DEIM Forum 2012 E11-4

XMLストリームに対する省電力を考慮した問合せ処理

清野　真奈† 　天笠　俊之†† 　北川　博之††

† 筑波大学大学院システム情報工学研究科　〒 305–8573 茨城県つくば市天王台 1–1–1
†† 筑波大学 システム情報系　〒 305–8573 茨城県つくば市天王台 1–1–1
E-mail: †mana@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp

あらまし　近年，XML はデータの記述フォーマットとして普及してきている．特に，連続的に送信されるストリーム型の XML データが急増しており，XML ストリームデータに対するフィルタリング処理システムの研究が多くなっている．本研究では，フィルタリングの要求に必要のない XML データ要素の構文解析をスキップし，フィルタリングにかかる処理を削減することで省電力を図るスキップ処理手法を提案する．フィルタリング要求である XPath 式をオートマトンの形式に変換し，次に遷移可能な状態を確認することで要求にマッチしない要素をスキップする．また，次に遷移可能な状態が一意に定まらない XPath 式は，送信されてくる XML データのスキーマ定義を用いて書換えを行いスキップを可能にする．最後に，既存のフィルタリング手法と比較実験を行い，提案手法の有効性を示す．

キーワード　XML, XML ストリーム処理，省電力

Mana SEINO†, Toshiyuki AMAGASA††, and Hiroyuki KITAGAWA††

† Graduate School of Systems and Information Engineering, University of Tsukuba Tennoudai 1–1–1, Tsukuba City, Ibaraki, 305–8573 Japan
†† Faculty of Engineering, Information and Systems Tennoudai 1–1–1, Tsukuba City, Ibaraki, 305–8573 Japan
E-mail: †mana@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp

1. はじめに

近年，XML [1] は様々なデータを柔軟に表現可能なため，多くの分野でデータの記述フォーマットとして急速に普及してきている．XML はデータの意味や構造をタグで記述することが可能であるため，データ構造の把握が容易になり，電子データを柔軟に表現できる．特に Web サービスの技術として XML が利用されており，XML データは連続的に送受信が行われるようになってきた．このような XML データのストリームが急増すると共に，XML ストリームデータのフィルタリングに関心が高まってきている．ストリームデータは常時送信されてくるため，フィルタリングシステムの電源も常にオンの状態にしておかなければならない．そのため，フィルタリングシステムの電力を抑えることが課題であると言える．

既存のフィルタリングシステムでは，全ての XML データを構文解析し，イベントの発生や内容の処理を行なっている．既存手法である YFilter [2] におけるフィルタリングの各処理時間の割合を図 1 に示す．フィルタリング処理のうち，最も時間がかかる処理は XML パーサであり，その中でも要素名の解析，改行や空白等のリテラル解析，イベントの発生処理に時間がかかっている．しかし，フィルタリング対象の XML データは XML ストリームデータのごく一部であるため，フィルタリング対象となる部分 XML データ以外は本来処理する必要はない．そこで，本来処理する必要のない不要な処理をスキップすることで，システムの性能向上や消費電力の削減を図る．

また，近年，エネルギー不足や，地球温暖化等による環境破壊や環境汚染が大きな問題となっている．問題の対策として省エネルギーが推進されている．省エネルギーの一つに電力の節約が挙げられる．東日本大震災による電力供給量の低下もあり，ますます節電や省電力が重要になってきている．

そのため，本研究では，XML ストリームについてのフィルタリングを例に挙げ，XML ストリームデータ処理に関して省電力を考慮した手法について提案する．既存手法 [2] で提案されている変換方法に基づいて，フィルタリング要求である XPath 式をオートマトンの形式に変換する．オートマトンを利用して，現在の状態から次に遷移可能な状態を取得する．それらの状態から XPath 式にマッチしない XML 要素を探し出す手法を提案する．現在の状態から次に遷移可能な状態を取得し，それをフィルタリングシステムで保持しておく．XML パーサから要素の開始タグのイベントを受け取ると，遷移可能であるかを

調べる．遷移できない要素の場合は，その要素と内容全てにスキップ処理を行う．しかし，次に遷移可能な状態が一意に定まらないような XPath 式が与えられた場合，XPath 式にマッチしない XML 要素を探すことができないという問題点がある．この問題点に対し，送信されてくる XML データのスキーマ定義を用いて書換えを行う前処理手法の提案を行い，スキップ処理を可能にする．スキップ処理手法でフィルタリングの処理を削減することで CPU にかかる負荷を削減し，省電力を図った．最後に，既存のフィルタリング手法と比較実験を行い，提案手法の有効性を示した．

本論文の構成を以下に示す．まず，2. 節で本研究の前提知識について述べ，3. 節で関連研究について述べる．次に本論文の提案する手法について 4. 節で述べる．フィルタリング要求にマッチしない XML 要素の構文解析を行わないスキップ処理手法について説明する．また，経路が一意に定まらない XPath 式があるとスキップ処理手法を行うことができないという問題に対するアプローチとして， XML データのスキーマ定義から XPath 式を書き換える前処理を行う手法について説明する．5. 節では既存手法である YFilter と本研究の提案手法の CPU 消費電力に関する実験と考察について述べる．最後に，6. 節で本研究についてまとめ，今後の課題について述べる．

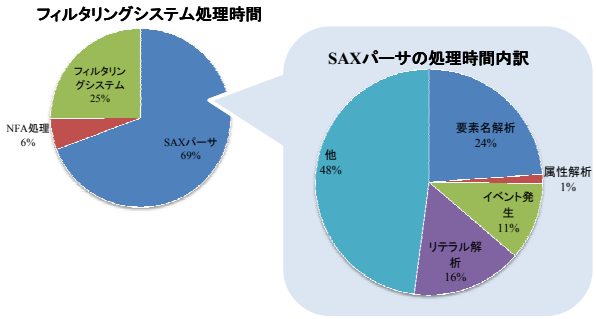

図 1　フィルタリングシステムの処理時間

2. 前提知識

2.1 XML ストリーム

連続した XML データを XML ストリームデータと呼ぶ．XML データベースは蓄積された XML データに対して検索を行うが，XML ストリーム処理では蓄積された問合せに対して入力 XML ストリームデータを処理する．XML ストリーム処理の特徴として，問合せ式が予め蓄積されており，入力される XML ストリームデータを問合せ式が受理するか否かというオートマトンの問題であるという点がある．そのため，現在提案されている XML ストリーム処理のアプローチはオートマトンを拡張した手法が多くなっている．

2.2 SAX (Simple API for XML)

XML データを処理する手段として SAX [3] がある．SAX は XML データをイベントとして表現するイベント駆動型のアプリケーションプログラミングインタフェース (API) である．

SAX ベースの XML パーサは，XML データを先頭から順次読み込み，ドキュメントの開始や要素の開始等のイベントが発生する度に対応するハンドラを呼び出す．そのため，構文解析を行いながら XML データに対する処理を実行することができるという特徴を持つ．また， 解析結果をイベントとして発生させた後はそれを保持しないため，巨大な XML データでも小さなメモリで高速な処理が可能となる． これらの特徴から，連続的にデータを受信する XML ストリームに対する構文解析器として適していると言える．

SAX ベースの XML パーサの例としては Apache Xerces [4] や Crimson [5] がある．Apache Xerces は SAX だけでなく，XML ドキュメントの構文解析を行うための様々な API を実装しているライブラリを提供している．また，XML データの生成機能も提供している．対して，Crimson は XML パーサの機能のみを提供するシンプルな実装となっている．

本研究では，XML パーサとして Crimson を用いて実装を行った．

3. 関連研究

3.1 YFilter

XML ストリームデータのフィルタリングに関する研究として Diao らの研究がある [2]．連続的に到着する XML データについて効率的かつスケーラブルにフィルタリングを行うために，YFilter と呼ばれる手法を提案している．YFilter は XFilter [6] を改良したもので，非決定性有限オートマトン (NFA) ベースの有限状態マシン (FSM) を用いて問合せ XPath 式の共有化を行い，同じ構造を持つ問合せ式の処理の冗長性を回避している．また，NFA を用いることで処理性能は XPath 式の件数に依存することになる．YFilter では，SAX ベースの XML パーサを利用しており，イベントが発生すると，そのイベントを入力として NFA を遷移させて XPath 式とのマッチングを行う．

YFilter による XML ストリームデータに対するフィルタリングの流れを図 2 に示す．まず，予めモニタリング要求である XPath 式を XPath パーサに入力し，NFA の形式に変換しておく．フィルタリングを行う XML ストリームデータは，まず XML パーサに入力される．フィルタリングシステムは XML パーサから渡ってきたイベント情報と，既に所持しているオートマトン形式に変換した要求 XPath 式を基にマッチング処理を行い，結果を出力する．

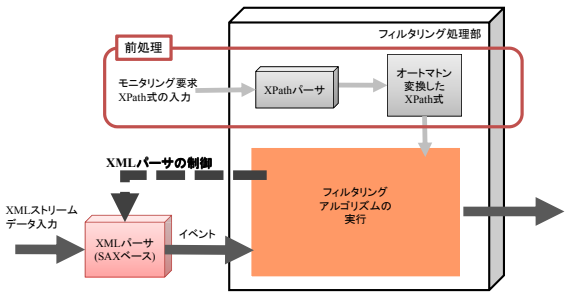

図 2　YFilter の処理の流れ

問合せ式 (図 3) を共有化し，NFA の形に変換した例を図 4 に示す．Q1 から Q8 の八個の問合せ式が与えられたとき，初

期状態から同一な状態を共有していく．問合せは全て “a” から始まっているため，初期状態からの遷移先は一つとなる．問合せで “a” の次に現れる要素名は “b”，“c”，“d” の三パターンが存在する．そのため，遷移先は三つとなる．残りも同様に共有を行っていくと図 4 に示した状態遷移図になる．このように，共有化を行うことで状態の数を減らし，処理の高速化を図っている．

本研究では，YFilter をベースに省電力を考慮した処理について検討していく．

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

図 3　問合せ式群

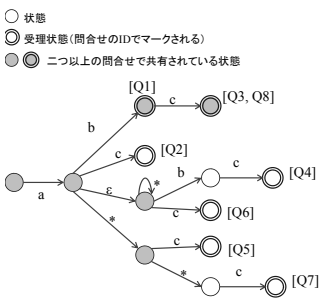

図 4　問合せ式の共有化例

3.2　Stream Index (SIX)

ストリーム処理の高速化に用いられる手法として Stream Index (SIX) [7] がある．SIX は XML ドキュメントにバイトオフセットのペアの順序集合を付与する．バイトオフセットのペアは $(beginOffset, endOffset)$ で表され，これらの数値はバイナリで記述される．beginOffset は開始タグのバイトオフセットであり，endOffset は beginOffset の開始タグに対応する終了タグのバイトオフセットである．作成された SIX は XML ストリームと共に送信され，SIX に対応しているアプリケーションで利用することが可能である．一方，SIX に対応していないアプリケーションでは無視することができる．

この手法では SIX を作成する際に XML データの構文解析を行っているが，一度も解析されたことのない XML ストリームデータでは利用することができない．本手法では XML ストリームデータの前処理を必要としない手法を提案する．

4. 提 案 手 法

4.1　概　　要

省電力を考慮するために提案する XML ストリーム処理に関するアプローチについて説明する．

図 5 の XML データに対するモニタリングの要求例を次に示す．

$$/ItemSearch \tag{1}$$

$$/ItemSearch/Request/Keywords \tag{2}$$

```
<?xml version="1.0"?>
<ItemSearch>
  <Request>
    <SearchIndex>Books</SearchIndex>
    <Keywords>XML</Keywords>
    <Author>筑波花子</Author>
  </Request>
</ItemSearch>
```

図 5　XML データの例

式 (1) は XML データが到着したかどうかを知りたいという要求で用いる．式 (1) にマッチした XML データの数を数えることで，一定期間でどれだけリクエストが送信されてきたかが分かる．式 (2) は検索されたキーワードを調べる要求である．式 (1) の場合，ルート要素が “ItemSearch” である XML データが送信されてきたかが分かれば良いため，“Request” 要素及び子孫要素についての内容は必要ない．そのため，“Request” 要素とその子孫要素の構文解析はしなくても良い．また，式 (2) の場合は “Request” 要素の子要素である “Keywords” 要素のみが必要である．そのため，同じ “Request” の子要素である “SearchIndex” や “Author” 要素について構文解析を行う必要がない．このように，要求によっては XML データの構文解析の処理を行わなくても良い要素が存在することになる．ある要素に対して構文解析処理とそれに伴うイベントの発生処理を行わないことを **スキップ処理**と定義する (定義 4.1)．

［定義 4.1］（スキップ処理） ある要素 “a” に対し，XML パーサで要素としての構文解析を行わず，タグ及び要素の内容全てをテキストとみなして構文解析を行うことをスキップ処理と定義する．スキップ処理中は XML パーサのイベントの発生処理も行わない．

図 5 の XML データが XML パーサに入力された場合のイベントの発生例について図 6 に示す．XML データが XML パーサに入力されると，通常は図 6(a) のように全ての要素に対してイベントが発生する．しかし，モニタリング要求が D1 の場合，“ItemSearch” 要素以外はスキップ可能なため，図 6(b) のようにイベントの処理数を減らすことができる．モニタリング要求が D2 の場合も同様に，“Request” の子要素である “Keywords” 要素以外はスキップ可能なため，イベントの発生は図 6(c) のようになる．

また，XML パーサではイベント発生の処理に多く時間がかかっている．そこで本研究では，要求に必要のない要素をスキップ処理することで不要な要素のイベント発生数を抑え，構文解析にかかる処理を減らし，CPU のアイドル状態を長くすることで電力の節約を図る．

スキップ処理を行うために，フィルタリング処理部から XML パーサに命令を出して通常処理との切換えを行う枠組みを提案する．XML パーサはフィルタリング処理部からのスキップ命令を受け取ると，構文解析及びイベントの発生処理を行わずに

start document
start element: ItemSearch
start element: Request
start element: SearchIndex
characters: Books
end element: SearchIndex
start element: Keywords
characters: XML
end element: KeyWords
start element: Author
characters: 筑波花子
end element: Author
end element: Request
end element: ItemSearch
end document

(a)

start document
start element: ItemSearch
end element: ItemSearch
end document

(b)

start document
start element: ItemSearch
start element: Request
start element: Keywords
characters: XML
end element: KeyWords
end element: Request
end element: ItemSearch
end document

(c)

図 6　SAX ベースの XML パーサから発生するイベントの例．(a) 通常時のイベント，(b) 要求 D1 に対する省略処理を行った場合のイベント，(c) 要求 D2 に対する省略処理を行った場合のイベント

次の要素の処理に移る．次の要素もスキップ可能な要素だった場合はスキップ不可要素になるまで構文解析処理を行わない．

4.2　スキップ処理可能なケース

ある要求式に対してどのような XML データの構文解析のスキップができるかを例を挙げて示す．ここではスキップ処理可能なケースとして要求式のみで判断できるケースとスキーマ知識が必要なケースの二つを挙げる．まず，スキップ処理可能ケース 1 として，要求式のみでスキップ可否が判断できるケースについて説明する．図 7 は XML データとそれに対するフィルタリング要求式である．この要求式は “ItemSearch” 要素の子要素である “Request” 要素の更に子要素である “Title” 要素または “Keywords” 要素を指定している．図 7 の XML データでは，“Request” の子要素として “SearchIndex”，“Keywords” 及び “Author” が存在する．要求式で求められている要素は “Title” または “Keywords” のため，“SearchIndex” と “Author” 要素はスキップ処理が可能である．ここで，“/ItemSearch/Request/Title” のような XPath 式を単純パスと呼ぶこととする (定義 4.2)．

［定義 4.2］（単純パス） “//” や “*” がロケーションステップに含まれない XPath 式を単純パスと定義する．

XMLデータ

```
<ItemSearch>
  <Request>
    <SearchIndex>Books</SearchIndex>
    <Keywords> XML </Keywords>
    <Author>筑波花子</Author>
  </Request>
</ItemSearch>
```

フィルタリング要求式

1: /ItemSearch/Request/Title
2: /ItemSearch/Request/Keywords

図 7　スキップ処理可能ケース 1

次に，スキップ処理可能ケース 2 として，スキーマの知識が必要なパターンについて説明する．図 8 は XML データとそれに対するフィルタリング要求式である．要求式 1 は “ItemSearch” 要素の子要素である “Request” 要素の更に子要素である “Author” 要素の子要素である “Name” 要素を指定している．また，要求式 2 は XML データのどこかにある “Name” 要素を指定する式である．図 8 の XML データにおいて，“Request” 要素の子要素としてまず “SearchIndex” 要素がある．“SearchIndex” は要求式で指定されていない．しかし，開始タグを解析しただけでは子要素に “Name” 要素が存在するかどうかは判断できないため，スキップ処理することができない．そのため，XML データのスキーマを利用する．XML にはデータの構造定義を行うためのスキーマ言語として DTD [8] や XML Schema [9] などがある．この構造定義を利用すると，XML データがどのような要素を持っているか，要素と要素の関係についてを知ることができる．図 8 の XML データに対するスキーマとして図 9 を利用する．図 9 から，“SearchIndex” 要素は子要素を持たないことがわかるため，スキップ処理が可能になる．また，“Title” 要素の子要素には “Name” と “Yomi” がある．DTD から “Title” 要素の子要素である “Name” 要素は一回しか出現しないことがわかる．そのため，“Yomi” 要素はスキップ処理することができる．このように，要求式のみでは判断できず，入力される XML データのスキーマ知識を利用すればスキップ可否が判断できるケースがある．

XMLデータ

```
<ItemSearch>
  <Request>
    <SearchIndex>Books</SearchIndex>
    <Author>
      <Name>筑波花子</Name>
    </Author>
    <Title>
      <Name>XML解説</Name>
      <Yomi>XMLかいせつ</Yomi>
    </Title>
  </Request>
</ItemSearch>
```

フィルタリング要求式

1: /ItemSearch/Request/Author/Name
2: //Name

図 8　スキップ処理可能ケース 2

```
<!ELEMENT ItemSearch (Request)>
<!ELEMENT Request (SearchIndex, Keywords?, Author+, Title?)>
<!ELEMENT SearchIndex (#PCDATA)>
<!ELEMENT Keywords (#PCDATA)>
<!ELEMENT Author (#PCDATA | Name)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Title (#PCDATA | (Name, Yomi))>
```

出現回数

記述なし	1回
?	0または1回
+	1回以上

図 9　図 5 の DTD

以上より，次のことが言える．

単純パスである XPath 式に対するスキップ処理可能ケース
フィルタリングシステムに “/a” や “/a/b” といった単純パスの XPath 式が与えられたとき，式にマッチしない要素が存在し，スキップ処理を行うことができる．

単純パスでない XPath 式に対するスキップ処理可能ケース
フィルタリングシステムに “//a” や “/*” といった XPath 式が与えられたとき，式のみでマッチしない要素を確定させることはできない．しかし，XML データのスキーマ知識を利用することでスキップ処理を行うことが可能になる．

本研究では，まず，要求式のみのでスキップ判断が可能なケースを考慮した手法の説明を行い，次にスキーマ知識を用いた前処理手法について説明を行う．

4.3 スキップ処理手法

XML ストリームデータに対するスキップ処理手法について説明する．フィルタリングシステムには要求 XPath 式が与えられており，NFA の形に変換されているものとする．スキップ可能か否かの判断材料として以下を定義する．

次状態要素集合 (nextStateElements) NFA において，次に遷移可能な状態である要素名の集合を次状態要素集合 (nextStateElements) と定義する．

スキップフラグ (skipFlag) スキップすることができる場合，スキップフラグは true になる (skipFrag = true)．

ある要素がスキップ可能であるという判断は (1) スキップフラグ，または (2) 次状態要素集合に対象の要素名が含まれているか，という条件で決める．この二つの条件をスキップ判断条件と呼ぶこととする．(1) はスキップフラグが true であるときはスキップ可能である．また，(2) では次状態要素集合に対象の要素名が含まれているときスキップ可能である．

XML ストリームデータ処理のアルゴリズムを図 10 に示す．スキップ判断条件でスキップ処理に移るか判断し，スキップが開始した場合は，開始したときの要素の終了タグが現れるまでスキップ処理を繰り返す．また，次状態要素集合の更新は，フィルタリングシステムで NFA が遷移した際に同時に行う．

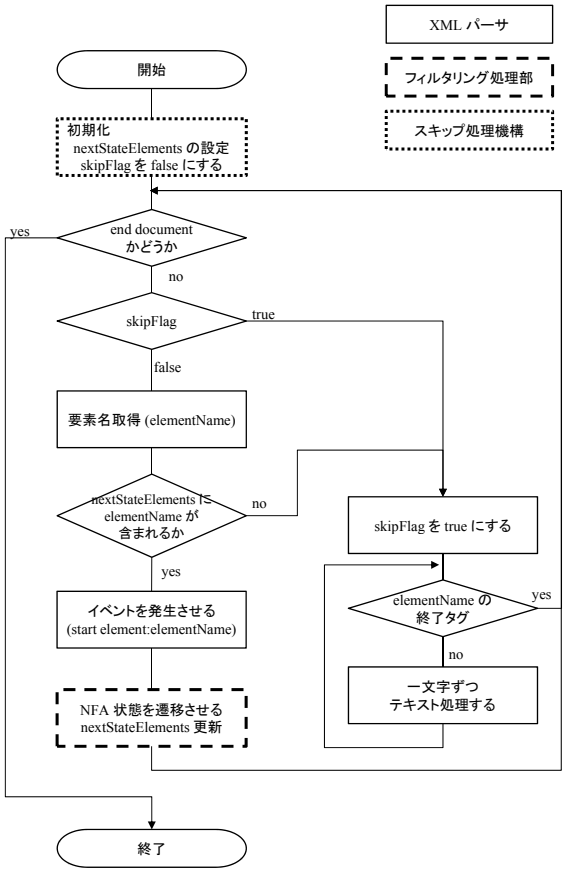

図 10　提案手法のアルゴリズム

本手法の適用例として図 5 の XML データに対して “/ItemSearch/Request/Keywords” という XPath 式でモニタリングを行う場合を挙げる．XML データに対する XML パーサとフィルタリングシステムにおけるイベント発生例及びスキップ判断材料である skipFlag と nextStateElements の値変遷について図 11 に示す．“Request” 要素の “Keywords” 要素以外の子要素は次状態要素集合に出現しないため，その要素はスキップする．

XML データ	SAX ベースの XML パーサ	フィルタリング処理部
<?xml version=“1.0”?>	start document skipFlag = false	start document skipFlag = false nextStateElements = {ItemSearch}
<ItemSearch>	start element: ItemSearch skipFlag = false	start element: ItemSearch skipFlag = false nextStateElements = {Request}
<Request>	start element: Request skipFlag = false	start element: Request skipFlag = false nextStateElements = {Keywords}
<SearchIndex>	skipFlag = true	
Books	skipFlag = true	
</SearchIndex>	skipFlag = false	
<Keywords>	start element: Keywords skipFlag = false	start element: Keywords skipFlag = false nextStateElements = {}
XML	characters: XML skipFlag = false	characters: XML skipFlag = false nextStateElements = {}
</Keywords>	end element: Keywords skipFlag = false	end element: Keywords skipFlag = false nextStateElements = {Keywords}
<Author>	skipFlag = true	
筑波花子	skipFlag = true	
</Author>	skipFlag = false	
</Request>	end Element: Request skipFlag = false	end element: Request skipFlag = false nextStateElements = {Request}
</ItemSearch>	end Element: ItemSearch skipFlag = false	end element: ItemSearch skipFlag = false nextStateElements = {ItemSearch}
	end document	end document

図 11　提案手法におけるイベント発生の流れ

本手法は YFilter と XML パーサである Crimson を拡張することで実装を行った．図 12 は本手法を実装したフィルタリング処理部の概要である．図 12 において，拡張した処理を太線で示した．

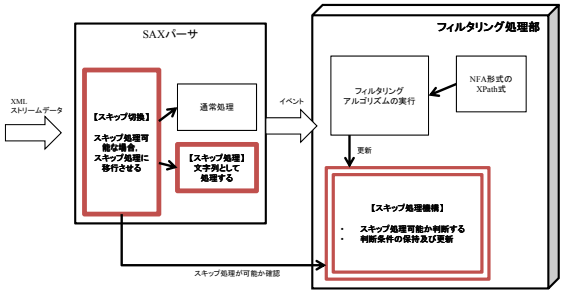

図 12　実装したシステム

拡張した処理について詳しく説明する．

まず，スキップ処理判断条件として，スキップ判断条件に用いる次状態要素集合とスキップフラグを定義するスキップ処理機構の実装を行った．次状態要素集合の要素名はハッシュマップのキー値として保持し，スキップの判断を行う際に現在の要素名とのマッチングを行うこととした．スキップフラグは boolean 型で実装を行った．

また，フィルタリング処理部と XML パーサに拡張した操作を以下に示す．

- スキップ処理の切換え

XML パーサは，スキップ判断条件を確認し，通常処理からスキップ処理への切換えを行う．判断は (i) 開始タグの “<” を読み取った直後と，(ii) 開始タグの要素名を読み取った直後に行う．(i) ではスキップフラグが true かどうかで判断し，(ii) では次状態要素集合に開始タグの要素名が含まれているかで判断する．スキップ処理が終了したら通常の処理に戻る．

- スキップ処理

スキップを開始した要素名の終了タグまで，または同じ深さレベルに戻るまでの XML データを文字列として処理する．

- 次状態要素集合の設定処理

YFilter で NFA を遷移させた際に，次状態要素集合の更新を行う．この処理は YFilter の NFA 遷移処理に追加することで実現させる．NFA が遷移すると，その状態から次に遷移可能な状態を取得する．それらの要素名を引数として与え，スキップ処理機構の次状態要素集合更新操作を呼び出す．次に遷移可能な状態の中に “//” もしくは “*” が存在する場合，“$$” を次状態要素集合に追加する．そして，“//” もしくは “*” である状態の次に遷移することができる要素名を探して次状態要素集合に含める．

また，スキップ処理機構に作成した操作を以下に示す．

- スキップ可否判断

スキップ判断条件に基づいて，スキップ処理に切換えできるかの判断を行う．判断の内容を以下に示す．重要度は 1,2,... の順で低くなる．

（1） スキップフラグが true であるか．
（2） 次状態要素集合が空集合になっていないか．
（3） 次状態要素集合に現在の要素名が含まれていないか．
（4） 次状態要素集合に “$$” が含まれていないか．

- 次状態要素集合の更新

スキップ処理機構で定義されている次状態要素集合を，引数として与えられた新しい要素集合に書き換える．

- スキップフラグの設定

スキップフラグを設定する．初期設定は false である．

4.4 スキーマを考慮した前処理手法

4.3 節では，NFA において次に遷移することが可能な状態である要素名からスキップ処理の判断を行う手法を提案した．しかし 4.2 節でも述べたとおり，“//” や “*” といった要素が一意に定まらない XPath 式が与えられた場合，全ての要素が次に遷移できる状態となってしまうためスキップ処理ができない．そこで，本節では “//” や “*” が含まれる XPath 式に対応したスキップ処理の前処理手法の提案を行う．

XPath 式における “//” は要素自身またはその子孫要素を表し，“*” は要素名を限定しないワイルドカードである．例えば，“/*//a” という XPath 式は “/a/a”，“/b/a”，“/b/b/b/b/a” 等，マッチする要素までの経路は何通りも存在する．また，スキーマからは “Request” 要素の子要素 “Keywords” は 0 または 1 回出現するといった情報を得ることができ，XPath 式の述語を用いて絞込みを行うことが可能になる．そこで，本手法ではスキーマを用いて XPath 式を書き換えることでスキップ処理をより行いやすくする手法を提案する．XML ストリームデータに対するスキーマ知識は前処理としてフィルタリングシステムに入力しておき，要求 XPath 式の書換えに用いる．

XML データ (図 5) に対するスキーマとして図 9 が与えられているとする．このスキーマから，“Keywords” 要素は “Request” 要素の子要素であるといった要素同士の関係性や，“Keywords” や “Title” の出現回数は 0 または 1 回であり，“Request” や “SearchIndex” は 1 回であるといった出現回数の情報を得ることができる．このとき，式 (3) ，式 (4) がフィルタリング要求式として与えられた場合を考える．

$$//Name \tag{3}$$

$$/ItemSearch//Keywords/Hoge \tag{4}$$

まず，式 (3) について考える．DTD から，“Name” 要素は “Author” 要素及び “Title” 要素の子要素であり，それぞれ一回しか出現しないことがわかる．また，“Author” 要素と “Title” 要素は “Request” 要素の子要素であり，“Request” 要素は “ItemSearch” の子要素であることがわかる．そのため，式 (3) を書き換えた XPath 式は式 (5) と式 (6) の二つとなる．

次に，式 (4) について考える．式 (4) は，“ItemSearch” 要素の子孫要素である “Keywords” 要素の子要素 “Hoge” を指定している．しかし，“Keywords” 要素は子要素を持たないため，この式は入力される XML データにマッチすることはありえない．このように，スキーマにマッチしない XPath 式は該当なしとし，DFA に変換しないことで無駄な状態を削除する．

$$/ItemSearch/Request/Author/Name[1] \tag{5}$$

$$/ItemSearch/Request/Title/Name[1] \tag{6}$$

5. 評価実験

5.1 実験環境

ストリームは，PC を二台用意し，XML データを分割して送信するクライアントと XML データを受信してフィルタリング処理を行うサーバで実装した．クライアント PC とサーバ PC の通信はソケット通信で行った．サーバ PC には Intel Core2 Duo (3Ghz) の CPU と 4GB のメインメモリを持つ Windows 7 マシンを用い，Java SE 1.6.0_30 でフィルタリングシステムの実装を行った．また，クライアント PC は Intel Core i5 (2.40GHz) の CPU と 2GB のメインメモリを持つ Windows 7 のマシンを用い，Java SE 1.6.0_30 で XML データ送信処理の実装を行った．

実験に用いた XML データは XML ドキュメントのベンチマークである XMark [10] でランダムに作成した．XPath 式は YFilter によって提供されている PathGenerator を用いて作成した．PathGenerator では作成する際に問合せ数，問合せパスの深さ，“*” の発生確率及び “//” の発生確率を指定することができる．

CPU の電力測定には Microsoft より提供されている Joulemeter [11] を用い，PC 全体の電力測定には HIOKI 3334 AC/DC POWER HiTESTER を用いた．

5.2 既存システムと提案手法の比較

XML データの送信時間を一定にし，サーバ側の CPU 及び PC 全体にかかる電力を測定した．XPath 式は要求式の数を 5, 10, 15 個，要求式の深さを 5，“*” 及び “//” の確率は 0 で作成したデータを使用した．

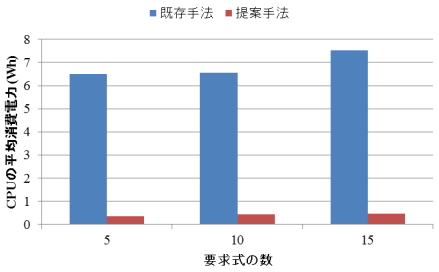

図 13 既存手法と本手法の CPU 平均消費電力

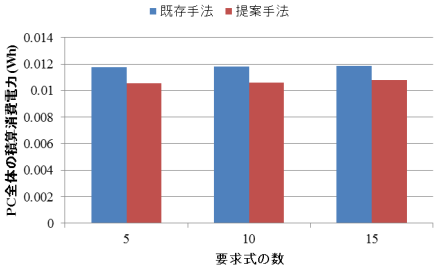

図 14 既存手法と本手法の PC 全体の積算消費電力

CPU の平均電力を図 13 に，PC 全体の積算消費電力を図 14 にグラフ化した．CPU の消費電力は，既存手法と比べて，およそ 95%削減している．また，PC 全体の消費電力は，既存手法と比べて，およそ 10%削減していることがわかった．これにより，不要な要素のスキップを行うことで CPU にかかる電力が削減できていることが分かる．提案手法では，次状態要素集合からスキップ可能な要素を判断しているため，要求式の数が少ないほどスキップ処理できる要素が増える．そのため，要求式の数が増えると消費電力も増加していると推測される．

5.3 スキーマの有効性実験

スキーマを考慮した XPath 式の書換えの有効性を検証した．既存手法と提案手法であるスキップ処理手法に対し，書換え前と書換え後の要求式をそれぞれ与え，XML データの送信時間を一定にして，サーバ側の CPU 及び PC 全体にかかる電力を測定した．XPath 式は要求式の数を 5，10，15 個，要求式の深さを 5 とし，“*” 及び “//” の発生確率を 0.2 とした．また，要求式の重複はないものとし，要求式は必ず XML データにマッチするデータを用いた．

前処理として XPath 式の書換えを行った．書換えによる

表 1 XPath 式の数とオートマトン状態数の変化

XPath 式の数	書換え前	5	10	15
	書換え後	12 (12)	24 (22)	60 (47)
状態数	書換え前	9	23	29
	書換え後	22	22	23

XPath 式の数及びオートマトンの状態数の変化を表 1 にまとめた．書き換え後の式には重複した XPath 式も存在したため，重複した XPath 式は一つの式とみなした数を括弧で示した．

XPath 式の書換えを行うことで，XPath 式の数は全体的に大きく増加した．これは，XPath 式の書換えで “//” や “*” といった一意に決まらない状態を削除し，XML データのスキーマから新たな経路を作成しているからである．オートマトンの状態数は XPath 式の数が 5 のときは増加しているが，10，15 のときは減少している．今回の実験では要求式の数が少なかったこと，式の書換えで重複が生じたことや同じ経路を持つ XPath 式が多かったことが原因であると考えられる．また，“//Name” のような要求式があった場合，“Name” 要素が XML データ内の複数箇所に存在するならば，書換え後の XPath 式が増加して状態数も増加する可能性がある．今回の実験ではこのケースが存在しなかったことも原因として考えられる．

既存手法と提案手法において電力を測定し，CPU の平均電力を図 15 に，PC 全体の積算消費電力を図 16 にグラフ化した．グラフより前処理ありの提案手法が電力消費を抑えることができていることがわかる．スキップ処理できなかった要素をスキップ可能にすることで処理を削減しているからだと推測される．また，XPath 式の書換えを行わない場合，既存手法よりも提案手法の方が消費電力が高くなっている．これは，既存手法に比べ，提案手法ではスキップ処理可能か判断する処理が増えているためであると考えられる．

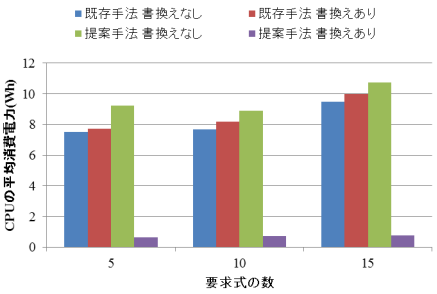

図 15 スキーマの有効性実験における CPU 平均消費電力

6. ま と め

本稿では，フィルタリング要求式に不要な要素をスキップすることで電力消費を抑える手法を提案した．本手法は NFA 変換された XPath 式から，次に遷移可能な状態の集合を取得することで XPath 式にマッチしない要素の構文解析処理をスキップする．また，XPath 式において経路が一意に定まらない式を書き換えることでスキップ処理の効率化を図っている．最後に，ベンチマークデータを用いた実験によって，電力消費の面にお

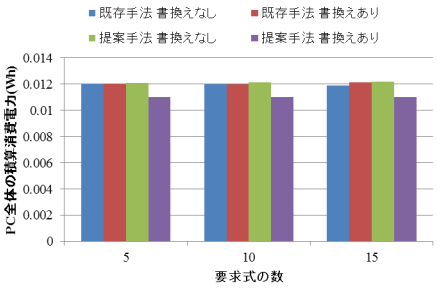

図 16　スキーマの有効性実験における PC 全体の積算消費電力

いてスキップ処理を行う手法が省電力に有効であることを確認し，XPath 式を書き換えることでスキップ処理が効率化することを確認した．

XPath 式の書換えの際，書換え後の経路が無限に存在する可能性がある．そのため，今後の課題として，スキップ処理を無効化している “//” や “*” を含む XPath 式の処理を見直し，更なる消費電力の削減が挙げられる．また，様々な要求に対してスキップ処理を可能にするため，XPath 式の述語及び完全構文に対応するスキップ処理についても今後の課題である．

謝辞　本研究において有益なご助言を頂いた東京工業大学の横田治夫教授，NTT サイバースペース研究所の鬼塚真氏に感謝の意を表す．

本研究の一部は科学研究費補助金基盤研究 (A) (#21240005)，基盤研究 (A) (#22240005) 及び若手研究 (B) (#23700102) による．

文　　献

[1] World Wide Web consortium. Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/xml/`.

[2] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, Vol. 28, pp. 467–516, December 2003.

[3] Simple API for XML. `http://www.saxproject.org/`.

[4] The Apache Xerces Project. `http://xerces.apache.org/`.

[5] Crimson. `http://xml.apache.org/crimson/`.

[6] Mehmet Altinel and Raymond To Franklin, Michael J. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pp. 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[7] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, Vol. 29, pp. 752–788, December 2004.

[8] Document Type Definition. `http://www.w3.org/TR/html4/sgml/dtd.html`.

[9] XML Schema. `http://www.w3.org/XML/Schema`.

[10] XMark-An XML Benchmark Project. `http://www.xml-benchmark.org/`.

[11] Joulemeter: Computational energy measurement and optimization. `http://research.microsoft.com/en-us/projects/joulemeter/`.