feasibility equal zero, for speculation.

2.2 Formulation of Performance Expectation

In the performance formulation, we assume that T threads try to process an evenly divided input string based on a given FSM. We use *expected makespan*[1] as the performance metric, defined as follows:

Definition 2. *Given an FSM application, the* makespan *of a thread's speculative execution is the time elapsed from its start to its finish; the* makespan *of the FSM application's speculative execution is the time elapsed from the starting of the first started thread to the end of the last finished thread. The* expected makespan *is the statistical mean of the makespans of all executions of the application on inputs of a given length. It is denoted as EM.*

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

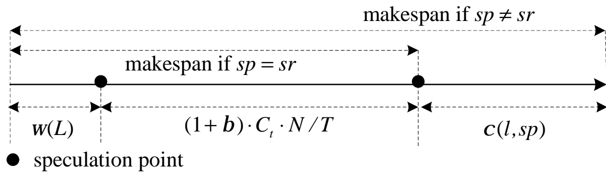


Figure 4. The makespans of a thread, in cases of correct speculation ($sp = sr$) and wrong speculation ($sp \neq sr$), where, sp and sr are the speculation and real states respectively.

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

2) Workload Processing Time The workload processing time includes the time needed by a state transition, and often some additional operations to consume the current results of the FSM execution. Given an input segement with length N, The processing time can be expressed as $(1 + \beta) \cdot C_t \cdot N/T$, where C_t is the time consumed by one state transition, and β is the cost for additional operations.

[1] Makespan is a standard term in scheduling, referring to finish time.

3) Reexecution Time To compute the reexecution time, one must consider state convergence. Even though the speculation state sp may differ from the real state sr, state transitions starting from them tend to converge gradually. For example, when the FSM in Figure 1 sees string "001000", no matter it starts with state 4 or state 5, after processing the first three characters "001", it always reaches state 6. We call the number of state transitions needed before two states converge *the merging length* of the two states.

Apparently the merging length depends on input strings and the real state sr. To compute the expected makespan, we use the expectation of the merging length across all inputs and all possible true states, denoted as $L_M(sp)$.

Suppose after an l-long lookback, the feasible states set (i.e., the set of states whose feasibilities are positive) is S_l and feasibilities are $\{P^l(s)|s \in S_l\}$. The expected merging length between state sp and real state sr is:

$$L_M^l(sp) = \sum_{s_i \in S_l} L_M(sp, s_i) \cdot P^l(s_i), \tag{1}$$

where, $L_M(sp, s_i)$ is the statistical expectation of the merging length of sp and s_i on all possible inputs.

As the actual reexecution length cannot be larger than the length of the input segment (N/T), $min\{L_M^l(sp), N/T\}$ 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

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

Putting All Together The sum of the three components gives the makespan of a thread. For the makespan of the entire execution, note that the reexecutions have to happen in serial due to the state dependence happened at each speculation point.

Without loss of generality, assume that all threads start at the same time. Let $\mathbb{S}$ be a vector containing the speculated states of all threads in one execution. The expected makespan is as follows:

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

where $sp(i)$ is the speculated state of thread i, that is, the ith element in $\mathbb{S}$.

This formulation of performance expectation plays a fundamental role by formalizing the goal of an optimal design of speculation schemes. 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 $P^l(.)$) may not be the best strategy.

3. Towards Optimal Designs

In this section, we first discuss the major dimensions in designing a speculation scheme for FSM computations. Then, we demonstrate the use of the formulations provided in earlier sections to optimally configure two speculation schemes.

3.1 Design Dimensions

There are three main dimensions in configuring a lookback-based speculation scheme for FSM computations. The first is lookback length. The effects of a long lookback are mixed: By exploiting a long suffix, it tends to reduce misspeculations and hence reexecution time, but meanwhile, it 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 starting state for lookback, which restraints the lookback benefit. As we will show, a larger starting state set tends to yield a better speculation result, by paying off higher overhead.

The third dimension of design is the selection of lookback results for speculation. When the starting 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 0 and 3 as the starting states for lookback for the FSM in Figure 1, on a suffix "010", the two lookbacks will end up at states 1 and 5 respectively. Choosing the best lookback ending state for speculation is the core question in this dimension.

These three dimensions interrelate with one another. Designs in all these dimensions together determine the quality of a speculation scheme. In this section, we concentrate on two configurations of the second dimension. One uses the complete state set S as the lookback starting state set, the other uses a single state (adaptively determined) as the lookback starting state set. Our analysis will demonstrate how the formalization described in the previous sections makes it possible to configure the two speculation schemes optimally.

Using a subset of S as lookback starting state set leaves some state transitions unexamined during the lookback, Some approximations may have to be used as remedy when computing conditional state feasibilities. Details are out of the scope of this paper.

3.2 Speculation through All-State Lookback

In this scheme, the lookback uses the complete state set as the starting state set. The key design questions are the determination of the optimal lookback lengths and the selection of the optimal lookback ending states for speculation.

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: After the lookback, each state s would get a latest state feasibility, the expected merging length can be computed by Equation 1, that is the expected rexecution length when s is selected for speculation. The opitmal speculation state is the one with minimal $L_M^l(s)$, noted as s^*.

Finding Optimal Lookback Length The selection of the optimal lookback length is based on the expectation of makespan (i.e., Equation 3.) The first two components of the makespan are easy to compute. The third component is the sum of all threads' reexecution overhead, which is unavailable before the execution finishes. To approximate it, we use a number of suffixes to do l-long lookbacks and get the average rexecution overhead. Then, the curve fitting is applied to the first and third components. The optimal lookback length can be obtained by solving Equation 3.

3.3 Speculation through Single-State Lookback

In the prior schemes [14, 19], only the default initial state is used as the starting state for lookback. In this sub-section, we show that when the probabilistic analysis applies to single-state lookback, it can easily enhance the quality of such schemes.

We start with its makespan. If we use l_b and l_x to represent the lookback length and expected reexecution length respectively, we can rewrite the makespan equation, Equation 3, to

$$EM(l_b) = l_b{\cdot}(1{+}\lambda){\cdot}C_t{+}(1{+}\beta){\cdot}C_t{\cdot}N/T{+}(T{-}1){\cdot}l_x{\cdot}(1{+}\beta){\cdot}C_t. \tag{4}$$

Let s'_d represent the starting state of a lookback. The makespan equation can be simplified with the following lemma:

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

Putting l_x values from Lemma 1 into Equation 4, the makespan 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 optimal lookback length equals $L_M^0(s'_d)$, and the expected makespan equals $L_M^0(s'_d){\cdot}(1{+}\lambda){\cdot}C_t{+}N/T{\cdot}(1{+}\beta){\cdot}C_t$, where s'_d is the lookback starting state.*

We prove the lemma and theorem in our technical report [25].

All parameters in the theorem, including $L_M^0(s)$ ($s \in S$), can be obtained through profiling (Section 4.) Based on this theorem, one can easily compute the minimum expected makespan $min_em(s)$ for each s. The optimal state to use for lookback is just the one whose $min_em(s)$ is the smallest; its corresponding optimal lookback length is the overall optimal lookback length. This gives the configuration that minimizes the expectation of makespan.

4. 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, starting states, selection of speculation states) on the fly to suite the properties of the FSM and inputs.

The profiler can run either online or offline. The online profiler has an adaptive switch. If the overhead is larger than 10% of the single-thread workload processing time, it stops and falls back to the default simple heuristic-based parallelization.

OptSpec ***Library*** 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. Its most important API are

OptSpec_offlineProf (string* input, FILE* profFile, FSM* fsm, void* action, void* args, int scheme);
OptSpec_specPar (string* input, FILE* profFile, FSM* fsm, void* workload, void* args, int scheme).

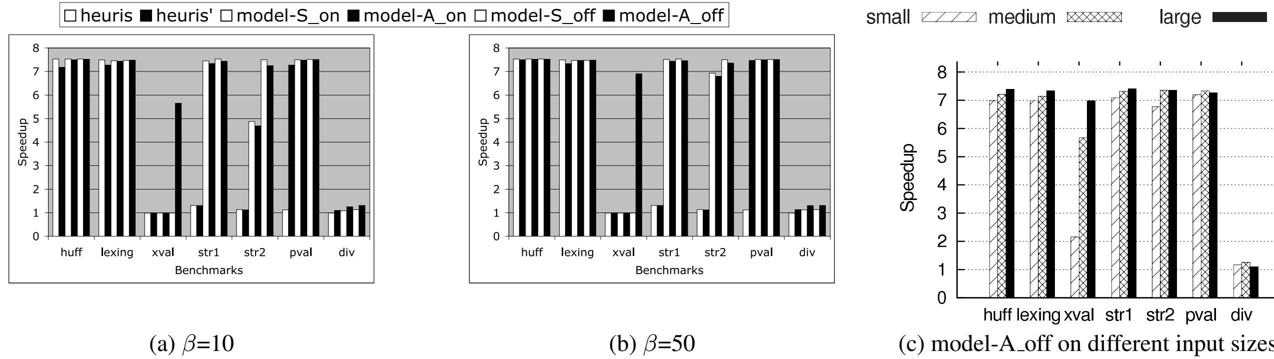
Figure 5 shows how the library can be used for parallelizing a regular expression matching application. As demonstrated, to use *OptSpec* for parallelization, users do not need parallel programming or debugging. Details are in our technical report [25].

5. Evaluation

We evaluate the proposed techniques on seven FSM applications listed in Table 1. These programs come from the web XML processing community (e.g., *lexing* and *xval* [24]), mathematics (e.g.,

Table 1. Benchmarks

Name	Description	$\|S\|$	$L_M(s,r)$	$P^0(s)$	L^*	Training Input	Testing Input
huff	Huffman Decoding	46	4∼25	0∼0.21	23	1.60MB	209MB
lexing	XML Lexing	3	1.0∼6.8	0.06∼0.5	2	1.60MB	76MB
xval	XML Validation	742	∞	0∼0.054	229	1.70MB	57MB
str1	String Pattern Search 1	21	1.9∼4298	0.016∼0.066	1279	1.60MB	96MB
str2	String Pattern Search 2	41	1000∼40000	0.008∼0.033	32767	1.64MB	96MB
pval	Pattern Validation	28	0∼∞	0∼0.50	0	1.7MB	96MB
div	Unary Divisibility	7	∞	0.143∼0.144	0	1.7MB	97MB



(a) β=10 (b) β=50 (c) model-A_off on different input sizes

Figure 6. The overall speedup when 8 threads are used. **heuris:** Previous scheme [19]. **heuris':** Previous scheme [19] with simple extensions. **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.

```
void * recPos (char * buff){...} // user defined action function
...
OptSpec_setThdNum (8);// thread number
read (regex);
OptSpec_regex2fsm (regex, fsm);// build FSM
read (input);
// speculative execution. Replace it with OptSpec_offlineProf for offline profiling.
OptSpec_specPar (input, f, fsm, recPos, buffArr,  OptSpec_ALL);
highlight (buffArr, ...);
```

Figure 5. Using *OptSpec* to parallelize a string matching program.

div [3]), classical Huffman decoding (*huff* [16]), and string pattern matchings (*str1*, *pval*, and *str2* [1].)

Our experiments run on a 8-core machine equipped with two Intel Xeon E5620 processors. The machine runs Linux 2.6.22 and has GCC 4.4.1 as the compiler (optimization "-03" is used for all.)

For each benchmark, we compare the results from different speculative executions: The *heuris* shows the performance from the state-of-the-art scheme described in recent work [19]. It has lookback and other recent techniques incorporated, but relies on simple heuristics and is not adaptive to FSM properties or input strings. We implement the scheme with three lookback lengths, 32, 128, 512, that are used by the previous study [19] and use the best performance for *heuris*. The *heuris'* version is a simple extension to the previous technique in that it uses the state with the largest context-free probability $P^0(s)$ as the starting state for lookback. Again, we try the three lookback lengths and report the best results of this version. We include this version to see whether a simple extension is sufficient to address the limitations of the previous techniques. The other four versions are our methods with either online or offline profiling.

Figures 6 (a) and (b) report the overall speedups compared with the sequential performance when 8 threads are used with different β values. Figure 6 (c) reports the influence of input size on the performance of model-A_off with β=10, where the sizes are in gemometric progression with the common ratio 5. The "medium" size is the input size listed in Table 1.

Overall, the model-based approaches outperform the previous techniques significantly, regardless β values, and input sizes. The online profiling incurs some noticeable overhead on *str2* when β is small. For a large FSM, it shuts down automatically and is remedied by our offline model-based scheme, as illustrated on *xval*. The single-state model performs as well as the all-state methods in most cases. An exception is *xval*, whose average state merging length is infinity; speculation accuracy is hence critical. Thanks to the use of all states for lookback, the all-state method is able to select the state with the highest speculation accuracy (57% versus 0), producing the largest speedup. The results of *heuris'* show that the simple extension to the previous techniques is insufficient.

6. Conclusion

This paper introduces formal analysis into speculative parallelization by formulating FSM speculative executions and the connections between the design of speculation schemes and the characteristics of FSM and their inputs. It deepens the understanding to speculative execution of FSM computations with a series of theoretical findings, including the essence and effects of lookback for speculation, the connections between state transitions and conditional feasibilities, the relation between partial committing and overall running times. It provides two model-based speculation schemes, with suitable configurations automatically determined. Experiments show that the new techniques outperform the state of the art substantially. By integrating the techniques into a library, this work makes the techniques easy to adopt. The findings may apply to applications beyond FSM, prompting more studies in shifting speculative parallelization to a rigorous paradigm. In [25], we demostrates this potential through a case study, parallelizing command dialogue systems, and leave full generalization to the furture.

References

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

[2] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, 2008.

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

[4] J. Allen, D. Byron, M. Dzikovska, G. Ferguson, L. Galescu, and A. Stent. Towards conversational human-computer interaction. *AI Magazine*, 2001.

[5] 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.

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

[7] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming langauges. In *PLDI*, 2006.

[8] 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.

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

[10] M. Feng, R. Gupta, and Y. Hu. Spicec: Scalable parallelism via implicit copying and explicit commit. In *PPOPP*, 2011.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, 1998.

[12] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *HPCA*, 1993.

[13] Y. Jiang and X. Shen. Adaptive software speculation for enhancing the cost-efficiency of behavior-oriented parallelization. In *ICPP*, 2008.

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

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

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

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

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

[19] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *PLDI*, 2010.

[20] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS*, 2010.

[21] 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.

[22] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Micro*, 2008.

[23] 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.

[24] 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.

[25] Z. Zhao, B. Wu, and X. Shen. Probabilistic Models towards Optimal Speculation of Finite State Machine Applications Technical Report WM-CS-2011-03, The College of William and Mary, 2011.