Parallel XML Matching Algorithm for Publish/Subscribe Systems

Susana Cardoso Ferreira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Doctor João Carlos Serrenho Dias Pereira

Examination Committee

Chairperson: Doctor João Emílio Segurado Pavão Martins
Supervisor: Doctor João Carlos Serrenho Dias Pereira
Member of the Committee: Doctor Luís Manuel Antunes Veiga

October 2015
Acknowledgements

I would like to thank to my supervisor Professor João Dias Pereira for his guidance, knowledge and motivation throughout the course of this thesis. This outcome would not be possible without his wisdom and criticism.

I am most grateful to all my friends, especially to my best friend Paulo Figueiredo for his invaluable support and endless friendship.

I would like to thank to my family for always being there for me not only in the bad moments, but also in the good ones.

I can not thank enough to my mother, my sister and my faithful companion Æon for their trust, support and unconditional love.

Lisboa, November 9, 2015
Susana Cardoso Ferreira
For my mother and sister.
Resumo

O paradigma distribuído mais adequado para processamento complexo de eventos e disseminação em larga escala de informação é o modelo de comunicação Publish/Subscribe. O crescimento contínuo de interesse em XML como a linguagem padrão para representação de informação e intercâmbio pela internet aumentou a importância de sistemas Publish/Subscribe baseados em XML, onde os utilizadores registam os seus interesses com expressões XPath complexas sobre o conteúdo e estrutura de eventos sobre a forma de documentos XML. A principal função de sistemas Publish/Subscribe reside no algoritmo de matching que é responsável por, quando novos eventos são publicados, determinar as subscrições verificadas por cada evento. Utilizadores cujas subscrições são satisfeitas serão posteriormente notificados com o evento publicado.

Vários algoritmos eficientes de matching XML emergiram recentemente e, apesar das suas diferenças em aspectos consideráveis, a maioria destes trabalhos foca-se em soluções sequenciais. Em aplicações no mundo real, com uma enorme quantidade de subscrições armazenadas e a chegada contínua de eventos, o algoritmo de matching torna-se facilmente num obstáculo que afecta o desempenho global do sistema. Uma maneira de obter um sistema escalável, mantendo simultaneamente alto desempenho, consiste em tirar partido de arquitecturas multi-core presentes em grande parte dos computadores da atualidade.

Nesta tese, propomos e implementamos três técnicas de processamento paralelo de eventos sobre DeltaFilter, um algoritmo de matching XML altamente eficiente. Realizamos avaliações experimentais numa máquina com 48 núcleos, para estudar a escalabilidade e desempenho das técnicas propostas com um número variável de threads em diferentes cenários de aplicação. Os resultados revelam um ganho no desempenho de 20 vezes mais eventos processados por segundo e uma redução de quase 74% no tempo de matching por evento quando na presença de 48 threads.
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, where users register their interests with subscriptions expressed as complex XPath expressions on the content and structure of events expressed as XML documents. 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.

Many efficient XML matching algorithms have emerged in recent years and, despite different in considerable aspects, the majority of these works only rely on sequential solutions. 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 multi-processors architectures already present in today’s computers.

In this thesis, 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.
Palavras Chave

Disseminação Seletiva de Informação
Publish/Subscribe
Algoritmos de matching XML
DeltaFilter
Processadores Multi-core
Processamento Paralelo de Eventos

Keywords

Selective Dissemination of Information
Publish/Subscribe
XML Matching Algorithms
DeltaFilter
Multi-core Processors
Parallel Event Processing
4.3.1 Impact of Event Properties .................................................. 79
4.3.2 Impact of Attributes ............................................................... 81
4.4 Summary ................................................................................. 82

5 Conclusions and Future Work .................................................. 83
5.1 Conclusions ........................................................................... 83
5.2 Future Work ........................................................................... 84
List of Figures

1.1 Publish/Subscribe paradigm overview ............................................. 2
1.2 Example of a well-formed XML document ........................................ 5
1.3 XML tree representation and XPath expressions example ...................... 6
1.4 XML-based Publish/Subscribe Architecture ...................................... 8
1.5 SAX parser events resultant from parsing XML document of figure 1.2 ...... 9
2.1 NFA example from the combination of a set of XPath subscriptions .......... 18
2.2 Runtime stack values during parsing of an incoming XML document ........ 19
2.3 Static and dynamic task allocation for NFA decomposition .................... 23
2.4 Data structures for a predicate-based clustering algorithm .................... 27
2.5 Multiple Event Independent Processing ......................................... 29
2.6 Single Event Collaborative Processing .......................................... 30
2.7 Stages of the Path Specific Stack (PSS) when evaluating event XML doc over subscription a/c/a/c/b ....................................................... 33
2.8 Overview of a GPU architecture, according to NVIDIA’s CUDA model and terminology ............................................................. 35
3.1 Internal events representation in DeltaFilter of the XML example of figure 1.2. . 38
3.2 Indexes of the XPath subscriptions presented in figure 1.3b .................... 40
3.3 Multi-level index I of the position records illustrated in figure 3.2 ............ 41
3.4 Cluster Vector and cluster family of access position record p_3 of the first three subscriptions of example of figure 1.3b ......................... 45
3.5 Filter structure when evaluation internal event 3 of figure 3.1. 49
3.6 Parallelism levels with the respective event processing approaches decomposition and granularity. 51
3.7 Multiple Event Independent Processing (ME-IP) approach. 52
3.8 Single Event Collaborative Processing (SE-CP) approach. 54
3.9 Thread distribution in static ME-CP approach. 59
3.10 Dynamic ME-CP approach. 60

4.1 Sequential Optimizations: $T = 1, N \in [1.000.000, 10.000.000], \Delta* = 0.2, \Delta// = 0.6, \Delta@ = 0.2$. 68
4.2 Varying Number of Threads: $T \in [1, 48], N = 5.000.000, \Delta* = 0.2, \Delta// = 0.6, \Delta@ = 0.2$. 70
4.3 Varying Block Size: $T = 48, blocksize \in [1, 2120], N = 5.000.000, \Delta* = 0.2, \Delta// = 0.6, \Delta@ = 0.2$. 73
4.4 Work and Wait times of SE-CP with IE (Internal Event) and SPR (Satisfied Position Record): $T \in [1, 48], N = 5.000.000, \Delta* = 0.2, \Delta// = 0.6, \Delta@ = 0.2$. 74
4.5 Varying Number of Threads per Event: $T = 48, N = 5.000.000, \Delta* = 0.2, \Delta// = 0.6, \Delta@ = 0.2$. 75
4.6 Varying Number of Subscriptions: $T = 48, N \in [1.000.000, 10.000.000], \Delta* = 0.2, \Delta// = 0.6, \Delta@ = 0.2$. 77
4.7 Distribution of the number of nodes and internal events of the three generated event workloads. 79
4.8 Impact of Event Properties: $T \in [1, 48], N = 5.000.000, \Delta* = 0.2, \Delta// = 0.6, \Delta@ = 0.2$. 80
4.9 Impact of Attributes: $T = 48, N = 5.000.000, \Delta* = 0.2, \Delta// = 0.6, \Delta@ \in [0, 0.6]$. 81
List of Tables

1.1  Result of applying XPath expressions of figure 1.3b to XML document of figure 1.2  

4.1  Workload parameters description and range values  
4.2  Event workload properties  
4.3  Threads per Event for Dynamic Hybrid technique  
4.4  System throughput and average matching time improvements with varying number of subscriptions.  
4.5  Minor, Standard and Major event workload properties
In the Information Age, the first step to sanity is FILTERING. Filter the information: extract for knowledge. Filter first for substance. Filter second for significance. These filters protect against advertising. Filter third for reliability. This filter protects against politicians. Filter fourth for completeness. This filter protects against the media.

– Marc Stiegler

The Internet is an ever growing source of knowledge at an unprecedented scale, with an exponential quantity of information available each passing day. The increasingly popularity and magnitude of the internet lead to a diversified environment containing users with completely different interests and, consequently, an enormous diversity of information. With so many information available, there is a high risk of information overload towards the users, where they are overwhelmed by an excessive amount of data, unable to remove the irrelevant content.

This led to the concept of Information Retrieval (IR) (Salton and McGill 1986) and Information Filtering (IF) (Belkin and Croft 1992) systems whose purpose is to reduce the quantity of irrelevant information from the large quantity of information available, through user needs. The main differences between these two systems is that, whereas IR are designed for one-time user interests to query static information, IF are designed for long-term user needs to query incoming dynamic information. IR systems already encompass a broad research and study area and, as stated by Belkin and Croft (Belkin and Croft 1992), IF systems can be viewed as a specialized type of IR. This way, IR research problems, as well as results, can also be applied to IF systems.

IR systems are deployed as traditional databases, where documents are persistently stored and user queries trigger searches for information of interest. An effective and scalable alternative to avoid a sequential scanning to the database consists in the application of indexes over the stored information. As pointed out in (Yan and García-Molina 1994), the key to scalable IF systems lies in the observation that the filtering of IF systems is the inverse problem of query-
ing in IR systems. Thus, an efficient selective matching of incoming documents for IF systems requires indexing the user queries.

IF systems aim at filtering the relevant information present in large quantities of data through the use of user profiles, which constitute long-term interests expressed by users. IF systems are employed on a broad range applications, such as browser filters for blocking non-valuable information, email filters, search result filters or e-commerce filters. An important research field for IF systems is Selective Dissemination of Information which guarantees that users only receive information in which they have expressed their interests in.

1.1 Selective Dissemination of Information

Selective Dissemination of Information (SDI) is a concept first introduced by Hans Peter Luhn in 1958 (Luhn 1958) as part of the business intelligence system. However, one of the first real dissemination service attending to a large number of users in the internet was only developed in 1994, with the name Stanford Information Filtering Tool (SIFT) (Yan and Garcia-Molina 1999). In this SDI system, users express their interests as sets of weighted keywords which are then stored in the system. When new documents arrive to the system from different data sources, these documents are evaluated against the user profiles through similarity-based techniques and users whose queries are matched are notified with the respective document (Yan and Garcia-Molina 1999).

With new information being made available each passing day and users with high demands of information, these 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 paradigm overview](image)

Figure 1.1: Publish/Subscribe paradigm overview

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, as illustrated in figure 1.1. 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. This many-to-many communication model allows for publishers and subscribers to be time and space decoupled, i.e., neither parties need to be actively participating at the same time and neither parties need to know of one another (Eugster, Felber, Guerraoui, and Kermarrec 2003).

A wide-range of applications use publish/subscribe in order to disseminate information, namely e-commerce and internet applications, alerts and notification services (e.g., Google Alerts), monitoring applications, financial information systems, areas concerning real-time data (such as RSS feeds) and several others. These applications require the development of software systems that enable scalable and efficient matching of a high rate of events against millions of subscriptions.

The logic of a publish/subscribe system resides in the matching engine which can be seen as the constitution of two modules:

- Subscription Storage: module responsible for storing subscriptions in a format that can be efficiently evaluated by the event processor.
- Event Processor: module responsible for implementing the matching algorithm, where stored subscriptions are matched against a stream of incoming events.

The core function of the matching algorithm is to solve the matching problem, ‘given an event, determine all subscriptions matched by this 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., TIBCO (Tibc
CHAPTER 1. INTRODUCTION

1999), Scribe (Rowstron, Kermarrec, Castro, and Druschel 2001)). The limitation of this model is mainly its expressiveness, which can be enhanced by introducing hierarchies to group topics in a tree fashion. This way, subscriptions referring to a node, contain all the subtopics of that node.

Next, **content-based**, or **property-based** (Rosenblum and Wolf 1997), 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 (Eugster, Felber, Guerraoui, and Kermarrec 2003). These properties can be associated to the actual content of the event, where events are represented as **attribute-value** pairs and subscriptions as conjunctions of conditions over attributes, (e.g., Gryphon (Banavar, Chandra, Mukherjee, Nagarajarao, Strom, and Sturman 1999), Elvin (Segall, Arnold, Boot, Henderson, and Phelps 2000), Siena (Carzaniga, Rosenblum, and Wolf 2000), Le Subscribe (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000) and Jedi (Cugola, Di Nitto, and Fuggetta 2001)), or to the meta-data of the event (e.g., Java Messaging Service (Microsystems 1998)).

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 (Pereira, Fabret, Jacobsen, Llirbat, and Shasha 2001), ONYX (Diao, Rizvi, and Franklin 2004), XTreeNet (Fenner, Rabinovich, Ramakrishnan, Srivastava, and Zhang 2005) and Sonnet (Zhou, Qian, Gong, and Zhou 2007)), where events are represented as **eXtensible Markup Language** (XML) (Bray, Paoli, and Sperberg-McQueen 1998) documents and subscriptions as **XML Path Language** (XPath) (Clark and DeRose 1999) expressions. This combination results in a good trade-off between the expressiveness of the subscription language and the required complexity for the matching algorithm.

### 1.2 XML-based Publish/Subscribe

The advent of XML\(^1\) on the web enhanced the way applications exchange, integrate data and how information is managed and processed on the internet. XML is a natural choice for Enterprise Application Integration (EAI), web services and event-based processing applications

---

\(^1\)http://www.w3.org/XML/
due to its expressive power, simplicity and heterogeneity qualities. XML documents are defined in a hierarchical organization with structured elements that can be nested to any depth and can contain value elements, representing the actual content. An XML element has the form:

\[<\text{tag } \text{attribute } = 'value' > \text{content} < \text{/tag}>\]

\(<\text{tag}>\) is referred to as open tag, \(<\text{/tag}>\) as close tag and the actual content is represented in \text{value} and \text{content} as textual tokens. Content \text{value} corresponds to an attribute element, whereas content \text{content} can also be composed by a sequence of other XML elements, defining a hierarchy between the context nodes. Consequently, an element can appear inside the same enclosing element at any level of depth, situation referred to as recursion.

```
1 <xml version="1.0" encoding="utf-8"?>
2  <A attr1="31">
3      <B>19</B>
4  </B>
5   <B attr2="14">90</B>
6  <C>
7       <A></A>
8   </C>

Figure 1.2: Example of a well-formed XML document
```

An XML document is said well-formed if it respects XML syntax rules, such as, that all documents must have an unique root element and all elements must have a closing tag. Optionally, the arrangement between the structure and value elements can be specified by a set of predefined rules in a schema specification, such as XML Schema (Consortium 2004). Figure 1.2 illustrates a well-formed XML document without a predefined schema where, for instance, \text{content} of root element \text{A} (line 2) are elements \text{B} (line 3 and 4) and \text{C} (line 5–7). A recursion situation is present with element \text{A}, since the root element at line 2 contains a sub-element \text{A} in line 6.

The most popular XML query language is XPath\(^2\), which is a declarative language that treats XML documents as tree structures and has the ability of selecting parts of this XML tree through XPath expressions. There are three types of nodes: (1) element nodes which only contain structural information and are specified as XML tags, (2) attribute nodes which contain values corresponding to the attributes of a tag and (3) text nodes which contain values and

\(^2\)http://www.w3.org/TR/xpath/
CHAPTER 1. INTRODUCTION

(a) XML tree representation of XML document 1.2

(b) XPath expressions example

Figure 1.3: XML tree representation and XPath expressions example

are represented between an open and close XML tag. This way, XPath is able to exploit the structure of an XML document as relationships between nodes are represented in the form of tree edges. It is also worth noting that each XML document contains a single root node, which encompasses all other elements and is always at depth 1 in the XML tree.

An XPath expression is defined as a location path that results of the combination of one or more location steps. A location step results of the combination of an axis, a node test and zero or more predicates, represented in the form:

\[
\text{axis node test [predicate]} ... [predicate_n]
\]

An axis establishes a hierarchical relationship between nodes and the most prominent axis are the parent-child (’/’) and ancestor-descendant (’//’) axis. Node node test refers to specific nodes, for instance, A matches element nodes with name A, text() matches only text nodes and the wildcard operator (’*’) matches any element node. Predicates are used to further refine the selected node set and are represented between ‘[ ]’. In turn, a location path can be an absolute path, beginning from the root node (e.g., figure 1.3 XPath expressions lines 0 and 1), or a relative path, beginning in any context node (e.g., figure 1.3 XPath expressions lines 2–5).

To sum up, XPath expressions are able to express restrictions over an XML document structure, defined in this context as structure matching and over an XML document content, defined in this context as value matching. Structure matching is expressed through the use of XPath relationships, whereas value matching is expressed through the used of predicates. Predicates are represented in the form [attribute operator value] where attribute corresponds to an attribute node, operator is one of the following ’=, <, <=, > or >=’ and value corresponds to an attribute
value. Predicates can occur in two modes regarding the operator used: (1) exact mode (operator ‘=’) and (2) range mode (operators ‘<, ≤, >, ≥’).

Figure 1.3 illustrates the XML tree resultant from the XML document of figure 1.2 and an example of XPath expressions defined in the context of this XML tree. The results of applying the XPath expressions of figure 1.3b to the XML document of figure 1.2, are expressed in table 1.1.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/A/*/A</code></td>
<td>‘∗’ in this expression matches elements B and C, but then the parent-child relationship ∗/A specifies that element A must appear at the depth level immediately after the node matched by ‘∗’, in this case element C. As a result, path /A/C/A is matched.</td>
</tr>
<tr>
<td><code>/A//A</code></td>
<td>The ancestor-descendant relationship specifies that the second element A must appear at a higher depth level than the first element A in the tree. As a result, path /A/C/A is matched.</td>
</tr>
<tr>
<td><code>//C/A</code></td>
<td>Because this represents a relative location path (starts with ‘//’), element C can be found at any level of depth, immediately followed by an element A. As a result, path /A/C/A is matched.</td>
</tr>
<tr>
<td><code>//B[@attr2&lt;90]</code></td>
<td>Because this represents a relative location path, element B can be found at any level of depth. The predicate [@attr2 &lt; 90] specifies that only B elements with attribute attr2 containing a number lower than 90 are matched. As a result, the second path /A/B is matched.</td>
</tr>
<tr>
<td><code>//*[@attr1=31]</code></td>
<td>Because this represents a relative location path, ‘∗’ can match any element at any level of depth. The predicate [@attr1 = 31] specifies that only elements with attribute attr1 equal to 31 are matched. As a result, path /A containing only the root element is matched.</td>
</tr>
<tr>
<td><code>//B[@attr2&gt;10][text()&lt;=90]</code></td>
<td>Because this represents a relative location path, element B can be found at any level of depth. The two predicates [@attr2 &gt; 10] and [text() ≤ 90], specify that only B elements with attribute attr2 containing a number higher than 10 and with a text element lower or equal to 90 are matched. As a result, the second path /A/B is matched.</td>
</tr>
</tbody>
</table>

Table 1.1: Result of applying XPath expressions of figure 1.3b to XML document of figure 1.2

XML-based pub/sub systems often follow an architecture similar to the one depicted in figure 1.4. In this system, a subscriber expresses subscriptions as XPath expressions forwarded to
the Subscription Service in the intermediary agent. This service then dispatches the subscription to the XPath parser in order to be converted to the engine’s internal subscription representation and stored in the subscription storage module. After that, the subscription storage sends an unique identifier representing the subscription in the system to the subscriptions service, which then associates this identifier with the respective subscriber.

When a publisher publishes an event, this event is forwarded to the Publish Service which then dispatches it to the XML parser in order to be processed by the matching algorithm in the event processor module. Furthermore, when the matching algorithm finishes and the subscriptions matched by the event have been computed, the event processor module sends these matched subscriptions to the Notification Service, which is then responsible for disseminating the event to the subscribers with matched subscriptions. These components are now described in more detail.

**XML Parser.** Several XML Parsers are available, but one that stands out is the event-driven parser SAX Parser (Megginson et al. 2001), which allows the matching to start alongside with the parsing. SAX parser contains a set of event methods for which the system must implement handlers:

- `startDocument()`: event that reports that the parser will start parsing a new document.
- `endDocument()`: event that reports that the parsing of a document finished.
- `startElement(tag, attributes)`: event that reports that a start tag with name `tag` and with a
set of attributes attributes was parsed.

- \textit{endElement(tag)}: event that reports that a close tag with name \textit{tag} was parsed.

- \textit{characters(string)}: event that reports that characters \textit{string} were parsed between a start and a close tag.

Figure 1.5 demonstrates the resultant SAX events of parsing the XML document of figure 1.2.

```xml
1 startDocument()
2   startElement(A, attr1="31")
3       startElement(B)
4           characters(19)
5       endElement(B)
6   startElement(B, attr2="14")
7           characters(90)
8       endElement(B)
9   startElement(C)
10      startElement(A)
11     endElement(A)
12   endElement(C)
13   endElement(A)
14 endDocument()
```

Figure 1.5: SAX parser events resultant from parsing XML document of figure 1.2

\textbf{Notification Service.} The notification service is responsible for the dissemination of events to the subscribers whose subscriptions have been matched. In a centralized pub/sub system, this service forwards the events directly to the subscribers. However, in distributed pub/sub systems, this service routes events to the responsible broker. This event routing in content-based pub/sub is referred to as content-based routing (CBR) and exploits content information to route events to their respective subscribers (Coulouris, Dollimore, Kindberg, and Blair 2011). In some systems, the matching algorithm operates as a form of event routing mechanism in order for events to be forwarded only to brokers with a path to subscribers.

With the increasingly use of XML as the language for data exchange and dissemination, several XML message routing applications emerged, namely XML routing over a mesh-based overlay network (Snoeren, Conley, and Gifford 2001), reusing matching information from preceding brokers (Chan 2007) and distribution of the matching algorithm over different brokers through the use of distributed hash-tables (DHTs) (Miliaraki, Kaoudi, and Koubarakis 2008) (Miliaraki and Koubarakis 2010). In all these systems, when a subscription is matched by
an event, the entire XML document is disseminated to the respective user. Grummt (Grummt 2011) proposed a system capable to delivery only the parts of XML documents that have in fact been matched by the conditions expressed by the subscribers.

**XML Matching Engine.** The use of XML in combination with XPath in pub/sub systems allows not only content matching, as occurred in simple content-based systems, but also structure matching relatively to the hierarchy present on XML documents. Consequently, the XML matching problem is more complex and more difficult to resolve in acceptable time than content-based. The way subscriptions are stored greatly influences the time and space complexity of the matching algorithm, since when an event arrives to the system these subscriptions will have to be evaluated as fast as possible. A brute-force approach where all the subscriptions are evaluated is out of question due to its inability to scale in the presence of a huge amount of subscriptions and a high rate of incoming events. Therefore, a direction was chosen with the goal of reducing the subscription search space for incoming events, mainly by taking advantage of similarities between different XPath expressions.

Different XML matching algorithm have been proposed in the literature, from automata solutions where subscriptions are stored in state machines with states as location steps and transitions triggered by SAX parser events (Altinel and Franklin 2000)(Diao, Fischer, Franklin, and To 2002), to solutions that convert simple content-based to XML-based systems (Pereira, Fabret, Jacobsen, Llirbat, and Shasha 2001)(Hou and Jacobsen 2006)(Sadoghi, Burcea, and Jacobsen 2011). In the latter, XPath expressions are converted to sets of boolean predicates, that additionally to expressing the actual content, establish the hierarchy of the different XPath elements, resulting in complex and numerous predicates that limit performance. DeltaFilter (Martins, Pereira, and Wichert ) is an XML matching engine able to efficiently adapt a content-based to an XML-based system by converting XPath expressions to very fast binary operations that are set and evaluated throughout the parsing of XML documents. This way, DeltaFilter manages to exploit cache conscious indexes proven efficient in content-based systems in an XML matching engine of high performance.

As concluded in the study presented in (Mignet, Barbosa, and Veltri 2003) 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 matching algorithm. For instance, there have been numerous contributions to improve previous matching algorithms, namely more efficient data structures (Diao and Franklin 2003), better index structures (Kwon, Rao, Moon, and Lee 2008), better clustering criteria (Sadoghi, Burcea, and Jacobsen 2011), lower memory usage (Xiaochuan and Alvin 2011) and cache-conscious indexes (Martins, Pereira, and Wichert). 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.

1.3 Thesis Contributions

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 (Johnson and Welser 2005). In fact, today, most commercialized computing architectures already encompass Chip-level Multiprocessors (CMP), i.e., multi-core processors. It is important for the next generation of XML matching algorithms to evolve to this type of architecture, as a solution to improve efficiency and, above all, scalability.

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., (Farroukh, Ferzli, Tajuddin, and Jacobsen 2009), (Qian, Yin, and Dong 2011), (Margara and Cugola 2014) 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 thesis, we propose and implement a parallel matching engine by adapting DeltaFilter (Martins, Pereira, and Wichert), an highly efficient and cache aware XML matching algorithm, to a multi-processor environment for performance enhancement. DeltaFilter is an XML matching algorithm that is able to efficiently tailor the content-based matching problem to an XML-based matching problem without the additional complexity associated to structure and value matching of XML documents. By validating subscriptions over incoming events as very fast binary operations and employing index and clustering techniques proven efficient in previous works, e.g., (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000)(Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001)(Pereira, Fabret, Jacobsen, Llirbat, and Shasha 2001), both time and temporal complexity decrease.

Based on the nomenclature presented by Farroukh et al. (Farroukh, Ferzli, Tajuddin, and Jacobsen 2009) for content-based pub/sub systems, three approaches concerning parallel XML matching algorithm 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. Consequently, this approach is suitable for constant matching times in scenarios where large numbers of events overwhelm the system.

A detailed study is presented for each technique with special focus on how the addition of more threads to a matching algorithm can improve its performance, from up to which points
the parallel overheads do not impact performance, what is the best task granularity and how are tasks distributed. Experiments, run on a 48 multi-core machine, demonstrate how the three approaches perform when dealing with large collections of subscriptions and events.

1.4 Thesis Organization

This document is organized as follows:

- **Chapter 2** presents the major categories of XML matching algorithms, matching algorithms related to other subscription languages and an overview of different techniques that employ XML, namely XML query processors.

- **Chapter 3** presents a more meticulous analysis of DeltaFilter and describes the parallel event processing techniques in detail.

- **Chapter 4** illustrates the performance and scalability results of the parallel event processing techniques in different application scenarios.

- Finally, in **Chapter 5** conclusions and directions for future work are presented.

XML query processors consist in the mapping of XML to conventional query engines of databases, where XML documents are persistently stored and incoming XML queries trigger searches to find all matched documents. This way, as it happens in traditional databases, XML documents are indexed in order to quickly find all occurrences of incoming XML queries. Among the most popular XML processors there are TwigStack (Bruno, Koudas, and Srivastava 2002) and, more recently, Twig$^2$Stack (Chen, Li, Tatemura, Hsiung, Agrawal, and Candan 2006) that proposes a holistic XML query processor restricted to ancestor descendant relationships and PRIX (Rao and Moon 2004)(Rao and Moon 2006) that transforms and indexes XPath queries into Prüfer sequences. Both these approaches use XPath as query language. In (Ludäscher, Mukhopadhyay, and Papakonstantinou 2002), Ludäscher et al. proposed a query processor with an XPath superset, XQuery$^1$, by exploiting internal buffers. However, it is unable to process the ancestor-descendant relationship in queries.

More recently, the area of parallel XML query processing has been gaining attention. For instance, ParaXML (Li Lu W. and 2008) traverses XML documents in parallel by employing a fine-grained work stealing technique as a way to obtain more balanced workloads.

\footnote{http://www.w3.org/TR/xquery/}
In XML matching systems, the roles of the XML query processor trade places, i.e., while XML query processing evaluates incoming XML queries over stored XML documents, XML matching evaluates incoming XML documents over stored XML queries. In the former, the result of evaluating an XML document over an XPath expression is a set of nodes, whereas in the latter, the result consists only in a boolean referring if the set of nodes is (subscription not matched), or not (subscription matched), empty. In that sense, contrarily to indexing XML documents as in XML query processors, in XML matching 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 four main classes: (1) automaton-based, where subscriptions are stored as state machines with states as location steps and transitions between states triggered by parser events; (2) index-based, where subscriptions exploit high performance indexes; (3) sequence-based focus on processing twig patterns against incoming XML documents; and (4) all other approaches.

2.1 Automaton-based

Automaton-based matching algorithms are based on the idea that XPath expressions can easily be mapped to Moore Machines (Hopcroft and J.D. Ullman 1979), 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. For instance, startElement(tag) events trigger transitions with ‘tag’ or ‘*’, whereas endElement(tag) backtrack to previous states.

The first solution of this category was XFilter (Altinel and Franklin 2000) 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 (Diao, Fischer, Franklin, and To 2002) where common prefixes are represented and processed only once in a global Non-deterministic Finite Automaton (NFA).
2.1. **AUTOMATON-BASED**

2.1.1 **YFilter**

YFilter is considered the standard of the XML matching community, mainly because it was made available by the authors in the YFilter package\(^2\). YFilter stores all subscriptions in a single NFA, in order to share the storage and the processing of prefixes between subscriptions, resulting in a performance improvement of structure matching when compared to XFilter.

Figure 2.1 illustrates a set of eight XPath subscriptions (figure 2.1a) and the NFA resultant from combining such subscriptions (figure 2.1b), where states are represented as circles and, for further analysis, each state has a number placed inside the circle, in order to be uniquely identified. Two types of states take place: (1) non-shared states, i.e., states that only encompass a subscription, represented as white circles (e.g., state 7); and (2) states shared by two or more subscriptions, i.e., states that encompass more than one subscription (e.g., state 9), represented as gray circles. Note that prefixes shared by different subscriptions are represented only once, e.g., state 2 corresponds to the location step ‘/a’ shared by all eight subscriptions. A state is considered an accepting state, represented with a bold line, if it corresponds to the matching of one (non-shared state) or more (shared state) subscription(s). Subscriptions’ identifiers are represented with curly brackets next to the respective accepting state. For example, state 4 is an accepting state for subscription Q2, while state 5 is an accepting state for both subscriptions Q3 and Q8. In turn, transitions between states are representing as arrows connecting two states. Two types of transitions exist in this context: (1) normal transitions corresponding to element nodes or the wildcard ‘*’ operator; and (2) ε-moves, or empty transitions, which allow the support of the non-determinism introduced by the XPath ancestor-descendant relationship.

In the presence of non-determinism on a NFA, for each SAX parser event, several transitions can be triggered and, consequently, several states can be active at a time, resulting in multiple active paths in the NFA. Consequently, when an `endElement(tag)` SAX parser event is generated, all currently active states must backtrack to previous states. For this purpose, a runtime stack structure is used to bookkeep previously active states and current active states are maintained in the top of the stack (TOS). This way, when an `endElement(tag)` event occurs, the NFA backtracks and the TOS is popped. On the other hand, whenever a SAX parser event `startElement(tag)` is generated, all active states, i.e., all states in the TOS, are traversed and for each state containing transitions labeled with ‘tag’, ‘*’ or ‘ε’, the correspondent target states

\(^2\)YFilter Package: http://yfilter.cs.umass.edu/code_release.htm
CHAPTER 2. RELATED WORK ON XML MATCHING ALGORITHMS

(a) XPath subscriptions
Q1=a/b
Q2=a/c
Q3=a/b/c
Q4=a/b/c
Q5=a/*/c
Q6=a/c
Q7=a/*/c
Q8=a/b/c

(b) NFA representation

Figure 2.1: NFA example from the combination of a set of XPath subscriptions

are pushed to the TOS. Note that ancestor-descendant relationships are represented in the NFA as an empty transition to a state with a ‘*’ self-loop. Therefore, when an \( \varepsilon \)-move transition is present the target state needs to be added to the stack from this path on whenever a new \texttt{startElement} \((\text{tag})\) event occurs, since the location steps corresponding to its subsequent states can appear at any level of depth.

An XML document is said to match a subscription when its accepting state is reached. Contrarily to finishing execution when an accepting state is reached, as occurs in regular NFAs, in YFilter the NFA execution must continue until all possible accepting states have been visited, since all possible matched subscriptions must be discovered. Therefore, the set of all subscriptions matched is only complete when the parsing finishes.

Figure 2.2 illustrates the runtime stack values for the NFA exemplified in 2.1 resultant from processing the XML fragment ‘<a><b><c><e><f><g</a>’. The first step consists in pushing the initial NFA state, state 1, to the TOS. Secondly, the SAX parser generates a \texttt{startElement}(\textit{a}) event, resultant from parsing the open tag <a>. Since the set of active states is only composed by state 1 and, this state contains an a transition, state 2 is pushed onto the TOS. Thirdly, a \texttt{startElement}(\textit{b}) event is raised, resultant from parsing the open tag <b>. Therefore, for each state in the TOS all matched transitions are triggered. State 2 is the only on the TOS and contains a transition b to state 3. Because state 3 is an accepting state for subscription Q1, this subscription is added to the set of matched subscriptions. From state 2 there are another two matched transitions, transition ‘*’ to state 9 and an empty transition to state 6. In the latter, because state 6 contains a transition b to state 7, this state is also added...
to the TOS. Recall that this situation is to match subscriptions containing ancestor-descendant relationships, meaning in this example that state 7 corresponds to subscriptions whose tag $b$ appears at any level of depth higher than $a$. In brief, the resultant TOS of the third step contains states $\{3, 9, 7, 6\}$.

Similarly to the previous step, when the SAX parser generates a \textit{startElement}(c) event, resultant from parsing open tag \texttt{<c>}, states $\{5, 10, 12, 8, 11, 6\}$ are pushed onto the TOS. Because states $\{5, 10, 8, 11\}$ constitute accepting states for subscriptions $\{Q_3, Q_8, Q_5, Q_4, Q_6\}$ these subscriptions are added to the set of matched subscriptions. Finally, SAX parser generates events \textit{endElement}(c), \textit{endElement}(b) and \textit{endElement}(a) performing three consecutive \textit{pop}() operations. As expected, the content of the TOS in the last step is solely the initial state, state 1.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{runtime_stack.pdf}
\caption{Runtime stack values during parsing of an incoming XML document}
\end{figure}

Although this shared processing solves the structure matching in an efficient way, it complicates the handling of value matching. If the same approach was used to value matching, the NFA would be extended with additional transitions to states corresponding to predicates, resulting in a huge increase in the number of states and, consequently, destroying the shared prefixes among subscriptions, which is the main reason why a NFA is used. Consequently, three different alternatives for value-based evaluation have been proposed in the literature: \textit{bottom-up evaluation} (Miliaraki and Koubarakis 2010), \textit{Selection Postponed} (SP) (Diao and Franklin 2003) (or top-down evaluation (Miliaraki and Koubarakis 2010)) and \textit{Inline} (Diao and Franklin 2003) (or step-by-step evaluation (Miliaraki and Koubarakis 2010)).

\textbf{Bottom-up evaluation.} Bottom-up evaluation is based on a widely employed technique in traditional relational query processing, where \textit{cheap selections} are evaluated as early as possible
as a way to prune future work. In an NFA execution this is translated in first performing value matching, followed by structural matching. Although efficient in relational query processing, here a lot of work is put into evaluating value-based predicates for subscriptions whose structure may not be matched.

**Selection Postponed or top-down evaluation.** Conversely, SP evaluation exploits the NFA’s structure matching to prune subscriptions and delays predicate evaluation only to subscriptions whose accepting state is reached. This approach outperforms Inline by pruning the NFA with structure matching instead of premature value matching.

**Inline or step-by-step evaluation.** These previous techniques aim at pruning the search states using either structure or value matching evaluation. An alternative consists in performing structural alongside with value matching in each state. That is, when a state is reached the associated predicates are evaluated over the list of candidate predicates contained in the XML document, meaning that all queries must maintain information concerning its satisfied predicates. However, this evaluation does not necessarily stop structure or value matching of subscriptions sharing the same path, since the predicates associated to a state may not be shared by other subscriptions. Additionally, backtracking and ancestor-descendant relationships further increase complexity and decrease performance.

In order to attenuate this problem, instead of in each state evaluating all value-based predicates, (Miliaraki and Koubarakis 2010) proposes an user-defined selectivity criterion that evaluates value-based predicates only on the most selective states, as a way to prune active paths. Inspired by this idea, (Antonellis, Makris, and Pispirigos 2012) goes one step further and instead of user-defined selectivity criterion uses a popular criteria, where only the states that have been activated a lot in previous XML documents are considered and, a semantic similarity criteria, where only the value-based predicates semantically similar to the values of the candidate predicates of the XML documents are evaluated. This technique, however, presents a major drawback in that a post-processing phase is required to evaluate the candidate predicates of the XML documents against all matched subscriptions, since not all predicates may have been evaluated in unselected states.

Although scalable with the number of subscriptions, the performance of YFilter’s NFA decreases for deep XML documents, since the deeper in the document 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 (Diao, Altinel, Franklin, Zhang, and Fischer 2003). In (Green, Miklau, Onizuka, and Suciu 2003) 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 (Green, Gupta, Miklau, Onizuka, and Suciu 2004). XPush (Gupta and Suciu 2003) transforms this previous system into a pushdown automata in order to support a more expressive XPath set but, with several flexibility restrictions especially concerning the addition and remotion of subscriptions from the built automata.

2.1.2 Parallel YFilter

In (Miliaraki, Kaoudi, and Koubarakis 2008), YFilter is adapted to a distributed environment on top of Chord (Stoica, Morris, Karger, Kaashoek, and Balakrishnan 2001), a distributed lookup protocol for peer-to-peer applications. In order to achieve an efficient structural XML matching this solution exploits Distributed Hash Tables (DHTs), where the NFA is statically decomposed in sub-machines processed by different peers. The processing of the NFA is implemented using two possible methods, iterative method where the publisher peer is responsible for the parsing, runtime stack management and forwarding of SAX events to the correspondent peers and, a recursive method where the publisher peer forwards the XML document to the peer containing the initial state, which then notifies subsequent peers in a recursive way. The iterative method results in very poor performance since all peers are dependent of the publisher peer which in some cases can become a bottleneck. On the other hand, the recursive method obtains better results since it is able to execute multiple active paths in parallel.

More recently, an extension to combine structural and value XML matching in a distributed environment was proposed in (Miliaraki and Koubarakis 2010) by exploiting the structured P2P system Pastry (Rowstron and Druschel 2001) DHT. In brief, this type of approach to split the NFA, although simple, does not guarantee that each thread will have workloads of equivalent sizes, as the actual NFA active paths may occur only in a very small part of the whole NFA, which results in poor speedups when compared to the original YFilter.
A different approach, concerning vertical scalability, consists in adding more cores to a single machine and executing the NFA in parallel in a multi-core system, e.g., (Zhang, Pan, and Chiu 2010) and (Antonellis, Makris, and Pispirigos 2012). In (Zhang, Pan, and Chiu 2010) the structural XML matching of a single event is performed in parallel by decomposing the NFA in independent and concurrent sub-machines, corresponding in this context to tasks, to be further processed by available threads. These tasks are placed in a global queue by the main thread and worker threads are continuously checking for new tasks to process. Two approaches have been proposed for task allocation: (1) static, that decomposes the NFA based on the number of threads and (2) dynamic, that decomposes the NFA at runtime using a work-stealing technique (Blumofe and Leiserson 1999).

In the static approach, in order to obtain good load balancing it is imposed a limit for the number of tasks of \( k \) times the number of threads \( t \) used, i.e., \( k \times t \). Constant \( k \) establishes to which number of location steps the decomposition depends on. This way, if a location step has more transitions than there are threads, these transitions (and respective states) are grouped so as to equal the number of tasks. In contrast, if a location step has a small number of transitions, the next location step is used to generate more tasks. This procedure continues until \( k \) is reached. To demonstrate consider figure 2.3a and assume \( k = 2 \) and \( t = 2 \). Considering the first location step, three tasks are obtained \( T_0 = \{1, 2, 3, 4, 5, 6, 7\}; T_3 = \{8, 9, 10\}; T_4 = \{11, 12, 13\} \). However, task \( T_1 \) is more coarse-grained than tasks \( T_3 \) and \( T_4 \). For this reason the second location step is taken into account and task \( T_0 \) is decomposed into two more fine-grained subtasks \( T_1 \) and \( T_2 \), resulting in the expected number of four tasks \( k \times t = 2 \times 2 = 4 \).

In order to obtain a better load balancing, statistical information concerning incoming XML documents could be used to optimize the division of the NFA. Because several paths can be active in an NFA, without information concerning the incoming events, there is no prior knowledge on which sub-machines will have active paths. Therefore, in the worst case scenario a thread can get only tasks corresponding to active paths, while the remaining threads get tasks corresponding to inactive paths resulting in the increase of idle time.

In the dynamic approach, to avoid large idle times, the NFA is initially decomposed in \( t \) parts, one for each thread and, when a thread finishes processing its associated task, a steal work request is sent to all busy threads. Subsequently, busy threads overwhelmed by its associated task will decompose it into two subtasks and place one in the task queue in order to be
2.1. AUTOMATON-BASED

(a) Static task allocation

(b) Dynamic Task allocation

Figure 2.3: Static and dynamic task allocation for NFA decomposition

processed by an idle thread.

The dynamic allocation is illustrated in figure 2.3b, where the incoming XML document and XPath subscriptions are represented on the left side. Assuming that two threads are being used, the NFA is initially decomposed into two tasks, task \(-1\) and task \(-2\), which are attributed to threads \(1\) and \(2\), respectively. Since task \(-2\) is more fine-grained than task \(-1\) it is more likely that thread \(2\) finishes executing task \(-2\) before thread \(1\) finishes task \(-1\). Thus, when thread \(2\) finishes task \(-2\), it broadcasts a steal work signal. Thread \(1\) responds to this signal by decomposing task \(-1\) into two more fine-grained tasks, tasks \(-3\) and \(-4\), and adding one of these subtasks into the task queue.

Although some optimizations to prevent an explosion of subtasks, such as restrictions on which threads respond to steal work requests based on its global processing, in the worst case scenario a thread can obtain a very time-consuming task and the remaining threads finish their associated tasks and overwhelm the busy task with several steal work signal, impacting both busy (unable to process work, because of task decomposition) and idle threads (waiting for new tasks to be added to the queue).

Antonellis et al. proposed in (Antonellis, Makris, and Pispigrig 2012) a different approach that, instead of dividing the NFA into several subsets, whenever a startElement(tag) SAX event
is raised, the set of active states in the TOS is divided in subsets to be further evaluated by available threads. Subsequently, each thread will be responsible for evaluating the candidate transitions of the obtained states and the states resultant from the matched transitions are pushed to a new level of the TOS. Take the case of the NFA illustrated in figure 2.1b and lets consider that 3 threads are being used. Using the XML fragment of figure 2.2, when the SAX event \texttt{startElement}(c) is generated, the set of active states \{3, 9, 7, 6\} is decomposed in subsets \{3\}, \{9\}, \{7\} and \{6\}. Suppose thread 1 gets state \{3\}, thread 2 gets state \{9\} and thread 3 gets state \{7\}. Each thread will evaluate candidate transitions (transition \(c, * \) and \(\varepsilon\)) of the assigned state and push the resultant states into the TOS. The first thread to finish processing a state, is assigned with the last state, state \{6\}. Despite obtaining better speedups than the solution proposed in (Zhang, Pan, and Chiu 2010), this solution still suffers from some load balancing issues, since active states may have different numbers of candidate transitions, which may not result in the best distribution of work.

2.2 Index-based

Index-based matching algorithms exploit index structures for a more efficient filtering by sharing the processing of structure matching. MatchMaker (Lakshmanan and Parthasarathy 2002) was among the first approaches of this category to efficiently match shared patterns between subscriptions. However, it is focused on disk-oriented solutions that decrease the matching performance when compared to memory-based matching algorithms (Diao, Altinel, Franklin, Zhang, and Fischer 2003). XTrie (Chan, Felber, Garofalakis, and Rastogi 2002) 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 (Bruno, Gravano, Koudas, and Srivastava 2003) 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.
2.2. INDEX-BASED

2.2.1 Predicate-based

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., Le Subscribe (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000), (Pereira, Fabret, Llirbat, and Shasha 2000), (Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001), WebFilter (Pereira, Fabret, Jacobsen, Llirbat, and Shasha 2001), (Ashayer, Leung, and Jacobsen 2002), (Hou and Jacobsen 2006), GPX-Matcher (Sadoghi, Burcea, and Jacobsen 2011), DeltaFilter (Martins, Pereira, and Wichert)). In this matching algorithms because predicates are the basic unit of processing, subscriptions are defined as conjunctions of predicates validated as regular boolean expressions over properties of incoming events. The execution of predicate-based matching algorithms follows two phases:

1. 1st phase - Predicate Update: predicates are evaluated over properties of incoming events;

2. 2nd phase - Subscription Evaluation: subscriptions matched are computed based on the predicates satisfied in the 1st phase.

Different systems have proposed different techniques for the 2nd phase, namely, counting algorithms that count if the number of satisfied predicates is the same as the number of predicates contained in a subscription (e.g., (Ashayer, Leung, and Jacobsen 2002), (Sadoghi, Burcea, and Jacobsen 2011)) and, a more efficient alternative, clustering algorithms that group subscriptions based on common predicates that comply with a selectivity criteria, defined access predicates, e.g., (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000), (Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001), (Martins, Pereira, and Wichert).

The main idea behind the concept of access predicates is to reduce the subscription search space for the Subscription Evaluation phase. The best solution consists in grouping subscriptions based on their size in clusters according to an unique access predicate shared by all subscriptions of the same cluster. For each event, clusters of satisfied access predicates have a high probability of containing subscriptions matched by the event. For this purpose, it is important for an access predicate to have a powerful pruning capability that significantly reduces the number of candidate subscriptions. Otherwise, the overhead of maintaining specialized clusters could potentially deteriorate the performance since nearly all subscriptions need to be evaluated.
Predicate-based matching algorithms date back to content-based publish/subscribe (e.g., (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000), (Pereira, Fabret, Llirbat, and Shasha 2000), (Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001)). Le Subscribe (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000), 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 (Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001) a highly efficient and scalable clustering algorithm that exploits space efficient and cache-aware structures. In (Ashayer, Leung, and Jacobsen 2002) the authors assume that predicates are expressed over reasonably-sized finite domains, hence exploiting highly efficient table-based predicate data structures in DBMS-based matching algorithms.

An overall view of the main data structures used in predicate-based clustering algorithms is depicted in figure 2.4. A 2-dimensional table for each attribute, where rows represent a relational operator and columns contain the possible values, acts as an index on predicates to efficiently compute the satisfied predicates by an incoming event. Each predicate is uniquely identified and mapped to a dynamic predicate bit vector that establishes if a predicate is satisfied (bit to 1) or not (bit to 0) by the event currently being processed. Subscriptions with the same access predicate and number of predicates are grouped in clusters implemented as a matrix of size \((subscribersize + 1) \times (numbersubscriptionscluster)\), where each column contains a subscription’s unique identifier and respective references to its predicates in the predicate bit vector. Since subscriptions with different sizes can have the same access predicate, an access predicate is associated to a list of clusters of different sizes, defined in this context as a cluster family. These cluster families are maintained in a \textit{cluster vector} where each index corresponds to an access predicate and contains its correspondent cluster family.

The execution flow of predicate-based clustering algorithm is as follows: when an incoming event arrives to the matching engine, in the 1\textsuperscript{st} phase, event predicates \((attribute, value)\) are processed over the indexes on predicates to compute satisfied predicates. When a predicate is matched the corresponding bit in the predicate bit vector is set to 1. In the end of the 1\textsuperscript{st} phase all satisfied predicates have been discovered, i.e., predicates with bit set to 1 in the predicate bit vector and, only the subscriptions of this satisfied predicate that correspond to access
predicates are in fact evaluated. In the 2nd phase, the list of satisfied predicates is traversed and predicates corresponding to access predicates trigger the evaluation of its clusters with size less or equal to the event’s predicates. A subscription is considered a match if all its predicates bits are set to 1. In this case, the subscription’s identifier is added to the list of matched subscriptions. After all subscriptions matched have been discovered, all indexes of the predicate bit vector are reseted to 0.

![Data structures for a predicate-based clustering algorithm](image)

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 (Pereira, Fabret, Jacobsen, Llirbat, and Shasha 2001) that translates XML event paths to a set of \((\text{attribute}, \text{value})\) pairs to be further processed using techniques employed in Le Subscribe (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000). In (Hou and Jacobsen 2006) and later on GPX-Matcher (Sadoghi, Burcea, and Jacobsen 2011), a solution with a complete XPath subset is proposed by adapting the clustering algorithm presented in (Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001). In these solutions, XPath subscriptions are translated to sets of \((\text{attribute}, \text{operator}, \text{value})\) tuples, whereas each path of an XML event is converted to a set of \((\text{attribute}, \text{value})\) pairs.

Additionally to value-based predicates, 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 detection of false positives, result in more difficult to evaluate and index predicates. 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.

More recently, an XML matching algorithm able to efficiently convert the XML matching problem into a simple predicate matching problem was proposed in DeltaFilter (Martins, Pereira, and Wichert) where subscriptions are processed over events as simple binary operations. The best way to obtain an efficient XML matching algorithm is by reducing the search space through the exploitation of similarities between XPath subscriptions, this is obtained in DeltaFilter by using the unidimensional solution of Le Subscribe (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000). The access predicate selectivity criteria in DeltaFilter is based on the fact that the more deeper in the XPath subscription the more selective, since if a subscription is matched this location step is definitely verified. It also takes advantages of cache-conscious data structures presented in Le Subscribe (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000), the clustering criteria presented in (Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001) and a sophisticated holistic handling of both structural and attribute location steps. As the authors demonstrated, DeltaFilter (Martins, Pereira, and Wichert) is not only up to 45 times faster than YFilter, but also uses up to 18 times less memory. This results from the evaluation being done only to candidate subscriptions through very fast binary operations. An overview of this system is presented in more detail in section 3.1.

2.2.2 Parallel Predicate-based

Farroukh et al. proposed in (Farroukh, Ferzli, Tajuddin, and Jacobsen 2009) three parallel techniques for the event processing of the matching engine presented in (Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001): Multiple Event Independent Processing (ME-IP), Single Event Collaborative Processing (SE-CP) and Multiple Event Collaborative Processing (ME-CP). In the Multiple Event Independent Processing (ME-IP) technique, threads process events separately in parallel in order to increase system throughput, by reducing the total matching time. Each thread gets an event and executes phases 1 and 2 as it happens in the sequential solution. In that sense, in phase 1 each thread computes its list of satisfied predicates by updating its local
2.2. INDEX-BASED

bit vector, whereas in phase 2 this list of satisfied predicates is traversed and evaluated over
the bit vector, in order to obtain the matched subscriptions. This technique requires minimal
synchronization only related to the concurrent accesses to the global structure storing unpro-
cessed events. Figure 2.5 illustrates this technique with 4 threads, each one processing an event
independently of the others.

Figure 2.5: Multiple Event Independent Processing

In the Single Event Collaborative Processing (SE-CP) technique, threads cooperate in the
processing of an event in order to reduce the average matching time of that event. Recall that
the subscription language of this matching engine is content-based therefore, events are ex-
pressed as a conjunction of \((\text{attribute, value})\) pairs. In that sense, in the 1\(^{st}\) phase each thread
evaluates a pair and updates its local bit vector according to the assigned pair. When all event’s
pairs have been evaluated, each thread updates a global bit vector according to their local
one. At this point a synchronization barrier is introduced, since threads must wait for all other
threads to merge their local bit vectors, before proceeding to the next phase. After all bit vectors
are merged into the global bit vector, 2\(^{nd}\) phase takes place by distributing the list of satisfied
predicates by all thread in order for each thread to evaluate a subset of the candidate clusters
over the global bit vector. Another synchronization barrier is introduced at this point since all
threads must wait for all the partial results of the 2\(^{nd}\) to be merged, before starting the process-
ing of a new event. The satisfied predicate list division results in poor load balancing, since a
thread can get a satisfied access predicate with a numerous set of candidate clusters, whereas
another thread can get a predicate that does not constitutes an access predicate. The poor load
balancing and synchronization requirements greatly impact performance and scalability for
this technique. Figure 2.6 illustrates this technique with 4 threads, where in the 1\(^{st}\) phase each
thread gets a single pair and in the 2\(^{nd}\) phase each thread evaluates the cluster family of the
assigned access predicate.
Last but not least, in the Multiple Event Collaborative Processing (ME-CP) technique, a hybrid combination of the previous two techniques takes place. More concretely, threads are divided into groups and, a single event is assigned to each group. The parallelization levels can be easily customized, depending on the requirements of the events, in terms of priority and resources. Because it is a hybrid of the previous techniques, it is able to increase system throughput, since several events are processed in parallel, while reducing average matching time, since inside each group a single event is validated concurrently.

Since the data structures are shared, three synchronization paradigms have been studied: static, lock, or dynamic and, Software Transactional Memory (STM).

In the static-based approach, it is known a priori, the number of events/predicates each processor will be responsible for. The distribution is based on the number of events/predicates and not on the processing time. This approach is lock-free and requires minimal synchronization. The main downside of this approach is the case of unbalanced workloads. For instance, some threads may finish before others and, therefore, must wait for them in order to continue.

In the lock-based, or dynamic-based, approach, each thread processes the next available part of the workload, called chunks. As a result, the distribution is based on the processing time, instead of the number of events/predicates. The downside of this approach is that it is necessary to have shared counters for the available chunks and locks to ensure that only one thread updates the counter in an instance of time. Compared with the static-based approach, the lock-based achieves better resource utilization, but it has a bigger overhead for the data synchronization of the shared variables.

Finally, a technique that has been proposed as an alternative to lock-based synchronization
is STM. In STM systems, read and write operations over shared variables are executed as transactions. Each transaction executes their own operations independently but, before committing it is checked if a conflict occurred, i.e. if a variable accessed by a transaction was modified by a different transaction. If a conflict occurs the transaction is aborted and all changes are rolled back, otherwise the transaction is committed successfully. Due to this properties, STMs have recently been used as a way to facilitate the parallelization of applications (e.g. (Marathe, Scherer, and Scott 2005)(Olszewski, Cutler, and Steffan 2007)(Fahmy, Ravindran, and Jensen 2009)).

2.3 Sequence-based

The first algorithm to introduce sequence-based matching was FiST (Kwon, Rao, Moon, and Lee 2005) which focuses on the processing of twig patterns, i.e., nested path expressions. Contrarily to the query decomposition technique used by previous algorithms (e.g. YFilter (Diao, Fischer, Franklin, and To 2002), XTree (Chan, Felber, Garofalakis, and Rastogi 2002)), 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 (Moro, Bakalov, and Tsotras 2007), iFiST (Kwon, Rao, Moon, and Lee 2009)), or by employing a holistic processing of twig patterns (e.g., XFiS (Antonellis and Makris 2008), pFiST (Kwon, Rao, Moon, and Lee 2008)).

2.4 Others

In the database community, researchers have tried to employ relational database techniques in order to solve the matching problem. In (Ashayer, Leung, and Jacobsen 2002) the authors proposed two DBMS-based matching algorithms where subscriptions are stored and manipulated as database relations. For each event, an SQL query performs the required matching operations outputting the set of matched subscriptions. In (Tian, Reinwald, Pirahesh, Mayr, and Myllymaki 2004) and (Zhao, Yang, Gao, and Wang 2007) two relational database algorithms that exploit database operators for structure and value-based matching of incoming XML documents are presented. Furthermore, in the database research area, Continuous
Queries (CQ) (e.g., NiagaraCQ (Chen, DeWitt, Tian, and Wang 2000)) and Triggers (e.g., TriggerMan (Hanson, Al-fayoumi, Carnes, Kandil, K, Liu, Lu, Park, and Vernon 1997)) have also evolved to support XML. These techniques are based on the concept of storing user subscriptions as database conditions which are continuously running and when an operation occurs in the database interested users are notified. The aforementioned solutions come as an alternative to specialized data structures and, despite being easier and cheaper to implement, are unable to scale in scenarios with a large number of stored subscriptions and high rates of incoming events.

Parallel applications are very machine dependent so, by implementing the logic of matching engines directly into hardware, the main software overheads, such as communication and synchronization, do not impact the implementation, allowing for better performance results. One non-traditional highly parallel hardware architecture that has, recently, been made available, is Field Programmable Gate Arrays (FPGA). FPGAs are integrated circuits with programmable logic components, Configurable Logic Blocks (CLBs), connected through programmable interconnections, Switched Boxes (SB). In an overall perspective, FPGAs can be viewed as chips containing thousands of processing units (CLBs) that communicate over SBs.

An important area of research on FPGAs has been on implementing regular expressions on CLBs (e.g., (Sidhu and Prasanna 2001), (Mitra, Najjar, and Bhuyan 2007)). In (Mitra, Najjar, and Bhuyan 2007), a method that compiles PERL Compatible Regular Expressions operation codes directly to VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) for parallel implementation was presented. Towards the XML matching problem, Mitra et al. proposed in (Mitra, Vieira, Bakalov, Najjar, and Tsotras) a pure hardware solution, where the logic of each subscription is deployed directly into a single CLB. This way, multiple queries can be processed over a single XML document. For this to be possible, XPath expressions must be translated to PCRE forms, which are implemented in CLBs as regular expression state machines.

As pointed out by the authors, a limitation of this approach is that it is unable to handle XPath subscriptions with recursive elements and wildcards. In order to overcome this, the authors proposed a new FPGA solution (Moussalli, Salloum, Najjar, and Tsotras 2010). In this approach, each XPath subscription is mapped to a customized 2-dimensional stack, defined as Path Specific Stack (PSS). The size of the PSS depends on the size of each XML documents and
each subscription, i.e., its maximum height is the same as the depth of an XML document and, its width is equal to the size of the XPath subscription, where each location step is mapped to a single column. When the SAX parser generates a `startElement()` event, a `push()` operation is performed and the bits of all satisfied location step are set to 1 in the new PSS level. On the other hand, when an `endElement()` event is triggered, a `pop()` operation is carried out. An example of the state of the PSS throughout the matching process of XPath expression `a/c/a/c/b` given the document `XML doc` is illustrated in figure 2.7. Notice that, because `a/c/a/c/b` is a relative location path, more than one match of the same XPath are possible. By storing all location step occurrences at each level in the PSS, this solution guarantees that more than one path can be processed concurrently.

Figure 2.7: Stages of the Path Specific Stack (PSS) when evaluating event `XML doc` over subscription `a/c/a/c/b`

In both presented FPGA version, after the subscriptions are offloaded to the FPGA, the system can become online and start processing events over the installed subscriptions. Whenever a new event arrives, the document is streamed over the SBs, through the I/O blocks and, all subscriptions implemented in CLBs are evaluated over the event in a parallel fashion. Therefore, the only supported level of parallelism for these devices is the parallel processing of a single event.

In conclusion, FPGAs are very flexible devices in the sense that all of its components can be programmed by the user in order to implement specific logic. However, FPGAs are unable to update, remove, or even add new subscriptions in an online fashion, i.e., the entire system needs to be re-compiled, a process that can take up to several hours.
Recently, there has been an exponential increase in the computational power of GPUs over CPUs. Furthermore, vendors are now offering APIs for GPU programming, e.g., CUDA of NVIDIA. Therefore, a new programming model was created, General Purpose Graphics Processing Unit (GPGPU), that allows the use of GPUs for general purpose computation (e.g., (Kim, Chhugani, Satish, Sedlar, Nguyen, Kaldewey, Lee, Brandt, and Dubey 2010), (Ao, Zhang, Wu, Stones, Wang, Liu, Liu, and Lin 2011)).

The main architectural differences between GPU and CPU, is that in CPU a big area is responsible for control units and cache and, of course, an area to Arithmetic Logic Units (ALU), whereas in the GPU most of its area is composed of computation units and small areas devoted to control units and cache.

GPUs are an example of a parallel architecture of a fine-grained Single Instruction Multiple Data (SIMD) (Flynn 1972) parallelism, containing large data arrays. In this type of architecture all processing units execute the same instruction but over independent data. The main elements of a GPU architecture are depicted in figure 2.8. GPUs are constituted by a set of $M$ Streaming Multiprocessors (SM) which are clusters of $N$ Streaming Processors (SP). All SPs within one SM execute instruction from the same memory block. SMs include a shared memory, which is used for the communication between SPs inside the same SM and, a constant cache. On the other hand, communication between SPs from different SMs is achieved through the global memory (Moussalli, Najjar, Halstead, Tsotras, and Salloum 2011).

The GPU-based solution for the XML matching problem in (Moussalli, Najjar, Halstead, Tsotras, and Salloum 2011) is based in the algorithm of the previous FPGA version (Moussalli, Salloum, Najjar, and Tsotras 2010). A stack-based approach is used where each query is mapped to a 2-dimensional stack and location steps are mapped to a dimension of the stack. As it happens in the previous version, the top of the stack (TOS) is updated with every `startElement()` and `endElement()` events generated by the SAX parser. In this system, two levels of parallelism are used: \textit{inter-query} parallelism, where all queries are processed in parallel by different SMs and \textit{intra-query} parallelism, where each state of location steps of a query, i.e., each column of the 2-dimensional TOS, are updated concurrently within SPs of the same SM.

Contrarily to what happens in (Moussalli, Salloum, Najjar, and Tsotras 2010), where the XML documents are directly streamed to the engine, here the XML documents have to be stored in the GPU memory, since it is processed by software implementations of the stack. To achieve
2.5. Summary

In this chapter, we presented the main categories for the sequential XML matching problem: (1) automaton-based, where subscriptions are stored as state machines and SAX events trigger transitions between states; (2) Index-based, where subscriptions exploit high performance indexes; (3) sequence-based focus on processing twig patterns against incoming XML documents; and (4) all other approaches.

The state of the art algorithm for the XML matching problem, YFilter (Diao, Fischer, Franklin, and To 2002) is presented. Due to the popularity of this algorithm some parallel
approaches have been proposed in recent years, both centralized solutions (Zhang, Pan, and Chiu 2010) (Antonellis, Makris, and Pispirigos 2012) and distributed solutions (Miliaraki and Koubarakis 2010). These approaches are compared and analyzed in detail.

The parallel matching algorithm for content-based publish/subscribe systems proposed in (Farroukh, Ferzli, Tajuddin, and Jacobsen 2009) is presented. As parallel algorithms are very machine dependent, some hardware solutions based on FPGAs and GPUs are also considered.
Redesigning your application to run multithreaded on a multicore machine is a little like learning to swim by jumping into the deep end.

– Herb Sutter

### 3.1 DeltaFilter Overview

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 LeSubscribe (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000) and the efficient clustering of (Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001). 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. Furthermore, since DeltaFilter handles value-based predicates holistically, these predicates are considered refinements to a path structure and are, therefore, stored at the same depth of the respective parent tag.

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*. Figure 3.1 illustrates DeltaFilter’s representation of the internal events of the XML tree illustrated in figure 1.3a.

As previously stated in section 1.1, 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*, which are now presented.
3.1.1 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 the nodes and the relationship between them, as discussed in section 2.2.1 regarding adaptation of the content-based matching problem to an XML one, in DeltaFilter an XPath expression is converted to a simple binary operation represented as a set of operators over position records. The correlation between DeltaFilter’s operators and position records and, the XPath syntax can be defined as:

- **position records** correspond to XPath’s elements, or attributes, nodes to which an axis is applied;
- **operators** correspond to XPath’s parent or ancestor axes.

### 3.1.1.1 Position Record

A position record is responsible for storing at which depth an unique tag \( \text{pos}_{\text{tag}} \), an unique root tag \( \text{root}_{\text{tag}} \) or an unique attribute \( \text{pos}_{\text{tag}}[@\text{attribute} \text{ operator value}] \) 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\(^1\). Thereby, frequent XML matching problems, such as evaluation of recursive tags, value-based predicates and wildcards, are dealt efficiently and without employing complex data structures. However, this solution

\(^1\)Section 3.1.2 explains in more detail how these bits are set throughout the parsing of an XML document.
leads to a potential problem, where it is not possible to evaluate internal events with a depth higher than 63 but, as pointed out in (Mignet, Barbosa, and Veltri 2003), about 99% of the documents on the web have less than 8 levels of depth.

**Position Record Vector.** 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 \( p_i \). The first two positions of the position record vector are reserved for two important position records: (1) wildcard root \( (p_0) \) and (2) wildcard position record \( (p_1) \). These two position records must be added to the position record vector in order to support subscriptions containing wildcards at any level of depth. Because the wildcard operator matches *any* element name, when an internal event with depth \( n \) is discovered, all the \( n \) lower bits of \( p_1 \) are set to 1, as to say that at each level of depth lower than \( n \) there is a possible match.

**Position Record Indexes.** Three main indexes are used for a fast retrieval of a position record’s identifier to which a 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 the value is 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 \((tag, attribute, operator, value)\). With that in mind, two auxiliary indexes, index \( A \) and \( P \), are constructed in combination with the main index \( I \). Firstly, index \( A \) creates an unique attribute identifier \( a_{id} \), by combining the tag name with the attribute name, as represented in expression 3.1.

\[
A(tag, attribute) = a_{id}
\] (3.1)

Secondly, index \( P \) creates the attribute position record’s unique identifier \( p_{id} \) by combining the attribute identifier \( a_{id} \), resulting of index \( A \), with the attribute’s operator and value, as expressed in expression 3.2.

\[
P(a_{id}, operator, value) = p_{id}
\] (3.2)

Root, tag, \( A \) and \( P \) indexes are illustrated in figure 3.2 for the XPath expressions presented in figure 1.3b. Root and tag indexes store the position record identifiers \( p_{id} \) of tag names. Index \( A \) stores the attribute identifiers \( a_{id} \) of \((tag, attribute)\) pairs, which is then used as a key for index \( P \). Index \( P \) stores the attribute position record identifiers \( p_{id} \) of \((a_{id}, operator, value)\) tuples.
CHAPTER 3. DELTAFILTER

Figure 3.2: Indexes of the XPath subscriptions presented in figure 1.3b.

Index $I$ is a multi-level index that maps the $p_{id}$s of attribute position records with the corresponding attribute elements of both indexes $A$ and $P$. DeltaFilter supports value-based predicates of exact (operator $=$) and range mode (operators $>$, $\geq$, $<$, and $\leq$) therefore, for an efficient processing when evaluating attributes, index $I$ consists of five 2-dimensional low-level indexes $I_{op}$, one for each operator. 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. This way, for attribute position records in the exact mode, the $a_{id}$ constitutes a key for an hash table of pairs $(value, p_{id})$ and, $value$ is used as key to lookup the respective $p_{id}$. For attribute position records in the range mode, the $a_{id}$ constitutes a key for a vector of pairs $(value, p_{id})$, ordered by $value$ and, $value$ is used as key to lookup the range of verified $p_{id}$s. Figure 3.3 illustrates how the multi-level index $I$ is organized with the position records present of figure 3.2. Low level indexes of the exact mode, operator $=$, store $(value, p_{id})$ pairs in an hash table, whereas low level indexes in the range mode, operators $>$, $\geq$, $<$, $\leq$, store $(value, p_{id})$ pairs in an ordered vector.

Access Position Record. Until now it was demonstrated how position records are stored in DeltaFilter but, not how the system is aware that a set of position records composes a subscription. 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. Several clustering criteria have been studied in previous matching algorithms (e.g., LeSubscribe (Pereira, Fabret, Llirbat, Preotiuc-Pietro, Ross, and Shasha 2000), (Fabret, Jacobsen, Llirbat, Pereira, Ross, and Shasha 2001)). In large scale systems there is a high probability of common-
3.1. DELTAFILTER OVERVIEW

Figure 3.3: Multi-level index $I$ of the position records illustrated in figure 3.2.

...
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. This way, a single cluster can be uniquely identified by a 2-tuple \((\text{access position record}, \text{subscriptions size})\). Each cluster is implemented as a matrix, where subscriptions are stored column-wise, i.e., column \(i\) contains all the \(p_{id}\)'s of subscription \(i\) with the last row containing the subscription’s id. Hence, each cluster has \(\text{subscriptions size} + 1\) rows (+1 represents the subscription’s identifier) and \(\text{number subscriptions cluster}\) columns, resulting in a matrix of size \((\text{subscriptions size} + 1) \times (\text{number subscriptions cluster})\). 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 but, 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 \(p_i\). As a result, 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.

### 3.1.1.2 Operators

Operators express how position records are related to represent a subscription. Since position records are binary numbers, the logic implementation for operators is binary operations. Recall that the bits set to 1 in a position record 64-bit number correspond to the depths at which a unique tag name occurred in an internal event. Since DeltaFilter supports three XPath axis (parent, ancestor and attribute) three operators can take place (1) parent, (2) ancestor and (3) attribute operators, which are now explained in more detail. For each subscription, when the combination of the operators over the position record is different than 0, it means that the subscription was matched at a certain level of depth.

**Parent operator.** Expression 3.3 corresponds to the XPath’s expression \(a/b\) and is responsible for finding out if tag \(b\) immediately follows \(a\), i.e., if it is at the next level of depth. This operation receives the position records of tags \(a\) and \(b\), and returns a 64-bit number with the bits set to 1 corresponding to the positions in the internal event where tag \(b\) follows tag \(a\).
3.1. DELTAFILTER OVERVIEW

\[ \text{pos}_a \oplus \text{pos}_b := (\text{pos}_a \ll 1) \& \text{pos}_b \] (3.3)

To exemplify, consider the subscription 0 (\(S_0 = /A/ * /A\)) of figure 1.3b and internal event 3 (\(IE_3 = <A \text{ attr1 = "31" } <C > <A>\)) of figure 3.1. For simplicity assume a depth of 4.

\[
\begin{align*}
\text{root}_A &= 0001 \\
\text{pos}_A &= 0100 \\
p_1 &= 0111, \text{ because } \text{depth}(IE_3) = 3
\end{align*}
\]

\[
\begin{align*}
S_0 &= \text{root}_A \oplus p_1 \oplus \text{pos}_A \\
S_0 &= 0001 \oplus 0111 \oplus 0100 \\
S_0 &= (0001 \ll 1) \& 0111 \oplus 0100 \\
S_0 &= ((0001 \ll 1) \& 0111) \oplus 0100 \\
S_0 &= 0010 \oplus 0100 \\
S_0 &= (0010 \ll 1) \& 0100 \\
S_0 &= 0100
\end{align*}
\]

\(S_0 \neq 0 \Rightarrow \text{Matched}\)

**Ancestor operator.** Expression 3.4 corresponds to the XPath’s expression \(a//b\) and is responsible for finding out if tag \(b\) follows tag \(a\), at any level of depth. This operation receives the position records of tags \(a\) and \(b\), and returns a 64-bit number with the bits set to 1 where \(b\) is at a higher level of depth than \(a\). Method \(\text{mask()}\) returns a 64-bit number that contains all bits set to 0 from the least significant bit to the first bit set to 1 and, all following bits set to 1. The \(\text{mask()}\) operation has some costs, but it is efficiently implemented by using binary operations.

\[ \text{pos}_a \otimes \text{pos}_b := \text{mask(\text{pos}_a)} \& \text{pos}_b \] (3.4)

To exemplify, consider the subscription 1 (\(S_1 = /A//A\)) of figure 1.3b and internal event 3
(IE₃ = < A attr1 = "31" > < C > < A >) of figure 3.1. For simplicity assume a depth of 4.

\[
\begin{align*}
\text{root}_A &= 0001 \\
\text{pos}_A &= 0100 \\
S_1 &= \text{root}_A \otimes \text{pos}_A \\
S_1 &= 0001 \otimes 0100 \\
S_1 &= \text{mask}(0001) \& 0100 \\
S_1 &= 1110 \& 0100 \\
S_1 &= 0100 \\
\text{if } S_1 \neq 0 \Rightarrow \text{Matched}
\end{align*}
\]

**Attribute operator.** Expression 3.5 corresponds to the XPath’s expression \(a[@c \text{ op } \text{val}]\), where \text{op} refers to one of the operators \(>, \geq, <\) or \(\leq\) and \text{value} refers to an integer value. This expression is responsible for finding out if attribute \(c\) has \(a\) as parent tag, i.e., occurs at the same level of depth. This operation receives the position records of tag \(a\) and attribute \(a[@c \text{ op } \text{val}]\), and returns a 64-bit number with all the bits set to 1 where \(c\) is at the same level of depth as \(a\).

\[
pos_a \bullet pos_a[@c] := pos_a \& pos_a[@c \text{ op } \text{val}] \tag{3.5}
\]

To exemplify, consider the subscription 3 (\(S_3 = //B[@attr2 < 90]\)) of figure 1.3b and internal event 2 (\(IE_2 = < A \text{ attr1 = "31" } > < B \text{ attr2 = "14" } > 90\)) of figure 3.1. For simplicity
assume a depth of 4.

\[ pos_B = 0010 \]
\[ pos_{B\text{attr2}} = 0010 \]
\[ S_3 = pos_B \oplus pos_{B[\text{attr2}="14"]} \]
\[ S_3 = 0010 \bigoplus 0010 \]
\[ S_3 = 0010 \& 0010 \]
\[ S_3 = 0010 \]
\[ S_3 \neq 0 \Rightarrow \text{Matched} \]

In order to reduce time and space complexity, operators are stored implicitly in clusters and in the position record vector:

- Parent operator \( pos_a \oplus pos_b \): default operator where no flag is set.

- Ancestor operator \( pos_a \otimes pos_b \): for this operator, the position record identifier \( p_b \) is stored in the cluster with a negative identifier, i.e., \(-p_b\).

- Attribute operator \( pos_a \ominus pos_{a[c \text{ op val}]} \): all position records concerning attributes matched by the event have the leftmost bit set to 1.

Figure 3.4: Cluster Vector and cluster family of access position record \( p_3 \) of the first three subscriptions of example of figure 1.3b.
3.1.2 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, updating position records and evaluating stored subscriptions. Recall that position records are dynamic structures that must be updated at every tag parsed of the event. Therefore, the recommended parser is an event-driven one, such as the SAX parser, as explained in section 1.2. Since subscriptions are expressed as XPath path 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 1st phase constitute the access position records that need to be evaluated. All cluster families whose access position record is not verified can be ignored.

These two phases are now described in more detail.

3.1.2.1 Position Record Update

Before parsing starts, it is necessary to add the wildcard position record to the list of satisfied position records in order to obtain subscriptions that contain only wildcards, i.e., with access position record the wildcard position record $p_1$. Additionally, the position record vector must be initialized with the values of the global position record vector, i.e. position records with all bits set to 0, except the leftmost bit of attribute position records. Subsequently, the event is parsed and managed through the `startElement(tag, attributes), characters(string)` and `endElement(tag)` handlers. Consider $depth$ the depth at which a parser event was triggered.

`startElement(tag, attributes)`. When a `startElement` event is triggered the position record $pos_{tag}$ and/or the respective attribute position records of $attributes$ need to be updated by set-
ting their \( depth^{th} \) bit to 1. Firstly, if \( depth \) is 1, it is checked if a \( root_{tag} \) already exists in the root index and then, it is checked if a \( pos_{tag} \) exists in the tag index. For both cases, if the position record exists then a match at the \( depth^{th} \) level is added to \( pos_{tag} \). In the case these position records are matched for the first time, they are also added to the list of satisfied position records.

Secondly, for each attribute it is checked if an attribute identifier exists for \((tag, attribute)\) tuple through index \( A \). If the attribute identifier \( (a_{id}) \) is present in the engine, the entries of the multi-level index \( I \) corresponding to the supported operators are traversed. Subsequently, as explained in section 3.1.1.1, for each operator, the \( a_{id} \) constitutes a key for a low level index and the value of the attribute is used as a key to look up the respective \( p_{id} \). This operation returns all the attribute position records satisfied by the current internal event and, therefore, a match at their \( depth^{th} \) bit is added. As it happens with tag position records, if an attribute position record is matched for the first time it is also added to the list of satisfied position records. This procedure also takes place for attribute position records whose tag is the wildcard operator \((*)\).

If no satisfied position record is found, the parsed elements do not represent position records stored in the system.

\texttt{characters(string)}. When a \texttt{characters(string)} event is triggered a value predicate is discovered between a start and a close tag, so the tag of the previous start element must be maintained. In this case, the same procedure regarding the identification of attribute position records of the previous handler is employed with the predefined attribute associated with this type of attributes \((text())\).

\texttt{endElement(tag)}. When an \texttt{endElement(tag)} event is triggered, after a \texttt{startElement} event, it means an internal event has been discovered in the XML document. As such, all the position records satisfied by this internal event have been matched and the algorithm can proceed to the 2\textsuperscript{nd} phase called Subscriptions Evaluation, where this set of satisfied position records is traversed and the associated clusters are evaluated. The simplest implementation consists in evaluating all satisfied position records of a given internal event however, because the access position record choice lies in the last position record of subscriptions, an interesting property arises concerning satisfied position records shared by different internal events:

\textit{Except subscriptions ending with the wildcard position record \( p_{1} \), only the new satisfied position records need to be evaluated, since the ones shared by previous internal events have already been evaluated. In cases where a wildcard operator is present in the last position record, two situations}
occur: (1) an internal event is deeper than a the previous one, case where all satisfied position records have to be evaluated; (2) an internal event is less deep than a previous one, case where only the new satisfied position records need to be evaluated.

Taking this property into account, three different Filter algorithms are presented regarding the evaluation of satisfied position records:

- **Filter**: for each internal event, evaluates all its satisfied position records.

- **DeltaFilter**: an optimization of the Filter algorithm consists in verifying if the current internal event is deeper than its previous one, case where all satisfied position records of this internal event are evaluated. Alternatively, if an internal event is less deep than its previous one, the satisfied position records of the common path do not need to be evaluated once more.

- **OptimizedDeltaFilter**: employs the same optimization as DeltaFilter, but taken one step further, in the sense that instead of comparing only with the previous internal event, compares with the deeper internal event found to the moment.

### 3.1.2.2 Subscriptions Evaluation

First, it is necessary to initialize the wildcard position record \(p_1\) taking into account the depth \(depth\) of the concerned path, setting all its \(depth^{th}\) lower bits to one. By doing this, we are able to match subscriptions containing wildcards. Then, with all the satisfied position records discovered in the 1\(^{st}\) phase we are able to filter which cluster families have to be evaluated, narrowing this way the search space. This justifies why the most selective access position record should be used and why the best solution lies in the last position record of a subscription, because a subscription is only satisfied if this position record is matched. For each of the satisfied position records obtained in the 1\(^{st}\) phase, if it corresponds to an access position record all its clusters whose subscriptions’ size is smaller than \(depth\) of the current internal event, are evaluated.

As previously stated, subscriptions are stored in cache-friendly clusters. These clusters were designed in order to obtain better performances when traversing its subscriptions. By storing the subscriptions column-wise, cache locality is promoted and cache misses reduced
since, when a position record of a subscription is read, $K$ elements (value $K$ depending of the system’s cache mechanism) at the same row are loaded into cache. When evaluating a subscription, if a given position record is not verified the evaluation of this subscription is aborted and the evaluation of the next one begins. When a subscription is evaluated till the end and the result of applying the operators to the position records is different than 0, this subscription is considered a match and is added to the list of subscriptions matched. This way, the result of an internal event consists in all the subscriptions that were matched by the associated XML path. As such, the total set of subscriptions matched by an event is the combination of all the partial subscriptions resulting from the evaluation of each internal event.

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 $depth^{th}$ level 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.

Figure 3.5 illustrates the Filter structure when evaluating the third internal event illustrated in figure 3.1. In the 1$st$ phase, Position Record Update, the satisfied position records ($spr$) are stored and the position record vector ($prv$) is updated according to the SAX events triggered. In the 2$nd$ phase, Subscriptions Evaluation, the satisfied position records computed in the 1$st$ phase are verified in the cluster vector and the corresponding clusters with size less or equal than the internal event depth, are evaluated using the values of the updated position record.
3.2 Parallel Event Processing

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. However, the use of this technique arises two main concerns: (1) task decomposition and (2) task granularity. Theoretically, tasks can be decomposed up to instruction level nevertheless, very fine-grained tasks decrease performance, as the parallel overheads, such as task creation and synchronization costs, do not make up for the actual work. On the other hand, very coarse-grained tasks result in more unbalanced workloads as the probability of tasks with very different sizes and complexities is higher.

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 (Farroukh, Ferzli, Tajuddin, and Jacobsen 2009) 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.

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 (Farroukh, Ferzli, Tajuddin, and Jacobsen 2009) 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 the evaluation of a candidate cluster family. As it happens in the previous levels, a cluster family can further be decomposed into clusters subtasks, which can finally be decomposed into subscription subtasks\(^2\). Although all these parallelism levels can be considered, the presented study only delves into the satisfied position record evaluation subtasks since, as demonstrated in section 4.2.3, from this level on the granularity of tasks is so fine that already limits, or even decreases, performance. The event processing parallel approaches ME-IP, SE-CP and each of their associated parallelism levels are illustrated in figure 3.6.

![Figure 3.6: Parallelism levels with the respective event processing approaches decomposition and granularity.](image)

All these parallelism levels can be used independently, i.e., only one level is used (e.g. only the first level is used), or in conjunction, i.e. combining several levels (e.g. using the first and the second levels). The latter represents the final parallel approach presented, defined as Multiple Event Collaborative Processing (ME-CP) (Farroukh, Ferzli, Tajuddin, and Jacobsen 2009), where events are processed in parallel by group of threads.

For each parallelism level a centralized dynamic load balancing mechanism is accomplished by using a Thread Pool Pattern, where threads are continuously checking the task queue for tasks to process. Concerning the addition of tasks to the queue a simple `add(task)` operation is performed. For the removal of tasks from the task queue, two operations with different behaviors concerning the impossibility to remove a task are possible: `remove()` and `take()`. The `remove()` operation constitutes a non-blocking procedure, i.e., if no tasks are available in the queue, the thread continues its execution flow. On the other hand, the `take()` operation constitutes a blocking procedure, i.e., if no tasks are available in the queue, the thread blocks switching to a `WAITING` state, until it is able to successfully remove a task from the queue.

\(^2\)Decomposition can still go further on till instruction level. However, this level of parallelism is out of the scope of this thesis.
Moreover, all task queue operations need to be synchronized in order to ensure consistency among threads.

### 3.2.1 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 (eventQueue) 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.

![Figure 3.7: Multiple Event Independent Processing (ME-IP) approach.](image)

In an algorithmic perspective, the processing of a single event in this context is equivalent to the sequential algorithm presented in section 3.1.2. As such, for each internal event, in the 1\textsuperscript{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\textsuperscript{nd} phase
3.2. PARALLEL EVENT PROCESSING

It takes place, using the satisfied position records computed in the 1st phase to evaluate the candidate cluster families and obtain the matched subscriptions 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 3.7 illustrates the work flow of the ME-IP technique and algorithm 1 demonstrates filter threads execution.

Algorithm 1 Filter Thread execution

1: function FILTER_THREAD::RUN
2: while state ≠ FINISHED do
3: event ← eventQueue.take() ▷ If no events are available, block until new events are added to the queue
4: filter.PROCESS_EVENT(event)
5: end while
6: end function

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, which is now described in more detail.

3.2.2 Single Event Collaborative Processing

As demonstrated by the authors in (Martins, Pereira, and Wichert) the 1st phase of the matching algorithm, position record update, is in fact fast and the real demanding computation lies in the 2nd phase, subscription evaluation. With that in mind, the SE-CP approach parallelizes the 2nd phase of DeltaFilter’s event processing algorithm. Two main parallelization levels for SE-CP are presented: (1) internal event level (2nd level) dependent of the XML structure of events, where a task corresponds to the processing of an internal event and (2) satisfied position record level (3rd level) dependent of the position records satisfied by an internal event, where a task corresponds to the evaluation of candidate cluster families associated to a satisfied access position record.
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 1st phase of the matching algorithm and creating 2nd or 3rd level tasks, depending on the parallelization level. Additionally, a set of worker threads is responsible for executing the 2nd phase of the matching algorithm, by processing 2nd or 3rd level tasks.

![Diagram of Single Event Collaborative Processing (SE-CP) approach.](image)

The main thread starts the processing of an event by executing the 1st phase of the matching algorithm and, when an internal event is discovered, instead of progressing to the 2nd phase, it creates and adds 2nd/3rd level 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 1st phase of the remaining internal events and, consequently updating these structures, as worker threads execute the 2nd phase of previously created tasks. The result of processing these tasks constitute partial sets of subscriptions matched by the current event.
3.2. PARALLEL EVENT PROCESSING

While the main thread is performing the 1st phase and creating new tasks, worker threads are continuously checking the task queue to process 2nd or 3rd level tasks. This processing corresponds to the 2nd phase of the matching algorithm so, worker threads are responsible for computing in parallel matched subscriptions of individual 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, threads add its set of subscriptions matched ($psm$) to the $psmQueue$ queue and reset its $psm$ structure. 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 $psmQueue$ 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. Furthermore, the cost of the merge operation is directly related to the Filter algorithm employed, since the merge can be slower if several subscriptions are duplicated among partial results (case of simple Filter algorithm) or faster if only a few subset of partial results overlap (case of OptimizedDeltaFilter algorithm). Nevertheless, the merge operation is still, usually, faster than the 2nd 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 $psmQueue$ is not empty, the main thread merges partial results to the set of final subscriptions matched ($sm$). When all the partial subscriptions matched have been merged and all worker threads are in a waiting state, i.e., worker threads are waiting for new tasks, 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 3.8 for both 2nd and 3rd levels, i.e., internal event and satisfied position record levels, respectively.

3.2.2.1 Internal Event Level

In the internal event level, 2nd level, of the SE-CP approach, when an internal event is discovered, additionally to bookkeeping information featuring internal event properties, an
internal event task is created and added to the task queue \textit{ieQueue}. This internal event task is uniquely identified and contains information regarding the state of the internal event, i.e., depth ($d$), position record vector ($prv$) and the list of satisfied position records ($spr$). When a worker thread removes an internal event task from \textit{ieQueue}, processes it by traversing its list of satisfied position records and, for each satisfied position record that is an access position record, evaluate its correspondent cluster family and respective subscriptions. The execution of the main thread and worker threads for matching events through the SE-CP technique in the internal event level is presented in algorithm 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 (Mignet, Barbosa, and Veltri 2003), 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:

\begin{algorithm}
\caption{Internal Event Main and Worker Threads execution}
\begin{algorithmic}[1]
\Function{FILTER::PROCESSEVENT}{\textit{event}}
\State $n_{ie} \leftarrow 0$
\ForEach{\textit{internal event} in \textit{event}}
\State $1^{st}$ phase: update position record vector ($prv$) and satisfied position records ($spr$)
\State $2^{nd}$ phase: when an internal event is found at depth $d$
\State $task \leftarrow \text{addIE}(n_{ie}, d, prv, spr)$
\State $n_{ie} \leftarrow n_{ie} + 1$
\EndFor
\While{$n_{ie} > 0$}
\State $\text{while} \ !\text{psmQueue.isEmpty}()$
\State $\text{sm.add(psmQueue.remove())}$
\State $n_{ie} \leftarrow n_{ie} - 1$
\EndWhile
\EndFunction
\State \textbf{shared variables:} \textit{ieQueue}, \textit{filter}, \textit{psmQueue}
\Function{RUNIEMAINTHREAD}{ }
\State $ie \leftarrow \text{ieQueue.remove}()$
\If{$ie$ valid}
\State $sm.add(\text{filter.processIE}(ie))$
\State $n_{ie} \leftarrow n_{ie} - 1$
\EndIf
\EndFunction
\Function{RUNIEWORKERTHREAD}{ }
\State $\text{while } \text{state} \neq \text{FINISHED}$
\State $ie \leftarrow \text{ieQueue.take}()$
\State $psm \leftarrow \text{filter.processIE}(ie)$
\State $\text{psmQueue.add(psm)}$
\State $\text{reset}()$
\EndWhile
\EndFunction
\end{algorithmic}
\end{algorithm}
• $ie < t$. 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, $ie = k \times t + 1 : k \in \mathbb{N}$, 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 transform these tasks into smaller and simpler ones by considering the next parallelism level: satisfied position record. Although decomposing internal event tasks into satisfied position record tasks is unable to remove the non-uniform distribution of clusters, this distribution has less influence in this level than in the internal event level.

### 3.2.2.2 Satisfied Position Record Level

In the satisfied position record level, 3rd level, of the SE-CP approach, when an internal event is discovered, instead of creating a single task representing an internal event, $spr$ tasks are created, one for each satisfied position record of the internal event. Thus, each thread is responsible for the evaluation of the cluster family associated to the obtained satisfied position record. However, recall that some satisfied position records may not correspond to access position records, so there is a probability of some threads working more than other, resulting in
load balancing problems, as occurred in the internal event level.

As fine grained tasks tend to maximize lock contention, the task and result queue become a major bottleneck specially with high number of threads. With this in mind, a block size optimization was create with the purpose of more balanced workloads and to reduce workers’ idle times. A block in this context consists in grouping satisfied position records of the same event (possibly from different internal events) into a single task. Therefore, an optimal block size is not too small, as it increases lock contention, neither too large, as it translates to the internal event level, but with a size resulting in balanced workloads.

Despite this optimization, as later validated in section 4.2.3, the actual work is unable to overcome the parallel overheads originated from task granularity, resulting in worse performance gains than the internal event level. To sum up, whilst in the internal event parallelism level a limitation in scalability is present, here a degradation in scalability is already imminent. As a result, a different direction was taken: the conjunction of the ME-IP and SE-CP techniques into a single one.

### 3.2.3 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 at the internal level), into a single technique. 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. Concerning thread distribution, two hybrid approaches are presented: (1) a static approach resultant from a simple association of the ME-IP and the SE-CP techniques, where a predefined number of threads is allocated to the independent processing of events and to the collaborative process of a single event; (2) a dynamic approach resultant from a more complex association of the two main techniques, where the number of threads allocated per event is defined at runtime.
3.2.3.1 Static ME-CP

In the static approach a predefined number of threads is allocated for both the event (e threads) and internal event (ie threads) parallelism levels. This way, ie threads are organized in e groups, with a main thread per group that is responsible for executing the 1st phase of the matching algorithm, managing its ie − 1 internal event worker threads and merging the partial results into the final set of subscriptions matched. Hence, for e threads at the event level and ie threads at the internal event level, a total of e × ie threads are used. In a global perspective, depending on the distribution of threads among the two parallelization levels, this approach is able to operate as the ME-IP, when a single thread is allocated per event and, as the SE-CP technique, when all available threads are allocated per event. It is worth noting that the groups of threads are always the same throughout execution, i.e., sets of ie threads always belong to the same group. A generic scheme of the predefined distribution of threads per event and internal event levels in the static approach is illustrated in figure 3.9.

![Figure 3.9: Thread distribution in static ME-CP approach.](image)

In brief, this approach should be able to obtain a higher system throughput whilst, simultaneously, reducing the matching time per event. However, this last metric still suffers from the load balancing problems present in the SE-CP internal event level, depending in the number of allocated threads for this parallelism level. An alternative to enhance this metric is to compute at runtime the number of available threads that can help in the processing of a single event, in order for the waiting times of worker threads to be reduced. This alternative is defined as the dynamic hybrid approach and is explained in the next section.

3.2.3.2 Dynamic ME-CP

The dynamic hybrid approach is based on a more interesting concept of adapting execution according to event requirements. In that sense, for each event, a versatile number of threads is
employed, while the remaining threads process separate events. As validated in section 4.2.2, 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.

![Diagram showing event processing](image)

Figure 3.10: Dynamic ME-CP approach.

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 the parsing finished, i.e., all internal events have been discovered but, there are still internal events to process;
- `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 1st 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 updated.

Algorithm 3 Dynamic Hybrid Thread execution.

```plaintext
1: function GETEVENT(thread) 1: shared variables: ieQueue, psmQueue
2:     while !events[thread.e]. 2: function HYBRIDTHREAD::RUN
3:         isEVENTAVAILABLE(thread) do 3:     while state ≠ FINISHED do
4:             thread.e ← thread.e + 1 4:         e ← GETEVENT(Thread)
5:         end while 5:             if e inexistent then
6:     return events[thread.e] 6:                 Thread.wait()
7:     end function 7:                     continue
8: 7: function isEVENTAVAILABLE(thread) 8:     end if
9:     if CAS(state, AVAILABLE, 9:         if mainThread then
10:         PARSING) then 10:             filter.processevent(e)
11:         atomicInc(nThreads) 11:             mainThread ← false
12:     thread.mainThread = true 12:     else
13:     return true 13:         while e.state ≠ FINISHED
14:     end else 14:             do
15:         if state ≠ FINISHED 15:                 ie ← ieQueue.remove()
16:             AND conditionalAtomicInc( 16:                     if ie valid then
17:                 nThreads < 16) then 17:                     psm ← filter.processIE(ie)
18:             return true 18:                     e.psmQueue.add(psm)
19:         end if 19:                     reset()
20:     end if 20:                     else
21:     return false 21:                     if e.state ≠ PARSING
22: end function 22:                         then break
23: 14: end if 23:                     end if
24: end if 24:             end if
25: while state ≠ FINISHED do 25:         end if
26:     e.state ← FINISHED 26:     end while
27: end if 27: 28: end while
28: end while
29: end function
```

To better understand the procedure of the dynamic hybrid approach consider the example of figure 3.10 and, for simplicity assume only 2 available threads. When a hybrid thread \(ht1\) 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 at position e, 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 parsing event $E_n$ by executing the 1st 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 \texttt{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 \texttt{ieQueue}, processes them and adds its partial results to the \texttt{psmQueue} 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. The algorithm illustrating the dynamic hybrid approach is delineated in algorithm 3.

In conclusion, the dynamic 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).
3.3 Summary

In this chapter, we started by giving an overview of the sequential processing of DeltaFilter. We described the two modules of the matching engine: (1) Subscription Storage responsible for efficiently storing XPath subscriptions in the engine; and (2) Event processing module that contains the Filter structure that implements the XML matching algorithm and is, therefore, responsible for the processing of incoming events to the publish/subscribe system. For each module the main data structure were presented and how they are organized within the matching engine. The three matching algorithms, i.e., Filter, DeltaFilter and OptimizedDeltaFilter, are introduced and, their main differences highlighted.

Concerning the parallel event processing of DeltaFilter, three parallel techniques have been described: Multiple Event Independent Processing (ME-IP), Single Event Collaborative Processing (SE-CP) and Multiple Event Collaborative Processing (ME-CP). The purpose of the ME-IP technique is to enhance system throughput by employing threads in the processing of independent events in parallel. Since the ME-IP technique is unable to reduce average matching time, a SE-CP technique is introduced with the purpose of reducing matching time per event through the concurrent processing of a single event in parallel. Last but not least, the ME-CP or hybrid technique intends to incorporate the advantages of the two main techniques by grouping threads that match events in parallel and, at a deeper parallelization level, these threads process a single event in parallel.

In an overall perspective the ME-IP and SE-CP can be seen as two opposite concepts that ME-CP tries to merge into a single technique. This way for the ME-CP technique, the system throughput ideal is the same as the ME-IP technique, whereas the average matching time ideal is the same as the SE-CP technique. For the static hybrid approach in the case more threads are allocated per event better results concerning average matching times but, smaller system throughputs. Conversely, by reducing the number of threads per event better results for system throughput but, slower average matching times per event. The dynamic hybrid approach is able to find a balance between the metrics benefits of the two main approaches, as it obtains good system throughput while reducing average matching time.
The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it.

– Steve Jobs

In this chapter a set of experiments is conducted to measure performance and scalability of the optimized sequential version, as well as, the parallel event processing techniques implemented in DeltaFilter. We compare the three filtering structures, Filter, DeltaFilter and OptimizedDeltaFilter, with the well-known XML matching algorithm YFilter. The popularity of YFilter is, mainly, due to the fact that the code was released by the authors in the YFilter package and is, therefore, a popular benchmark in this research area. However important the analysis of the sequential optimizations of DeltaFilter, the real purpose of this chapter is to understand the benefits in parallelizing the XML matching process, along with studying implementation choices that mostly impact performance and limit scalability. In that sense, we compare system throughput and average matching time of the parallel techniques described in the previous chapter and present, not only the strong points, but also the main limitations of each technique.

4.1 Experimental Setup

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.

1YFilter Package: http://yfilter.cs.umass.edu/code_release.htm
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 are 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. Since XML parsing performance is out of focus of this thesis, we used the SAX parser made available in the YFilter package that is able to minimize parsing overheads. Experiments concerning the parallel approaches try to demonstrate how these technique operate 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.

4.1.1 Workload Generation

The News Industry Text Format (NITF) DTD\(^2\) 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 and is used as a standard benchmark in many research studies (e.g. (Miliaraki, Kaoudi, and Koubarakis 2008), (Chan, Felber, Garofalakis, and Rastogi 2002), (Diao, Altinel, Franklin, Zhang, and Fischer 2003), (Hou and Jacobsen 2006), (Sadoghi, Burcea, and Jacobsen 2011), (Martins, Pereira, and Wichert )).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>1,000,000 − 10,000,000</td>
<td>Number of XPath subscriptions</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
<td>Maximum depth of XPath subscriptions</td>
</tr>
<tr>
<td>Δ*</td>
<td>0.2</td>
<td>Probability of ‘*’ operator occurring at a location step</td>
</tr>
<tr>
<td>Δ//</td>
<td>0.6</td>
<td>Probability of ‘//’ operator being the operator at a location step</td>
</tr>
<tr>
<td>Δ@</td>
<td>0 − 0.6</td>
<td>Probability of a location step to have at least one attribute</td>
</tr>
</tbody>
</table>

Table 4.1: Workload parameters description and range values

The subscription workloads were generated using the XPath query generator released in the YFilter package within the range of workload parameters expressed in table 4.1. Because both YFilter and DeltaFilter share the storage and processing of duplicate subscriptions, it becomes more interesting to run these experiments using unique subscriptions. Hence, an ad-

\(^2\)News Industry Text Format: https://iptc.org/standards/nitf/
ditional 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.

An event workload was used and generated using IBM’s XML generator tool and the workload parameters employed in (Diao, Altinel, Franklin, Zhang, and Fischer 2003), i.e., maximum depth ranges between 6 and 10, and the maximum number of times a recursive element can appear in a simple path is established as 3. A total of 5,000 XML documents were generated with these specifications resulting in a collection of XML documents with its main properties illustrated in table 4.2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>5,000</td>
<td>Number of events</td>
</tr>
<tr>
<td>( \text{avg}(\text{node}) )</td>
<td>68.54</td>
<td>Average number of nodes in an event</td>
</tr>
<tr>
<td>( \text{max}(\text{node}) )</td>
<td>239</td>
<td>Maximum number of nodes in an event</td>
</tr>
<tr>
<td>( \text{min}(\text{node}) )</td>
<td>7</td>
<td>Minimum number of nodes in an event</td>
</tr>
<tr>
<td>( \text{avg}(\text{ie}) )</td>
<td>38.45</td>
<td>Average number of internal events in an event</td>
</tr>
<tr>
<td>( \text{max}(\text{ie}) )</td>
<td>131</td>
<td>Maximum number of internal events in an event</td>
</tr>
<tr>
<td>( \text{min}(\text{ie}) )</td>
<td>4</td>
<td>Minimum number of internal events in an event</td>
</tr>
</tbody>
</table>

Table 4.2: Event workload properties

4.1.2 Metrics

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.

To show the effectiveness of the parallel event processing techniques, experiments were divided into two experimental sets. The first experimental set, Performance and Scalability Analysis, presents a detailed study of the main performance gains of the implemented approaches and how these approaches scale when in the presence of varying number of threads. The second experimental set, Workload Properties Impact Analysis, reports the impact of subscription and event workload parameters on the parallel event processing techniques.
4.2 Performance and Scalability Analysis

The aim of this experimental set is 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. Accordingly, the workload generated for this section tries to simulate real world scenarios by matching 5,000 events over a significant amount of subscriptions (1,000,000 – 10,000,000 subscriptions). The subscription configuration parameters consist in complex XPath subscriptions as a way to reproduce real user interests with the following configuration parameters that remain constant throughout this experimental set: $\Delta_\ast = 0.2; \Delta/= 0.6$ and $\Delta_@ = 0.2$.

4.2.1 Sequential Optimizations

The purpose of this experiment is to compare the sequential algorithms of DeltaFilter, i.e., Filter, DeltaFilter and OptimizedDeltaFilter, with the previously published state-of-the-art for sequential XML matching, YFilter. Furthermore, in order to evaluate how these algorithms scale when in the presence of enormous quantities of user interests, a subscription workload varying between 1,000,000 and 10,000,000 subscriptions is selected.

![Figure 4.1: Sequential Optimizations: $T = 1, N \in [1,000,000, 10,000,000], \Delta_\ast = 0.2, \Delta/= 0.6, \Delta_@ = 0.2.$](image)

Figure 4.1 presents how the different algorithmic techniques behave when scaling from scenarios with a relatively small number of subscriptions (1,000,000 subscriptions) to a significant amount of subscriptions (10,000,000 subscriptions). More specifically, figure 4.1a shows
the number of events processed per second and figure 4.1b shows the average matching time of a single event for the listed approaches. Note that the vertical axes are in a logarithmic scale.

**YFilter.** As explained in section 2.1.1, YFilter stores all subscriptions in an unique Non Deterministic Finite Automaton (NFA), where common structural prefixes are stored and processed only once. Despite being highly scalable when dealing with only structural predicates, YFilter suffers with attribute matching. Diao *et al.* proposed two different approaches concerning value-based matching, Inline and Selection Postponed (SP). According to the authors in (Diao, Altinel, Franklin, Zhang, and Fischer 2003), SP outperforms Inline by delaying value-based matching after pruning subscriptions that do not match structural requirements, instead of evaluating value-based predicates alongside with structural predicates as occurs with Inline. Additionally, with enormous quantities of transitions to consider in each parsing step, it is hard for YFilter to scale even when in the presence of 1,000,000 subscriptions, as it can only process 1.9 events/second under these conditions. YFilter’s scalability decreases even further as the number of subscriptions increases, taking more than 5 seconds to process a single event with 10,000,000 subscriptions, as shown in figure 4.1a. Concerning average matching time, figure 4.1b demonstrates that YFilter takes 10 times more to process a single event with 10,000,000 subscriptions than with 1,000,000 subscriptions stored in the NFA. This increase is directly related to the low rate of events per second.

**Filter, DeltaFilter and OptimizedDeltaFilter.** Contrarily to YFilter, Filter, DeltaFilter and OptimizedDeltaFilter are capable of efficiently processing events with large quantities of subscriptions stored. Moreover, these sequential algorithms employ very selective rules that significantly reduce the subscription’s search space for each internal event. The Filter algorithm already presents extraordinary improvements over YFilter, as it holistically evaluates XML attributes and processes predicates with very fast binary operations. However, since Filter does not take into account shared sub-paths of internal events, it remains in a lower level of efficiency when compared with DeltaFilter and OptimizedDeltaFilter. These two optimizations have similar results with a performance gain in system throughput of approximately 1.5 against the Filter algorithm, as showed in figure 4.1a. When compared to YFilter, these algorithms are able to process 64 more events/second with 1,000,000 and nearly 9 more events/second with 10,000,000 subscriptions.

Regarding average matching time, figure 4.1b demonstrates that all the algorithms take
roughly 7 times more to process a single event with 10,000,000 subscriptions when compared with 1,000,000 subscriptions. Nonetheless, OptimizedDeltaFilter presents slightly improvements over DeltaFilter as it goes one step further of taking into account the shared sub-paths of all previous internal events, whereas DeltaFilter only considers the last processed internal event.

As more and more subscriptions are added to the system, more clusters and cluster families are created and more similarities between subscriptions are present. Consequently, this results in more dense clusters and more subscriptions that need to be evaluated for each internal event. This explains why system throughput decreases and average matching time increases for larger numbers of subscriptions. Despite this limitation, since OptimizedDeltaFilter presents better results for both system throughput and average matching time, this algorithm is the one used throughout the remaining experiments.

4.2.2 Varying Number of Threads

The goal of this experiment is understanding how the number of threads impacts performance and scalability of Multiple Event Independent Processing (ME-IP), Single Event Collaborative Processing (SE-CP) and Dynamic Multiple Event Collaborative Processing (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.

Figure 4.2: Varying Number of Threads: $T \in [1, 48], N = 5,000,000, \Delta* = 0.2, \Delta/\ = 0.6, \Delta@ = 0.2$.

Figure 4.2 shows the number of events processed per second and the average matching
time to process a single event as the matching engine scales from a single-thread to a fully parallel approach where 48 threads are used simultaneously. Note that the horizontal axes are in a logarithmic scale. The sequential OptimizedDeltaFilter algorithm is also illustrated as a way to demonstrate the parallel performance gains.

**ME-IP.** 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 4.2a 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 4.2b, the average matching time per event increases alongside with the number of threads, with a 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.

**SE-CP.** At a different baseline, figure 4.2a illustrates that the 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 performance 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 4.2b 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
CHAPTER 4. EVALUATION AND RESULTS

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 16 threads is the optimal number of threads per event.

**Dynamic Hybrid.** The first observation to make regarding the dynamic hybrid approach 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 4.2a 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 dynamic hybrid approach. This way, we are able to avoid as fast as possible idle times resultant from task availability.

When analyzing average matching time, figure 4.2b depicts that the dynamic 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 dynamic hybrid approach will have more groups of threads processing events in parallel. As a result, for more than 16 threads, the dynamic 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.
4.2. PERFORMANCE AND SCALABILITY ANALYSIS

4.2.3 Varying Block Size

The previous experiment showed that the SE-CP technique at the internal event granularity is unable to scale efficiently due to the unbalanced distribution of satisfied position records among internal events, resulting in unequal internal event complexities. Hence, a deeper level of granularity was taken into account, satisfied position record, with a block size parameter that tries to achieve uniform distribution of workload between threads. In that sense, the purpose of this experiment is to explore if the satisfied position record task granularity in combination with the block size optimization is able to enhance SE-CP performance and scalability.

![Graphs showing throughput and average matching time](image)

Figure 4.3: Varying Block Size: \(T = 48, \Delta \text{blocksize} \in [1, 2120], N = 5.000.000, \Delta* = 0.2, \Delta// = 0.6, \Delta@ = 0.2.\)

Figure 4.3 compares the system throughput (figure 4.3a) and average matching time per event (figure 4.3b) of the internal event level with the satisfied position record level using different block sizes. The domain values for the block sizes range from 1 to 2120 blocks, where 2120 represents the maximum number of satisfied position records present in an event of the event workload. The usage of a block size of this magnitude in this context is equivalent to a sequential algorithm execution, since a single massive task is created per event.

The graph of figure 4.3a illustrates that employing a block size of 5 to the satisfied position record level results in the most events processed per second. In a similar fashion, figure 4.3b demonstrates that the satisfied position record level with block size 5 obtains the better results for average matching time and is approximately the same as the internal event level. However, for both metrics, this granularity level does not achieve significant improvements when compared with the internal event level granularity.
CHAPTER 4. EVALUATION AND RESULTS

Figure 4.4: Work and Wait times of SE-CP with IE (Internal Event) and SPR (Satisfied Position Record): $T \in [1, 48], N = 5.000.000, \Delta* = 0.2, \Delta/ = 0.6, \Delta\oplus = 0.2$

Figure 4.4 illustrates the average work and wait times of threads for the internal event and satisfied position record level with block size 5, when varying the number of threads between 1 and 48. The first observation to make is that for both granularity levels, as the number of threads increases, in the matching of an event, work time decreases and wait time increases, hindering scalability for several threads. The second observation is that, despite work times between the two approaches remaining almost identical, the main difference occur in the wait time with the satisfied position record level obtaining worse results as the number of threads increases.

In conclusion, the granularity level of satisfied position records still encompasses complex tasks, due to the non-uniform distribution of clusters among position records which, similarly to what happened in the internal event level, results in unbalanced workloads. This in combination with the increased parallel overheads associated with task creation and, the task and result queues becoming a major bottleneck, results in longer idle times that prevent scalability. Consequently, the internal event granularity constitutes a stagnation level for the SE-CP technique.

4.2.4 Varying Number of Threads per Event

In this experiment we focus on the ME-CP approaches and how the distribution of threads per event influences performance and scalability. Accordingly, comparisons between static and dynamic approaches are presented in detail.


4.2. PERFORMANCE AND SCALABILITY ANALYSIS

Figure 4.5 illustrates the number of events processed per second (figure 4.5a) and matching time for a single event (4.5b) as the matching engine scales from a single threaded to a 48 threaded system. The static number of threads employed per event in this experiment are 2, 4, 8 and 16 threads/event. Since the version with a single thread per event corresponds to the original ME-IP technique and the with all 48 threads/event corresponds to the original SE-CP technique, these approaches are not represented. The version with 32 threads/event presents a similar behavior to the 16 threads/event not adding relevant information.

The static approach with 2 threads/event presents the best system throughput for all techniques till a total of 32 threads is reached in the matching engine. As it happens with a single thread per event (ME-IP technique), this increase occurs as several groups of 2 threads will be processing independent events simultaneously. However, in both metrics, for 48 threads this approach presents worse results than with 32 threads. We associate this unfavorable result to a cache problem, as it is most likely that 2 threads allocated to the same event are scheduled for cores of different sockets, impacting the cache mechanism that deteriorates performance.

At the other end, the static approach with 16 threads/event presents the worst results for system throughput, but the best results concerning average matching time. This was expected, as less groups process events concurrently, resulting in less events processed per second and, more threads collaborate in the context of a single event, reducing the matching time per event.

As expected, when comparing the static approach of 4 threads/event with the 8 threads/event it is observable that the 4 threads/event approach presents better system
Chapter 4. Evaluation and Results

throughput, but the 8 threads/event approach presents better matching times per event. In these approaches, the cache mechanism has less influence, as it is less likely that the threads allocated to the same event are scheduled for cores of different sockets, when compared with the static 2 threads/event approach. However, a visible increase in average matching time from 32 to 48 threads is present, which is directly related to slower increases in system throughput.

The dynamic hybrid approach is able to maintain a balance between system throughput and average matching time. This approach obtains better results than static 8 threads/event approach till a total of 32 threads is present in the matching engine. Table 4.3 demonstrates that the dynamic hybrid approach, for a total of 32 threads employs an average of 14.167 threads/event and for a total of 48 threads employs an average of 14.995 threads/event. Despite having similar threads employed per event in the two scenarios, the dynamic hybrid approach is able to obtain a more balanced workload than static 8 threads/event approach with 32 threads. The same does not occur with 48 threads, as static 8 threads/event obtains better system throughput. However, the dynamic hybrid approach is able to obtain better average matching time than static 8 threads/event, since more threads are allocated for the processing of a single event.

When compared to the 16 threads/event approach, dynamic hybrid is able to obtain a more efficient scheduling of threads, as the system throughput of figure 4.5a shows. Nonetheless, the static 16 threads/event obtains faster average matching times per event than the hybrid approach. Note that, despite the dynamic hybrid without the imposed limit of 16 threads/event, for more than 16 threads this approach presents a similar behavior to the SE-CP technique since larger number of threads are allocated to single events, resulting in a decrease of both system throughput and average matching time. In brief, dynamic hybrid approach is more adaptive to the workload needs, whereas static approach is more dependent of the workload properties.

<table>
<thead>
<tr>
<th>Threads per Event</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Minimum</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.8</td>
<td>1.8</td>
<td>3</td>
<td>2.8</td>
</tr>
</tbody>
</table>

Table 4.3: Threads per Event for Dynamic Hybrid technique
4.2. PERFORMANCE AND SCALABILITY ANALYSIS

4.2.5 Varying Number of Subscriptions

The objective of this experiment is to evaluate how the number of subscriptions stored in the system impacts performance of parallel approaches. As seen in the experiment of section 4.2.1, for the sequential algorithms, 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. An identical scenario takes place for the parallel approaches ME-IP, SE-CP and Dynamic Hybrid. Figure 4.6 shows the behavior of ME-IP, SE-CP and Dynamic Hybrid approaches as the number of subscriptions stored in the system increases from 1.000.000 to 10.000.000 subscriptions.

![Graphs showing throughput and average matching time](image)

Figure 4.6: Varying Number of Subscriptions: \( T = 48, N \in [1.000.000, 10.000.000], \Delta* = 0.2, \Delta/\ = 0.6, \Delta@ = 0.2.\)

**ME-IP.** 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 dynamic hybrid. Additionally, as demonstrated in table 4.4, the increase of the number of subscriptions stored improves speedup, reaching a speedup of almost 25 with 48 threads in a system with 10.000.000 subscriptions stored. 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.

**SE-CP.** For the SE-CP technique the graph of figure 4.6b 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, as shown in table 4.4, it is up to almost $4$ times faster than the sequential OptimizedDeltaFilter algorithm in the presence of $10,000,000$ subscriptions.

**Dynamic Hybrid.** As expected for the dynamic 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 dynamic 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.

<table>
<thead>
<tr>
<th>Number of subscriptions (x 1,000,000)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME-IP system throughput speedup</td>
<td>7.9</td>
<td>14.3</td>
<td>16.8</td>
<td>16.4</td>
<td>17.3</td>
<td>18.1</td>
<td>20.6</td>
<td>21.7</td>
<td>23.2</td>
<td>24.5</td>
</tr>
<tr>
<td>SE-CP average matching time speedup</td>
<td>2.4</td>
<td>3.2</td>
<td>3</td>
<td>3.3</td>
<td>3.2</td>
<td>3.6</td>
<td>3.5</td>
<td>3.4</td>
<td>3.9</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Table 4.4: System throughput and average matching time improvements with varying number of 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.

### 4.3 Workload Properties Impact Analysis

The purpose of this experimental set is to understand how workload properties impact performance and scalability of the parallel event processing techniques.
4.3 WORKLOAD PROPERTIES IMPACT ANALYSIS

4.3.1 Impact of Event Properties

In this experiment we aim at evaluating the parallel approaches under event workloads with different properties. Although the number of subscriptions has a greater impact for algorithm performance and scalability, event complexity also constitute a major factor. For this purpose, we assume that the number of nodes and internal events of an event constitute an important aspect of event complexity. In this line of reasoning, starting from the original set of 5,000 events, two additional event workloads of different complexities were created: Minor and Major. The Minor event workload is composed by the 100 least complex events replicated 49 times, whereas the Major event workload is composed by the 100 most complex events replicated 49 times. Table 4.5 represents Minor, Standard and Major event workloads properties and, figure 4.7 illustrates the distribution of event’s complexity, i.e., the number of nodes and internal events, over the three event workloads.

Figure 4.8 shows the course of the ME-IP, SE-CP and dynamic hybrid techniques under the three proposed event workloads. The graphs from figure 4.8a to 4.8c show the effect of event complexity on system throughput and, the graphs from figure 4.8d to 4.8f show the effect of event complexity on average matching time.

<table>
<thead>
<tr>
<th>Property</th>
<th>Minor</th>
<th>Standard</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>5,000</td>
<td>5,000</td>
<td>5,000</td>
</tr>
<tr>
<td>$\text{avg}(\text{node})$</td>
<td>15.08</td>
<td>68.544</td>
<td>139.97</td>
</tr>
<tr>
<td>$\text{max}(\text{node})$</td>
<td>26</td>
<td>239</td>
<td>239</td>
</tr>
<tr>
<td>$\text{min}(\text{node})$</td>
<td>7</td>
<td>7</td>
<td>105</td>
</tr>
<tr>
<td>$\text{avg}(\text{ie})$</td>
<td>10.04</td>
<td>38.454</td>
<td>76.49</td>
</tr>
<tr>
<td>$\text{max}(\text{ie})$</td>
<td>16</td>
<td>131</td>
<td>131</td>
</tr>
<tr>
<td>$\text{min}(\text{ie})$</td>
<td>4</td>
<td>4</td>
<td>55</td>
</tr>
</tbody>
</table>

Table 4.5: Minor, Standard and Major event workload properties

In general, similarly to what took place when increasing the number of subscriptions in experiment of section 4.2.5, for more complex events system throughput decreases and average
matching increases for all techniques. This was expected, as similarly to the variation in the number of subscriptions, more complex events result in longer matching times, since more nodes and internal events have to be considered and processed.

Concerning average matching time, the graph of figure 4.8d show an improved average matching of the dynamic hybrid over the SE-CP approach. Since events are less complex, the difference between using a lock solution, as in SE-CP, versus a Compare-And-Swap (CAS) solution, as in dynamic hybrid, is more visible. Additionally, considering the graph of figure 4.8f, for 1 thread the average matching time of ME-IP technique is faster than the ones from the SE-CP and dynamic hybrid techniques. This can be explained as both SE-CP and dynamic hybrid with a single thread need to create internal event tasks, whereas ME-IP technique does not account this additional overhead. Despite this situation, as the number of threads increase this parallel overhead becomes less significant. Furthermore, notice that the dynamic hybrid approach is always able to balance system throughput with average matching independently of the event workload employed.
4.3. WORKLOAD PROPERTIES IMPACT ANALYSIS

4.3.2 Impact of Attributes

The purpose of this experiment is to show the impact of the attribute XPath parameter in the parallel approaches. As demonstrated in the study (Mignet, Barbosa, and Veltri 2003), the proportion of attribute nodes in the content of XML documents ranges from 35% to 51% which already constitutes a large amount of the document. This way the probability of subscriptions containing attributes is higher as users are more interested in value-based predicates than in structure-based predicates. Recall that the XPath workload parameter $\Delta@$ represents the probability of an attribute appearing in a location step. In order to avoid all location steps to have attributes the $\Delta@$ parameter employed in this experiment ranges from 0 to 60%.

Figure 4.9: Impact of Attributes: $T = 48, N = 5.000.000, \Delta* = 0.2, \Delta// = 0.6, \Delta@ \in [0, 0.6]$}

Figure 4.9 shows the impact of the attribute parameter in system throughput (figure 4.9a) and average matching time (figure 4.9b) of ME-IP, SE-CP and dynamic hybrid techniques. Since DeltaFilter treats attribute elements holistically, the attribute position records constitute more selective predicates, resulting in faster matching times. There is a visible difference between 0% and 10% for both ME-IP and dynamic hybrid techniques. This peak results from evolving from an environment where no attributes are considered to an environment where there is a probability of 10% of a location step having at least one attribute. As a result, the position record vector increases size and the 1st phase of the matching algorithm takes longer, since additional comparisons have to be made in order to obtain candidate attribute position records. Recall that, only the 2nd phase of the matching algorithm is parallelized, meaning that the additional operations of the 1st increase total matching time per event. From this point on, the performance increases as the probability of attribute position records being defined as access
position records increased resulting in more selective position records.

4.4 Summary

In this chapter, we evaluated the parallel event processing techniques detailed in the previous chapter. The experiments here presented demonstrate how the parallel matching algorithm scales from a single thread to a full parallel implementation of 48 threads.

The ME-IP technique is able to efficiently scale as the number of threads increases making it suitable for environments with high rates of incoming events of uniform sizes, where the fast processing of a set of events is more relevant than the fast processing of a single event. This technique is able to process nearly 20 times more events per second with 48 threads, when compared with a single thread. For faster matching times per event the SE-CP technique is recommended as it is able to reduce by 77% the average matching time when employing 16 threads per event. For balanced metrics, the ME-CP techniques constitute the best approach, with the dynamic approach processing 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.
Conclusions and Future Work

5.1 Conclusions

In this thesis, 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. There have been some attempts to parallelize the XML matching process in centralized environments, with the main approaches focusing on automata systems based on the well-known sequential XML matching algorithm YFilter (Zhang, Pan, and Chiu 2010)(Antonellis, Makris, and Pispirigos 2012). However, these solutions fail to scale when in the presence of millions of XPath subscriptions and do not explore different parallel levels for the matching process.

Based on the work presented in (Farroukh, Ferzli, Tajuddin, and Jacobsen 2009) in the context of content-based publish/subscribe systems, 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. Moreover, two approaches were implemented concerning this technique: static, where a predefined number of threads is allocated to events and, dynamic, where the number of threads allocated to events is determined in runtime in accordance with event complexity.

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.
As such, this technique is suitable for scenarios with high rates of incoming events of uniform sizes. SE-CP presents more limitations concerning scalability due to the increase of synchronization costs alongside with the addition of threads, resulting in larger wait times and smaller work times. Despite the low performance gains, it is still able to efficiently reduce up to 77% the average matching time with 16 threads. This makes the SE-CP technique appropriate to complex event workloads, where a single event has to be processed as fast as possible. Finally, the ME-CP technique is able to incorporate ME-IP and SE-CP qualities making it convenient for systems with diversified workloads where collections of events should be processed instantly with small matching times per event. This approach 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.

5.2 Future Work

In this section, we enumerate possible future work. The work presented in this thesis has some limitations that suggest possible future directions. The current algorithm only matches events against subscriptions formerly stored, i.e., does not support a dynamic environment where subscriptions can be added, removed or updated while the matching engine is processing events. For the addition of subscriptions at runtime a simple solution consists in placing the new subscriptions in a queue to be further added to the system by an assigned thread. The removal of subscriptions leads to a more complex situation with clusters including empty entries. A solution to prevent this scenario is to set the first position record of the subscription with a position record that is always invalid, avoiding the evaluation of the remaining position records of this subscription. However, this solution could lead to substantial clusters with large subsets of invalid subscriptions and, can further decrease the cache optimization technique employed by the clusters. A simple solution to overcome this problem is to fill out previous subscriptions with new ones. The update of subscriptions subsists in two main situations, one where the update is only on a single position record that does not constitute the subscription’s access position record and, another one, where the update adds/removes position records and/or changes the access position record. In the former, a single update of the respective position record takes place, since the subscription remains in the same cluster. In the latter, two operations have to be performed, a subscription removal in the original cluster and
5.2. FUTURE WORK

a subscription addition in the destination cluster.

Regardless of the update operation, it is crucial that no threads are processing events as the subscription set is modified otherwise, inconsistent results may take place, as users might get subscriptions they did not subscribe to, or not receive notification they subscribed to. In this line of reasoning, when a subscription operation is inserted into the system, threads are notified to pause after the processing of the event they are currently matching. Additionally, instead of modifying a single subscription at a time a more efficient solution is to apply a set of modifications from time to time.

The standard cache mechanism can sometimes degrade the algorithm performance for large numbers of threads as less resources are available for threads. In this line of reasoning, an interesting future direction would be to investigate ways of optimizing thread scheduling as a way to improve the cache mechanism and obtain optimum performance. For instance, guarantee that threads allocated to the cooperative processing of a single event are scheduled within cores of the same NUMA node.

Another suggestion is to support parallel twig patterns processing, i.e., nested path queries with additional structural constraints, usually specified as tree structured relationship in an XML document. Some sequential solutions concentrate on this problem employing query decomposition techniques (Chan, Felber, Garofalakis, and Rastogi 2002) or holistic structure matching (Kwon, Rao, Moon, and Lee 2005)(Kwon, Rao, Moon, and Lee 2008). In the area of parallel processing of twig pattern most system are based on hardware solutions, e.g., FPGA (Moussalli, Salloum, Najjar, and Tsotras 2011) or GPU (Absalyamov, Moussalli, Tsotras, and Najjar).

Another appealing direction would be to implement parallel matching engines in a distributed version of the algorithm, as a way to improve scalability at an Internet-scale. Here, subscriptions would be clustered according to their resemblance and, subscribers would be assigned to a set of brokers. As a result, it would be possible to achieve better scalability, better coordination of events and subscriptions and, event routing mechanism, similarly to BlueDove (Li, Ye, Kim, Chen, and Lei 2011).


International Conference on Distributed Event-Based Systems, DEBS ’09, New York, NY, USA, pp. 8:1–8:4. ACM.


Megginson, D. et al. (2001). Sax 2.0: The simple API for XML. SAX project.


