Cache-Conscious Automata for XML Filtering

Bingsheng He, Qiong Luo
Hong Kong University of Science and Technology
{saven,luo}@cs.ust.hk

Byron Choi
University of Pennsylvania
kkchoi@seas.upenn.edu

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.

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 emerging applications of XML filtering, such as selective information dissemination and content-based XML routing. Because 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 [2, 3, 8, 9, 10], we find that they are all based on automata (either an NFA, a Non-deterministic 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 since their state transitions are deterministic. Because the conversion from an NFA to a DFA increases the number of states exponentially [23], recent work such as the lazy DFA [10] 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 form of the YFilter [8, 9]. 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 [23] 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.

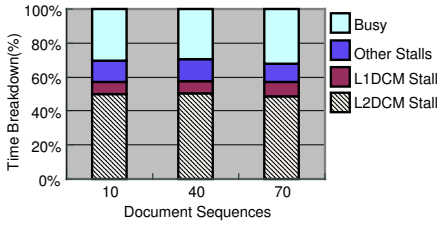

Figure 1. Execution Time Breakdown

Figure 1 shows the time breakdown of a memory-resident XPath-DFA filtering sequences of XML documents (Sequences 10, 40, and 70) on a Pentium-III machine. Details of the experiment are described in Section 5. In this figure, the filtering time of each sequence is divided into four categories: L2 data stall, L1 data stall, other stalls (e.g., stalls due to L1 instruction misses and branch misprediction), and CPU busy time. This figure shows that the XPath-DFA spends a large portion of the running time - more than 55% - on stalls due to data cache misses, especially L2 data cache misses. Hence, there is considerable room for improvement of the cache performance of automaton-based XML filtering.

Cache-conscious techniques [1, 5, 6, 11, 19, 20, 21, 24] have been shown to be 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 [4], to automaton-based XML filtering. General-purpose cache-conscious techniques such as blocking [21] are also inapplicable in this case.

In order to propose cache-conscious techniques that work for automaton-based XML filtering, we start by developing an analytical model of its cache performance. We define a *round* as the engine filtering one document. We then estimate in our model the number of cache misses in three categories: compulsory misses, capacity misses and conflict misses [12]. The total number of cache misses is the sum of these three categories. Our model estimates the total number of cache misses in filtering a single document (*intra-round filtering*) and a sequence of documents (*inter-round filtering*).

Based on the insights gained from the model, we propose to extract *hot spots* (frequent state transitions) in an automaton into a contiguous memory area called a *hot buffer*. This hot buffer aims at improving the spatial locality of the filter. Through experiments, we have validated the accuracy of our model and the effectiveness of the hot buffer.

This paper makes the following three contributions: (1) we develop the first analytical model for the cache performance of XML filtering; (2) we propose a cache-conscious technique for the performance improvement of automaton-based XML filtering; and (3) we conduct an empirical study to verify the accuracy of our model and the effectiveness of our cache-conscious technique.

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 proposed cache-conscious technique. 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.

2.1 XPath Queries Handled

As the first step in studying cache-conscious automata for XML filtering, we focus on a basic subset of XPath queries [7] defined as follows:

$q ::= w^+$,
$w ::= /l \mid //l \mid /* \mid \epsilon$.

In this subset, an XPath query, q, consists of a sequence of *location steps*. A location step, w, can be either empty (ϵ) or non-empty. A non-empty location step, w, consists of (1) a downward XPath axis (either "/", a child axis, or "//", a descendant axis) and (2) l or *, denoting an XML element name and a wildcard, respectively.

2.2 System Overview

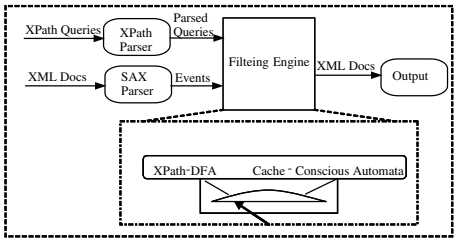

Figure 2. The XML Filtering System Architecture

The system architecture of our XML filtering system is shown in Figure 2. The major components of this system include an XPath parser for parsing user queries, an event-based SAX parser [18] 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 automata. We describe our cache-conscious automata in detail in Section 4.

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, when a start-of-element event comes to the engine, it triggers a state transition in the automaton. When an end-of-element event occurs, the automaton *backtracks* to the previous state. A runtime *stack* is used to keep track of the current and previously visited states. Upon an end-of-document event, the system checks if any accepting state has been reached and disseminates the document to users whose XPath queries are satisfied.

An example of the filtering process is shown in Figure 3. On the top-left are the XPath queries. On the bottom-left is the fraction of XPath-DFA that represents the XPath-queries. The circles in the DFA represent the states and the shadowed circles represent the accepting states. A number in "{}" 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. 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.

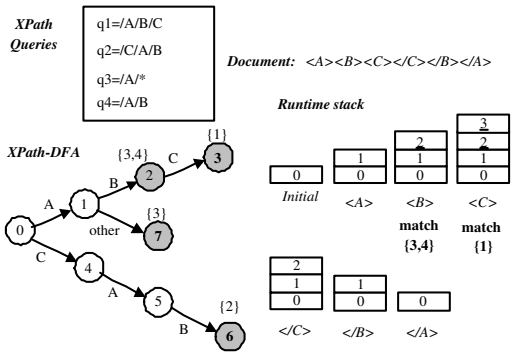

Figure 3. An XML Filtering Process

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

2.3 Automata Implementation

We briefly review three commonly used alternatives for automaton implementation, including the map, the hash table and the matrix.

A map for a DFA (or a multi-map for an NFA) is implemented with a red-black tree (e.g., the ASTL [17]). It has a good balance between speed and memory usage, but the lookup time is still significant. In contrast, a hash table with a good hash function requires almost constant time on a lookup. However, due to the random access nature of hash tables, a state transition causes two cache misses on average, one for fetching the bucket and the other for fetching the values. Finally, all state transitions in a matrix are performed in constant time. Although state transitions in a matrix are also random access, one state transition causes only one cache miss on average (directly fetching the state). In addition, the space efficiency of the matrix is acceptable in practice, as the matrix is dense in a large filtering system with diversified user interests and heterogeneous XML documents.

We have experimented with all three implementation alternatives and have found that matrices performed the best in automaton-based XML filtering. Hence, in the remainder of this paper, we use a matrix-based implementation.

3 Modeling Cache Performance

In this section, we propose an analytical model to estimate the total number of cache misses in XML filtering. We estimate the number of compulsory misses and capacity misses by modeling the intra-round and the inter-round filtering processes. We then give a coarse estimation on the number of conflict misses in association with the number of compulsory misses.

Table 1. Cache Configuration

Parameters	Description
C	The cache capacity (bytes)
B	The cache line size (bytes)
N	The number of cache lines ($\frac{C}{B}$)
A	The degree of set-associativity

3.1 Modeling 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:

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

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 cache-conscious techniques aim at minimizing the cache miss rates and the total number of cache accesses.

As we have seen in Figure 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 unified memory shared by data and instructions. *In the remainder of this paper, "cache" is short for "L2 data cache" unless specified otherwise.*

We define the *cache configuration* as a four-element tuple <*C*,*B*,*N*,*A*> (Table 1). *A*=1 for a direct-mapped cache; *A*=*N* for a fully associative cache; and *A*=n for an n-associative cache ($1 < n < N$). In addition, we model the cache replacement policy as LRU (Least Recently Used).

3.2 Modeling Intra-round Filtering

We start with intra-round filtering, during which a single document is filtered and there is no data reuse from the previous rounds. Because a single document is small in practice, we assume that there are no capacity misses in intra-round filtering. Subsequently, we focus on compulsory misses in modeling intra-round filtering and give a coarse estimation on conflict misses based on compulsory misses in later discussions.

3.2.1 Definitions

We model the cache behavior of a state transition, i, in an automaton implementation using three metrics,

$<F_i, R_i, M_i>$:

•*F_i represents the footprint of i.* Footprint F is the mean number of the cache lines that contain the information for a state transition. Specifically, F_i has a constant value of 1 in the matrix-based implementation and 2 in the hash table-based implementation. It varies for the map-based implementation. *For simplicity, we model F_i as a constant and use F_i and F interchangeably.*

•*R_i represents the reuse of i.* Reuse R_i is the number of the cache lines that are required for i and have already been brought into the cache by previous state transitions. If $R_i = F_i$, we call it *perfect reuse*.

•*M_i represents the number of compulsory misses caused by i.* $M_i = F_i - R_i$.

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

3.2.2 Intra-round Estimation

From the definitions of $<F_i, R_i, M_i>$ of a state transition, *i*, and the working set, ws, of a document, d, we have two ways to estimate the number of compulsory misses, Com, in intra-round filtering d: (1) $Com = \sum_{i=1}^{p} M_i = \sum_{i=1}^{p} (F - R_i)$, where p is the total number of state transitions in filtering the document; and (2) $Com = |ws| \cdot F$. We can use either way to validate the other; for simplicity, we used (2) in our estimation algorithm.

We propose a data structure called the *tag tree* of a document to estimate the working set of the document over an XPath-DFA. A *tag tree* of a document is a tree that satisfies the following two properties: (1) each node represents a state in the automaton and has a unique ID (the root has an ID 0); (2) each edge represents a state transition in the automaton and is labeled with a tag in the document.

We say a node of a tag tree resides in the cache if the corresponding state in the automaton resides in the cache. Similarly, we say a node of a tag tree is fetched into the cache if the corresponding state in the automaton is fetched into the cache. To simulate the LRU replacement policy, we add a *timestamp* to each node. The timestamp is initialized with the value -1, which means that the node has not been fetched into the cache. Whenever a state is accessed, the timestamp of its corresponding node of the tag tree is updated to a positive value equaling the total number of state transitions that occured. Consequently, a node with a positive timestamp indicates that the node resides in the cache. In addition, the more recently a node has been accessed, the larger the value of the timestamp.

An example of constructing a tag tree is shown in Figure 4. At the top of the figure is a document. On the left is the constructed tag tree of the document. The rectangles in the tree represent tag tree nodes. The number in a node is the tree node ID and the number beside a node is the timestamp. The tag tree construction process is similar to the filtering process, and a runtime stack (on the right of the figure) is used to hold the active and previously accessed nodes. The '$' in the content of the stack indicates an update of the timestamp.

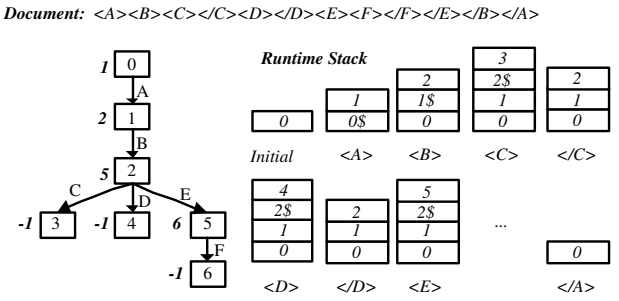

Figure 4. Tag Tree Construction

Algorithm 1 Tag Tree Construction

Input: streaming document events, footprint F
Output: ws, Com

1: Upon a start-of-document event, initialize a tag tree T with Node 0 only, an empty stack s, $i = 0$, s.push(Node 0);
2: **while** e is not an end-of-document event **do**
3: **if** e is a start-of-element event **then**
4: i++, $R_i = 0$; //Processing the i^{th} state transition
5: $curNode$=$s.top()$;
6: **if** $ChildOf(curNode, e)$ == $null$ **then**
7: $NewChild(curNode, e)$;
8: s.push($ChildOf(curNode, e)$);
9: **if** $curNode$.timestamp ≥ 0 **then**
10: $R_i = F$;
11: $curNode$.timestamp= i;
12: **else**
13: s.pop(); //e is an end-of-element event
14: add all nodes of a positive timestamp to ws;
15: $Com = |ws| \cdot F$.

Algorithm 1 constructs the tag tree, computes the working set of a document, and estimates the number of compulsory misses. We build the tag tree by parsing the document once. Initially, the root (Node 0) is the active node. For each start-of-element event, e, we simulate the state transitions using the tag tree. First, we check if the active node has a child node connected by an e edge (Line 6). If not, we create a new child for the active node with an e edge (Line 7). Then, we push the child node (either already existing or newly created) into the runtime stack (Line 8). Next, we determine the reuse, R_i: if the node resides in the cache

already, a perfect reuse occurs (Line 9-10). At Line 11, we update the active node's timestamp. When an end-of-element event comes in, we pop the stack.

At the end, all nodes with a positive timestamp, which represent the states accessed when filtering the document, form the working set (Line 14). Finally, we estimate the number of compulsory misses at Line 15. Using this algorithm, we estimate that the working set size of the example document in Figure 4 is four and that the number of compulsory misses is also four (given $F = 1$). Note, the reuse, R_i, calculated in the algorithm can be used to compute the number of compulsory misses in the other way for validation.

Since our algorithm assumes that there is a one-to-one mapping between a tag tree node and an automaton state, the accuracy of our estimation may be affected when multiple tag tree nodes correspond to one state in an XPath-DFA. Fortunately, in a large filtering system with all sorts of XPath queries, this mapping is close to one-to-one correspondence.

3.3 Modeling Inter-round Filtering

Having modeled cache misses of intra-round filtering, we then model the cache behavior of inter-round filtering. There are two major differences between intra-round and inter-round filtering:

• Each round (except the first round) in inter-round filtering is likely to have reuse of the working sets of previous documents.

• As multiple documents come, the size of all states referenced during filtering may exceed the capacity of the cache. Therefore, capacity misses should be considered in inter-round filtering.

3.3.1 Definitions

Given a sequence of documents $(doc_1, doc_2, ..., doc_n)$, we estimate the corresponding working sets $(ws_1, ws_2, ..., ws_n)$ using Algorithm 1. We define the intersection, union, and difference operations on the sequence of working sets to be ordinary n-ary set operations: *Intersection* ($ws_1 \cap ws_2 ... \cap ws_n$); *Union* ($ws_1 \cup ws_2 ... \cup ws_n$); and *Difference* ($ws_1 - ws_2 ... - ws_n$).

Denote $rs_{i,1}$, $rs_{i,2}$,..., $rs_{i,i}$ as the subsets of ws_1, ws_2,..., ws_i that reside in the cache at the beginning of the i^{th} round. We call $rs_{i,j}$ the *cache set* of document j at the beginning of the i^{th} round. The set of cache sets, $\{rs_{i,j} | 1 \leq j \leq i\}$, constitutes the *cache content* at the beginning of the i^{th} round. The cache content at the beginning of $(i+1)^{th}$ round is $\{rs_{i+1,j} | 1 \leq j \leq i+1\}$, which is also the cache content at the end of the i^{th} round. Since we assume that there is no capacity miss in intra-round filtering, $rs_{i,i} = \phi$ at the beginning of the i^{th} round while $rs_{i+1,i} = ws_i$ at the beginning of the $(i+1)^{th}$ round.

3.3.2 Inter-round Estimation

We start by modeling the changes in the cache contents in inter-round filtering. Based on the modeled inter-round change of cache contents, we estimate the number of compulsory misses and capacity misses. Finally, we discuss how to implement the inter-round modeling using tag tree operations.

Cache Content. Given a sequence of n documents $(doc_1, doc_2, ..., doc_n)$ with corresponding working sets $(ws_1, ws_2, ..., ws_n)$, we examine the change in the cache content in each round. This change will be affected by the working set of the document as well as the cache replacement policy. Specifically, given the working set, ws_i, of document i at the beginning of the i^{th} round, the cache content at the beginning of the $(i+1)^{th}$ round is estimated as follows:

• If no replacement occurs, $rs_{i+1,j} = rs_{i,j}$ $(1 \leq j < i)$, $rs_{i+1,i} = ws_i$ and $rs_{i+1,i+1} = \phi$.

• If replacement occurs, $rs_{i+1,j}$ is a subset of $rs_{i,j}$ due to the replacement. With the LRU replacement policy, the candidate states for replacement in document j's cache set at the beginning of the i^{th} round are $Cand_{i,j} = (rs_{i,j} - rs_{i,j+1} - ... - rs_{i,i}), 1 \leq j < i$. A $Cand_{i,j}$ with a smaller j has a higher priority to be replaced. Within a $Cand_{i,j}$, an earlier state has a higher priority to be replaced. The number of states need to be replaced is $\#toReplace_i = |rs_{i,1} \cup rs_{i,2} \cup ... \cup rs_{i,i-1} \cup ws_i| - \frac{N}{F}$. Replacement is done by going through $Cand_{i,1}$, $Cand_{i,2}$,..., until the number of states replaced reaches $\#toReplace_i$. This way, $rs_{i+1,j} (1 \leq j < i)$ is updated at the end of replacement. Finally, we set $rs_{i+1,i} = ws_i$ and $rs_{i+1,i+1} = \phi$.

Estimation. After modeling changes in the cache contents between rounds, we can estimate:

• *the number of compulsory misses in the i^{th} round* as $|ws_i - (rs_{i,1} \cup rs_{i,2} \cup ... \cup rs_{i,i})| \times F$, which is the number of the new states in the automaton referenced in filtering doc_i;

• *the number of capacity misses in the i^{th} round* as considered in two cases:

(1) If $|rs_{i,1} \cup rs_{i,2} \cup ... \cup rs_{i,i} \cup ws_i| \times F \leq N$, no replacement occurs and the number of capacity misses is 0;

(2) If $|rs_{i,1} \cup rs_{i,2} \cup ... \cup rs_{i,i} \cup ws_i| \times F > N$, there is replacement in the cache and capacity misses occur. The set of the states that is swapped out of the cache in this round is $Rp_i = ((rs_{i,0} \cup rs_{i,1} ... \cup rs_{i,i-1}) - (rs_{i+1,0} \cup rs_{i+1,1} ... \cup rs_{i+1,i-1}))$. The set of the states that is reloaded into the cache is $toReload_i = Rp_i \cap ws_i$. The number of capacity misses in the i^{th} round is $|toReload_i| \times F$.

Implementation. We implement the estimation using the tag tree operations of intersection, difference and union.

Two nodes, n_i and n_j, from two tag trees, T_i and T_j, are considered to *overlap* if the following two conditions hold: (1) there exists a path from the root of T_i to n_i and one from the root of T_j to n_j, both of which contain the same sequence of tags; (2) both n_i and n_j have positive timestamps. By this definition, two root nodes with positive timestamps always overlap. A node with a negative timestamp is ignored because it is not in the cache.

Given two tag trees, T_m, T_n ($m < n$), we define the three operations of the two tag trees. (1) *Intersection*. *I*(T_m, T_n) results in a new tag tree that contains all overlapping nodes from T_n. The timestamp of an overlapping node is set to be the same as that in T_n. (2) *Difference*. *D*(T_m, T_n) results in a new tag tree that contains all nodes from T_m, with the timestamp of every overlapping node being set to -1. (3) *Union*. *U*(T_m, T_n) results in a new tag tree that contains all overlapping nodes whose timestamps are set to be the same as those in T_n, as well as all other nodes from the two tag trees. We have developed algorithms for these tag tree operations. Since they are quite straightforward, we omit the algorithms but show an example of these operations in Figure 5.

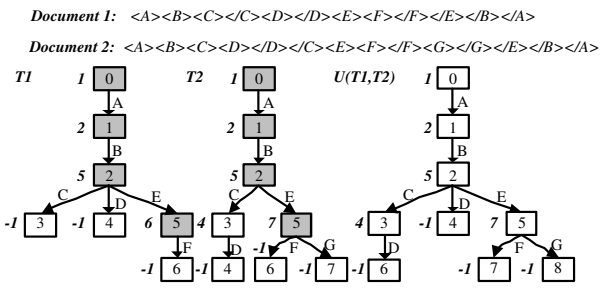

Figure 5. Operations on Two Tag Trees

Figure 5 shows two tag trees, T_1 and T_2, constructed from the two XML documents at the top of the figure. The shadowed rectangles represent the overlapping nodes of T_1 and T_2. Note that Node 3 in T_1 and Node 3 in T_2 do not overlap, as one of them has a negative timestamp. On the right of the figure is the union of T_1 and T_2. The difference of T_1 and T_2 has the same set of nodes as T_1 but all nodes have a timestamp -1.

Given a sequence of documents, doc_1, doc_2,...doc_i, the correspondence between working set operations and tag tree operations is listed in Table 2. The number of states in the result of the operations on the working sets is the number of nodes with a positive timestamp in the result of the corresponding operations on the tag trees.

Similarly, we implement the operations on cache sets using the operations on tag trees. Denote $rt_{i,j}$ to be the tag tree that corresponds to the cache set $rs_{i,j}$ and $\{rt_{i,j}|1 \leq j \leq i\}$ to the cache content at the beginning of the i^{th} round. When the cache content changes between rounds (including cache replacement), the tag trees need to be maintained to reflect the change. For this purpose, we add a round ID to each node in the tag tree. The round ID indicates in which round the node has been referenced most recently.

Table 2. Correspondence of Working Set and Tag Tree Operations

Operations on working sets	Operations on tag trees
$ws_1 \cap ws_2 \cap ... \cap ws_i$	$I(I(T_1, ..., T_{i-1}), T_i)$
$ws_1 \cup ws_2 \cup ... \cup ws_i$	$U(U(T_1, ..., T_{i-1}), T_i)$
$ws_1 - ws_2 - ... - ws_i$	$D(T_1, U(T_2, ..., T_i))$

In the i^{th} round, given the tag tree, T_i, that corresponds to the working set of document i, we set each node of $rt_{i,j}$ ($1 \leq j \leq i$) that overlaps with a node of T_i to have a round ID of i and a timestamp the same as that of the node in T_i. When cache replacement occurs, a node with a smaller round ID has a higher priority to be replaced. Among nodes with the same round ID, the node with a smaller timestamp value has a higher priority to be replaced. When a node is swapped out of the cache, its timestamp is set to be -1. At the end of the i^{th} round, $rt_{i+1,i} = T_i$.

With these tag trees, we can estimate the compulsory misses in the i^{th} round as $(|T_i - (rt_{i,1} \cup rt_{i,2} \cup ... \cup rt_{i,i})|) \times F$. The number of capacity misses can be estimated in a similar way.

3.4 Discussion

Estimation of Conflict Misses. Having modeled compulsory misses and capacity misses, we now discuss conflict misses. Conflict misses depend on a number of factors, such as cache set-associativity, data layout and access patterns. However, once the data layout and alignment are fixed in the main memory, for a random data access pattern, the probability of causing a conflict miss by fetching a cache line into the cache is uniform across all cache lines. Since state transitions in an automaton can be treated as a random access pattern, the ratio of conflict misses to compulsory misses is expected to be quite stable, as we observed in the experiments. This ratio allows us to estimate the conflict misses from the compulsory misses.

In summary, the total number of cache misses in the i^{th} round, TM_i, is estimated as follows, where Com_i and Cap_i are respectively the numbers of compulsory misses and capacity misses estimated by the inter-round model and r is the ratio of conflict misses to compulsory misses.

$$TM_i = Com_i + Com_i \times r + Cap_i \qquad (2)$$

Finally, even though we essentially simulate the filtering process in the tag tree construction, 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. For

instance, PAPI [15], the profiling tool we used, can report the total number of misses, but cannot divide them into the three categories. In contrast, our model can estimate the three categories of misses based on the processing flow of the filtering automaton. Second, our approach is more flexible. Our model can run on multiple platforms without much modification, while profiling tools need to be carefully studied and the application code needs to be carefully instrumented for profiling. 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 in the application code.

4 Cache Conscious Automata

Our cache behavior model for filtering makes it clear that locality (e.g., the state reuse) is essential for reducing cache misses. In addition, we know that sequential access is much more cache friendly than random access. These two observations motivate us to develop a cache-conscious automaton organization technique, which we call the hot buffer.

A *hot buffer* is a contiguous area that stores *hot spots*, which are the state transitions frequently performed in a filtering engine. For example, the state transitions that correspond to the root element of a document are accessed more frequently than others. By putting these frequent state transitions together into a hot buffer, the number of compulsory misses and capacity misses is reduced. Furthermore, we try to pack the hot spots in some frequent transition paths in an almost contiguous way. This packing reduces the conflict misses. Finally, we put a size limit on the hot buffer, which aims at reducing the capacity misses.

In this section, we first describe the hot buffer technique using a simple example and then present an algorithm for constructing the hot buffer. Last, we discuss our hot buffer technique in association with our model.

4.1 An Example of a Hot Buffer

Let us describe the hot buffer using an example. We start from the original automaton without a hot buffer. We use the matrix implementation for this automaton: the row index is the current state, the column index is the symbol (XML element name, or XML tag), and each matrix cell is the next state to go to from the current state upon the symbol. We add a counter to each cell in this matrix in order to record the frequency statistics of the corresponding state transition. A fragment of such an automaton is shown on the left of Figure 6. The content of each cell in the automaton is in the form of <next state, counter>.

In order to construct a hot buffer, we first need to identify the hot spots. Suppose we set the *threshold* of hotness to be

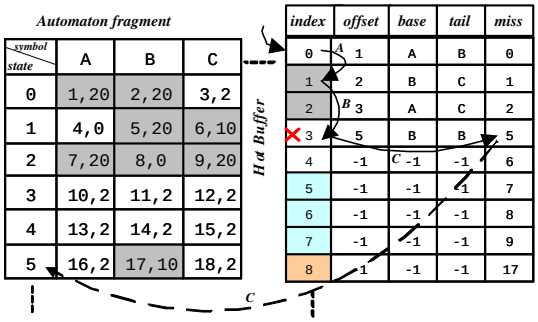

Figure 6. Hot Buffer Construction

an access frequency of 10. We identify the hot spots by scanning the matrix row by row and recording the *hot base* (the left-most cell that is hot) and the *hot tail* (the right-most cell that is hot) of the candidates in each row. To preserve the fast-transition advantage of a matrix, we identify all cells between the hot base and the hot tail (both ends inclusively) in a row to be hot spots and call this fragment a *compact row*. This also improves the construction speed. For example, the shadowed cells in the original automaton in Figure 6 are identified as hot spots. Note that some shadowed cells in a compact row (e.g., the cell <8,0> in Row 2 Column B) have an access frequency of less than 10.

After identifying the hot spots, we pack them into the hot buffer. The hot buffer on the right of Figure 6 is constructed from the hot spots in the original automaton on the left.

As shown in Figure 6, the hot buffer is an array structure starting from index 0. Each element of the array is a tuple with four fields, <*offset*, $base$, $tail$, $miss$>. All values are initialized to be -1. The $base$ and $tail$ of an array element in the hot buffer correspond to the starting position and the ending position of a compact row in the original automaton, if any. That is, the values of $base$ and $tail$ are the smallest symbol and the largest symbol respectively in the compact row in the original automaton. In our implementation, the neighboring symbols are coded as consecutive integers.

For each compact row in the original automaton, $(tail - base + 1)$ array elements are added to the hot buffer. The value of *offset* is set during the hot buffer construction process. The use of *offset* is to lead the current index to move forward *offset* array elements during filtering. Given the current index, cur, in the hot buffer and the current symbol, tag, if $base \neq -1$ and $base \leq tag \leq tail$, the next array element to visit is the one with an index of $(cur$+*offset*$+tag - base)$ in the hot buffer; otherwise, its $miss$ field points to the state in the original automaton to resolve the miss on the array element in the hot buffer.

Filtering the hot buffer. After the hot buffer is constructed, we can then perform filtering on the hot buffer using a runtime stack.

If a transition can be finished in the hot buffer (a *hot buffer hit*), the transition will not reference the original automaton. When there is a hot buffer miss, the original au-

tomaton has to be visited. Before visiting the original automaton, we save the context of the stack, the size of the runtime stack and the content on the top of the runtime stack as $(mSize, mTop)$. Next, we replace the top of the runtime stack with the state in the original automaton to which the $miss$ field of the $mTop$ array element in the hot buffer points. The transitions continue on the original automaton until the size of the stack becomes $mSize$ again. At this point, we replace the top of the runtime stack with $mTop$ and resume the transitions on the hot buffer. After an entire document is filtered, the queries that are associated with an accepting state are satisfied.

The arrows shown in Figure 6 illustrate the filtering process of an XML document, $<A><B><C></C></B><B></B></A>$, with the hot buffer. The arrows with solid lines represent the transitions on the hot buffer and the arrow with a dashed line redirects the transitions to the original automaton. The filtering process starts from index 0 ($cur = 0$) on the hot buffer. When the $<A>$ tag comes in, we calculate the next index by (cur+*offset*+$tag - base$), where $tag = A$, *offset*=1, $base = A$ (in element 0). The next index is thus 1. The subsequent event, $<B>$, is processed in a similar way. When processing the third event, $<C>$, there is a hot buffer miss and the transition is redirected to the original automaton. After processing the fourth event, $</C>$, the runtime stack is restored and the transitions resume on the hot buffer.

4.2 Hot Buffer Construction

To make the hot buffer cache-resident and achieve a low miss rate, we put two constraints on the construction of the hot buffer:

• *The size of the hot buffer is smaller than the cache capacity,* C. If the size is close to or larger than C, the hot buffer will thrash. If the size is too small, the miss rate of the hot buffer will be high. In our implementation, we set the size limit of the hot buffer to be one half the size of the L2 cache.

• *The hot spots are selected with a threshold on the access frequency.*

With these two constraints considered, Algorithm 2 constructs the hot buffer in a top-down manner. It uses a queue ,$wqueue$, to store the *waiting states* of the automaton, M. A waiting state is a state from which some state transition (a hot spot) will be put into the hot buffer. The algorithm uses another queue, $pqueue$, to store the positions (indexes) at which the hot spots are put in the hot buffer.

Before we put a new hot spot into the hot buffer, we check if any existing hot spot in the buffer transits to the state (the $miss$ value) to which the new hot spot transits (Line 5). If this is true, we set the content ($base$, $tail$, and $miss$) of this new hot spot in the buffer to be the same as the existing one except that its *offset* value is adjusted (Line 12). Otherwise, the following operations are performed (Lines 6–10). First, we extract the compact row, cR, from the original automaton using the threshold, t. Then, we set the current hot buffer element. Next, we put the waiting states in the compact row, cR, into $wqueue$ and put the indexes ranging in [$curSize$, $curSize+|CR|-1$] into $pqueue$. Finally, the algorithm updates the current size of the hot buffer and checks if the size limit is exceeded. If so, we use procedure $SetMiss(pqueue, wqueue)$ to set the field, $miss$, of the hot buffer elements with indexes in $pqueue$ to be their corresponding waiting states in $wqueue$, and the hot buffer construction ends.

Algorithm 2 Hot Buffer Construction

Input: Automaton M with counters, the size limit of the hot buffer $sizeLimit$, and threshold t

1: Initialize $wqueue$ and $pqueue$, $curSize$=1;
2: Insert M's start state to $wqueue$ and 0 to $pqueue$;
3: **while** $wqueue$ is not empty **do**
4: $curState = wqueue$.dequeue(), $curIndex = pqueue$.dequeue();
5: **if** $curState$ has not appeared in $miss$ in the hot buffer **then**
6: Generate the compact row cR from row $curState$ in M using t;
7: Set the hot buffer's $curIndex$ element: *offset*= $curSize - curIndex$, $miss = curState$, and $base$, $tail$ are the minimum and maximum symbol in cR.
8: Insert waiting states in cR into $wqueue$ and indexes ranging in [$curSize$, $curSize + |CR| - 1$] $pqueue$;
9: $curSize = curSize + |cR|$;
10: if $curSize \geq sizeLimit$, $SetMiss(pqueue, wqueue)$, END;
11: **else**
12: Set the hot buffer's $curIndex$ element based on the existing hot spot;

Hot buffer construction can be done online or offline. The tradeoff is between adaptivity and overhead. Online methods may adapt to the locality patterns well, but they come with a runtime overhead. Furthermore, adding one counter per transition increases the size of the automaton. Therefore, we used an offline method to identify the hot spots and to construct the hot buffer in our current implementation.

4.3 Model and Hot Buffer

Having presented both the cache miss estimation model and the hot buffer technique, next we show how to use the model to estimate the effectiveness of a hot buffer under

a certain workload and how to determine the threshold for selecting hot spots.

Given a certain workload (a sequence of documents), we define the effectiveness of the hot buffer using $speedup$. $speedup$ is the ratio of 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:

$$speedup = \frac{TM}{BM} \quad (3)$$

TM is estimated using our cache miss model. BM is estimated as follows:

$$BM = hbsize + tc \cdot m_{buf} \cdot (1 + r) \cdot F \quad (4)$$

In this equation, $hbsize$ is the size of the hot buffer in terms of the number of cache lines, tc is the total number of state transitions of the workload, m_{buf} is the hot buffer miss rate (not the cache miss rate), r is still the ratio of the conflict misses to the compulsory misses, and F is the footprint of a state transition in the original automaton.

This equation estimates the cache misses with the hot buffer in two parts. The first part, $hbsize$, includes the compulsory cache misses of those state transitions that are hot buffer hits. As the hot buffer is contiguous and is smaller than the cache size, we ignore the capacity and conflict misses of a hot buffer hit. The second part, $tc \cdot m_{buf} \cdot (1 + r) \cdot F$, is the estimated compulsory and conflict misses of a hot buffer miss. $m_{buf} = 1 - h_{buf}$, where h_{buf} is estimated as the ratio between the sum of the counter values of the hot spots and tc. Note that we ignore the capacity cache misses of a hot buffer miss transition.

Given a threshold, t, we can obtain $hbsize$ and m_{buf} using a straightforward algorithm similar to Algorithm 2. Then, we can compute the $speedup$ using Equations 3 and 4. If $speedup > 1$, the hot buffer can improve the cache performance to the $speedup$ scale. Otherwise, the filtering should be done on the original automaton without the hot buffer.

Finally, we give a simple iterative algorithm to find the threshold, t, for the maximum $speedup$. We define the threshold, t, to be an integer in the range of $[x, y]$, where x is the minimum counter value (if the minimum is 0, we set $x = 1$) and y is the maximum counter value in the warmup engine. We vary t from x to y and obtain $(y-x+1)$ $speedup$ values and find the threshold that results in the maximum speedup.

5 Experimental Evaluation

In this section, we first evaluate the accuracy of our cache performance model and then study the effectiveness of the hot buffer.

5.1 Setup and Methodology

All of our experiments were conducted on a 1GHz Pentium III machine with 512MB memory running Red Hat Linux 9.0 . 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 was 64 bytes. Our XML filtering system was written in C++ and was compiled using g++ 3.2.2-5 with optimization flags (O3, foptimize-sibling-calls and finline-functions). The filtering experiments were always memory-resident and the memory usage never exceeded 80%.

We used a hardware profiling tool, PAPI [15], to count cache misses and CPU execution cycles. Table 3 lists the main performance metrics used in our experiments.

Table 3. Performance Metrics

Metrics	Description
#L1DCM	Number of L1 data cache misses
#L2DCM	Number of L2 data cache misses
#TOTCYC	Total running time (in CPU cycles)

To evaluate the performance of a filtering engine comprehensively, there are quite a few parameters to be varied, e.g., the number of queries, the number of documents, and the characteristics of the queries and the documents. Furthermore, for cache performance evaluation, we need to consider the working set size as well as the locality in the access pattern. We believe that designing microbenchmarks to evaluate the cache performance of automaton-based XML filtering is an interesting direction for future work; in this paper, we used a set of simple but representative workloads as a start.

To simplify the query workload setup, we fixed the XPath-DFA to have 16 symbols and to be in the shape of a complete tree (states as nodes and transitions as edges) with the depth of six. The resulting number of states in the automaton was around 16 million $(1+16+16^2+...+16^6)$. This automaton represents millions of simple XPath queries over the 16 XML elements with the maximum depth of 6 location steps. These parameter values are comparable with those in previous work [8, 10]. We used the matrix implementation for the DFA. Since each state in our filtering system was four bytes, the DFA was memory-resident but not cache-resident.

For the document workloads, we varied three parameters: the document size (in the number of elements), the document working set size (in the number of cache lines), and the number of documents in a sequence. The elements in a document were generated from the 16 symbols in the XPath-DFA and the depth of a document was set randomly between 1 and 6.

Specifically, we generated the following two document sets:

(1) 40 XML documents of various sizes for intra-round modeling validation. Each document k ($1 \leq k \leq 40$) contained about ($40 \cdot k + 1$) XML elements with a working set size around ($32 \cdot k$). The document of the largest working set size (32×40=1280 cache lines) was much smaller than the L2 cache size (512KB/64B = 8092 cache lines).

(2) 80 sequences of XML documents for inter-round modeling validation and the hot buffer. Each sequence contained 40 documents of the same size. A document in sequence s contained ($5 \cdot s + 1$) elements with a working set size of around ($4 \cdot s$). Each of sequences 40 to 80 was expected to cause capacity misses.

Figure 1 in the Introduction shows the time breakdown of filtering sequences 10, 40, and 70. The numbers of misses were obtained from PAPI and the miss penalty of the L1 and L2 caches were set to 10 cycles and 100 cycles, respectively, according to reference manuals.

5.2 Model Validation

We evaluated the accuracy of our model for both intra-round filtering and inter-round filtering. The accuracy is defined as follows. $accuracy = \frac{min(measurement, estimation)}{max(measurement, estimation)}$.

Intra-round Modeling Validation. In intra-round filtering, we restarted the filtering engine with each document in the first document set. For each document, we first measured the total number of cache misses. Then, we copied the states accessed by the document to a contiguous area and measured the number of cache misses when accessing these states sequentially. This number is a good approximation of the number of compulsory misses in filtering this document, because the number of conflict misses is small with sequential access and the number of capacity misses is zero.

Figure 7 shows both the measured and the estimated numbers of compulsory misses in intra-round filtering for each document. The average accuracy on these 40 documents was 90.6%. The estimation was always lower than the measurement. Possible reasons include some simplifying assumptions that we used in our modeling (e.g., perfect reuse).

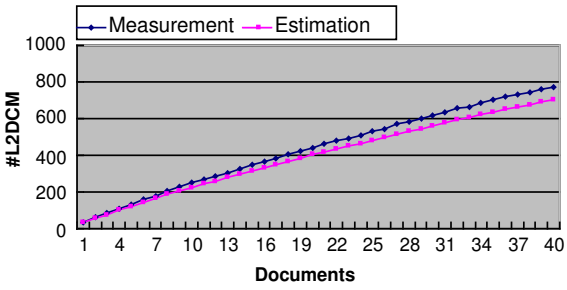

Figure 7. Intra-round Modeling Validation

We calculated the number of conflict misses in filtering a single document by subtracting the number of compulsory misses from the total number of misses, as the number of capacity misses was zero. We noted the ratio between the number of conflict misses and the number of compulsory misses was stable (mean: 0.84, stdev: 0.08) for all documents. We used the mean ratio in the inter-round modeling to estimate the number of conflict misses from that of compulsory misses.

Inter-round Modeling Validation. We experimented with inter-round filtering using the second document set. For each sequence, we restarted the filtering engine after filtering the entire sequence. We used the mean ratio of conflict misses to compulsory misses obtained from the intra-round filtering experiments to estimate the conflict misses in this experiment.

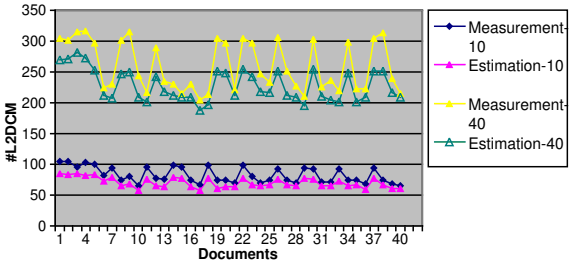

Figure 8. Inter-round Modeling Validation

Figure 8 shows the measured and the estimated numbers of L2 data cache misses in filtering sequences 10 and 40. In our model, L2 cache replacement did not occur in filtering sequence 10 but it did in filtering sequence 40. The average accuracy of our model for each sequence was 85.5% and 88.5%, respectively. Note that the inflection points in the curves were due to the difference in data reuse of each round from its previous round. The estimation was also consistently lower than the measurement. Results from other sequences of documents in the second document set were similar to those shown in Figure 8.

5.3 Cache Conscious Automata

After validating our analytical model for XML filtering, we evaluated the effectiveness of the hot buffer using sequences 10, 40, and 70 of the second document set. We chose these three sequences to examine the effects of both compulsory misses and capacity misses. We filtered each sequence five times (five runs), each without and with the hot buffer. In the construction of the hot buffer for each sequence, we warmed up the engine by filtering the sequence once and performed our speedup estimation to obtain the threshold for selecting hot spots. The size limit on the hot buffer was set to be 4096 cache lines, which is one half of the L2 cache size. The resulting threshold values of the hot buffers for sequences 10, 40, and 70 were all 1.

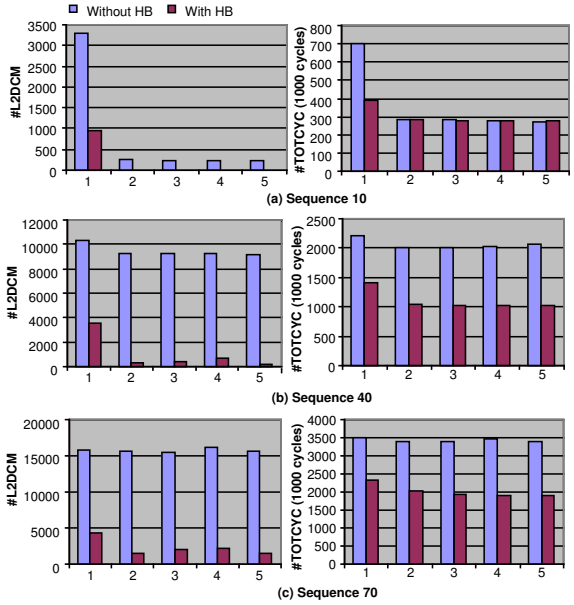

Figure 9. Inter-round Filtering: Left: L2 Cache Performance; Right: Overall Performance

The comparison results on the L2 data cache misses and the overall performance are shown on the left and right of Figure 9, respectively. "With HB" means that the number was measured when filtering using the hot buffer and "Without HB" is the number without the hot buffer. The hot buffer improved the cache and overall performance of inter-round filtering sequences 10, 40, and 70, with an average improvement of 75.0%, 89.2%, and 85.4%, respectively, on the cache performance and 16.7%, 46.8%, and 40.9%, respectively, on the overall performance.

In our experiments, we observed that sequence 10 had a working set smaller than the cache capacity (no cache replacement occurred in our model) and that sequences 40 and 70 had a working set larger than the cache capacity. Moreover, the hot buffer sizes of sequences 10 and 40 were smaller than the limit (642 and 2292 cache lines, respectively), whereas the hot buffer size of sequence 70 reached the limit of 4096 cache lines. As the threshold for the hot spots was one for each sequence, the hot buffers of sequences 10 and 40 contained all referenced transitions but that of sequence 70 contained only part of all referenced transitions (due to the hot buffer size limit).

Consequently, the three sequences represent three different cases: (a) the working set of the sequence is cache-resident (sequence 10); (b) the working set of the sequence is not cache-resident, but the hot buffer contains all states that are referenced by the sequence (sequence 40); and (c) the working set of the sequence is not cache-resident, and the hot buffer contains only part of all states that are referenced by the sequence (sequence 70).

With the categorization of these three cases, we analyze the results in Figure 9 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 documents becomes L2 cache-resident. The small number of cache misses in the later runs mainly includes conflict misses without the hot buffer. 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 in case (b) suffers from cache thrashing without the hot buffer. 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). Similar to case (b), the filtering in case (c) suffers from cache thrashing without the hot buffer. The thrashing effect is more severe in this case due to the increased working set size. The hot buffer reduces the cache thrashing significantly, but there are still a small number of cache misses due to the transition misses when filtering with the hot buffer.

Finally, for all three cases, the hot buffer improves cache misses (mainly compulsory misses) in the first run of the sequence to a large extent. After the first run, the number of cache misses becomes stable, both with the hot buffer and without. The overall performance follows the cache performance consistently.

In addition, we performed similar experiments varying the number of documents in a sequence and using some other DTD-generated document sets to evaluate the hot buffer technique. The results were similar.

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. [16]. Our work follows this direction but models the cost of XML filtering specifically. Successful research has been reported on reducing cache misses through specialized data structures, such as the CSS-trees used in decision support systems [19]. Several data layout techniques, either static [19] or adaptive [11, 20], have also been proposed for improving cache performance. Typical cache-conscious techniques including blocking [21], data partitioning [21], loop fusion [21], data clustering [21], prefetching [5, 6] and buffering [24] 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

non-traditional 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 [14]. Watson proposed general guidelines for cache-conscious automata, such as reorganizing the automata according to the access patterns and popularity information [22]. Compression techniques for automata [13] 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. Finally, we used the matrix-based automata implementation. This layout is related to nonlinear array layouts [4] in general.

Automaton-based XML filtering has emerged as a fruitful research area. XFilter [2] uses one NFA for each path query and uses list-balancing to speed up the processing. Both the YFilter [8, 9] and the XTrie [3] support path sharing and convert large numbers of XPath queries into a single NFA. XFilter, YFilter and XTrie use various indexing and optimization techniques to further improve the throughput. The lazy DFA [10] attempts to perform the subset construction only when needed in order to improve scalability. Our model and technique are applicable to these existing engines for further performance improvement.

7 Conclusions

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.

Acknowledgements

The authors are grateful to Derick Wood for his advice on automata theories and to Yi Wang for his discussions. The work of Byron Choi was done in part while he was visiting HKUST and was supported by grant HKUST6197/01E. The authors thank the reviewers for their insightful suggestions, all of which improved this work.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. VLDB, 2001.

[2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. VLDB, 2000.

[3] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In VLDB Journal, Special Issue on XML, Volume 11, Issue 4, 2002.

[4] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. 13th International Conference on Super-Computing, 1999.

[5] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. ICDE, 2004.

[6] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-trees: Optimizing both cache and disk performance. SIGMOD, 2001.

[7] J. Clark and S. DeRose. XML path language (XPath) - version 1.0. http://www.w3.org/TR/xpath.

[8] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. TODS, December 2003.

[9] Y. Diao, P. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. ICDE, 2002.

[10] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. ICDT, 2002.

[11] R. A. Hankins and J. M. Patel. Data morphing: An adaptive and cache-conscious storage technique. VLDB, 2002.

[12] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. IEEE Transactions on Computers 38(12):1612-1630, 1989.

[13] G. A. Kiraz. Compressed storage of sparse finite-state transducers. Workshop on Implementing Automata, 1999.

[14] N. Klarlund and T. Rauhe. BDD algorithms and cache misses. Technical report, BRICS Report Series RS-96-5, University of Aarhus, 1996.

[15] I. C. Laboratory. PAPI: Performance application programming interface. http://icl.cs.utk.edu/projects/papi/.

[16] S. Manegold, P. Boncz, and M. Kersten. Generic database cost models for hierarchical memory systems. VLDB, 2002.

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

[18] T. S. P. Organization. Sax: Simple API for XML. http://www.saxproject.org.

[19] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. VLDB, 1999.

[20] J. Rao and K. A. Ross. Making B+ trees cache conscious in main memory. SIGMOD, 2000.

[21] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. VLDB, 1994.

[22] B. W. Watson. Practical optimizations for automata. Second International Workshop on Implementating Automata, 1997.

[23] D. Wood. *Theory of Computation*. John Wiley, New York, NY, 1987.

[24] J. Zhou and K. A. Ross. Buffering access to memory-resident index structure. VLDB, 2003.