シーケンス演算子の効率的実装と適応的処理機構

島貫 稔之† 川島 英之‡

†筑波大学 情報科学類
305-8573 茨城県つくば市天王台 1-1-1
shimanuki@hpcs.cs.tsukuba.ac.jp

‡筑波大学システム情報系（計算科学研究センター）
305-8573 茨城県つくば市天王台 1-1-1
kawashima@cs.tsukuba.ac.jp

あらまし 端的な攻撃手法の一つにポートスキャンがある。この検知にはシーケンス演算子が利用可能である。本研究ではまず代表的なCEPエンジンであるEsperシステムに、シーケンス演算の省メモリな処理方式であるAIS方式を実装する。我々の実装がEsperシステムのデフォルト実装に比して省メモリかつ高性能であることを実験により示す。次に、Esperシステムのデフォルト実装を改善した実装を行い、それがAIS方式よりも最大で12倍ほど高速である事を示す。最後に同方式をAIS方式と組み合わせて適応的に動作させる技術を開発する。我々の提案方式は高速かつロバストであることを実験により示す。

Efficient Implementation and Adaptive Processing Architecture of Sequence Operator

Toshiyuki Shimanuki† Hideyuki Kawashima‡

†College of Information Science, University of Tsukuba
1-1-1, Tennodai, Tsukuba, Ibaraki 305-8573, Japan
shimanuki@hpcs.cs.tsukuba.ac.jp
‡Faculty of Information, Systems and Engineering / Center for Computational Sciences, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan
kawashima@cs.tsukuba.ac.jp

Abstract Detection of port scan is one of the most straight forward attack detection techniques. This paper first implements AIS which is a memory efficient query processing technique of sequence operator. We show that our implementation is not only more memory efficient but also faster than default implementation of Esper system. Second, we make Esper implementation efficient, and show that it is faster than AIS by a factor of 12. Finally we develop an architecture that adaptively adopt the two techniques, and show that the architecture is not only fast but also robust through experiment.

1　はじめに

攻撃検知をエンドポイントでリアルタイムに行うことは肝要である。攻撃には様々なものが存在するため、それぞれに適した検出技術が利用される。例えば分類、クラスタリング、異常検知、そして複合イベント処理（CEP）[1-6] などである。標準的な攻撃としては DSTPORT を順々にスキャンしていく方式が考えられるが、CEP におけるシーケンス演算を用いればこれを検知する問合せを容易に記述できる。

本研究で述べるシーケンス演算とは順序性を有するパタンの検出である。例えばイベントAの後にBが発生するようなイベント（シーケンスイベント）を検出するためにシーケンス演算は2006年 Diao らにより提案がされた [1]。例えば DSTPORT アクセスの単調増加を記述するには、ユーザがクエリ 1 に示すような問合せを記述すれば良い。SEQ がイベント間の順序を表す。

```
SELECT ＊
FROM   packet
PATTERN SEQ(A, B, C, D, E)
WHERE B. dstport ＞ A. dstport
AND C. dstport ＞ B. dstport
AND D. dstport ＞ C. dstport
AND E. dstport ＞ D. dstport
```

クエリ 1：DSTPORT 単調増加の検知

シーケンス演算の実現にはそれ以前に存在していたリレーショナルストリーム処理 [7-11] でも可能である点には注意が必要である。ただしそれには自己結合演算を行う必要がある。自己結合演算の計算量は N 件のタプルに対して $O(N^2)$ であり高価である。シーケンス演算の対象はストリームデータである。ストリームデータは頻繁に到着するために N は大きくなるのが通例である。従ってシーケンス演算のためにリレーショナルストリーム処理を用いることは性能の観点からは現実的ではない。

本研究の貢献は次の 2 点である。第 1 の貢献は、代表的なCEPエンジンであるEsperに、シーケンス演算の省メモリな処理方式であるAIS 方式 [1] を実装することである。我々の実装が Esper のデフォルト実装（「Esper 方式（既存実装）」と以後は表記）に比して省メモリかつ高性能であることを実験により示す。第 2 の貢献は、Esper 方式（既存実装）を改善した実装（「Esper 方式（効率的実装）」と表記）を行い、それを AIS 方式と組み合わせて適応的に動作させる技術を開発する。我々の提案方式は高性能かつロバストであることを実験により示す。

本論文の構成は次の通りである。2 節では関連研究を述べる。3 節ではシーケンスパタン検知方式を述べる。4 節では Esper システムの性能評価を行う。5 節では適応的機構について述べる。最後に 6 節では論文をまとめる。

2　関連研究

2.1　リレーショナルストリーム処理

流れ来るストリームデータからリアルタイムにイベントを検出するために様々な技術が開発されてきた。当初提案された技術はリレーショナルストリーム処理である。リレーショナルストリーム処理はリレーショナルデータモデル、即ちイベント間の順序性を考慮しないモデル、に基づいてストリーム処理を行う。ただしストリームの長さは無限であり、既存の演算の一部（例：集約、結合）は処理不能であった為、新たなリレーショナル演算子としてウィンドウ演算子を導入し、無限長のストリームを有限長に変換することで、それらのリレーショナルデータ処理を可能にしている。これに伴いウィンドウ集約やウィンドウ結合などの演算子が開発され、それらはイベント検知に使われてきた。

これまでに様々なリレーショナルストリーム処理エンジンが開発されてきた。汎用システムとしては STREAM [9], TelegraphCQ [8], Aurora [7], そして Borealis [10] がある。一方、専用システムも存在する。車載システムに特化した例として eDSMS [11] がある。

ストリームデータは時々刻々と到着するため、時間的依存性を考慮することはストリームデータ処理において自然である。ところが、リレーショナルデータモデルにおけるデータの最小単位たるタプルにおいて、タプルの間に依存性は理論的に存在しない。従ってリレーショナルデータモデルを用いて順序関係を扱うことは自然ではない。これを扱う一つの方法は自己結合であるが、その計算量は $O(N^2)$ と高価である。これは頻繁に到着するストリームデータにはそぐわない。

2.2 複合イベント処理(CEP)

時間的依存性を有する問合せ、例えば、アクセスポート番号が順々に大きくなるようなデータパタンの検出、を行うために、複合イベント処理(CEP)が提案された。CEP 技術の例としては、SASE [1], SASE++ [2], ZStream [3], NEEL [5], MQOS [6], E-Cube [4]などがある。

順序性を検知するために SEQ なる演算子を提案したのは SASE である。SEQ 演算子中ではクリネ閉包を用いることも許される。例えば (A, B+, C) なるパタン問合せを記述すると、Aの後→Bの繰返し→Cというシーケンスイベントを取得可能になる。SASE [1]には後継技術として SASE+ [14] と SASE++ [2]がある。

本研究ではアクセスする DSTPORT 番号が徐々に増加するパタンを検出することを考える。すなわち、(A, B, …, N)なるパタンのみ考える。このパタンはSASEにおいて提案されたAISを用いることで効率的に達成可能である。それゆえ本研究では AIS に焦点を当てる。

2.3 CEP エンジン Esper

シーケンスパタンを検知する方式として、学術的にはSASEで[1]で導入されたAIS が広く知られている。一方、学術界でも産業界でも幅広く使われている CEP エンジンに Esper システム[12] がある。Norikra においても Esper システムが使われている [13]。Esper は広く利用されている一方、その実装方式あるいは設計方式について詳細が公開されていない。3.2 節では我々が Esper システムのソースコードを読んで分析した結果を示す。

3 シーケンスパタン検知方式

シーケンスパタン(例:A → B → C)の検知を、自己結合を使わずに効率的に処理する方式として、AIS 方式ならびに Esper システムが用いている方式(これを以後は **Esper 方式**と記述する)がある。AIS 方式と Esper 方式の違いを概説する。AIS 方式は実体化を遅らせることにより中間データ領域を省メモリ化して管理する。すなわち処理時間を犠牲にしてメモリ使用量を削減する方式と見なせる。Esper 方式はイベント到着に伴い実体化を行うため、中間データ領域は AIS 方式に比べて大きくなる。ただし最終イベント到着に伴う Esper の実体化時間は即時実体化のために AIS 方式よりも短くなる。以上より、両者を比較すると AIS は lazy な方式であり、Esper は eager な方式であると考えられる。以下では AIS 方式と Esper 方式について詳述する。

3.1 AIS 方式(lazy)

AIS 方式においてシーケンスパタンは NFA (Non Deterministic Finite Automata)として表現される。NFA の各状態に AIS と呼ばれるスタックを用意し、各状態に適合するイベントは該当スタックに格納される。スタック中のイベントは一つ前のスタックにおいて最も時間の近いイベントである RIP (the most Recent Instance in the Previous stack)[1] にリンクを張る。NFA の受理状態に対応する AIS にイベントが push されると、そのイベントから RIP を利用してシーケンスパタンを構築し、結果を出力する。この方式は受理状態に対応するイベントが到着するまではイベントを展開せず、当該イベントが到着してから初めて構築処理を行う。そのため Esper 方式に比べて AIS 方式は構築処理に長時間を要する。AIS 方式はイベント到着間隔を無駄にする点が弱点である。

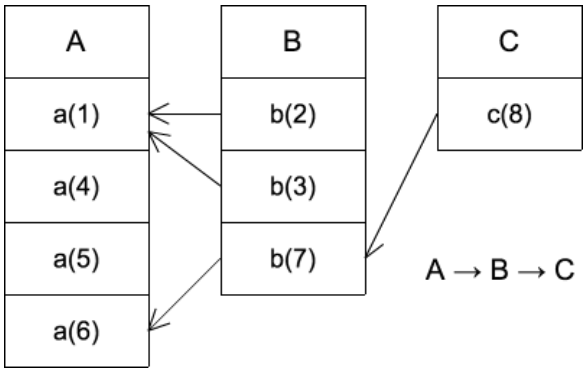

図 1 AIS 方式

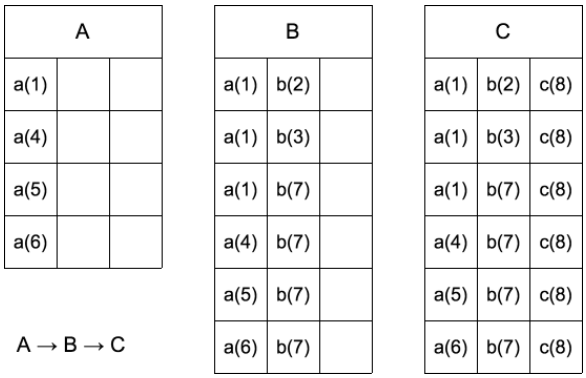

図 2 Esper 方式

図 1 に AIS による問合せ処理構造の例を示す。問合せイベントごとにスタック構造（AIS）が用意される。この構造は次のパタン問合せに対応する。

$$\mathrm{A} \rightarrow B \rightarrow \mathrm{C}$$

この場合のイベントの到着パタンは次の通りである。括弧内の数字は時刻印を表す。

$$\mathrm{a}(1), b(2), \mathrm{b}(3), a(4), a(5), a(6), b(7), c(8)$$

$b(2)$と$b(3)$の RIP は$a(1)$、$b(7)$の RIP は$a(6)$、そして$c(8)$の RIP は$b(7)$である。出力イベントは $\mathrm{a}(1), b(2), c(8)$をはじめとする 6 パタンである。6 パタンの詳細は図 2 の最右に示されている。

3.2 Esper 方式（eager）

Esper 方式においてもシーケンスパタンは AIS 方式同様に NFA として表現される。AIS では到着イベントは他のイベントと結合されずに保持されるが Esper 方式では AIS とは逆に即時結合を行う。それゆえ受理状態に該当するイベントが到着すると、既に構築してあった部分的シーケンスイベントにそのイベントを追加し、そして出力を行う。到着データレートが極めて短い時には Esper 方式が不利になる可能性もあるが、そのような状況は例外であると本論文は見なし、イベント到着の間に、到着イベントの実体化が完了する場合を本論文は想定する。

図 2 に Esper 方式の問合せ処理構造の例を示す。AIS 方式と異なり Esper 方式ではイベントが到着すると即座に部分シーケンスの構築を行う。この例における問合せ内容ならびにイベント到着パタンは図 1 における例と同一である。

Esper 方式においては検出対象のイベント毎に配列のリストを作成する。各配列のサイズは出力予定イベントシーケンスの長さである。この場合は問合せパタンが $\mathrm{A} \rightarrow B \rightarrow \mathrm{C}$ であるから配列のサイズは 3 となる。イベント到着毎に配列の構築を行う。受理状態であるイベント（この場合は C）以外は結果シーケンス構築補助のために用いられる。最終的に出力されるシーケンスパタンは最右に示されている。

4 Esper の性能評価

Esper 方式の性能を評価するために、Esper システムを修正して AIS 方式を導入し、性能比較を行った。ただし Esper 方式について本節では Esper 元々の実装を利用した。方式と実装を区別するため、本節で評価する Esper 方式を **Esper 方式（既存実装）**と表記する。

4.1 実装方法

この節では、AIS を Esper システムへ実装した方法について述べる。我々はイベント到着時にクエリの各シーケンスパタンにマッチするかを判定し、RIP にリンクを張る。この実現には RIP フィールドが必要であるので、
com.espertech.esper.event.bean.BeanEven

tBean を拡張して RIP フィールドを追加した。

また、AIS ではスタックが必要であるので、com.espertech.esper.pattern.EvalFollowedByFactoryNode を拡張し、ここにスタック構造を追加した。

AIS のスタックにイベントを追加するために、com.espertech.esper.core.service.EPRuntimeImpl 内の processMatches メソッドを拡張して実装を行った。

クエリにイベントがマッチするかは、Esper に従前より備わっている実装を利用した。照合判定処理の結果が真であり、到着イベントがシーケンスパタンの最終イベントである場合には、com.espertech.esper.core.service.EPRuntimeImpl 内の searchPath メソッドで RIP を検索し、結果を出力する方式を採用した。

4.2 実験(処理時間)

Esper 方式(既存実装)と AIS 方式の性能を比較するために、問合せ処理に要する時間を測定した。問合せ内容をクエリ2に示す。この問合せはシーケンスパタン(A → B → C)を検出する。この問合せを処理するに当たっては Esper システムの問合せ処理系を用いた。

```
SELECT *
FROM packet
PATTERN[every A = Event[id = 'a']
→ every B = Event[id = 'b']
→ every C = Event[id = 'c']]
```

クエリ 2: Esper システム評価で用いた問合せ

実験環境は次の通りである。CPU は 2.3GHz Intel Core i7 である。ガベージコレクション方式は Parallel Compacting Collector である。Java は SE7, version 1.7.0_65 である。Esper システムは version 5.0.0 である。実験データは筆者らが採取したものを用いた。

実験結果を図3に示す。図の横軸は JVM が使用可能なヒープメモリサイズを表し、縦軸は処理時間(ミリ秒)を表す。なお、縦軸はログスケールである点に注意されたい。Esper 方式と AIS 方式それぞれ 7 パタンずつ、合計で 14 パタンの測定結果を図 3 に示す。各パタンは技法と使用可能な JVM ヒープメモリのサイズから構成されている。例えば 128-Esper は JVM ヒープメモリとして 128MB を割り当てて Esper 方式を走らせた場合の実験結果を表している。即ち同じ数字を有する技法についての比較に意味がある。例えば 128-Esper と 128-AIS を比較することが意味のある分析である。

図 3 に示されるグラフが語る事実を述べる。まず、AIS 方式は全ての状況において Esper 方式(既存実装)よりも高い性能を示していることがわかる。最も性能差があるのはヒープサイズを 8192MB に設定した場合である。このとき AIS 方式は Esper 方式(既存実装)に比べて処理時間が 12.77 倍短い。使用可能なヒープサイズを大きくすれば性能差はさらに広がることをグラフは示唆している。

観察結果から次にわかることは、使用可能なヒープメモリサイズが同じであれば AIS 方式はより大きなウィンドウを処理することができることである。なぜならばグラフの右端がプログラムを実行できた最終状態だからである。従って、AIS 方式は Esper 方式(既存実装)よりもメモリ効率が良い。

4.3 実験結果に対する考察

実験結果はアルゴリズムから導かれる直観とは異なる。直観的には AIS 方式は使用メモリ量が小さい一方で処理時間は長く、Esper 方式は使用メモリ量が大きい一方で処理時間が短いと考えられる。しかしながら全ての場合において AIS 方式は Esper 方式(既存実装)よりも高い性能を示している。

この理由を端的に述べれば、Esper 方式(既存実装)は実装方法が極めて非効率的であるからである。Esper は高機能な複合イベント処理ライブラリである。そのため、本研究に必要の無い処理が多く含まれており、正確な値を測定することが困難である。

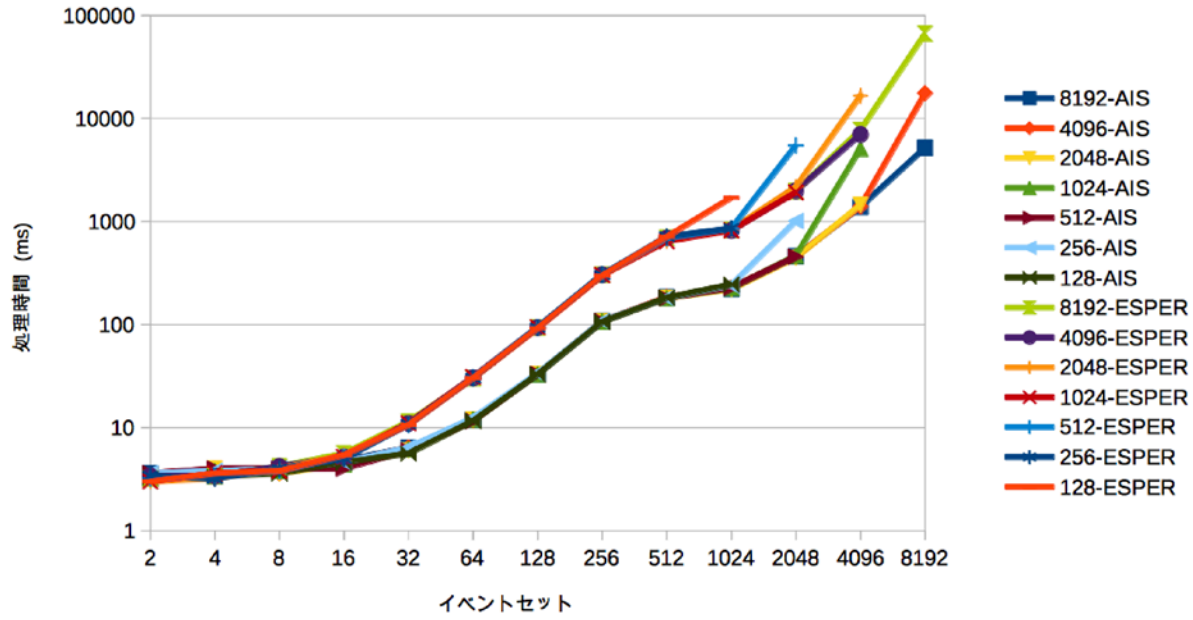

図 3 イベント処理時間

5 効率的実装と適応的機構

5.1 Esper 方式(効率的実装)

前節で述べたように Esper 方式(既存実装)には実装上の問題がある。そのために Esper 方式と AIS 方式を適切に比較することができなかった。そこで我々は Esper 方式と AIS 方式について、適切に比較することのできるよう、Esper 方式の効率的なプロトタイプシステムを実装した。これを **Esper 方式(効率的実装)**と表記し、以下で報告する。Esper 方式(効率的実装)では Esper 方式の基本原理のみを実装することで、測定に最適な環境を用意しているため、Esper 方式(既存実装)のように非合理的な低性能は示されず、モデルに合理的な性能が示される。

5.2 適応的機構

前述したように AIS 方式と Esper 方式には、空間効率と時間効率についてトレードオフの関係がある。これらの長所を適応的に用いる機構を実装した。

Esper 方式は理論的には処理時間が短い一方、メモリを大量に利用する必要がある。そのため、問合せにおけるシーケンス演算子中のイベント数が多い場合にはメモリが不足する可能性がある。Java 言語による実装においては OutOfMemory Exception によって処理が停止する可能性がある。AIS 方式は Esper 方式に比べて処理時間が長い一方、使用メモリ量が少ない。そのためにメモリ不足に対して AIS 方式は Esper 方式よりもロバストだと見なせる。

我々は Esper 方式と AIS 方式の長所を組み合わせる適応的機構を提案する。適応的機構はメモリが不足しない場合には Esper 方式同様の高速処理を実現するが、システム稼働中にメモリ不足が検知された場合には即座に問合せ処理方式を AIS 方式に切り替える。これにより AIS 方式同様の効率的メモリ管理を行う。

適応的機構の挙動を述べる。同機構は動作開始直後には Esper 方式と AIS 方式を共に動作させる。この時ユーザへ問合せ処理結果を返送するのは Esper 方式の結果のみである。AIS 方式ではスタックにイベントを格納するが、結果シーケンスイベントの展開処理は一切行わず、切り替え時に備えるのみである。

OutOfMemory Exception がシステム稼働中に発生した場合、Esper 方式の動作を停止させる。そして AIS 方式が即時に処理を引継ぐ。これが可能であるのは AIS 方式において全てのイベントを並行的に処理しているからである。引継ぎ処理以降、AIS 方式が結果シーケンスイベントをユーザに返送する。我々の現在の実装は Java 言語のみであるが、同様の実装は他言語でもメモリ不足を検知できれば可能である。

5.3 実験内容

適応的機構のパケット処理速度に関する性能を評価するために実験を行った。実験ではクエリ 1 に示した問合せであり、シーケンスパタン (A → *B* → C → D → E)を検知する。

実験環境は次の通りである。CPU は 2.3GHz Intel Core i7 である。ガベージコレクション方式は Parallel Compacting Collector である。Java は SE7, version 1.7.0_65 である。Esper システムは version 5.0.0 である。データセットとしては MWS Datasets 2014 [15]の Practice Dataset 2013/practice_2.pcap を用いた。そして JVM が使用可能ヒープメモリサイズは 1024MB と設定した。

5.4 実験結果

実験結果を図 4 に示す。横軸は経過時間、縦軸は総パケット処理量である。スループットではなく積算である点に読者は注意されたい。

Eager および Adaptive は時刻 121程度まではほぼ同一の処理量を示している。Eagerは時刻 121 近辺で停止している。そのため、その後 Eager は総処理量を増やせていない。停止の理由は Eager 方式でヒープメモリが不足したことである。

Lazy は Adaptive および Eager に比べて低い処理量を示している。ただし Lazy は Eager と異なり停止していない。そのため、時間が経過するにつれて Lazy のパケット総処理量は Eager のそれよりも大きくなると考えられる。一方、Adaptive は時刻 121 近辺で問合せ処理方式を Esper 方式(効率的実装)から AIS 方式に切り替えている。そのため時刻 121 以降の Lazy と Adaptive のグラフの増加率は同一であり、Lazy のパケット総処理量が Adaptive のそれを上回ることはない。総パケット処理量の増加率が時間と共に劣化する理由は中間結果数が増大するために、その維持・管理コストが増大することである。

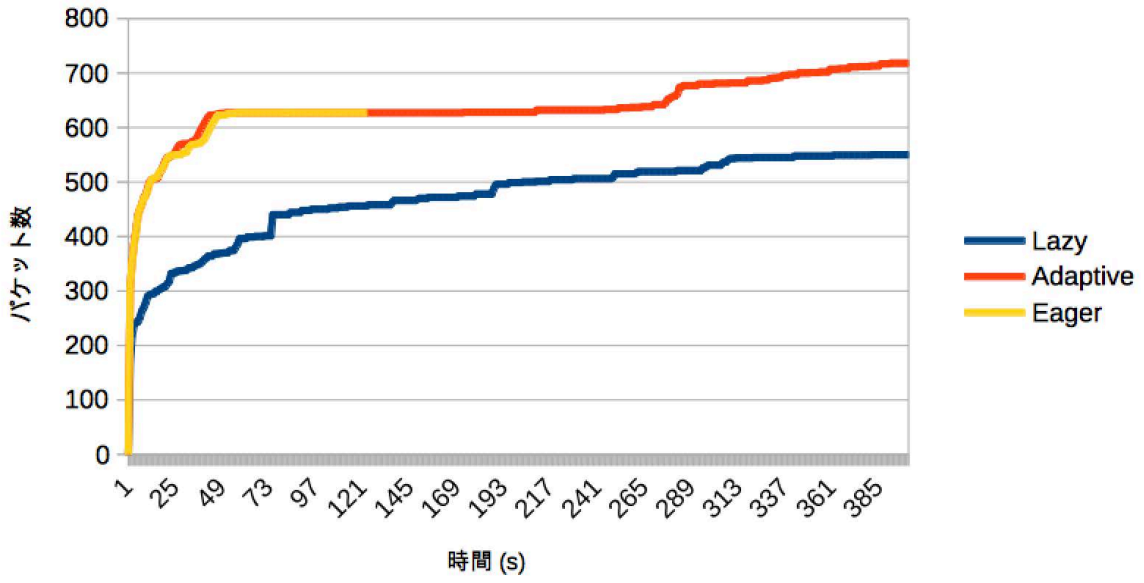

図 4 適応的機構(Adaptive)、AIS 方式(Lazy)、Esper 方式(効率的実装)(Eager)のパケット処理量

6 まとめ

本研究ではまず代表的な CEP エンジンである Esper システムに、シーケンス演算の省メモリな処理方式である AIS 方式を実装した。我々の実装が Esperシステムのデフォルト実装に比して省メモリかつ高性能であることを実験により示した。次に、Esper システムのデフォルト実装を改善した実装を行い、それが AIS 方式よりも 12 倍程度高性能である点を示した。最後に同方式を AIS 方式と組み合わせて適応的に動作させる技術を開発した。提案方式は高性能かつロバストであることを実験により示した。

今後の課題はシーケンス演算子による広範な攻撃検知と適応的機構の精緻化である。

謝辞

本研究は JSPS 科研費 25280043HA、24500106 の助成を受けたものである。貴重なデータセットをご準備下さった MWS2014 実行委員会に心からお礼申し上げる。

参考文献

[1] Eugene Wu, Yanlei Diao, Shariq Rizvi: High-performance complex event processing over streams. SIGMOD Conf. 2006: 407-418.

[2] Haopeng Zhang, Yanlei Diao, Neil Immerman: On complexity and optimization of expensive queries in complex event processing. SIGMOD Conf. 2014: 217-228.

[3] Yuan Mei, Samuel Madden: ZStream: a cost-based query processor for adaptively detecting composite events. SIGMOD Conf. 2009: 193-206.

[4] Mo Liu, Elke A. Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, Abhay Mehta: E-Cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing. SIGMOD Conf. 2011: 889-900.

[5] Mo Liu, Elke A. Rundensteiner, Daniel J. Dougherty, Chetan Gupta, Song Wang, Ismail Ari, Abhay Mehta: NEEL: The Nested Complex Event Language for Real-Time Event Analytics. BIRTE 2010: 116-132.

[6] Fuyuan Xiao, Masayoshi Aritsugi: Nested Pattern Queries Processing Optimization over Multi-dimensional Event Streams. COMPSAC 2013: 74-83.

[7] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, Stanley B. Zdonik: Aurora: a new model and architecture for data stream management. VLDB J. 12(2): 120-139 (2003).

[8] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel Madden, Frederick Reiss, Mehul A. Shah: TelegraphCQ: An Architectural Status Report. IEEE Data Eng. Bull. 26(1): 11-18 (2003).

[9] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, Jennifer Widom: STREAM: The Stanford Stream Data Manager. IEEE Data Eng. Bull. 26(1): 19-26 (2003).

[10] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, Stanley B. Zdonik: The Design of the Borealis Stream Processing Engine. CIDR 2005: 277-289.

[11] 勝沼 聡, 本田 晋也, 高田 広章:「リアルタイム性を考慮した車載システム向けDSMSのクエリ処理効率化」、DEIM, D3-5, Mar, 2014.

[12] Esper: http://esper.codehaus.org/

[13] Norikra: http://norikra.github.io/

[14] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom: SASE+: An Agile Language for Kleene Closure over Event Streams. UMass Tech. Rep. 07-03

[15] 秋山満昭, 神薗雅紀, 松木隆宏, 畑田充弘, "マルウェア対策のための研究用データセット～MWS Datasets 2014～," 情報処理学会 研究報告コンピュータセキュリティ(CSEC) Vol. 2014-CSEC-66, No. 19, pp. 1 - 7, 2014.