Cache-Conscious Automata for XML Filtering

Bingsheng He, Qiong Luo, and Byron Choi

Abstract—Hardware cache behavior is an important factor in the performance of memory-resident, data-intensive systems such as XML filtering engines. A key data structure in several recent XML filters is the automaton, which is used to represent the long-running XML queries in the main memory. In this paper, we study the cache performance of automaton-based XML filtering through analytical modeling and system measurement. Furthermore, we propose a cache-conscious automaton organization technique, called the hot buffer, to improve the locality of automaton state transitions. Our results show that 1) our cache performance model for XML filtering automata is highly accurate and 2) the hot buffer improves the cache performance as well as the overall performance of automaton-based XML filtering.

Index Terms—Cache-conscious, automata, XML filtering, query processing, cache behavior model, buffer.

1 Introduction

XML filtering is a newly emerged query processing paradigm, in which a large number of XML queries reside in the main memory and incoming XML documents are filtered through these queries. There are many applications of XML filtering, such as selective information dissemination and content-based XML routing. Since an XML filtering engine is typically memory-resident and long-running, we study the cache performance of such engines and investigate if their overall performance could be improved through cache performance improvement.

When we examine several recent XML filtering engines [3], [6], [12], [13], [14], we find that they are all based on automata (either an NFA, a Nondeterministic Finite Automaton, or a DFA, a Deterministic Finite Automaton). NFA-based approaches are space efficient because they require a relatively small number of states to represent a large set of queries. In comparison, DFA-based approaches are time efficient because their state transitions are deterministic. Since the conversion from an NFA to a DFA through subset construction [27] increases the number of states exponentially, recent work such as the lazy DFA [14] chose to convert an NFA into a DFA lazily at runtime.

To study the cache performance of automaton-based XML filtering, we implemented an *XPath-NFA*, a simplified version of the YFilter [12], [13]. This XPath-NFA represents a large number of XPath queries with the *path sharing* technique (sharing common expressions among queries), but without other optimization techniques. We then converted an XPath-NFA to an *XPath-DFA* through subset construction and studied the filtering performance of both the XPath-NFA and the XPath-DFA. Not surprisingly, the XPath-DFA outperformed the XPath-NFA to a large extent, as long as the subset construction succeeded without running out of memory. Therefore, we use an XPath-DFA for our subsequent studies on cache performance.

In automaton-based XML filtering, a commonly used data structure is the hash table [12], [13], [14], [22] for its space efficiency. Nevertheless, a traditionally used data structure for automata is the matrix [22]. In this paper, we investigate both implementation schemes. Figs. 1a and 1b show the time breakdown of filtering five sequences of XML documents using a memory-resident XPath-DFA with the matrix and the hash table implementation schemes, respectively, on a P4 machine. Details of the experiment are described in Section 5. In these figures, the filtering time of each sequence is divided into four categories: L2 data stalls (L2 DStalls), L1 data stalls (L1 DStalls), other stalls (e.g., stalls due to L1 instruction misses and branch mispredictions), and CPU busy time. These figures show that the XPath-DFA of these two implementation schemes spent a large portion of the running time—more than 50 percent —on stalls due to data cache misses, especially L2 data cache misses. Therefore, we see considerable room for improvement in the cache performance of automaton-based XML filtering.

Since cache-conscious techniques [2], [8], [9], [15], [23], [24], [25], [28] have been shown useful in performance improvement of various database structures and operations, we explore if and how some existing cache-conscious techniques are applicable to automaton-based XML filters. Unfortunately, due to the random access nature of automaton state transitions, it is difficult to apply state-of-the-art cache-conscious data layout techniques, such as the Morton layout [7], to automaton-based XML filtering. General-purpose cache-conscious techniques such as blocking [25] are also inapplicable in this case.

In order to propose cache-conscious techniques that work for automaton-based XML filtering, we developed an analytical model on its cache performance. In this model, we define a *round* as the engine filtering one document. We then estimate the total number of cache misses in filtering a single document (*intraround filtering*) and a sequence of

- B. He and Q. Luo are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, 1 University Road, Clear Water Bay, Kowloon, Hong Kong. E-mail: {saven, luo}@cse.ust.hk.
- B. Choi is with the Division of Information Systems, School of Computer Engineering, Nanyang Technological University, N4 02c-117b, Singapore. E-mail: kkchoi@ntu.edu.sg.

Manuscript received 22 Sept. 2005; revised 23 Apr. 2006; accepted 30 July 2006; published online 18 Oct. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0397-0905.

1041-4347/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

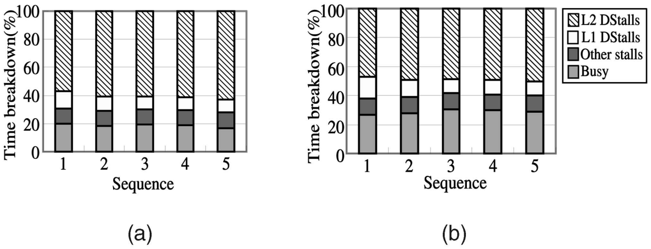


Fig. 1. Execution time breakdown of XML filtering engines on P4. (a) Matrix-based implementation. (b) Hash table-based implementation.

documents (*interround filtering*). We analyze the cache behavior of the state transitions of an XML filtering automaton and estimate the cache misses in three categories—compulsory misses, capacity misses and conflict misses [17]. Additionally, we apply our model to both the hash table and the matrix implementation schemes.

Based on the insights gained from the model, we propose to extract *hot spots* (frequent state transitions) in an automaton into an in-memory data structure called a *hot buffer*. This hot buffer aims at improving both the spatial and the temporal locality of the filter. We organize the hot buffer either contiguously, called a *contiguous hot buffer (C-Buffer)*, or in segments, called a *segmented hot buffer (S-Buffer)*. The hot spots in a C-Buffer are stored in an array whereas those in an S-Buffer are stored in a graph. The C-Buffer achieves a better cache performance than the S-Buffer at the absence of changes in the hot spots but lacks support for incremental maintenance upon these changes. In contrast, the S-Buffer supports incremental maintenance efficiently.

In comparison with our previous work [16], this paper makes the following four contributions:

1. we handle both simple XPath queries and twig queries with branches in our XML filters,
2. we compare the matrix-based and the hash table-based implementation schemes, and extend our analytical model to both implementation schemes,
3. we propose the segmented hot buffer to support incremental maintenance, and
4. we conduct a more comprehensive empirical study to validate the accuracy of our model and the effectiveness of our cache-conscious techniques.

The remainder of this paper is organized as follows: Section 2 introduces the preliminaries of this work and gives an overview of our filtering system. Section 3 presents our analytical model for estimating the cache misses in XML filtering. Section 4 presents our cache-conscious techniques, the C-Buffer and the S-Buffer. In Section 5, we experimentally validate our model and evaluate the performance of our approach. We briefly discuss related work in Section 6 and conclude in Section 7.

2 Preliminaries and Overview

In this section, we describe the XPath queries considered, provide an overview of our filtering system, and discuss representative automaton implementation alternatives.

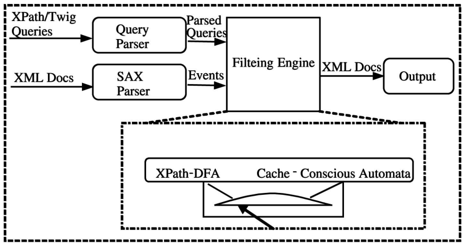


Fig. 2. Architecture of our XML filtering system.

2.1 Queries Handled

We handle a basic subset of XPath queries [11] without branch conditions, which we call *linear path queries*, or *path queries* in short, as well as twig queries [10] that have branch conditions. The set of queries handled is defined as follows:

$$\begin{aligned}&\text{Twig} ::= \text{Path}|\ \text{Path}\ ''[\,''\ \text{Twig}\ (\,''\text{and}''\ \text{Twig})^*\ ''\,]\,''\\&\text{Path} ::= \text{Step}\mid\text{Path Step}\\&\text{Step} ::= \text{Axis Node\_test}\\&\text{Axis} ::= ''/\,''\mid''//''\\&\text{Node\_test} ::= \text{Tag}\mid\ ''*\,''.\end{aligned}$$

This grammar defines a twig query consisting of a sequence of *step*s. A step in turn consists of 1) a downward XPath axis (either "/," a child axis, or "//," a descendant axis) and 2) a node test (either an XML element tag or a wildcard "*").

Given a twig query, we decompose it into path query conjuncts and transform its evaluation into the evaluation of these path queries. When all of these path queries are satisfied, the twig query is satisfied.

2.2 System Overview

The system architecture of our XML filtering system is shown in Fig. 2. The major components of this system include a query parser for decomposing twig queries into path queries and parsing path queries, an event-based SAX parser [1] for parsing incoming documents, and an automaton-based filtering engine that filters the documents. The automaton in the filtering engine can be either the XPath-DFA or our cache-conscious automaton with the C-Buffer or the S-Buffer. We describe our cache-conscious automata in detail in Section 4.

2.2.1 The Filtering Process of the XPath-DFA

The filtering process is driven by the events generated from the SAX parser. There are four types of events: start-of-document, end-of-document, start-of-element, and end-of-element. A start-of-document event triggers a new round whereas an end-of-document event indicates the end of a round. During a round, a runtime stack is used to keep track of the current state as well as previously visited states. When a start-of-element event occurs, it triggers a state transition in the automaton. When an end-of-element event occurs, the automaton backtracks to the previous state. Upon an end-of-document event, the system checks if any accepting state has been reached and disseminates the document to users whose queries are satisfied.

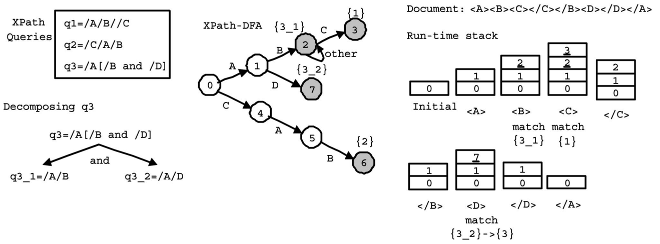


Fig. 3. XML filtering process.

An example of the filtering process is shown in Fig. 3. On the top-left are the queries. The twig query $q3$ is decomposed into two path queries $q3\_1$ and $q3\_2$. In the middle is a fraction of the XPath-DFA that represents the queries. The circles in the DFA represent states and the shadowed ones represent accepting states. An identifier inside "{}" on top of an accepting state is the ID of a query. We say this query is associated with the accepting state. When an accepting state is reached, the queries associated with this accepting state are satisfied. A twig query is satisfied only when all of its decomposed path queries are satisfied. In this example, if queries $q3\_1$ and $q3\_2$ are both satisfied, $q3$ is satisfied. The content of the runtime stack is shown on the right of the figure, as the XML document on the top-right streams through the XPath-DFA.

In the remainder of this paper, we use the term "symbol" from the automaton community and the terms "XML element" or "element" and "XML tag" or "tag" from the XML community interchangeably.

2.3 Automaton Implementation

We briefly review two commonly used alternatives for automaton implementation, the hash table [12], [13], [14], [22] and the matrix [22], as illustrated in Fig. 4. In the hash table implementation, buckets are indexed by states. A transition from one state is represented as a hash entry $<\text{symbol, next state}>$ and stored in the corresponding bucket. In the matrix implementation, neighboring symbols are coded as consecutive integers. Columns are indexed by symbols and rows are indexed by states. The cell at row i and column j in the matrix is the next state for the state i upon the symbol j.

Due to the random access nature of hash tables, a state transition causes more than two cache misses on average, one for fetching the bucket and the others for fetching the values. State transitions in a matrix are also random access and each causes one cache miss on average (directly fetching the state). However, regardless of the number of outgoing transitions from a state, the matrix implementation needs a row consisting of S cells for the state, where S is the number of symbols. Consequently, the matrix implementation often has a large space overhead.

We have experimented with these two implementation alternatives in automaton-based XML filtering. We find that matrices are mostly a winner in practice, as long as the matrix can fit into the memory. We have developed a model to compare the cache performance of both alternatives as well as hot buffer techniques to improve the overall performance of both alternatives.

3 Modeling Cache Performance

In this section, we propose an analytical model to estimate the total number of cache misses in automaton-based XML filtering. We estimate the number of compulsory misses and capacity misses by modeling the intraround and the interround filtering processes. The intraround model serves as part of the interround model. We then describe a coarse estimation technique for the estimation of conflict misses.

3.1 Modeling the Cache

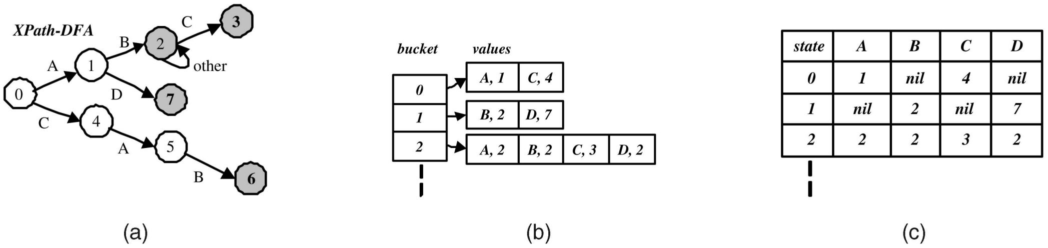
The memory hierarchy on a modern computer typically contains a Level 1 cache (L1 cache), a Level 2 cache (L2 cache), and the main memory. Access to a cache (either L1 or L2) is either a hit or a miss. Given the number of cache misses, $\#miss$, and that of cache hits, $\#hit$, the *miss rate*, m, is defined as $m = \frac{\#miss}{\#miss+\#hit}$. The average cost of one memory access T_{avg} is:

$$T_{avg} = t_{L1} + m_{L1} \cdot t_{L2} + m_{L1} \cdot m_{L2} \cdot t_M, \quad (1)$$

where t_{L1}, t_{L2}, and t_M are the access time of the L1 cache, the L2 cache and the main memory, respectively; m_{L1} and m_{L2} are the miss rates of the L1 cache and the L2 cache, respectively. Since the access time is determined by the hardware, software techniques aim at minimizing the miss rates and the total number of cache accesses.

We define the *cache configuration* as a four-element tuple $< C, B, N, A >$, where C is the cache capacity in bytes, B the cache line size in bytes, N the number of cache lines, and A the degree of set-associativity. $A = 1$ is a direct-mapped cache, $A = N$ a fully associative cache, and $A = n$ an n-associative cache $(1 < n < N)$.

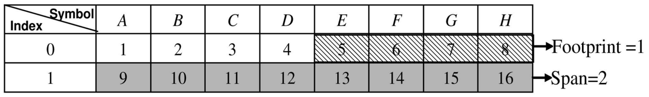
As we have seen in Fig. 1, L2 data stalls are the most significant stalls in the XML-filtering system. Consequently, we focus on the L2 data cache in our model, even though the L2 cache is usually a piece of unified memory shared by data and instructions. *In the remainder of this paper, "cache" is*



state	A	B	C	D
0	1	nil	4	nil
1	nil	2	nil	7
2	2	2	3	2

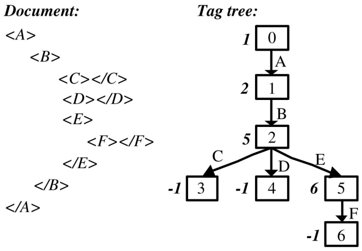
(a) (b) (c)

Fig. 4. An example automaton and its two implementation alternatives. (a) Example automaton. (b) Hash table. (c) Matrix.



Index \ Symbol	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	
0	1	2	3	4	5	6	7	8	→Footprint =1
1	9	10	11	12	13	14	15	16	→Span=2

Fig. 5. The footprint and the span of a matrix-based automaton (one cache line can hold four state transitions.



Document:

```
<A>
    <B>
        <C></C>
        <D></D>
        <E>
            <F></F>
        </E>
    </B>
</A>
```

Fig. 6. An example of tag tree.

short for "L2 data cache" unless specified otherwise. In addition, the cache in our model is fully associative and uses LRU (Least Recently Used) as the cache replacement policy.

3.2 Modeling Intraround Filtering

We start with intraround filtering, during which a single document is filtered and there is no reuse of data from previous rounds. Since a single document is small in practice, we assume that there are no capacity misses in intraround filtering. Subsequently, we focus on the estimation of compulsory misses in intraround filtering.

Definitions. *We model the cache behavior of a state transition, t, in an automaton implementation using three metrics, $< F_t, R_t, M_t >$:*

- F_t represents the footprint of t. *Footprint F is the mean number of cache lines that are required to access for a state transition. Specifically, F_t has a constant value of one in the matrix implementation, and varies for the hash table implementation. For simplicity, we model F_t as a constant and use F_t and F interchangeably.*
- R_t represents the reuse of t. *Reuse R_t is the number of cache lines that are required for t and have already been brought into the cache by previous state transitions. If $R_t = F_t$, we call it a perfect reuse.*
- M_t represents the number of compulsory misses caused by t. $M_t = F_t - R_t$.

To model the cache characteristics of a state, we define the *span* (denoted as P_s) of a state, s, as the number of cache lines required to store the transition information of s in an automaton implementation. P is the mean span of all states in the automaton. For instance, P_s has a constant value equal to the row size of the matrix in the matrix implementation, and varies for the hash table implementation. Again, for simplicity, we model P_s as a constant and use P_s and P interchangeably. Suppose there are eight symbols in a matrix-based automaton and a cache line can hold four state transitions, we have $F = 1$ and $P = 2$ (Fig. 5).

Finally, we define the *working set*, ws, of an XML document over an automaton as the set of states in the automaton that are referenced during filtering the document. The working set size, $|ws|$, is the number of states in the working set.

Estimation. Based on the definitions, we have two ways to estimate the number of compulsory misses, Com, in the intraround filtering of a document: 1) $Com = \sum_{t=1}^{p} M_t = \sum_{t=1}^{p} (F - R_t)$, where p is the total number of state transitions in filtering the document and 2) $Com = \sum_{s \in ws} CM(s)$, where $CM(s)$ gives the estimation of compulsory misses caused by the transitions made from s. Either method can be used to validate the other; for simplicity, we used the second way in our estimation algorithm.

Implementation. To simplify our estimation of the working set for a document on an XPath-DFA, we construct a DFA solely based on the document, to represent the access pattern of filtering the document on an XPath-DFA. Since this document DFA is acyclic and its symbols are element tags, we call it a *tag tree*.

To simulate the LRU replacement policy, we add a *timestamp* to each node. The timestamp of a node is -1 initially, indicating that the state it represents has not been fetched into the cache. Whenever a transition is made from a state, the timestamp of the corresponding node for the state is updated to a positive value equal to the total number of state transitions occured so far. A node with a positive timestamp indicates that the state it represents resides in the cache, and the working set can be represented as the set of tag tree nodes with a positive timestamp.

We use an example to illustrate the construction of a tag tree (Fig. 6) and the complete algorithm of tag tree construction can be found in our previous paper [16]. As shown in Fig. 6, the rectangles represent tag tree nodes. The number in a node is the state and the number beside a node is the timestamp. After the tree is constructed, we count the number of nodes with a positive timestamp, and obtain the working set size of the example document to be four.

The accuracy of our estimation is affected by the mapping between a tag tree node and an automaton state in the XPath-DFA. When this mapping is one-to-one, the working set size obtained from the tag tree equals to that on the XPath-DFA. With a large number of queries of diverse properties, this mapping will be close to a one-to-one correspondence. As we observed in our experiments, this mapping was close to one-to-one when the number of queries was larger than 160 K. Details of the experiment are described in Section 5.

3.3 Modeling Interround Filtering

Having modeled the cache misses in intraround filtering, we then model the cache behavior of interround filtering. There are two major differences between intraround and interround filtering:

- Each round, except the first round, of interround filtering is likely to reuse the working sets of previous rounds.
- As multiple documents are filtered, the total size of the states referenced during filtering may exceed the cache capacity, upon which cache replacement

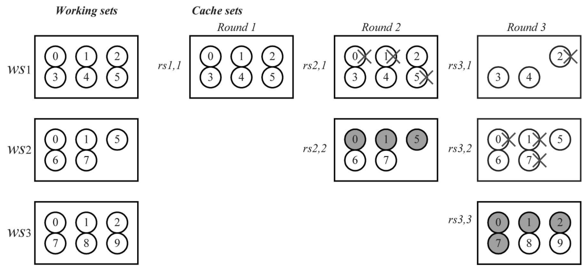


Fig. 7. An example of working sets and cache sets in interround modeling.

occurs. Therefore, capacity misses are considered in interround filtering.

We use a state as the unit in cache replacement for simplicity, as it is complex to estimate the replacement at a finer granularity, e.g., at the unit of a cache line. The accuracy of this simplified estimation is sufficient for our model, which is confirmed in our experiments.

The basic idea of our interround modeling is that we estimate the changes of the *cache content* (i.e., the set of states in the cache) for each round. Based on these changes, we estimate the number of compulsory misses and capacity misses.

Definitions. *Given a sequence of documents*

$$Seq = (doc_1, doc_2, \ldots, doc_n),$$

we obtain the corresponding working sets $(ws_1, ws_2, \ldots, ws_n)$ *using the intraround modeling. To estimate the changes of the cache content, we define* $rs_{i,j}$ $(i \geq j)$ *the cache set of* doc_j *at the end of the* ith *round so that* $rs_{i,j}$ *is the subset of* ws_j *that resides in the cache and is not accessed by any state transitions from the beginning of the* $(j+1)$th *round to the end of the* ith *round. Since we assume that there are no capacity misses in intraround filtering* doc_i*, we have* $rs_{i,i} = ws_i$.

Fig. 7 illustrates the working sets and the cache sets of three documents in a sequence. The circles represent the states and the shadowed ones represent the states reused in the current round. We assume the cache can hold more than 10 states so that all states referenced during filtering the three documents can fit into the cache. At the end of each round, the cache set of each document (if any) is computed by removing states that have been accessed in the current round. In Fig. 7, the state marked with a cross is the one to be removed from the cache set at the end of the current round.

We have defined the intersection, difference, and union operations on two cache sets and a sequence of cache sets to be ordinary binary and n-ary set operations, respectively. Similarly, we can define these operations on working sets [16]. The working set of sequence Seq is defined as $(ws_1 \cup ws_2 \ldots \cup ws_n)$. The cache content at the end of the ith round is $(rs_{i,1} \cup rs_{i,2} \ldots \cup rs_{i,i})$, which is also the cache content at the beginning of the $(i+1)$th round.

Estimation. We start by modeling the changes of the cache contents in interround filtering. Given the working set, ws_i, of doc_i at the beginning of the ith round, we estimate the cache content at the end of the ith round, $(rs_{i,1} \cup rs_{i,2} \ldots \cup rs_{i,i})$, based on the cache content at the beginning of the ith round, $(rs_{i-1,1} \cup rs_{i-1,2} \ldots \cup rs_{i-1,i-1})$. In the first step, we always set $rs_{i,i} = ws_i$. Next, we estimate the changes on cache sets $rs_{i-1,j}$ and obtain $rs_{i,j}$ $(1 \leq j < i)$:

- If no replacement occurs, $rs_{i,j}$ is obtained by removing the states accessed during the ith round from $rs_{i-1,j}$. That is, $rs_{i,j} = rs_{i-1,j} - rs_{i,i}$ $(1 \leq j < i)$.
- If replacement occurs, we consider the replacement in the cache content. We denote

  $$rs'_{i-1,j} = rs_{i-1,j} \ (1 \leq j < i),$$

  which contain the candidate states for replacement at the beginning of the ith round. With the LRU replacement policy, an $rs'_{i-1,j}$ with a smaller j has a higher priority to be replaced. Within an $rs'_{i-1,j}$, an earlier state has a higher priority to be replaced. The number of states that need to be replaced is

  $$\#toReplace_i = |rs'_{i-1,1} \cup rs'_{i-1,2} \ldots \cup rs'_{i-1,i-1} \cup rs_{i,i}| - \frac{N}{P}.$$

  Replacement is done by removing states from, $rs'_{i-1,1}$, $rs'_{i-1,2}, \ldots,$ until the number of states replaced reaches $\#toReplace_i$. Hence, $rs'_{i-1,j}$ $(1 \leq j < i)$ is updated at the end of replacement. The set of states that are swapped out of the cache in this round,

  $$Rp_i = ((rs_{i-1,1} \cup rs_{i-1,2} \ldots \cup rs_{i-1,i-1}) - (rs'_{i-1,1} \cup rs'_{i-1,2} \ldots \cup rs'_{i-1,i-1})).$$

  Finally, $rs_{i,j}$ can be obtained by removing the states accessed during the ith round from $rs'_{i-1,j}$. That is, $rs_{i,j} = rs_{i-1,j} - rs'_{i,i}$ $(1 \leq j < i)$.

Based on the modeled interround changes of cache contents, we estimate the number of compulsory misses and capacity misses:

- *the number of compulsory misses in the* ith *round* is $|rs_{i,i} - (rs_{i-1,1} \cup rs_{i-1,2} \ldots \cup rs_{i-1,i-1})| \times P$, which corresponds to the automaton states referenced for the first time by filtering doc_i.
- *the number of capacity misses in the* ith *round* is considered in two cases:
  1. If $|rs_{i-1,1} \cup rs_{i-1,2} \ldots \cup rs_{i-1,i-1} \cup rs_{i,i}| \times P \leq N$, no replacement occurs and the number of capacity misses is zero.
  2. If $|rs_{i-1,1} \cup rs_{i-1,2} \ldots \cup rs_{i-1,i-1} \cup rs_{i,i}| \times P > N$, there is replacement in the cache and capacity misses occur. The set of states that are reloaded into the cache, $toReload_i = Rp_i \cap rs_{i,i}$. The number of capacity misses in the i^{th} round is $|toReload_i| \times P$.

Implementation. We have developed the intersection, difference, and union operations on tag trees to implement the operations for working sets and cache sets, and have applied these tag tree operations to implement the interround model [16].

3.4 Estimations for Matrices and Hash Tables

We now apply our model to the matrix and the hash table implementation schemes, respectively. For simplicity, we assume that all outgoing transitions of a state have an equal probability to be accessed.

For the matrix implementation, we have $F = 1$. When $P = 1$, all reuses are a perfect one. If $P > 1$, a perfect reuse occurs when the transition is the same as a previous one, while a partial reuse may occur if the transition has the same originating state as some previous ones.

We apply Cardenas' formula [5] to estimate the partial reuses. Cardenas' formula was originally used to estimate the average number of blocks accessed by a query. Suppose α tuples are uniformly distributed into K blocks and there are β tuples satisfying the query, Cardenas' formula, $K \times (1 - (1 - 1/K)^{\beta})$, gives the average number of blocks that contain the β tuples. This scenario is similar to ours. We estimate the number of cache lines accessed by a number of distinct transitions having made from one state (denoted as Q_s). The number of satisfying tuples, the total size of all blocks and the block size correspond to the number of distinct transitions, the span of the state and the footprint of a transition, respectively. That is, β and K correspond to Q_s and P_s/F, respectively. The number of distinct cache lines visited by these Q_s transitions, $PM(P_s, Q_s)$, is estimated using (2):

$$PM(P_s, Q_s) = P_s \times (1 - (1 - 1/(P_s/F))^{Q_s}). \tag{2}$$

In intraround modeling, we estimate the number of compulsory misses as

$$Com = \sum_{s \in ws} CM(s) = \sum_{s \in ws} PM(P_s, Q_s),$$

where Q_s is estimated as the number of child nodes of the tag tree node corresponding to the state s. In interround modeling, the number of cache misses in each round can be estimated using (2) based on the modeled states and state transitions.

For the hash table implementation, each transition performs a linear scan on the values of a hash bucket to find the symbol of the transition. This scenario is different from that of the matrix implementation. Thus, we compute the number of hash entries visited by the Q_s distinct transitions. We present the abstraction of this problem and its solution in Proposition 1. Especially, given the number of hash entries that can be held in one cache line U, the number of hash entries in a hash bucket is $(P_s \times U)$. We have $(P_s \times U)$ and Q_s corresponding to n and m in Proposition 1. Hence, $E(P_s \times U, Q_s)$ gives the expected number of distinct hash entries visited by the Q_s transitions.

Proposition 1. *Suppose we randomly choose m distinct integers ranging from one to n $(m \leq n)$. The expected value of the maximum value of these m integers is given by $E(n, m) = \sum_{k=m}^{n}(k * \frac{C_{k-1}^{m-1}}{C_n^m})$.*

Proof. The maximum value of these m distinct integers should be no less than m. The probability that $k (k \geq m)$ is the maximum value of these m integers is $p_k = \frac{C_{k-1}^{m-1}}{C_n^m}$. Hence, $E(n, m) = \sum_{k=m}^{n}(k * \frac{C_{k-1}^{m-1}}{C_n^m})$. □

Therefore, the number of cache misses caused by these Q_s distinct transitions is given in (3). We can apply this formula to our cache behavior model on a hash table based automaton. The process is similar to applying (2) to the matrix implementation.

$$PM(P_s, Q_s) = 1 + E(P_s \times U, Q_s)/U. \tag{3}$$

Finally, we compare the cache performance of the matrix and the hash table using our model. We define the *density* of an automaton as the ratio of the number of transitions to the number of symbols multiplied by the number of states that have any outgoing transitions. We exclude the states without any outgoing transitions in this definition, because these states are not in the working set of any document. The density affects the span of the hash table implementation. The higher the density, the larger the span of the hash table-based automaton. In contrast, the density does not affect the span of the matrix implementation. We compute the threshold value of the density for the choice between the hash table and the matrix. When the density is larger than the threshold value, the matrix implementation has a better cache performance than the hash table implementation.

3.5 Putting It All Together

Having modeled compulsory and capacity misses in a fully associative cache, we now discuss the estimation of conflict misses in an A-associative cache. Conflict misses depend on a number of factors, including cache set-associativity, data layout, and access patterns. Since state transitions on an automaton are random access on a fixed data layout, the blocks in the main memory (i.e., memory blocks) referenced by state transitions are uniformly mapped to the sets in the cache. When more than A distinct memory blocks are mapped to one set in the cache, there must be cache line replacement. A conflict miss occurs when a discarded line is revisited.

Let Z be the number of sets in the cache $(\frac{N}{A})$. Denote Com_i, Con_i, and Cap_i as the estimated numbers of compulsory misses, conflict misses, and capacity misses in the ith round, respectively. Tot_i is the total number of cache accesses in the ith round. It is equal to the number of state transitions in the ith round multiplied by F. The number of revisited lines is $(Tot_i - Com_i - Cap_i)$.

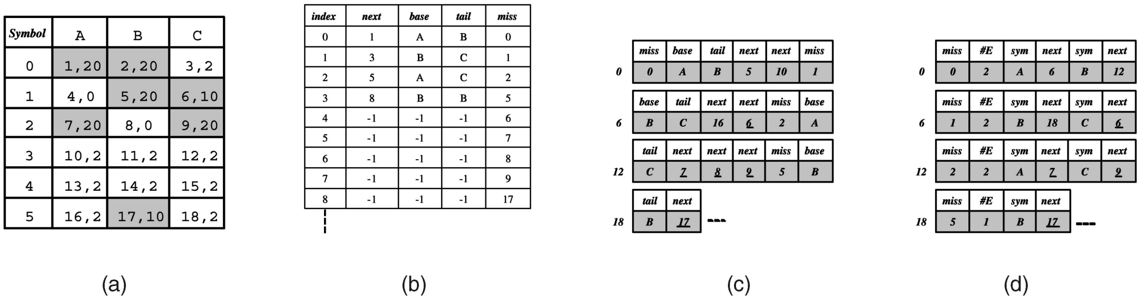
Given the total number of compulsory misses of the first i rounds, $TCom_i = \sum_{j=1}^{i} Com_j$, the probability of having x distinct memory blocks mapped to a set in the cache is:

$$P(x) = \left(\frac{1}{Z}\right)^x \cdot \left(\frac{Z-1}{Z}\right)^{TCom_i - x}. \tag{4}$$

The probability of a revisited line causing a conflict miss when there are x distinct memory blocks mapped to one set in the cache is:

$$C(x) = P(x) \cdot \frac{x - A}{x}, \text{given } x > A. \tag{5}$$

The number of conflict misses in the ith round can be estimated using (6). The item, $(\sum_{x=A+1}^{TCom_i} C(x))$, represents the probability of causing a conflict miss by a revisited line.



Symbol	A	B	C
0	1,20	2,20	3,2
1	4,0	5,20	6,10
2	7,20	8,0	9,20
3	10,2	11,2	12,2
4	13,2	14,2	15,2
5	16,2	17,10	18,2

(a)

index	next	base	tail	miss
0	1	A	B	0
1	3	B	C	1
2	5	A	C	2
3	8	B	B	5
4	-1	-1	-1	6
5	-1	-1	-1	7
6	-1	-1	-1	8
7	-1	-1	-1	9
8	-1	-1	-1	17

(b)

	miss	base	tail	next	next	miss
0	0	A	B	5	10	1
	base	tail	next	next	miss	base
6	B	C	16	6	2	A
	tail	next	next	next	miss	base
12	C	7	8	9	5	B
	tail	next				
18	B	17	---			

(c)

	miss	#E	sym	next	sym	next
0	0	2	A	6	B	12
	miss	#E	sym	next	sym	next
6	1	2	B	18	C	6
	miss	#E	sym	next	sym	next
12	2	2	A	7	C	9
	miss	#E	sym	next		
18	5	1	B	17	---	

(d)

Fig. 8. An automaton segment and hot buffers with different hot spot representations. (a) Automaton segment. (b) Fixed-size compact row representation. (c) Variable-size compact row representation. (d) Symbol table representation.

$$Con_i = \left(\sum_{x=A+1}^{TCom_i} C(x) \right) \cdot (Tot_i - Com_i - Cap_i). \qquad (6)$$

Finally, the total number of cache misses in the ith round, TM_i, is estimated as follows:

$$TM_i = Com_i + Con_i + Cap_i. \qquad (7)$$

Our approach to modeling cache performance has several advantages over the hardware profiling approach. First, our approach is more powerful because it can capture application semantics. Compared with general profiling tools, such as PCL [4], our model can estimate the three categories of misses based on the processing flow of the filtering automaton. This specialization allows us to develop specialized algorithms to reduce the most significant kind of cache misses. For example, our hot buffer techniques mainly improve the cache performance through reducing capacity misses and conflict misses. Second, our approach is more flexible. Our model can run on multiple platforms without much modification. In contrast, profiling tools need to be carefully configured to measure the performance for a specific hardware platform. Third, our modeling is much less intrusive than profiling, as the model can be run separately from the filtering engine, whereas hardware profiling requires instrumentation, which creates interference to the application code.

Compared with the cache simulation that can report the three categories of misses, our approach is faster in terms of the execution time and the time of examining the result. Additionally, our model captures the cache behavior of different automaton implementations and compares their cache performance. Through modeling, we understand the cache behavior of XML filtering and obtain insights for developing cache-conscious filtering techniques.

4 The Hot Buffers

Our model on the cache behavior of filtering makes it clear that locality, mainly state reuse, is essential for reducing cache misses. This observation motivates us to develop a cache-conscious automaton organization technique called the *hot buffer*. The basic idea is to replicate the frequent state transitions, or called *hot spots*, into a memory area outside the original automaton.

We have two design considerations on the hot buffer. First, hot spots from one state should be put together so that they can be efficiently retrieved. Second, hot spots should be extracted following the transition paths starting from the start state. There are two reasons for the second consideration: 1) The transitions near the start state are more likely to be hot than those far from the start state, because a round always starts from the start state and (2) all transition paths contained in the hot buffer should begin with the start state so that we can easily determine whether or not a certain transition is in the hot buffer.

Our hot buffer technique is orthogonal to the original automaton implementation. In particular, neither the hot buffer construction nor filtering with the hot buffer depends on the original automaton implementation. Given a matrix-based or a hash table-based automaton, we can construct a hot buffer and perform filtering with it. For the simplicity of the presentation, we describe our hot buffer technique using the matrix implementation.

4.1 Hot Spot Representations

To address the first design consideration of how to put the hot spots from one state together, we have designed two representations for hot spots, the *compact row* and the *symbol table*.

In order to identify hot spots in the original automaton, we add a counter to each cell in the matrix to record the access frequency of the corresponding state transition. We then set a threshold for the access frequency. If the access frequency of a state transition is larger than the threshold, it is a hot spot. The threshold value is determined to maximize the performance impact of the hot buffer [16].

Let us describe the two hot spot representations using an example in Fig. 8a. Each cell is in the form of $<next\ state,\ counter>$. The threshold on the access frequency is 10. The shadowed cells in the automaton are identified as hot spots.

For the compact row representation, we record the *hot base* (the left-most cell that is hot) and the *hot tail* (the right-most cell that is hot) in each row. To preserve the fast-transition advantage of a matrix, we treat all cells between the hot base and the hot tail (both ends inclusively) in a row as hot spots and call this fragment a *compact row*. This way, when checking whether a certain symbol is in the compact row, we only need to check whether the symbol is between the base and the tail. Note that the access frequency of some transition in a compact row, e.g., the cell $<8, 0>$ in row 2, column B, has an access frequency smaller than ten and is not really a hot spot.

We have designed two variants for the compact row representation, namely, the *fixed-size* and the *variable-size* compact row representations. For the fixed-size compact row representation, each element in the hot buffer is a tuple with four fields, $<next,\ base,\ tail,\ miss>$, as shown in Fig. 8b. The $base$ and $tail$ correspond to the hot base and the hot tail of a compact row in the original automaton, if any. For each compact row, $(tail - base + 1)$ elements are added to the hot buffer. The value of $next$ is set during the hot buffer construction process. Details of the construction algorithm are described in Section 4.2. The use of $next$ is to lead the current transition to move forward to a certain hot buffer element during filtering. Given the current symbol, tag, if $base \leq tag \leq tail$, a *hot buffer hit* occurs and the next hot buffer element to visit is determined based on the $next$ field; otherwise, a *hot buffer miss* occurs and the $miss$ field of the current element points to the state in the original automaton to resolve the hot buffer miss.

For the variable-size compact row representation, each element in the hot buffer is a variable-size tuple,

$$<miss,\ base,\ tail,\ next[0 : tail - base]>,$$

as shown in Fig. 8c. The definitions of $base$, $tail$, and $miss$ are the same as the ones in the fixed-size compact row representation. The array $next$ stores the transition information of a compact row, i.e., state transitions from the state $miss$ upon the symbols between $base$ and $tail$. Each $next$ element can be either a start position of a hot buffer element or a state in the original automaton. In Fig. 8c, the underlined $next$ elements are states in the original automaton, and other $next$ elements are positions in the hot buffer. Given the start position of the current hot buffer element, cur, and the current symbol, tag, if $base \leq tag \leq tail$, a hot buffer hit occurs, and the next hot buffer element or a state in the original automaton to visit is $next[tag - base]$; otherwise, a hot buffer miss occurs and the $miss$ field of the current hot buffer element is used.

For the symbol table representation, each element in the hot buffer is a variable-size tuple,

$$<miss, numEntries, tableEntry[0 : numEntries - 1]>,$$

as shown in Fig. 8d. The definition of $miss$ is the same as the one in the compact row representation. The array $tableEntry$ is the symbol table and each hot spot from the state $miss$ is represented as a table entry, $<symbol,\ next>$. The definition of $next$ is the same as the one in the variable-size compact row representation. The $numEntries$ value is the number of table entries in the symbol table. We add an entry to the symbol table only if the transition from the state $miss$ upon this symbol is hot. Compared with the compact row representation, the symbol table contains no false hot spots. Given the start position of the current hot buffer element, cur, the symbol table is stored from position $(cur + 2)$ to position $(cur + 2 + 2 \cdot numEntries)$. Given the current symbol tag, we scan table entries. If a table entry with tag is found, it is a hot buffer hit. Otherwise, it is a hot buffer miss.

We now compare the space efficiency of these compact row representations. We define the *hotness* of a compact row to be the ratio of the number of hot spots to the number of symbols in the compact row. The hotness of the automaton is defined as the average hotness of all compact rows in the automaton. Given the hotness of an automaton, h, the symbol table representation uses $(2 + (tail - base + 1) \times 2 \times h)$ integers to represent one compact row on average. The space required by the fixed-size and the variable-size compact row representations are around $(4 + (tail - base + 1) \times 4)$ and $(3 + (tail - base + 1))$ integers, respectively. Thus, the variable-size compact row representation always has a higher space efficiency than the fixed-size representation. When the hotness is larger than a threshold value ($\frac{tail-base+2}{2(tail-base+1)}$, around 50 percent), the variable-size compact row has a higher space efficiency than the symbol table.

Note that in our implementation, we encode the symbols in their alphabetic order as consecutive integers. If we have a priori knowledge of the access pattern on the symbols, such as DTDs/XML schemas, we can improve the efficiency of both representations by encoding more frequently accessed symbols as consecutive integers. However, the performance benefit of this careful encoding is quite limited compared with identifying hot spots, since the number of symbols is much smaller than the number of transitions.

4.2 Hot Buffers

Our hot buffer techniques address the second design consideration of how to extract the hot spots following the transition paths starting from the start state. Since sequential access is much more cache friendly than random access, one alternative is to pack hot spots contiguously into an array. We call this alternative a *contiguous hot buffer*, or a *C-Buffer*. However, the access patterns may change over time so that the hot buffer needs to be maintained. In order to balance the filtering and maintenance efficiency, we also propose an alternative called a *segmented hot buffer*, or an *S-Buffer*.

Since the algorithms for different hot spot representations are similar, we only present the algorithms for the fixed-size compact row representation.

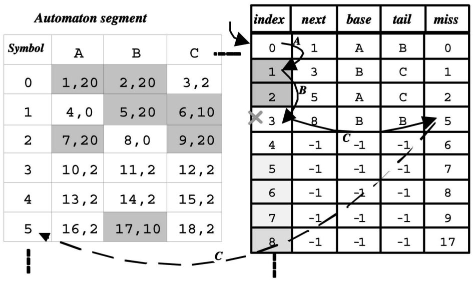
4.2.1 The C-Buffer

Figs. 8b, 8c, and 8d show the C-Buffers with different hot spot representations constructed from the hot spots in the original automaton in Fig. 8a. As shown in Fig. 8b, the C-Buffer with the fixed-size compact row representation is an array structure starting from index 0. We present the construction algorithm for the C-Buffer in Algorithm 1. We treat the original automaton as a graph with states as nodes and transitions as edges, and perform a breadth first traversal on the automaton. To assist the traversal process, the algorithm maintains two queue structures, $wqueue$ and $pqueue$. The queue $wqueue$ stores the *waiting states* from which a hot spot will be inserted into the C-Buffer. The queue $pqueue$ stores the row indexes in the C-Buffer at which the hot spots are inserted into the C-Buffer.

Algorithm 1 C-Buffer construction

Input: Automaton Au with counters, threshold for the access frequency t, an empty C-Buffer $cBuf$, the size limit of the C-Buffer in number of hot spots $sizeLimit$

```
$wqueue$.enqueue($Au$'s start state), $pqueue$.enqueue(0);
$curSize = 1$;        /* the total number of elements
                         having been inserted into $cBuf$ */
```



Automaton segment

Symbol	A	B	C
0	1,20	2,20	3,2
1	4,0	5,20	6,10
2	7,20	8,0	9,20
3	10,2	11,2	12,2
4	13,2	14,2	15,2
5	16,2	17,10	18,2

index	next	base	tail	miss
0	1	A	B	0
1	3	B	C	1
2	5	A	C	2
3	8	B	B	5
4	-1	-1	-1	6
5	-1	-1	-1	7
6	-1	-1	-1	8
7	-1	-1	-1	9
8	-1	-1	-1	17

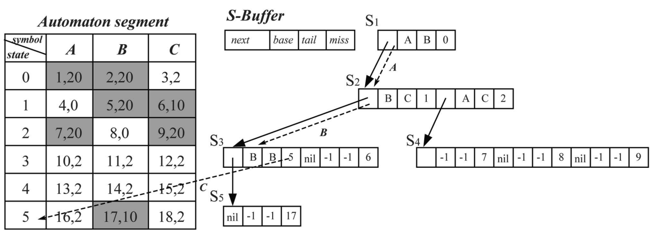
Fig. 9. Filtering on a C-Buffer with the fixed-size compact row representation.

```
3: while $wqueue$ is not empty do
4:   $curState = wqueue$.dequeue(), $curIndex = pqueue$.dequeue();
5:   let $p$ be the element in $cBuf$ so that $p.miss = curState$;
6:   if $p$ is $nil$ then
7:     if $curSize \leq sizeLimit$ then
8:       $cR$ is the compact row of the row $curState$ in $Au$ according to $t$;
9:       Set $cBuf$'s $(curIndex)$th element: $next = curSize$, $miss = curState$, and $base$, $tail$ are the smallest and the largest symbols of $cR$, respectively;
10:      for state $s$ in $cR$ do
11:        $wqueue.enqueue(s)$, $pqueue.enqueue(curSize)$;
12:        $curSize = curSize + 1$;
13:    else
14:      Set $cBuf$'s $(curIndex)^{th}$ element: $miss = curState$;
15:  else
16:    Set $cBuf$'s $(curIndex)^{th}$ element: $next = p.next$, $miss = p.miss$, $base = p.base$, and $tail = p.tail$;
```

During the traversal (Lines 5–16), the algorithm checks if any existing hot spot in the hot buffer and the new one transit to the same state. If so, the algorithm sets the fields of this new hot spot to be the same as the existing one (Line 16). Otherwise, it checks if the size limit has been reached. If so, it simply sets the $miss$ value for the new hot spot (Line 14). Note that all fields of a hot spot are initially -1. Otherwise, the following operations are performed. First, it extracts the compact row from the original automaton according to the threshold value. Then, it sets the current C-Buffer element. Next, it puts the waiting states in the compact row into $wqueue$ and puts the row indexes to insert the hot spots into $pqueue$. The construction completes when $wqueue$ is empty.

The size limit should be no more than the L2 cache capacity. With this size limit, the hot buffer is expected to be cache resident and has little cache thrashing. In practice, we set the size limit to be the L2 cache capacity.

After a C-Buffer is constructed, a state transition on the C-Buffer is performed as follows: Given the current index, cur, in the C-Buffer and the current symbol, tag, if $base \leq tag \leq tail$, a *C-Buffer hit* occurs and the next array element to visit is the one with an index of $(next + tag - base)$ in the C-Buffer; otherwise, a *C-Buffer miss* occurs and the $miss$ field of the current array element points to the state in the original automaton to resolve the C-Buffer miss.



Automaton segment

symbol / state	A	B	C
0	1,20	2,20	3,2
1	4,0	5,20	6,10
2	7,20	8,0	9,20
3	10,2	11,2	12,2
4	13,2	14,2	15,2
5	16,2	17,10	18,2

S-Buffer
next | base | tail | miss
S1
A | B | 0
A
S2
B | C | 1 | A | C | 2
B
S3
B | B | 5 | nil | -1 | -1 | 6
S4
-1 | -1 | 7 | nil | -1 | -1 | 8 | nil | -1 | -1 | 9
C
S5
nil | -1 | -1 | 17

Fig. 10. Filtering on an S-Buffer with the fixed-size compact row representation.

Filtering on the automaton with a C-Buffer is performed as follows: The filtering is performed using a runtime stack, which is the same as the one used in the filtering without a C-Buffer. If a transition is a C-buffer hit, it will not reference the original automaton. Otherwise, it is a C-Buffer miss and the original automaton is visited. Prior to visiting the original automaton, we save the context of the stack as $(mSize, mTop)$, where $mSize$ is the size of the runtime stack and $mTop$ is the content on the top of the runtime stack. Next, we replace the top of the runtime stack with the $miss$ value of the $mTop$th array element in the C-Buffer. The filtering process continues on the original automaton until the size of the stack becomes $mSize$ again. At this point, we set the content on the top of the runtime stack to be $mTop$ and resume the transition on the C-Buffer.

The arrows shown in Fig. 9 illustrate the filtering process of an XML document,

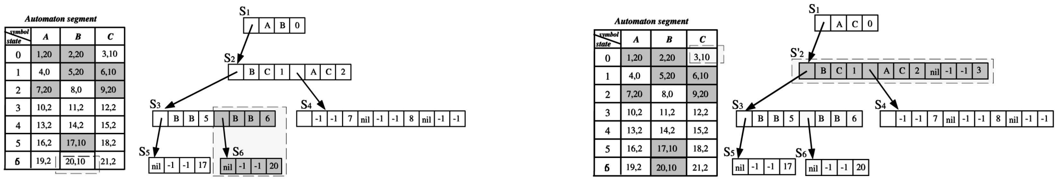
$$<A><B><C></C></B></A>,$$

with the C-Buffer. The arrows with solid lines represent the transitions on the C-Buffer and the arrow with a dashed line redirects the transition to the original automaton. The filtering process starts from index 0 $(cur = 0)$ on the C-Buffer. When the $<A>$ tag is encountered, we calculate the next index to be one by $(next + tag - base)$, where $next = 1$, $tag = A$, and $base = A$ in Element 0. The subsequent event, $<B>$, is processed in a similar way. When the third event is processed, $<C>$, there is a C-Buffer miss and the transition is redirected to the original automaton. After the fourth event, $</C>$, is processed, the runtime stack is restored and the transitions resume on the C-Buffer.

4.2.2 The S-Buffer

The construction process of the S-Buffer is similar to that of the C-Buffer. As shown on the right of Fig. 10, the S-Buffer is a graph structure. We call the segment (node) containing the start state of the original automaton the *root* segment. Each segment in the graph contains the hot spots from a compact row in the original automaton. The *next* field of each hot spot is the address of the first hot spot in the segment that stores the hot spots in the compact row of the state *miss* in the original automaton.

The filtering process using the S-Buffer is also similar to the one using the C-Buffer. The major difference is on the computation of the next hot spot. Given the current hot spot cur and the current symbol tag, if $base \leq tag \leq tail$, the transition is an S-Buffer hit, and the address of the next hot spot to visit is $(cur.next + (tag - base) \cdot v)$, where v is the size of a hot spot. Otherwise, it is an S-Buffer miss and the



Automaton segment

symbol / state	A	B	C
0	1,20	2,20	3,10
1	4,0	5,20	6,10
2	7,20	8,0	9,20
3	10,2	11,2	12,2
4	13,2	14,2	15,2
5	16,2	17,10	18,2
6	19,2	20,10	21,2

(a)

Automaton segment

symbol / state	A	B	C
0	1,20	2,20	3,10
1	4,0	5,20	6,10
2	7,20	8,0	9,20
3	10,2	11,2	12,2
4	13,2	14,2	15,2
5	16,2	17,10	18,2
6	19,2	20,10	21,2

(b)

Fig. 11. Examples of insertions into the S-Buffer.

transition is redirected to the original automaton. The arrows with dashed lines shown in Fig. 10 illustrate the filtering process of an XML document,

$$<A><B><C></C></B></A>,$$

with the S-Buffer.

We have developed algorithms for both insertion and deletion operations of the hot spots in the S-Buffer. The insertion and deletion algorithms are similar in that they first find the target segment and next perform insertions or deletions on that segment. Therefore, we present the insertion algorithms only. Insertion of hot spots can be done either one by one or in a batch. Again, since the processes of individual insertions and those of batch insertions are similar, we present Algorithm 2 for inserting the hot spots from a compact row in a batch. The algorithm takes the state s as an input parameter and inserts the hot spots from the compact row of the state s in the original automaton.

Algorithm 2 S-Buffer insertion: $InsertState(s)$

```
let $hp$ be the hot spot in the S-Buffer so that $hp.miss = s$;
if $hp$ is not $nil$ then
    $cR$ is the new compact row extracted from the state $s$ in the original automaton;
    Set $oldSeg$ to be the segment that $hp.next$ points to;
    $cR'$ is the old compact row constructed based on $oldSeg$;
    $start$ and $end$ are the smallest and largest symbols of $cR$, respectively;
    $start'$ and $end'$ are the smallest and largest symbols of $cR'$, respectively;
    if $cR$ is different from $cR'$ then
        Allocate a new segment $newSeg$ with a size of $(end - start + 1)$;
        for $i = 0; i < (end - start + 1); i++$ do
            $h$ is the $i$th hot spot of the $newSeg$;
            /* decide whether $h$ is in the old compact row */
            if $(i + start) \geq start'$ and $(i + start) \leq end'$ then
                Evaluate $h$ using the $(i + start - start')$th hot spot in $oldSeg$;
            else
                Evaluate $h.miss$ as the $i$th cell of $cR$;
        /* update $pList$ of the new segment */
        $newSeg.pList = oldSeg.pList$;
        for hot spot $pS$ in $newSeg.pList$ do
            $pS.next = newSeg$, $pS.base = start$, $pS.tail = end$;
        Release the space of $oldSeg$;
```

The algorithm first checks if any hot spot in the hot buffer originates from the state s. If so, it continues to check whether the new compact row is different from the old compact row. If so, the algorithm performs maintenance on the S-Buffer and inserts the new hot spots into the S-Buffer in two steps. First, the algorithm allocates a new segment to store the hot spots from the new compact row. Then, the algorithm sets the fields of the hot spots in the new segment. For the hot spots that are already in the old compact row, it sets their fields using the values from the old segment; otherwise, it sets their $miss$ values and leaves other fields to have the default values $(base = tail = -1, next = nil)$.

The algorithm maintains an auxiliary list $pList$ for each segment to keep the addresses of the hot spots with $next$ pointing to the segment. At the end, the algorithm sets the fields of the hot spots in $pList$ by setting their $next$ values addressing the new segment, and updating their $base$ and $tail$ values based on the new compact row. Finally, the algorithm releases the space of the old segment.

Let us further illustrate insertions using examples shown in Fig. 11. Starting with the original automaton and the S-Buffer shown in Fig. 10, we insert two hot spots into the S-Buffer: 1) A new hot spot $<6, B, 20>$ that originates from State 6 (Fig. 11a). The algorithm executes with $s = 6$. First, it finds the second hot spot in Segment S_3. Next, it allocates a new segment, S_6, and updates the $next$, $base$, and $tail$ values of the second hot spot of S_3. 2) A new hot spot $<0, C, 3>$ that originates from state 0 (Fig. 11b). The algorithm allocates a new segment S'_2 by copying the old segment S_2 and adding a new hot spot $<nil, -1, -1, 3>$ to S'_2, and updates S_1 accordingly.

Having developed the maintenance algorithms, we show how an S-Buffer handles a dynamic automaton with queries being added or removed. Adding or removing a query is handled using existing algorithms [14]. When queries are added, new state transitions and new states may be added to the original automaton, but we do not maintain the S-Buffer until we observe that the *buffer hit rate* (defined as $\frac{\#hit}{\#miss+\#hit}$, where $\#hit$ and $\#miss$ are the numbers of hot buffer hits and misses, respectively) is lower than a threshold value. When queries are removed, some states and state transitions may become invalid. If these invalid state transitions are in the buffer, we mark them as invalid but do not remove them until the buffer maintenance time.

4.3 Discussion

Given a workload, i.e., a sequence of documents, we define the effectiveness of a hot buffer using *speedup*, the ratio of

TABLE 1
The Choice of Hot Buffer Types

	Static hot spots	*Dynamic hot spots*
High hotness	C-Buffer with compact rows	S-Buffer with compact rows
Low hotness	C-Buffer with symbol tables	S-Buffer with symbol tables

TABLE 2
Performance Metrics

Metrics	*Description*
L1_DCM	Number of L1 data cache misses
L2_DCM	Number of L2 data cache misses
TOT_CYC	Total running time (in CPU cycles)

TM, the number of cache misses in the filtering without the hot buffer, to BM, the number of cache misses in the filtering with the hot buffer:

$$speedup = \frac{TM}{BM}. \qquad (8)$$

TM and BM can be estimated using our cache behavior model [16]. Given a hot buffer, BM is affected by the hot buffer hit rate. We maintain the hot buffer based on its hit rate. The threshold for the buffer hit rate is the one when $speedup$ is one. When the hit rate is lower than the hit rate threshold, the hot buffer needs to be maintained.

Combining two hot spot representations and two buffer organizations, we have developed four hot buffer types: C-Buffers with compact rows and with symbol tables, and S-Buffers with compact rows and with symbol tables. We choose a suitable buffer type to maximize the $speedup$. Intuitively, we use the compact row representation when the hotness is high, and use the symbol table representation otherwise. We use the C-Buffer when the set of hot spots is static, and use the S-Buffer otherwise. We summarize the choice of hot buffer types according to the hotness and the hot spot dynamics in Table 1.

5 Experimental Evaluation

In this section, we compare the performance of the matrix and the hash table implementation schemes, validate the accuracy of our cache behavior model, and study the effectiveness of the hot buffer.

5.1 Setup and Methodology

All of our experiments were conducted on a 2.8GHz P4 machine with 2.0GB memory running Red Hat Linux 9.0. The P4 processor has the hardware prefetching enabled. The L1 cache was 32K bytes (16K bytes for instructions and 16K bytes for data). The L2 cache was 512K bytes, unified, and 8-way associative. The cache line size for the L2 cache was 64 bytes. We also conducted our experiments on a P3 machine and obtained similar results as on P4 [16]. Our XML filtering system was written in C++ and compiled using g++ 3.2.2-5 with optimization flags *O3* and *finline-functions*. The filtering experiments were always memory-resident and the memory usage never exceeded 90 percent.

We used a hardware profiling tool, PCL [4], to count cache misses and CPU execution cycles. Table 2 lists the main performance metrics used in our experiments.

To evaluate the performance of a filtering engine, there are quite a few parameters to be varied for the automata and the documents. For the automaton setup, we consider the number of queries and the query characteristics including the number of symbols, the number of steps, and the probabilities of having a wildcard, a descendant axis and a branch. To simplify this setup, we treated the wildcard as a special symbol and fixed the maximum number of steps to be six [12]. The symbols are uniformly distributed in the queries. We varied the parameters listed in Table 3. *All parameters used in our experiments were in their default settings unless specified otherwise.* With these parameter values, the automaton can represent millions of path queries. For example, when the number of symbols is 16, the automaton can at most represent $(1 + 16 + 16^2 + \ldots + 16^6)$ states and $(16 + 16^2 + \ldots + 16^6)$ transitions, both of which are around 16 million. These parameter values are comparable with those in the previous work [12], [14].

For the comparison between the matrix and the hash table, we consider the density of the automaton. We varied the $Pr_{//}$ value to obtain different densities, as shown in Table 4, with other parameters in Table 3 under their default settings. The larger the $Pr_{//}$ value, the higher density of the automaton.

For the document setup, we consider the number of symbols, the distribution of the symbols in a document and the depth of the document tree. For simplicity, we used the same settings as those in the automaton setup. Additionally, given a sequence of documents, we consider its working set size and the hotness of the automaton after filtering this sequence. We define a parameter $fanout$ for a document sequence to be the average number of child nodes for a nonleaf node in the tag tree T_u, where T_u is the union of the tag trees of the documents in the sequence. Given the $fanout$ value, the hotness of the automaton after filtering the document sequence is around $\frac{fanout}{S}$.

We used three document sets in our experiments. Each document set consists of five sequences of documents.

TABLE 3
Parameters in Our Experiments

Parameter	*Description*	*Range*	*Default*
S	number of symbols in the symbol set	$16 - 80$	64
Q	number of queries	$160K - 800K$	$160K$
$Pr_{//}$	probability of the descendant axis in the query	$0 - 100\%$	5%
Pr_{twig}	probability of having branches	$0 - 100\%$	5%

TABLE 4
The Density Values for Different $Pr_{//}$ Values

$Pr_{//}(\%)$	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
$density(\%)$	3.2	4.3	5.5	6.6	7.7	8.8	9.9	11.0	12.2	13.2
$Pr_{//}(\%)$	*10*	*20*	*30*	*40*	*50*	*60*	*70*	*80*	*90*	*100*
$density(\%)$	14.1	24.3	34.4	44.4	54.2	63.7	73.3	82.7	91.8	100

TABLE 5
Document Sets in Our Experiments

Name	*Working set sizes (K)*	*Fanouts*	*S*	$Hotness$
COLD	2.0, 7.6, 11.2, 24.3, 33.6	5 each	64 each	0.08 each
HOT	2.0, 7.6, 11.2, 24.3, 33.6	40 each	64 each	0.63 each
SPAN	33.6 each	$2 \times k (1 \leq k \leq 5)$	$16 \times k (1 \leq k \leq 5)$	0.13 each

Table 5 lists some characteristics of each sequence in these document sets.

The hotness of the automaton after filtering each sequence in the document set COLD is around 0.08, which is smaller than the estimated hotness threshold (0.5). In contrast, the hotness of the automaton after filtering each sequence in the document set HOT is around 0.625, which is larger than the estimated hotness threshold. These two document sets test low and high hotness.

The sequences in the document set HOT have similar working set sizes to the document set COLD. Both document sets test performance impacts of different working set sizes. Take the document set COLD as an example. Sequence 1 has a working set smaller than the L2 cache capacity (no cache replacement occurred in our model), and other sequences have a working set larger than the L2 cache capacity.

For constructing the hot buffer for each sequence, we warmed up the engine by filtering the sequence once and performed our speedup estimation to obtain the frequency threshold. The resulting threshold values of the hot buffers for these sequences were all one. With the frequency threshold, we extracted hot spots from the warmed-up engine and constructed the C-Buffer. The C-Buffers with the fixed-size compact row representation of Sequences 1 and 2 are smaller than the size limit (i.e., the L2 cache capacity), whereas those of other sequences reach the size limit. The C-Buffers with the variable-size compact row representation of Sequences 1-3 are smaller than the size limit, whereas those of other sequences reach the size limit. The C-Buffers with the symbol table representation of Sequences 1-4 are smaller than the size limit, whereas that of Sequence 5 reaches the size limit. As the frequency threshold for the hot spots is one for each sequence, a C-Buffer contains all referenced transitions if it does not reach the size limit. Otherwise, it contains only part of all referenced transitions.

Consequently, these five sequences represent five different cases:

1. the working set of the sequence is cache-resident (Sequence 1),
2. the working set of the sequence is not cache-resident, but each C-Buffer variant contains all state transitions that are referenced by the sequence (Sequence 2),
3. the C-Buffer with either the variable-size compact row or the symbol table representation contains all state transitions that are referenced by the sequence (Sequence 3),
4. only the C-Buffer with the symbol table representation contains all state transitions that are referenced by the sequence (Sequence 4), and
5. no C-Buffer variant contains all state transitions that are referenced by the sequence (Sequence 5).

The sequences in the document set SPAN have different numbers of symbols. They test different spans. When $S = 16$, one row of the matrix-based automaton can fit into one cache line.

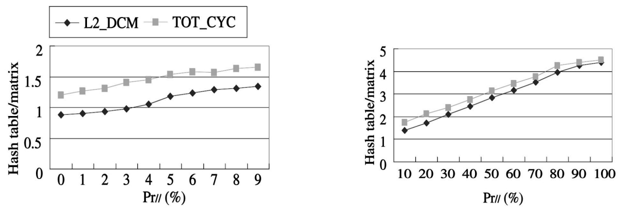
Fig. 1 shows the time breakdown of filtering the document set COLD. The numbers of cache misses were obtained from PCL. The L1 and L2 cache miss penalties were 10 and 200 cycles on P4, respectively, measured through a cache-memory calibration tool [20].

We define the accuracy of our model as follows. $accuracy = 1 - \frac{|measurement - estimation|}{measurement}$.

Finally, we summarize the three threshold values used in our experiments. The first one is the threshold for the access frequency. We select hot spots from the automaton according to this frequency threshold. The second one is the threshold for the density of an automaton. Based on this density threshold, we choose either the matrix or the hash table implementation scheme for the automaton. The third one is the threshold for the hotness of an automaton. Based on this hotness threshold, we choose either the compact row or the symbol table representation for the hot buffer.

5.2 Matrices versus Hash Tables

We experimentally compared the matrix and the hash table implementation schemes. Fig. 12 shows the cache and the overall performance ratios of the hash table over the matrix with $Pr_{//}$ varied. We report the results for the fifth sequence in the document set COLD only, because we obtained similar results for other sequences. The trend of the overall performance strictly follows that of the cache performance. Both ratios increase as the $Pr_{//}$ value increases. This indicates that the performance gap between the matrix and the hash table becomes large as the density increases.



(a) (b)

Fig. 12. The L2_DCM and TOT_CYC ratios for the hash table over the matrix. (a) Low $Pr_{//}$. (b) High $Pr_{//}$

TABLE 6
Estimated and Measured Density Threshold Values with S Varied

S	*16*	*32*	*48*	*64*	*80*
$measurement(\%)$	0.0	10.2	9.2	7.1	6.9
$estimation(\%)$	0.0	12.2	11.7	9.2	8.3

Comparing the cache performance, we find that the hash table outperforms the matrix only when the $Pr_{//}$ value is very low (i.e., the density is very low). Furthermore, the hash table has a worse overall performance than the matrix, due to its higher computation cost of each transition.

Table 6 shows the measured and the estimated density threshold values for the document set SPAN. Our estimation has an average accuracy of 81 percent.

5.3 Model Validation

Figs. 13a, 13b, 13c, 13d, 13e, and 13f show the estimated and the measured numbers of L2 data cache misses for the matrix and the hash table implementation schemes, respectively. We evaluated our model with documents and automata of different characteristics. First, we used the document set COLD to validate our modeling for different working set sizes, as shown in Figs. 13a and 13d. For both implementation schemes, we observed in our model that the L2 cache replacement did not occur in filtering Sequence 1 but happened in filtering other sequences. Our model achieved an accuracy of higher than 85 percent for both schemes. Second, we used the fifth sequence of the document set COLD and varied the number of queries to validate our modeling on different automata, as shown in Figs. 13b and 13e. We obtained similar results for the other document sequences in our document sets. Third, we used the document set SPAN to validate our modeling on different spans, as shown in Figs. 13c and 13f. Our model is consistently accurate regardless of the characteristics of the automata and the documents. The estimation is always lower than the measurement. One possible reason is that the cache misses caused by other data structures in the system (e.g., the runtime stack) are not counted in our model.

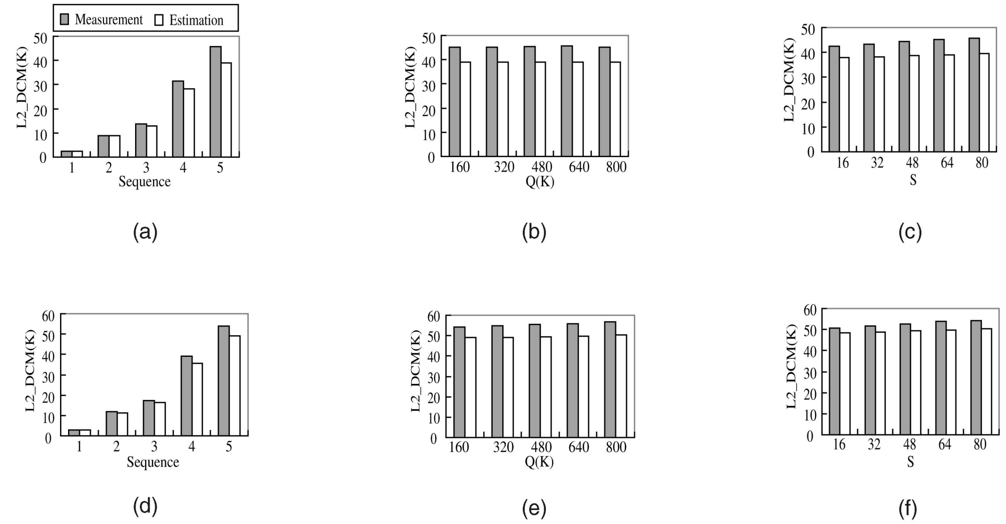
5.4 Hot Buffers

We evaluated the effectiveness of the hot buffer using two document sets, COLD and HOT. For each document set, we examined the effects of both capacity and conflict misses. Since the matrix outperformed the hash table for these two document sets, we used the matrix-based XPath-DFA as our comparison, and evaluated the filtering performance of the C-Buffer and the S-Buffer with static and dynamic workloads. Through these experiments, we find that 1) the hot buffer (either the C-Buffer or the S-Buffer) greatly improves the filtering performance and 2) the S-Buffer provides a good balance between the filtering and maintenance efficiency.

5.4.1 C-Buffer

We simulated a static workload by filtering a document sequence five times (five runs), with and without the hot buffer.

We first report the results obtained from the document set COLD. The performance comparison of filtering these five sequences is shown in Fig. 14. "w/ C-Buffer," "w/ C-Buffer(CR)," and "w/ C-Buffer(ST)," mean that the number was measured when filtering using the C-Buffer with the



(a) (b) (c)

(d) (e) (f)

Fig. 13. Model validation for the matrix and the hash table implementation schemes. (a), (b), and (c) Matrix. (d), (e), and (f) Hash table.

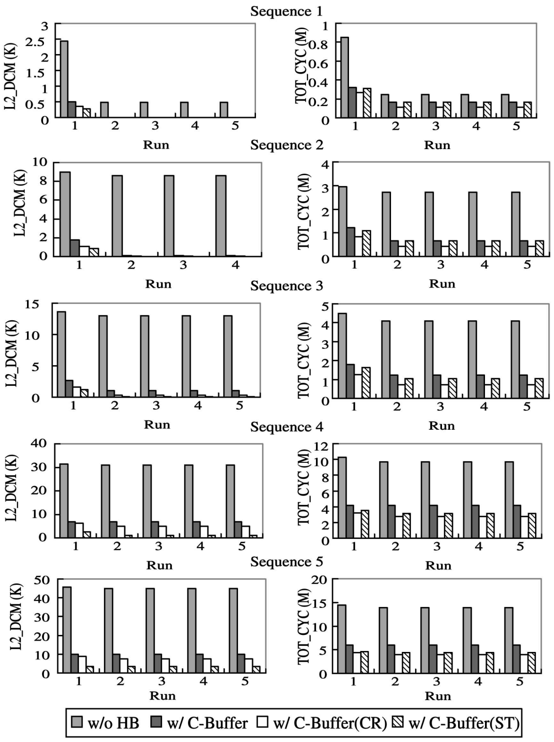


Fig. 14. Filtering with the C-Buffer. Left: L2 cache performance; Right: filtering performance.

fixed-size compact row, the variable-size compact row, and the symbol table representations, respectively, and "w/o HB" measured without the hot buffer.

All C-Buffer variants greatly improve the cache performance and the filtering performance. The main reasons include: 1) The C-Buffer has a good cache locality and reduces the number of capacity misses. 2) The contiguous layout of the C-Buffer reduces the conflict misses. 3) The hardware prefetching on P4 reduces the number of cache misses on the C-Buffer. In particular, the performance of the variable-size compact row representation is consistently higher than those of the other two hot spot representations.

We analyze the results for each case:

- **Case (a)**. Without the hot buffer, there are only a few cache misses in the second and later runs. The decrease in the number of cache misses from the first to the second run is because the working set of the documents becomes L2 cache-resident. The small numbers of cache misses in the later runs are mainly conflict misses. With the hot buffer, there are few cache misses in the later runs as few conflict misses occur. In this case, the improvement by the hot buffer is relatively insignificant, except for the first run.
- **Case (b)**. Different from Case (a), the filtering without the hot buffer in this case suffers from cache thrashing. The large numbers of cache misses in the second and later runs are capacity misses and conflict misses. In this case, the hot buffer helps greatly, as it contains all states that are referenced by the sequence.
- **Case (c)**. The hot buffer reduces the cache thrashing significantly, but the C-Buffer with the fixed-size compact row representation has a small number of cache misses due to the hot buffer misses.
- **Case (d)**. The hot buffer significantly reduces the cache thrashing, but the C-Buffers with the fixed-size and the variable-size compact row representations have a small number of cache misses due to the hot buffer misses. Although the variable-size compact row representation has more cache misses than the symbol table representation, it has a better overall performance due to its fast transition nature.
- **Case (e)**. The hot buffer reduces the cache thrashing significantly, but all C-Buffer variants have a small number of cache misses due to the hot buffer misses.

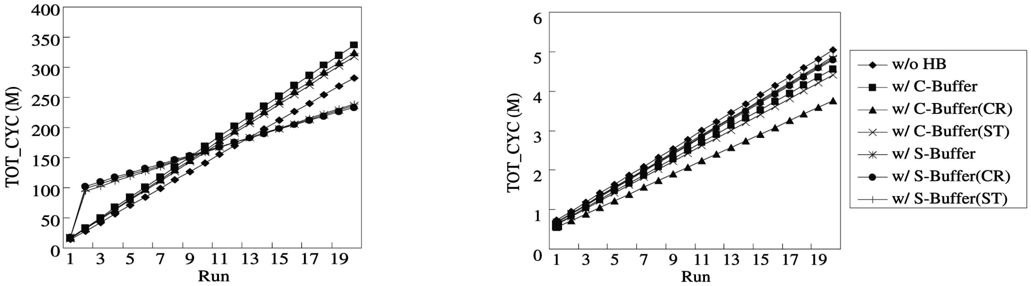
We further evaluated the C-Buffer with the document set HOT, and obtained similar results except that both the fixed-size and the variable-size compact row representations have a better cache performance as well as overall performance than the symbol table representation. We also performed the filtering with the S-Buffer and found that the S-Buffer improved the filtering performance but its performance improvement was smaller than the one with the C-Buffer.

5.4.2 S-Buffer

Our last experiment evaluated the overall performance of the hot buffer with dynamic workloads. For simplicity, we simulated a dynamic workload by filtering two sequences in a predefined order. We started by filtering one sequence (denoted as *Sequence A*) five times and the filtering performance became stable. Next, we filtered another sequence (denoted as *Sequence B*) repeatedly.

To understand how the hot spot dynamics affect the performance of the S-Buffer, we consider two kinds of dynamic workloads: one with sequences of increasing working set sizes (i.e., Sequence B has a larger working set size than Sequence A) and the other with sequences of decreasing working set sizes (i.e., Sequence B has a smaller working set size than Sequence A). We simulated the first workload using Sequences 1 and 5 in the document set COLD as Sequences A and B, respectively, and the second workload using these two sequences as Sequences B and A, respectively. We measured the accumulated execution time of each engine filtering Sequence B, shown in Fig. 15. "With S-Buffer," "With S-Buffer (CR)," and "With S-Buffer(ST)" mean that the number was measured when filtering using the S-Buffer with the fixed-size and the variable-size compact row, and the symbol table representations, respectively.

For both kinds of workloads, the S-Buffer improves the filtering performance. For the first workload (Fig. 15a), the change of the document sequences triggered the maintenance of the S-Buffer. This maintenance was because there was a dramatic drop in the hit rate of the S-Buffer (from around 100 percent to 11.5 percent) when the documents changed from Sequence 1 to Sequence 5. The performance of the XPath-DFA and the C-Buffer without maintenance were similar. In comparison, the accumulated time of the S-Buffer with maintenance had a sharp increase at the



(a) (b)

Fig. 15. Accumulated execution time of filtering dynamic workloads. (a) Workload 1. (b) Workload 2.

beginning due to the maintenance cost. After the maintenance was performed, the new S-Buffer was 50.4 percent faster than the C-Buffers without maintenance.

For the second workload (Fig. 15b), when the documents changed from Sequence 5 to Sequence 1, the hit rate of the S-Buffer increased (e.g., the hit rate of the S-Buffer with the symbol table representation increased from 75.4 percent to 95.2 percent) and no maintenance was triggered. This is because the S-Buffer constructed from Sequence 5 contained most of hot spots near the start state, which were reused during filtering Sequence 1.

6 Related Work

Cache-conscious techniques have been widely studied in the database area. A generic cost model for various database workloads was proposed by Manegold et al. [21]. Our work follows this direction but specifically models the cost of XML filtering. Specialized data structures, such as the CSS-Tree used in decision support systems [23], have been proposed to reduce cache misses. Several data layout techniques, either static [23] or adaptive [15], [24], have also been proposed for improving cache performance. Typical cache-conscious techniques, including blocking [25], data partitioning [25], loop fusion [25], data clustering [25], prefetching [8], [9], and buffering [28], were proposed for improving the cache behavior of traditional database workloads, such as joins. In contrast, we focus on the state transitions of automata, which are used as nontraditional query processing operations in XML filtering.

There has also been work from other areas on cache-conscious data structures. Klarlund et al. studied several cache-conscious techniques to reduce the pointer chasing in BDD (Binary Decision Diagram) applications [19]. Watson proposed general guidelines for cache-conscious automata, such as reorganizing the automata according to the access patterns and popularity information [26]. Compression techniques for automata [18] also improve the memory performance. As the automata in our work are used for XML filtering, we modeled their cache behavior with respect to the filtering workload and proposed techniques that take advantage of the locality patterns of the workload.

Automaton-based XML filtering has emerged as a fruitful research area. XFilter [3] uses one NFA for each path query and uses list-balancing to speed up the processing. Both YFilter [12], [13] and XTrie [6] support path sharing and convert a large number of XPath queries into a single NFA. XFilter, YFilter and XTrie use various indexing and optimization techniques to improve the filtering throughput. The lazy DFA [14] attempts to perform the subset construction only when needed in order to improve the scalability. Our model and technique are applicable to these existing engines for further performance improvement.

With the same focus on the cache-conscious XML filtering as this paper, our previous paper [16] modeled an automaton with a span one and developed the hot buffer technique to handle static hot spots. In contrast, this paper extends the model to the case where the span can be multiple cache lines and supports estimations for the matrix and the hash table implementation schemes. Moreover, this paper proposes the S-Buffer technique to support incremental maintenance for dynamic hot spots.

7 Conclusion

Automata are the key data structure in several XML filters. Through experiments, we find that cache stalls, especially L2 data cache stalls, are a major hurdle for further improving the performance of large automaton-based XML filters. To study the cache performance of automaton-based XML filtering, we have estimated the cache misses by modeling the filtering process. Furthermore, we have proposed a cache-conscious technique, the hot buffer, to improve the cache performance as well as the overall performance of automaton-based XML filtering.

Acknowledgments

The authors are grateful to Derick Wood for his advice on automata theories and to Yajun Wang, Yi Wang, and Jun Zhang for their discussions. This work was supported by the Hong Kong RGC grants HKUST6158/03E, HKUST6263/04E, and DAG05/06.EG11. The work of Byron Choi was done in part while he was visiting HKUST in 2003 and was supported by grant HKUST6197/01E. The authors thank the anonymous reviewers for their insightful comments on this paper.

References

[1] SAX: Simple API for XML, http://www.saxproject.org, 2005.

[2] A. Ailamaki, D.J. DeWitt, M.D. Hill, and M. Skounakis, "Weaving Relations for Cache Performance," *Proc. 27th Int'l Conf. Very Large Data Bases,* 2001.

[3] M. Altinel and M.J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," *The VLDB J.,* pp. 53-64, 2000.

[4] R. Berrendorf, H. Ziegler, and B. Mohr, "PCL: Performance Counter Library," http://www.fz-juelich.de/zam/PCL/, 2005.

[5] A.F. Cardenas, "Analysis and Performance of Inverted Data Base Structures," *Comm. ACM,* vol. 18, no. 5, pp. 253-263, 1975.

[6] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions," *VLDB J.,* special issue on XML, vol. 11, no. 4, 2002.

[7] S. Chatterjee, V.V. Jain, A.R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear Array Layouts for Hierarchical Memory Systems," *Proc. 13th Int'l Conf. Super Computing,* 1999.

[8] S. Chen, A. Ailamaki, P.B. Gibbons, and T.C. Mowry, "Improving Hash Join Performance through Prefetching," *Proc. Int'l Conf. Data Eng.,* 2004.

[9] S. Chen, P.B. Gibbons, T.C. Mowry, and G. Valentin, "Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance," *Proc. ACM SIGMOD Conf.,* 2001.

[10] Z. Chen, H. Jagadish, F. Korn, N. Koudas, R. Ng, S. Muthukrishnan, and D. Srivastava, "Counting Twigs in a Tree," *Proc. Int'l Conf. Data Eng.,* 2001.

[11] "XML Path Language (XPath)-Version 1.0," *W3C Recommendation,* J. Clark and S. DeRose, eds., http://www.w3.org/TR/xpath, 1999.

[12] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, and P. Fischer, "Path Sharing and Predicate Evaluation for High-Performance XML Filterin," *ACM Trans. Database Systems,* Dec. 2003.

[13] Y. Diao, P. Fischer, M.J. Franklin, and R. To, "YFilter: Efficient and Scalable Filtering of XML Documents," *Proc. Int'l Conf. Data Eng.,* 2002.

[14] T.J. Green, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams With Deterministic Automata," *Proc. Int'l Conf. Database Theory,* 2002.

[15] R.A. Hankins and J.M. Patel, "Data Morphing: An Adaptive and Cache-Conscious Storage Technique," *Proc. 29th Int'l Conf. Very Large Data Bases,* 2003.

[16] B. He, Q. Luo, and B. Choi, "Cache-Conscious Automata for XML Filtering," *Proc. Int'l Conf. Data Eng.,* 2005.

[17] M.D. Hill and A.J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. Computers,* vol. 38, no. 12, pp. 1612-1630, Dec. 1989.

[18] G.A. Kiraz, "Compressed Storage of Sparse Finite-State Transducers," *Proc. Workshop Implementing Automata,* 1999.

[19] N. Klarlund and T. Rauhe, "BDD Algorithms and Cache Misses," Technical Report, BRICS Report Series RS-96-5. Univ. of Aarhus, 1996.

[20] S. Manegold, "The Calibrator (v0.9e), a Cache-Memory and TLB Calibration Tool," http://www.cwi.nl/~manegold/Calibrator/, 2005.

[21] S. Manegold, P. Boncz, and M. Kersten, "Generic Database Cost Models for Hierarchical Memory Systems," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 191-202, 2002.

[22] V.L. Maout, *ASTL: Automaton Standard Template Library,* http://www-igm.univ-mlv.fr/lemaout/, 2005.

[23] J. Rao and K.A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," *Proc. 25th Int'l Conf. Very Large Data Bases,* 1999.

[24] J. Rao and K.A. Ross, "Making B+ Trees Cache Conscious in Main Memory," *Proc. ACM SIGMOD Conf.,* 2000.

[25] A. Shatdal, C. Kant, and J.F. Naughton, "Cache Conscious Algorithms for Relational Query Processing," *Proc. 20th Int'l Conf. Very Large Data Bases,* 1994.

[26] B.W. Watson, "Practical Optimizations for Automata," *Proc. Second Int'l Workshop Implementating Automata,* 1997.

[27] D. Wood, *Theory of Computation,* New York: John Wiley, 1987.

[28] J. Zhou and K.A. Ross, "Buffering Access to Memory-Resident Index Structure," *Proc. Int'l Conf. Very Large Data Bases,* 2003.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University (1999-2003). He is a PhD student in the Computer Science and Engineering Department, Hong Kong University of Science and Technology (HKUST). His research interests are in database systems, especially cache-centric query processing techniques.

Qiong Luo received the BS and MS degrees in computer sciences from Beijing (Peking) University, China, in 1992 and 1997 respectively, and the PhD degree in computer sciences from the University of Wisconsin-Madison in 2002. She is an assistant professor in the Computer Science and Engineering Department, Hong Kong University of Science and Technology (HKUST). Her research interests are database systems, with a focus on data management and analysis techniques related to network applications.

Byron Choi received the BEng degree in computer engineering from the Hong Kong University of Science and Technology (HKUST) in 1999 and the MSE and PhD degrees in computer and information science from the University of Pennsylvania in 2002 and 2006, respectively. He is currently an assistant professor in the School of Computer Engineering, Nanyang Technological University, Singapore.