Parallel XML Matching Algorithm for Publish/Subscribe Systems

Susana Cardoso Ferreira
Instituto Superior Técnico,
Universidade de Lisboa
susana.ferreira@tecnico.ulisboa.pt

***Abstract*—The Publish/Subscribe communication model is the most adequate distributed paradigm for complex event processing and large-scale dissemination of information to a variety of users. The continuous growing interest in XML as the standard language for information representation and exchange over the internet increased the importance of XML-based publish/subscribe systems. The core functionality of Publish/Subscribe systems lies in the matching algorithm that is responsible for, whenever new events are published, determine matched subscriptions. In real world applications with a huge amount of stored subscriptions and continuously arriving events, the sequential matching algorithm can easily become a bottleneck impacting the overall performance of the system. A way to achieve a scalable system, while maintaining high performance, is by exploiting chip multiprocessors architectures already present in today's computers. In this paper, we propose and implement three parallel event processing techniques for DeltaFilter, a highly efficient sequential XML matching algorithm. We perform experimental evaluations on a 48 core machine to study the scalability and performance of the proposed techniques with a varying number of threads in different application scenarios. The results show performance gains of 20 times more events processed per second and a reduction of almost 74% on the matching time per event when in the presence of 48 threads.**

Keywords. Selective Dissemination of Information, Publish/Subscribe, XML Matching Algorithms, DeltaFilter, Multicore Processors, Parallel Event Processing

I. Introduction

Everyday new information is made available in the internet and everyday more users wish to be notified when certain information is published. This led to the concept of *Selective Dissemination of Information* (SDI) which guarantees that users only receive information in which they have expressed their interests in. With new information being made available each passing day and users with high demands of information, SDI systems have to efficiently and quickly disseminate, at a large scale, huge amount of documents to enormous quantities of users. The best approach to obtain scalable SDI applications is by taking advantage of the full decoupling of the communication entities in time, space and synchronization, available in the *Publish/Subscribe* paradigm.

Publish/Subscribe (also known as pub/sub) is a distributed paradigm composed by an intermediary agent responsible for the indirect communication between two types of entities, *Publishers* and *Subscribers*. Publishers are data sources responsible for producing new information, defined in this context as *events*, whereas Subscribers correspond to users whose interests are expressed over the form of *subscriptions*.

The core functionality of Publish/Subscribe systems lies in the matching algorithm that is responsible for, whenever new events are published, determine matched subscriptions. Users whose subscriptions have been matched will be further notified by receiving the published event. In the presence of a large number of subscriptions and a high rate of events, the matching algorithm can easily become a bottleneck, so it is imperative for it to be as scalable and efficient as possible. The way subscriptions and events are represented is a relevant factor that greatly influences the complexity of the matching algorithm. This representation is defined in this context as *subscription model* and, as pub/sub systems evolved over time, this model also adapted accordingly to keep up with system and user needs.

The original subscription model is *topic-based*, or *subject-based*, where events are classified by topics and subscriptions are expressed as a collection of topics of interests (e.g., [28]). Next, *content-based*, or *property-based* [27], pub/sub systems emerged as an evolution of topic-based with more expressive and flexible subscriptions. Here, instead of using some predefined external criterion, such as topics, events are classified according to their own properties and subscriptions as constraints over these properties [10]. Several subscriptions schemes for content-based pub/sub have been proposed in the literature, but one that stands out is the XML content-based (e.g., WebFilter [23], ONYX [9]), where events are represented as *eXtensible Markup Language*[1] (XML) documents and subscriptions as *XML Path Language*[2] (XPath) expressions. This combination results in a good trade-off between the expressiveness of the subscription language and the required complexity for the matching algorithm.

Different XML matching algorithms have been proposed in the literature, from automata solutions where subscriptions are stored in state machines and transitions are triggered by SAX parser events [1] [7], to solutions that convert simple content-based to XML-based algorithms [23] [15] [29]. In the latter, XPath expressions are converted to sets of boolean predicates, whereas events are converted to sets of predicates that, additionally to expressing the actual content, establish

[1] http://www.w3.org/XML/
[2] http://www.w3.org/TR/xpath/

how the content is organized in the event, resulting in complex and numerous predicates that limit performance.

As concluded in the study presented in [20] relatively to XML-based systems, highly expressive and selective subscriptions require complex, and expensive, matching and routing algorithms. As such, the performance and scalability of either centralized or distributed pub/sub system is strongly affect by the cost of the matching algorithm. Recently, more attention has been given in implementing efficient techniques for constraining the complexity of the XML matching algorithm. For instance, there have been numerous contributions to improve previous matching algorithms, namely more efficient data structures [8], better index structures [18], better clustering criteria [29], lower memory usage [32] and cache-conscious indexes [19]. Despite these contributions being able to increase system throughput and reduce event matching time, they only focus on sequential computation and, thus, do not exploit the processing power of multi-core processors already present in today's computers.

With the exponential growing in computers performance in the past decades, mainly due to the transistor scaling which allows for more transistors to be placed in a single chip, chip manufacturers have found a stagnation point where chips are reaching physical limits in processing speed. Consequently, single thread performance growth will be slowing to a rate much lower than it has been in the history of computing. For this reason, instead of looking to improve a single core's performance, chip manufacturers have turned their attention to optimizing the number of cores on chips and improve scaling of those cores [16]. In fact, today most commercialized computing architectures already encompass Chip-level Multiprocessors (CMP), i.e., multi-core processors.

Event processing of XML matching engines is increasingly requiring more computing power than a sequential system can offer, so a viable solution consists in converting it to a parallel event processing one, where instead of a single thread processing all the events that arrive to the system, several threads cooperate among themselves to process incoming events. Lately, this field has received more attention with some efficient parallel engines proposed for content-based systems (e.g., [12], [25]) but, still with a long path ahead concerning parallel XML matching. The conversion of a sequential XML matching algorithm to a parallel one is no simple task, since several threads will be working concurrently to process complex events and will need synchronization mechanisms in order to preserve the integrity of shared information.

In this paper, we propose a parallel matching engine by adapting DeltaFilter [19], a highly efficient and cache aware XML matching algorithm, to a multi-processor environment for performance enhancement. Based on the nomenclature presented by Farroukh *et al.* [12] for content-based pub/sub systems, three approaches concerning parallel XML matching are explored:

- **Multiple Event Independent Processing (ME-IP)**: the purpose of this approach is to increase system throughput, i.e., number of events processed per second, by independently processing several events in parallel. In that sense, this approach is suitable for scenarios where a high rate of incoming events is present.
- **Single Event Collaborative Processing (SE-CP)**: at a different level, the purpose of this approach is to reduce event matching time by processing a single event in parallel. In that sense, this approach is suitable for scenarios where single events must be processed faster.
- **Multiple Event Collaborative Processing (ME-CP)**: this approach constitutes a hybrid of the previous two approaches. Therefore, its purpose is to simultaneously increment system throughput while reducing event matching time, making this approach suitable for constant matching times in scenarios where large numbers of events overwhelm the system.

II. Related Work

The advent of XML as the *lingua franca* of information integration and exchange on the web, led developers to adopt XML as data representation for Service-Oriented Architectures (SOA), Web Services (WS), content-based message routing [30], XML query processor databases [4] [26], Selective Dissemination of Information (SDI) applications [1] [6] [19], among several others.

In XML matching engines, XPath queries are indexed in order to quickly determine those that are matched by an incoming XML document. Depending on the way subscriptions are stored and event are processed, XML matching algorithms can be categorized in three main classes: (1) *automaton-based*, where subscriptions are stored as state machines and parser events trigger transitions between states; (2) *index-based*, where subscriptions exploit high performance indexes; and (3) *sequence-based* focus on processing twig patterns against incoming XML documents.

In the first category, automaton-based, matching algorithms are based on the idea that XPath expressions can easily be mapped to Moore Machines, where states correspond to location steps and transitions are triggered by the nodes tests (element nodes or '$*$') of these location steps. The processing of incoming XML documents follows an event-driven execution resultant from the events generated by the SAX parser.

The first solution of this category was XFilter [1] that converts each XPath subscription into an unique Finite State Machine (FSM). XFilter takes advantage of a novel index structure that, for each SAX parser event triggered, is able to locate and reduce the number of FSMs needed for evaluation. A subscription is considered a match when its correspondent FSM reaches an accepting state. Because XFilter creates a FSM per subscription, similarities between expressions are stored and processed independently, impacting both space and time complexities. To overcome this problem, an evolution of XFilter was proposed in YFilter [7] where common prefixes are represented and processed only once in a global Nondeterministic Finite Automaton (NFA).

YFilter stores all subscriptions in a single NFA, in order to share the storage and the processing of prefixes between sub-

scriptions, resulting in a performance improvement of structure matching when compared to XFilter. Although scalable with the number of subscriptions, the performance of YFilter's NFA decreases for deep XML documents, since the deeper in the XML tree the higher the number of active states and transitions to evaluate in each parsing step. A straightforward solution to avoid this overload of active states consists in converting the NFA into an equivalent DFA. However, this conversion could theoretically result in scalability problems due to an exponential blow-up in the number of states [6]. In [14] the authors explain that this explosion can be reduced in several scenarios by placing restriction on document types, through DTDs and supported queries, constructing the DFA at runtime in a lazy fashion. Although faster and more memory-efficient than the NFA, since only a small subset of the DFA is constructed in runtime, it is still significantly memory heavy [13].

In [21] YFilter is adapted to a distributed environment on top of Chord [31], a distributed lookup protocol for peer-to-peer applications. In order to achieve an efficient structural XML matching this solution exploits *Distributed Hash Table* (DHTs), where the NFA is statically decomposed in sub-machines processed by different peers. Similarly, but in a centralized environment, in [33] the structural XML matching of a single event is performed in parallel by decomposing the NFA in independent and concurrent sub-machines to be further processed by available threads.

In the second category, index-based, matching algorithms exploit index structures for a more efficient filtering by sharing the processing of structure matching. XTrie [5] exploits trie structures by treating the matching of XPath expressions as a form of string matching. Therefore, XPath expressions are decomposed in substrings and mapped to keys in a prefix tree. By removing the redundant processing of common prefixes between subscriptions the matching performance is increased. Despite having a throughput up to 4 times higher than YFilter, it only supports a small subset for structure matching through parent-child relationships.

The representative algorithm of this category is Index-Filter [3] which builds indexes over XML elements in order to avoid the processing of parts of the XML documents that are guaranteed to not match. It is specially efficient when dealing with big XML documents and a relatively small number of subscriptions.

Predicate-based matching algorithms are a specialization of index-based that heavily rely on special tailored data structures for high performance matching engines, by extending the idea of predicate codifications (e.g., [24], [11], [23], [2], [15], [29], [19]). This type of algorithms date back to content-based publish/subscribe (e.g., [24], [11]). *Le Subscribe* [24], proposes a clustering algorithm with two possible implementations: an unidimensional, where a single access predicate is used per cluster family, or a multidimensional, where several access predicates are used per cluster family. Fabret *et al.* introduced in [11] a highly efficient and scalable clustering algorithm that exploits space efficient and cache-aware structures.

Farroukh *et al.* proposed in [12] three parallel techniques for the event processing of the matching engine presented in [11]: *Multiple Event Independent Processing* (ME-IP), where threads process events separately in parallel in order to increase system throughput; *Single Event Collaborative Processing* (SE-CP), where threads cooperate in the processing of a single event in order for a single event to be processed faster; *Multiple Event Collaborative Processing* (ME-CP), a hybrid combination of the previous two techniques, where threads are grouped into groups and a single event is assigned to each group.

The fact that these content-based matching algorithms have proven to be highly scalable and efficient, in combination with the increasing popularity of XML-based pub/sub systems led researchers to take special attention in the adaptation of content-based matching algorithms techniques to XML-based. The first system that focused on this adaptation was WebFilter [23] that translates XML event paths to a set of *(attribute, value)* pairs to be further processed using techniques employed in *Le Subscribe* [24]. In [15] and, latter on GPX-Matcher [29], a solution with a complete XPath subset is proposed by adapting the clustering algorithm presented in [11]. In these solutions, XPath subscriptions are translated to a set of *(attribute, operator, value)* tuples, whereas each path of an XML event is converted to a set of *(attribute, value)* pairs.

Additionally to value-based predicates as in content-based matching algorithms, in XML-based the structure of XML documents is also taken into account. Consequently, the translated predicates must represent structural relationships, which results in very complex and large events, impacting both time and space complexity. Moreover, XPath '$*$' and '$//$' operators, which introduce non-determinism and, XML recursion which aggravates the false positive detections, results in predicates more difficult to evaluate and index. Despite achieving notable results, these solutions were not able to achieve an effective conversion without significant overhead, as the adaption results in numerous predicates complex to index resulting in slower indexes and weak cache performance.

In the final category, sequence-based, the first algorithm of this type was FiST [17] focused on the processing of twig patterns, i.e., nested path expressions. Contrarily to the query decomposition technique used by previous algorithms (e.g. YFilter [7], XTrie [5]), FiST employs a holistic processing of twig patterns. FiST encodes both events and subscriptions as unique Prüfer sequences which are then indexed to hash structures. Latter approaches take advantage of Prüfer sequence in combination with an early pruning technique to reduce the subscription search space (e.g., BoXFilter [22]), or by employing a holistic processing of twig patterns (e.g., pFiST [18]).

III. DeltaFilter

DeltaFilter is a predicate-based matching algorithm for very fast XML-based publish/subscribe that employs techniques proven efficient in content-based matching algorithms, namely the indexes proposed in *Le Subscribe* [24] and the efficient

clustering of [11]. One of the most important characteristics of DeltaFilter is its cache-conscious behavior in subscription storage to obtain better performance when evaluating subscriptions for the incoming events. As XML documents are constituted by an arranged set of complete root-leaf paths, DeltaFilter applies a two-phase algorithm to each of these paths, defined in this context as *internal events*.

In a global perspective the main components of a matching engine consists in the storage of users' subscriptions and the processing of events published by publishers. With that in mind, the architecture of DeltaFilter can be seen as the composition of two main modules: *Subscription Storage* and *Event Processing*.

A. Subscription Storage

The way subscriptions are stored greatly influences the performance of the matching algorithm, as it dictates how this information will be accessed at the event processing stage. In order to mitigate the complexity problems of converting an XPath expression into a conjunction of predicates addressing the names of XML nodes and the relationship between them, in DeltaFilter an XPath expression is converted to simple binary operations represented as sets of *operators* over *position records*.

A position record is responsible for storing at which depth an unique tag (pos_{tag}), an unique root tag ($root_{tag}$) or an unique attribute ($pos_{tag[@attribute\ operator\ value]}$) appeared in an internal event of the current XML document. An efficient and fast implementation for this dynamic structure is to represent each position record as a 64-bit number where the n^{th} bit is set to 1 if at depth n the correspondent tag or attribute, occurred; 0 otherwise. Position records are uniquely identified in the system by a position record identifier (p_{id}) and stored in a global *position record vector*, where position i corresponds to the position record with identifier p_i.

Three main indexes are used for a fast retrieval of a position record's identifier to which an XML tag or attribute corresponds, one for each position record type. The root and tag position record indexes are implemented as simple hash tables whose key is the tag name and value the position record's identifier.

For the attribute position records a simple hash table index is not possible, since an attribute is identified by a 4-tuple *(tag, attribute, operator, value)*. Therefore, a multi-level index I is employed that maps the p_{id}s of attribute position records with the corresponding attribute elements. DeltaFilter supports value-based predicates of exact (operator $=$) and range (operators $>$, $\geq$, $<$ and $\leq$) modes therefore, for an efficient processing when evaluating attributes, index I consists of five 2-dimensional low-level indexes I_{op}, one for each operator. In this low-level indexes, the $(tag, attribute)$ pair constitutes a key for an hash table of pairs $(value, p_{id})$ and $value$ is used as key to lookup the respective p_{id}'s. The implementation of the low-level indexes depends on the type of the operator of the indexed attribute position records: hash tables for exact mode and an ordered array for the range mode case.

Subscriptions are also uniquely identified in the system and indexed in *cache-friendly clusters*, where the association between position records and a subscription is established. In large scale systems there is a high probability of commonalities among user interests, which could result in redundant processing [8]. As such, for an efficient processing, it is important for position records to be evaluated only once throughout the processing of an internal event. An efficient solution that guarantees this single-evaluation is the grouping of clusters by a special predicate that is able to reduce *significantly* the search space of the subscriptions that have to be evaluated for a given internal event. In DeltaFilter, this predicate is called *access position record* and, all the subscriptions grouped in a cluster contain the cluster's access position record as position record.

In XML matching engines, where events are XML documents, the choice for the best access position records is highly associated with the fact that XML is a structured language. Furthermore, the deeper a predicate in an internal event, the more selective the predicate, as the probability of it belonging to other internal events is minor. With this in mind, the authors in DeltaFilter [19] proposed a simplification where the best solution lies in selecting the last position record of a subscription as its access position record.

Clusters store subscriptions with the same size, i.e., with the same number of position records and with the same access position record. Each cluster is implemented as a matrix, where subscriptions are stored column-wise, i.e., column i contains all the p_{id}'s of a subscription with the last row containing this subscription's id. Since several subscriptions may have the same access position record but different sizes, clusters are grouped in *cluster families* which can be defined as a set of clusters of different sizes with the same access position record.

A *Cluster Vector* structure was implemented in order to index access position records to its associated cluster families. This way, position i of the cluster vector contains the cluster family with access position record identifier p_i. This way, when evaluating an internal event, this vector is traversed and only the subscriptions of clusters with size equal or smaller than the path found and whose access position records are matched are evaluated.

Operators express how different position records are related to represent a subscription. Since position records are binary numbers, the logic implementation for operators is *binary operations*. In order to reduce time and space complexity, operators are stored implicitly in clusters and in the position record vector. Since DeltaFilter supports three XPath axis (parent, ancestor and attribute) three operators can take place:

- *parent* $pos_a \bigoplus pos_b := (pos_a \ll 1)\,\&\,pos_b$: returns a 64-bit number with the bits corresponding to the positions in the internal event where tag b immediately follows tag a set to 1. This is the default operator, where no flag is set.
- *ancestor* $pos_a \bigotimes pos_b := mask(pos_a)\ \&\ pos_b$: returns a 64-bit number with the bits where b is at a higher level

of depth than a set to 1. Method $mask()$ returns a 64-bit number with all bits set to 0 from the least significant bit to the first bit set to 1 and, all following bits set to 1. For this operator, the position record identifier p_b is stored in the cluster with a negative identifier, i.e., $-p_b$.

- *attribute* $pos_a \bigodot pos_{a@c} := pos_a \ \& \ pos_{a[@c \ op \ val]}$: returns a 64-bit number with all bits where c is at the same level of depth as a set to 1. All position records concerning attributes matched by the event have the leftmost bit set to 1.

For each candidate subscription, when the combination of the operators over its position records is different than 0, it means that the subscription was matched at a certain level of depth.

B. Event Processing

When an event first arrives to DeltaFilter it is placed in a queue to be further processed by an individual entity in charge of executing the matching algorithm, *Filter*. Filter is the entity responsible for parsing XML documents, updating position records and evaluating stored subscriptions. Since subscriptions are expressed as XPath expressions, all the internal events of the XML document have to be evaluated in order to obtain subscriptions relative to all possible paths. As a result, Filter processes internal events independently meaning that, for each internal event, two phases take place:

1) **Position Record Update**: throughout the parsing of an internal event the position records of every tag parsed, if existing, must be updated according to the depth at which they were discovered and added to the list of satisfied position records.
2) **Subscription Evaluation**: when an internal event is discovered, the satisfied position records obtained in the 1^{st} phase constitute the access position records that need to be evaluated. All cluster families whose access position record is not verified can be ignored.

After the 2^{nd} phase of an internal event is finished, all the position records corresponding to the closed tag must be reseted for the current depth, i.e., reset the $depth^{th}$ bit to 0. The execution of the matching algorithm continues for the remaining internal events and after parsing is completed, all matched subscriptions are outputted to the pub/sub system.

IV. Parallel XML Matching Algorithm

Parallel computing is considered one of the most difficult topics in computer science, not only concerning work decomposition and distribution, but also the management of shared resources. Withal, the complexity of converting a sequential algorithm to a parallel one, sometimes forces programmers to redesign algorithms in order to maximize the operations that can be processed in parallel and exploit the maximal processing power provided by the hardware. One technique commonly used to convert sequential algorithms in parallel ones is the *divide-and-conquer* technique, where a single time-consuming task is decomposed into several smaller and independent subtasks that can be executed in parallel.

In the sequential version of DeltaFilter's Event Processing presented in the previous section, a single thread is responsible for processing all the arriving events, one at a time. Depending on the desired task granularity, different levels of parallelism can be applied to the XML matching problem of predicate-based algorithms. The first level of parallelism consists in the decomposition of the arriving events into several event subtasks. According to the terminology presented by Farroukh *et al.* in [12] this decomposition can be defined as *Multiple Event Independent Processing* (ME-IP) since each thread will be responsible for processing a subset of the incoming events independently.

From the second level on, the parallelism is at the scope of an event, because threads collaborate with one another in the processing of a single event. In [12] this technique is referred to as *Single Event Collaborative Processing* (SE-CP). More specifically, the second level of parallelism consists in decomposing events in internal event subtasks, where each thread evaluates internal events independently. Following this course, internal events can be decomposed into satisfied position record subtasks where each thread is assigned to a candidate cluster family. Similarly to the previous levels, a cluster family can further be decomposed into clusters subtasks, which can finally be decomposed into subscription subtasks[3]. Although all these parallelism levels can be considered, the presented paper only delves into the internal event evaluation subtasks.

A. Multiple Event Independent Processing

The purpose of the ME-IP approach is to improve system throughput, i.e., increase the number of events processed per second, by maximizing the quantity of parallel computations while minimizing thread interaction. In that sense, each filter thread contains an unique Filter structure to processes events, one at a time, independently of all other threads. When an event arrives to the publish/subscribe system, it is assigned with an unique identifier and is automatically added to an event queue in the matching engine, where filter threads are continuously checking its content for new events to process. In the case the queue is empty, when trying to remove an event, threads block switching to a $WAITING$ state, waiting for new events to be added to the queue. On the other hand, if new events are available for processing, threads remove a single event from the queue, process it against the stored subscriptions and output its matched subscriptions.

In an algorithmic perspective, the processing of a single event in this context is equivalent to the sequential algorithm presented in the previous section. As such, for each internal event, in the 1^{st} phase each thread updates its unique position record vector and satisfied position records according to the triggered SAX events and, when an internal event is discovered, the 2^{nd} phase takes place, using the satisfied position records computed in the 1^{st} phase to evaluate the candidate cluster families and obtain the matched subscriptions

[3]Decomposition can still go further on till instruction level. However, this level of parallelism is out of the scope of this work.

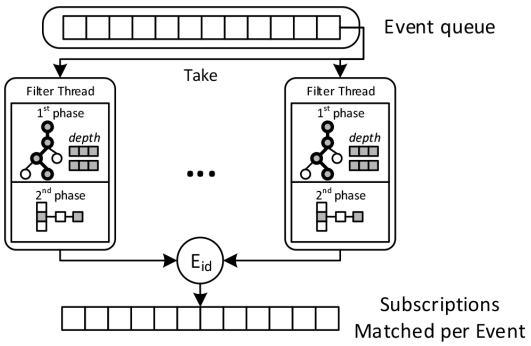


Fig. 1: Multiple Event Independent Processing (ME-IP) approach.

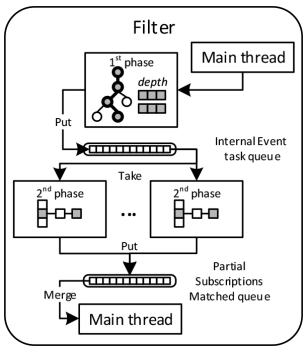


Fig. 2: Single Event Collaborative Processing (SE-CP) approach.

of the discovered internal event. When a filter thread finishes processing an event, the list of subscriptions matched is placed in a hash table indexed by the event's unique identifier, in order to be delivered to the subscription service, which will further notify interested users. Figure 1 illustrates the work flow of the ME-IP technique.

The idea behind this approach is simple as it only requires minimal synchronization associated with the operations of the global event queue. In conclusion, this approach is able to reduce the total matching time for a high rate of incoming events, while maintaining performance and improving scalability. Nevertheless, it is unable to reduce the matching time associated to a single event, since only one thread is responsible for each event, similar to what happens in the sequential case. In order to reduce matching time of an individual event, threads must cooperate in its processing. This cooperation constitutes the basic idea behind the SE-CP technique.

B. Single Event Collaborative Processing

As demonstrated by the authors in [19] the 1^{st} phase of the matching algorithm, is in fact fast and the real demanding computation lies in the 2^{nd} phase. With that in mind, the SE-CP approach parallelizes the 2^{nd} phase of the matching algorithm. In the internal event parallelization level, tasks correspond to the processing of internal events, i.e., this level is related to the XML structure of events.

The objective of the SE-CP approach is to reduce matching time of an event, by allocating threads to the collaborative processing of a single event. In this approach, an unique *main thread* containing a Filter structure is responsible for sequentially executing the 1^{st} phase of the matching algorithm and creating internal event level tasks. Additionally, a set of *worker threads* is responsible for executing the 2^{nd} phase of the matching algorithm, by processing the internal event level tasks.

The main thread starts the processing of an event by executing the 1^{st} phase of the matching algorithm and, when an internal event is discovered, instead of progressing to the 2^{nd} phase, it creates and adds internal event tasks into the task queue to be further processed by worker threads. Recall that, when an internal event is discovered, the current depth, position record vector and set of satisfied position records are in a state that defines the obtained internal event. Therefore, each task must maintain a reference of the current depth, current state of the position record vector and the current list of satisfied position records at which the corresponding internal event was discovered. Unfortunately, this bookkeeping is necessary as the main thread continues executing the 1^{st} phase of the remaining internal events and, consequently updating these structures, as worker threads execute the 2^{nd} phase of previously created tasks. The result of processing these tasks constitute partial sets of subscriptions matched by the current event.

While the main thread is performing the 1^{st} phase and creating new tasks, worker threads are continuously checking the task queue to process internal event tasks. This processing corresponds to the 2^{nd} phase of the matching algorithm so, worker threads are responsible for computing in parallel matched subscriptions of individual internal event tasks. Whenever a subscription is considered a match, it is stored in a local hash set of the thread to be further added to a queue containing the results of tasks. After finishing processing a task, worker threads add its set of subscriptions matched to the result queue. This procedure continues until no more tasks are available in the task queue and all worker threads are blocked in a $WAITING$ state, waiting for the main thread to add new tasks to the task queue.

When the main thread finishes parsing, i.e., all internal events have been identified and all tasks have been added to the task queue, it starts merging the partial results of the result queue into a single list that in the end will contain all the subscriptions matched by the current event. The reason why partial results are stored in hash sets is to minimize merge overheads resultant from duplicate matched subscriptions. Nevertheless, the merge operation is still, usually, faster than the 2^{nd} phase of the matching algorithm. With that in mind and the fact that the parsing is a considerably fast operation, in order to avoid big idle times for the main thread, if no results for merge are available it helps worker threads in the processing of the remaining tasks. Otherwise, if the

result queue is not empty, the main thread merges partial results to the set of final subscriptions matched. When all the partial subscriptions matched have been merged and all worker threads are in a waiting state, the main thread considers the current event as completed, outputs subscriptions matched and starts processing a new event. The presented procedure is depicted in figure 2.

This approach presents a problematic disadvantage regarding the internal event task balancing that hinders scalability when in the presence of a large number of threads. According to the study presented in [20], most XML documents found on the web are *narrow* in the sense that incorporate a small number of internal events. Taking that into consideration, depending on the number of internal events ie of an XML document and the number of threads t assigned to collaborate in the processing of the event, two main scenarios can take place:

- $ie < t$. In this scenario, $t - ie$ threads will not work throughout the matching of the event, which is essentially the same as using only ie threads.
- $ie \geq t$. In this case, most threads will process one or more internal events but, depending on the complexities of internal event tasks, some threads may get tasks more time-consuming than others, resulting in unbalanced workloads.

In this line of reasoning, two main variables influence load balancing: (1) if the number of internal events tasks per event is a multiple of the number of threads and, (2) the complexity of internal event tasks. Concerning the first variable, assuming that $ie = k \times t : k \in \mathbb{N}$, the ideal load balancing is only achieved if tasks have the same complexity. Otherwise, the matching time will always consider the time of the slower thread. Concerning the second variable, assuming that all internal event tasks have the same complexity, the ideal load balancing is only achieved if $ie = k \times t : k \in \mathbb{N}$. Otherwise, if for instance, $k \times t = ie + 1 : k \in \mathbb{N}$, the matching time considers the time of the thread obtaining the last internal event task.

In conclusion, the ideal case for this approach is when the number of internal event tasks is a multiple of the number of threads and internal event tasks have approximately the same complexity. Nonetheless, in most cases this parallelism level is unable to efficiently exploit parallel computations which limits scalability due, mainly, to the number and complexity of internal event tasks. An attempt to solve this problem is to combine the ME-IP and SE-CP techniques into a single one.

C. Multiple Event Collaborative Processing

The purpose of the ME-CP, or hybrid, technique is to combine the benefits of the previous two techniques, essentially improve system throughput (from ME-IP) and reduce matching time per event (from SE-CP). In that sense, this technique incorporates two parallelization levels, i.e., independent events are matched in parallel by groups of threads, and a single event is processed in parallel by an assigned group of threads.

As validated in section V-A, regarding the SE-CP technique, no significant improvements are achieved when employing more than 16 threads per event. As such, we impose a maximum limit of 16 threads per event in an attempt to avoid workers' idle times when processing single events.

A global structure $events$ maintains the set of all the events that arrive to the system and keeps record of each event's state, that can only be updated atomically by threads:

- $event.state = AVAILABLE$: when the event is already in the queue and its processing has not yet started;
- $event.state = PARSING$: when the event is currently being parsed and not all internal events have been discovered;
- $event.state = MATCHING$: when all the internal events have already been discovered but, the processing and merging of internal events is still in progress;
- $event.state = FINISHED$: when no more internal event tasks are available for distribution but, worker threads may still be processing internal events and the main thread can still be merging partial results. Note that this state is used to advice other threads that this event has no more available work and can, therefore, move forward to the next event. The event is completely finished when the main thread finishes merging and outputs the subscriptions matched.

As it happens in the ME-IP technique, each hybrid thread is associated to an unique *Filter* structure, in order to execute the 1^{st} phase of the matching algorithm of the events for which it is responsible. Furthermore, as a maximum limit of 16 threads per event is imposed, each event must bookkeep its number of threads.

To better understand the procedure of the hybrid approach consider the example of figure 3 and, for simplicity assume only 2 available threads. When a hybrid thread $ht1$

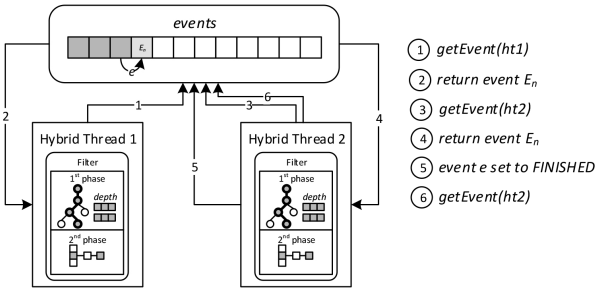


Fig. 3: ME-CP approach.

is available for work, the first step consists in requesting an event to the $events$ global structure by invoking the method $getEvent(ht1)$. In this method, $ht1$ traverses the $events$ structure through the thread variable e, that contains the position of its last event processed, till it finds an event whose state is either $AVAILABLE$, $PARSING$ or $MATCHING$. Event E_n with state $AVAILABLE$ is found, so $ht1$ atomically update the event's state to $PARSING$ through a Compare-And-Swap (CAS) operation. In this case, $ht1$ is

assigned as the main thread of E_n and atomically increments the number of threads associated to the event. Subsequently, $ht1$ starts executing the 1^{st} phase of the matching algorithm and creating internal event tasks.

In the meantime, hybrid thread $ht2$ requests to the $events$ structure an event by invoking $getEvent(ht2)$. Since event E_n is still currently being parsed, i.e., with state $PARSING$, $ht2$ performs an atomic conditional operation that only increments the number of threads associated to the event if this number is less than 16. In cases where the number of threads is equal to 16, the atomic conditional increment prevents threads from progressing and, threads continue searching for new events. However, since in this example only one thread is associated to E_n (thread $ht1$), $ht2$ is assigned to E_n as a worker thread and the number of threads assigned to E_n increments to 2. Afterward, $ht2$ removes E_n internal event tasks from the task queue, processes them and adds its partial results to the result queue of E_n.

When $ht1$ finishes parsing, meaning that all internal events have been discovered, it updates the event's state to $MATCHING$ and starts merging partial results. When no results are available for merge, $ht1$ helps worker thread $ht2$ in the processing of internal event tasks, as it happens in the SE-CP technique. When all internal event tasks have been processed, while thread $ht1$ finishes merging the last partial results, thread $ht2$ atomically updates E_n's state from $MATCHING$ to $FINISHED$ and requests a new event to the $events$ global structure. Since event E_n is already finished, the next event is considered and $ht2$ will constitute this new event's main thread.

In conclusion, the hybrid technique is able to minimize the internal event load balancing impact problem to a certain extent by being able to adjust the number of threads per event, according to event properties and available threads. Furthermore, it is able to efficiently integrate the two main techniques, gaining advantages concerning improved system throughput (ME-IP approach) and reduced matching time (SE-CP approach).

V. Experimental Results

We performed a set of experiments in order to evaluate performance and scalability of the different techniques under specific workload scenarios and to better clarify the advantages, as well as disadvantages, of each approach. The algorithms were implemented using Java 1.8 and, all experiments reported here were conducted on a hardware configuration comprising 48-cores AMD Opteron(tm) Processor 6168. This machine is composed by a 4-socket system, each one with 2 NUMA nodes of 6 single-threaded cores. Each core has 64KB data and instruction L1 caches and a 512KB L2 cache. Additionally, all 6 cores of the same NUMA node share an unified 5118KB L3 cache. The operating system is CentOS 64 bit with kernel version 2.6.32.

The execution of experiments is as follows: firstly, subscriptions are loaded to the system in order to build all data structures responsible for storing XPath subscriptions; secondly, a collection of XML documents to be processed is placed on a queue, as to simulate a scenario of streaming events; finally, the matching engine processes these events over the stored subscriptions and, for each event outputs its set of matched subscriptions.

The purpose of these experiments is to demonstrate how the parallel event processing techniques behave when scaling from a sequential system to a fully parallel system of 48 active threads. Thereby, the experiments were run using 1, 2, 4, 6, 8, 16, 32 and 48 threads. Each result presented in the experiments reflects an average of five runs.

The News Industry Text Format (NITF) DTD[4] was used to define the format and structure of both subscription and event workloads. This DTD is a popular format for XML news text interchange, used as a standard benchmark in many research studies (e.g. [21], [5], [6], [15], [29], [19]). The subscription workloads were generated using the XPath query generator released in the YFilter package[5] within the range of workload parameters expressed in table I. Because DeltaFilter shares the storage and processing of duplicate subscriptions, it becomes more interesting to run these experiments using unique subscriptions. Hence, an additional parameter that guarantees that no two subscriptions are identical is used. This way, when a subscription workload is presented with N subscriptions, N distinct subscriptions are used.

Parameter	Range	Description
N	$1.000.000 - 10.000.000$	Number of XPath subscriptions
D	10	Maximum depth of XPath subscriptions
$\Delta*$	0.2	Probability of '$*$' operator occurring at a location step
$\Delta//$	0.6	Probability of '$//$' operator being the operator at a location step
$\Delta@$	$0 - 0.6$	Probability of a location step to have at least one attribute

TABLE I: Workload parameters description and range values

A total of 5.000 XML documents was generated using IBM's XML generator tool and the workload parameters employed in [6], i.e., maximum depth ranging between 6 and 10, and the maximum number of times a recursive element can appear in a simple path is established as 3.

Based on measurements of previous works, system throughput and average matching time constitute the main performance metrics. The system throughput metric is concerned with the number of processed events per unit of time, whereas matching time is related to the time it takes for an event to be processed. The matching time of a single event includes removing the event from the queue, parsing it, matching it against stored subscriptions and returning its matched subscriptions.

[4]News Industry Text Format: https://iptc.org/standards/nitf/

[5]YFilter Package: http://yfilter.cs.umass.edu/code_release.htm

A. Scalability

The goal of this experiment is understanding how the number of threads impacts performance and scalability of ME-IP, SE-CP and ME-CP techniques. The number of subscriptions is fixed in $5.000.000$ subscriptions since it already encompasses a subscription set of significant magnitude, simulating a fairly realistic scenario.

As expected, for the ME-IP technique a system throughput increase is clearly visible, processing nearly 20 times more events per second with 48 threads when compared with a single thread. As the graph of figure 4a shows, from 1 to 8 threads an almost linear increase is present, with an obtained speedup of 7.1 for 8 threads. This near linear increase results from the fact that threads match events independently, i.e., as more threads are added more events are processed in parallel. Although minimal synchronization is required for this technique, when adding more than 8 threads, parallel gains attenuate as more concurrent access to the event queue take place, limiting scalability.

Theoretically, average matching time for this approach should remain constant as threads process events independently, without cooperating in the processing of single events. However, as shown by figure 4b, the average matching time per event increases alongside with the number of threads, with an accentuated growth for more than 16 threads. This situation occurs due to low level synchronization barriers and cache limitations, as all threads, despite processing independent events, perform read-only accesses to the global indexes, cluster families, correspondent clusters and subscriptions of the subscription storer module.

At a different baseline, figure 4a illustrates that SE-CP technique produces a similar behavior to ME-IP technique up to 4 threads, resulting at this point in a speedup of approximately 3. From this point on, the events per second metric growth is minimal reaching its peak at 16 threads with a speedup of only 4.3. This inability to enhance system throughput for more than 16 threads results from the combination of two variables: the probability of the number of internal events tasks per event not being a multiple of the number of threads and the probability of internal event tasks with non-uniform complexities. As a consequence, as the number of threads per event increases, a larger subset of threads will remain idle for quite large amounts of time and with very small, or even inexistent, work times.

Figure 4b shows that the SE-CP technique features a stable decrease from 1 to 4 threads with a reduction of 67% of matching time per event and remains almost constant from 4 to 48 threads with its peak again at 16 threads with a total reduction of 77% of matching time per event. It is worth noting that this technique does not present a drastic increase of average matching time for more than 16 threads as occurred in the ME-IP technique. This comes from the fact that, since threads are cooperating in the processing of a single event and some internal events have common subpaths there is, potentially, a better usage of the cache. As this solution presents better results with 16 threads we can assume that for this workload in particular 16 threads is the *optimal* number of threads per event.

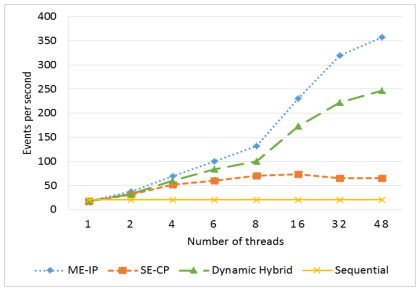
The first observation to make regarding the hybrid technique is that it is able to process a high rate of events while reducing the matching time of each event. As such, it tries to adapt the number of threads per event according to the number of available threads and event properties, namely, the number and complexity of internal event tasks. Figure 4a demonstrates that this approach presents a similar behavior to the ME-IP technique up to 16 threads, reaching at this point a speedup of 9.6. For more than 16 threads the number of events processed per second is slightly increased, reaching a maximum speedup of 13.8 with 48 threads. Additionally, the fact that a thread can only exit an event when all internal events have been discovered and distributed, also contributes to more synchronization costs in the scope of a single event. This reason, in combination with the fact that 16 threads per event presents better average matching time results, delegate the main reasons why a limit of 16 threads per event as been established for the hybrid approach. This way, we are able to avoid as fast as possible worker idle times resultant from task availability.

When analyzing average matching time, figure 4b depicts that the hybrid approach presents a similar behavior to the SE-CP technique up to 16 threads, with a reduction of 75% of matching time per event at this stage. For more than 16 threads, the hybrid approach will have more groups of threads processing events in parallel. As a result, for more than 16 threads, the hybrid approach suffers from the same problem of the ME-IP technique concerning synchronization overheads and cache performance. This increase in matching time for more than 16 threads, is the main influence for a smaller increase in system throughput starting from 16 threads. Despite this drawback, it still outperforms the ME-IP technique with 48 threads, by being 5.2 times faster when matching a single event.

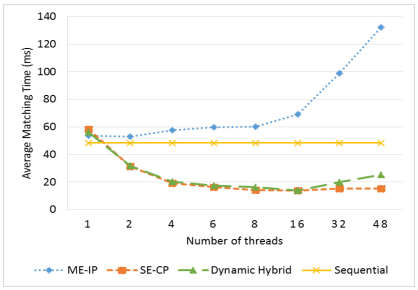
B. Subscriptions

The objective of this experiment is to evaluate how the number of subscriptions stored in the system impacts performance of parallel approaches. As expected, system throughput decreases and average matching time increases as the number of subscriptions expands, as a consequence of denser clusters that raise matching time per event.

For the ME-IP technique the system throughput decreases about half from $1.000.000$ to $10.000.000$. The main reason for this reduction comes from an accentuated increase in average matching time, from $80ms$ with $1.000.000$ subscriptions to almost $180ms$ with $10.000.000$ subscriptions. Nevertheless, with $10.000.000$ subscriptions, it is able to process 84% more events than SE-CP and about 56% more events than hybrid. This speedup improvement result from the fact that, since clusters are more compound, the sequential matching time per event increases, enhancing efficiency of parallel computations when processing events independently.

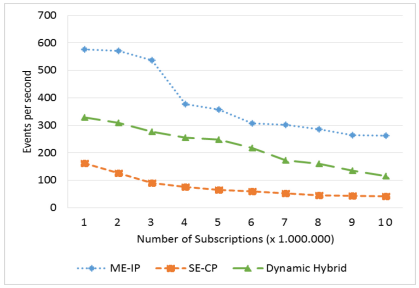


(a) Throughput

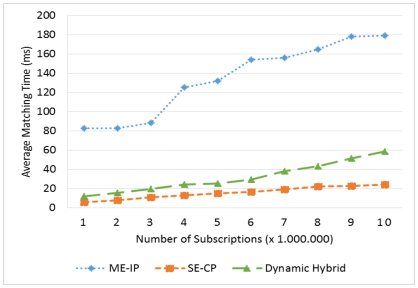


(b) Average Matching Time

Fig. 4: Varying Number of Threads: $T \in [1, 48]$, $N = 5.000.000$, $\Delta* = 0.2$, $\Delta// = 0.6$, $\Delta@ = 0.2$.



(a) Throughput



(b) Average Matching Time

Fig. 5: Varying Number of Subscriptions: $T = 48$, $N \in [1.000.000, 10.000.000]$, $\Delta* = 0.2$, $\Delta// = 0.6$, $\Delta@ = 0.2$.

For the SE-CP technique the graph of figure 5b shows a more moderate increase of average matching time from $6ms$ to $24ms$ when ranging from 1.000.000 to 10.000.000 subscriptions. In this technique, by comparison with ME-IP with 10.000.000 stored subscriptions, it is 86% faster when in the processing of a single event. Furthermore, it is up to almost 4 times faster than the sequential algorithm in the presence of 10.000.000 subscriptions.

As expected for the hybrid approach, since it incorporates the other two main techniques, ME-IP and SE-CP, an improvement in both the number of events processed per second and average matching time is visible. For system throughput it presents a similar behavior to ME-IP, with a rate of processed events lower than ME-IP but higher than SE-CP, whereas for average matching time an equivalent performance to SE-CP is present, with an average matching time higher than SE-CP but lower than ME-IP. This result reinforces the ideal in which hybrid is based, to evaluate high rates of events with small matching times even when in the presence of a large number of stored subscriptions.

In all cases, when increasing the number of subscriptions, system throughput decreases as a consequence of longer matching times per event. Although this may be true, a better performance and speedup is also visible, as the parallel overheads become negligible and the parallel computations have a greater impact in the overall performance and scalability.

VI. Conclusions

In this paper, we presented three parallel techniques to speedup the XML matching problem of publish/subscribe systems. The proposed solution is based on the highly efficient XML-based matching algorithm DeltaFilter, that evaluates XML events over stored XPath subscriptions through the use of very fast binary operations.

We introduce three parallel algorithms for an XML predicate-based matching engine: *Multiple Event Independent Processing* (ME-IP), *Single Event Collaborative Processing* (SE-CP) and *Multiple Event Collaborative Processing* (ME-CP) or simply *Hybrid*. The ME-IP technique aims to increase system throughput by scheduling threads to process independent events in parallel. On the other hand, the purpose of the SE-CP technique is to reduce average matching time by scheduling threads to collaborate in the process of a single event. The ME-CP technique results from the combination of the previous techniques in two parallel levels, achieving high system throughput with reduced matching times.

The experimental results showed that the ME-IP technique is highly scalable with up to 20 times more events processed with 48 threads when compared with the sequential version. SE-CP is able to efficiently reduce up to 77% the average matching time with 16 threads. Finally, the ME-CP technique was able to process 9.6 more events per second while maintaining a reduction of 75% of matching time per event with 16 threads when compared with the sequential version.

References

[1] Mehmet Altinel and Michael J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[2] G. Ashayer, H.K.Y. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 539–546, 2002.

[3] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. Navigation- vs. index-based xml multi-query processing. In *In Processings of Conference ICDE*, pages 139–150, 2003.

[4] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 310–321, New York, NY, USA, 2002. ACM.

[5] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. *The VLDB Journal*, 11(4):354–379, December 2002.

[6] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Transactions on Database Systems*, 28(4):467–516, December 2003.

[7] Yanlei Diao, P. Fischer, M.J. Franklin, and R. To. Yfilter: efficient and scalable filtering of xml documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 341–342, 2002.

[8] Yanlei Diao and Michael J. Franklin. High-performance xml filtering: An overview of yfilter. *IEEE Data Engineering Bulletin*, 26:41–48, 2003.

[9] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an internet-scale xml dissemination service. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 612–623. VLDB Endowment, 2004.

[10] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[11] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Record*, 30(2):115–126, May 2001.

[12] Amer Farroukh, Elias Ferzli, Naweed Tajuddin, and Hans-Arno Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 8:1–8:4, New York, NY, USA, 2009. ACM.

[13] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4):752–788, December 2004.

[14] ToddJ. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory — ICDT 2003*, volume 2572 of *Lecture Notes in Computer Science*, pages 173–189. Springer Berlin Heidelberg, 2003.

[15] Shuang Hou and H.-A. Jacobsen. Predicate-based filtering of xpath expressions. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 53–53, April 2006.

[16] Charlie Johnson and Jeff Welser. Future processors: Flexible and modular. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '05, pages 4–6, New York, NY, USA, 2005. ACM.

[17] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. Fist: Scalable xml document filtering by sequencing twig patterns. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 217–228. VLDB Endowment, 2005.

[18] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. Value-based predicate filtering of xml documents. *Data Knowledge Engineering*, 67(1):51–73, October 2008.

[19] Raul Martins, João Pereira, and Andreas Wichert. Deltafilter: A high performance cache-conscious xml filtering algorithm. Technical report, INESC-ID.

[20] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The xml web: A first study. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 500–510, New York, NY, USA, 2003. ACM.

[21] Iris Miliaraki, Zoi Kaoudi, and Manolis Koubarakis. Xml data dissemination using automata on top of structured overlay networks. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 865–874, New York, NY, USA, 2008. ACM.

[22] Mirella M. Moro, Petko Bakalov, and Vassilis J. Tsotras. Early profile pruning on xml-aware publish-subscribe systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 866–877. VLDB Endowment, 2007.

[23] João Pereira, Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, and Dennis Shasha. Webfilter: A high-throughput xml-based publish and subscribe system. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 723–724, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[24] João Pereira, Françoise Fabret, François Llirbat, Radu Preotiuc-Pietro, Kenneth A. Ross, and Dennis Shasha. Publish/subscribe on the web at extreme speed. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 627–630, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[25] Jianfeng Qian, Jianwei Yin, and Jinxiang Dong. Parallel matching algorithms of publish/subscribe system. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 638–643, April 2011.

[26] P. Rao and B. Moon. Prix: indexing and querying xml using prufer sequences. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 288–299, March 2004.

[27] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. *SIGSOFT Software Engineering Notes*, 22(6):344–360, November 1997.

[28] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *In Networked Group Communication*, pages 30–43, 2001.

[29] Mohammad Sadoghi, Ioana Burcea, and Hans-Arno Jacobsen. Gpx-matcher: A generic boolean predicate-based xpath expression matcher. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 45–56, New York, NY, USA, 2011. ACM.

[30] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh-based content routing using xml. *SIGOPS Operating System Review*, 35(5):160–173, October 2001.

[31] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Computer Communication Review*, 31(4):149–160, August 2001.

[32] Yu Xiaochuan and C.T.S. Alvin. A time/space efficient xml filtering system for mobile environment. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 184–193, June 2011.

[33] Ying Zhang, Yinfei Pan, and K. Chiu. A parallel xpath engine based on concurrent nfa execution. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 314–321, Dec 2010.