# Exact Multi-pattern String Matching on the Cell/B.E. Processor

集積システム専攻　田原慎也 (07M51298)

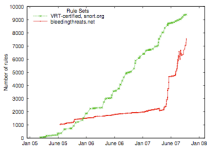
出展：Computing Frontiers 2008

## Exact Multi-pattern String Matching on the Cell/B.E. Processor

- 著者
  - Daniele Paolo Scarpazza (IBM)
  - Oreste Villa
  - Fabrizio Petrini (IBM)
- 概要
  - マルチコア環境における効果的な文字列探索の実装と評価
    - 文字列探索の重要性が高まる
      - サーチエンジン，ウィルス・スパイウェア探索など
    - データ量増加に伴う処理量，リアルタイム応答性への要求

## Introduction

- 取り扱う（処理する）データの増大
  - データの中に混在する脅威の探索とそれからの保護
    - ウィルス，マルウェア，スパイウェア，スパムなどの脅威
    - ネットワーク侵入探知システム（NIDS）
      - リアルタイム文字列探索
- ネットワークの高速化と脅威の増大
  - ネットワーク
    - 10Gbpsを超えるリンク速度
  - 増大する脅威
    - 右のグラフ参照

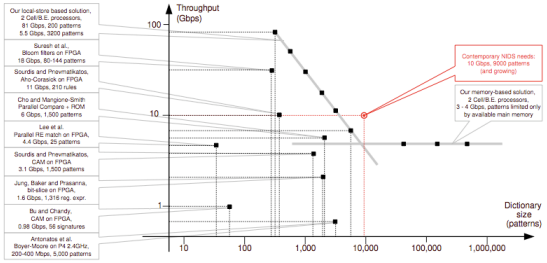



## Introduction

- 高性能（高速）な文字列探索
  - ハードウェアによる実装が伝統的
    - FPGAの利用（多くの場合）
      - 可能な限りの並列度を抽出したアルゴリズムが実装しやすい
    - Bloom filters
      - FPGAベースの文字列探索として有名なアルゴリズム
      - 空間利用効率がいい
  - 決定性有限状態オートマトン（DFA）の利用
    - Aho-Corasick アルゴリズム
      - DFAベースのアルゴリズムとして有名
      - 多くの派生アルゴリズムが存在
        - Knuth-Morrise Pratt, Boyer Moore, Rabin Karp, Commentz-Walter, Wu-Manber

## Introduction

- 文字列探索の解法に要求されるもの
  - スループットと辞書サイズ （下のグラフ参照）

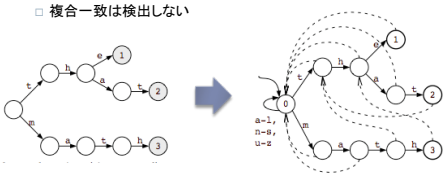



## Aho-Corasick Algorithm

- 長さ$m$ の入力テキスト $T$ に対して与えられた辞書に含まれるパターンの存在を探索するアルゴリズム
  - パターン $p$
    - アルファベットから構成される有限の連続した文字列
  - 辞書 $P$
    - パターンの集合 $P = \{p_1, p_2, \ldots, p_k\}$
  - 入力テキストをもとにオートマトンを進めていき，終了状態に至る度に辞書に含まれるパターンとの一致が起きたとする
    - AC-fail ：NFA
    - AC-opt ：DFA （AC-failの改良版）

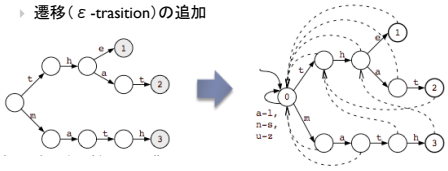
## Aho-Corasick Algorithm

- AC-fail オートマトン
  - 非決定性有限状態オートマトン（NFA）
  - 与えられた辞書のキーワード木（*trie*）がもとになったもの
    - *trie*
      - *root* からある*node*に至る*edge*にラベリングされた文字を並べると辞書に登録されたパターンとなるような木構造
      - 複合一致は検出しない



## Aho-Corasick Algorithm

- AC-fail オートマトン
  - *trie* からAC-fail オートマトンへの変換
    - ノード→状態
    - エッジ→遷移
    - *root* →初期状態
    - *i* ノード→終了状態
    - 遷移（ε -trasition）の追加

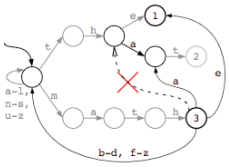


## Aho-Corasick Algorithm

- AC-opt オートマトン
  - 決定性有限状態オートマトン（DFA）
  - AC-fail オートマトンの改良版
    - AC-fail では，1入力に対し複数の遷移が起きる可能性がある
    - ε -遷移の除去

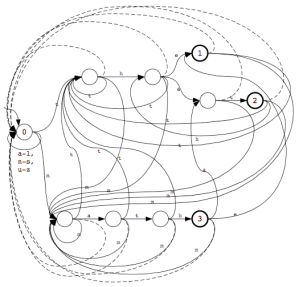
**Algorithm 1** The AC-fail variant of the Aho-Corasick algorithm.

```
1:  q ← 0; // initial state (root)
2:  for all i ∈ {1, 2, ... m} do
3:    while δ(q, T[i]) = {} do
4:      q := f(q); // failure transition
5:    end while
6:    q := δ(q, T[i]); // regular transition
7:    if out(q) ≠ {} then
8:      report i, out(q);
9:    end if
10: end for
```



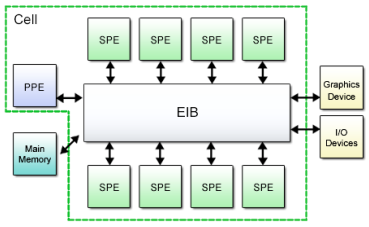
## Aho-Corasick Algorithm

- AC-opt オートマトン



## （参考）Cell/B.E.

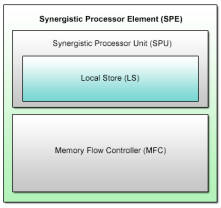
- Cell/B.E.の物理構成






## （参考）Synergistic Processor Element

- マルチメディア系演算を得意とするプロセッサコア
  - 演算系プロセッサコア
  - 単純な構成で高い計算能力
- 構成
  - Synergistic Processor Unit (SPU)
    - 演算処理
  - Local Store (LS)
    - メモリ領域 (256KB)
  - Memory Flow Controller (MFC)
    - 通信処理

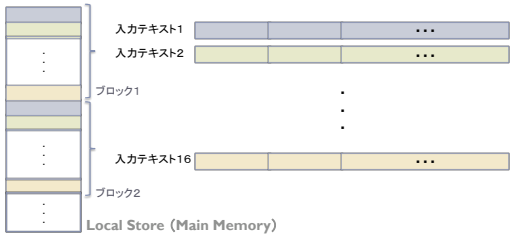


## A Fast, Local-Store Solution

- Local Store（以下LS）ベースの文字列探索法
  - メインメモリへのアクセスを避け，高い性能を発揮
    - 各SPEがLSに辞書情報（STT）を保持
    - 辞書がLSに保持できるほど小さい（もしくは分割されている）
  - タイル（*tile*）構成
    - 1つのSPEのLSに格納し実行できるDFAで実装
    - 200パターンの辞書，5Gbpsのスループット
- 状態遷移テーブル（State Transition Table, STT）
  - ライン（行）数は状態数で，1ラインは64byte長
    - 1ライン：32エントリ（32種のシンボル）
    - 1エントリ：2byte（16bit ポインタ）

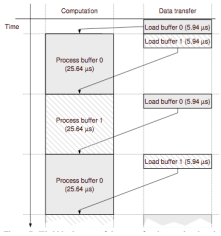
## A Fast, Local-Store Solution

- メモリ（データ）レイアウト
  - 入力データ配置の最適化
  - 16の入力テキストを1byteずつ16byteおきに配置




## A Fast, Local-Store Solution

- メモリ（データ）レイアウトの利点
  - 同じSTTで16オートマトンの最適な並列実行が可能
    - SIMD演算の有効利用
    - データ転送時間の隠蔽（右図）
    - 具体的には・・・（Cによる実装）
      - レイアウト前：
        - 19 clock cycles / 1 transition
        - 1.35Gbps
      - レイアウト後：
        - 5 clock cycles / 1 transition
        - 5.11Gbps
        - 3.79 × の高速化

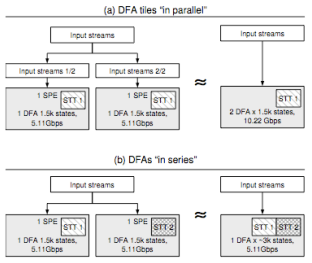



## A Fast, Local-Store Solution

- 複数SPEの活用
  - 1SPE：200パターンの辞書で5.11Gbpsのスループット
  - スループットを高めるか，辞書サイズを増やすか
  - スループット高（“in parallel”）
    - 同じSTTをLSに保存
    - 入力テキストを複数に分けて異なる部分を各SPEで文字列探索
  - 辞書サイズ大（“in series”）
    - 異なるSTTをLSに保存
    - 同じ入力テキストを各SPEで文字列探索
  - ハイブリットな実装も

## A Fast, Local-Store Solution

- 複数SPEの活用例（2SPEの場合）




## Using Larger Dictionaries

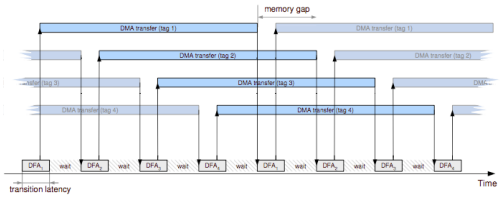
- メインメモリベースの文字列探索法
  - STTをメインメモリに保持し続ける探索法
    - 大きな辞書をサポート（STTがLSに格納できない）
    - 辞書サイズによらず，一定の性能を保証
  - ボトルネックは**メインメモリからのデータ転送**
- 考慮すべき点
  - メインメモリの利用を最大化にする並列戦略
  - メインメモリのデータ転送の混雑を減少させるテクニック

## Using Larger Dictionaries

- 各SPEで複数オートマトンの並列実行
  - 各オートマトンの挙動
    - 個々に異なる状態を持つ
    - 入力テキストの異なる部分に対して動作
  - 全てのSPEにあるオートマトンが同一のSTTを参照アクセス
- 1つのオートマトンに対する1状態遷移の時間的コスト
  1. 次の状態がSTTのどのラインかを決定
     - **数ナノsec**
  2. そのラインをメインメモリから読み込み
     - **数百ナノsec**
     - トラフィックやデータ密度に依存
     - 複数オートマトンの並列実行により改善

## Using Larger Dictionaries

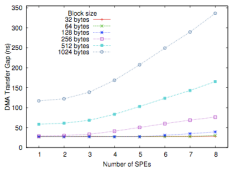
- *memory gap*
  - 2つの連続したデータ転送（メインメモリ）の完了時間の間隔
    - 性能を測る重要なパラメータ
    - 1入力文字の処理に*memory gap* かかる
      - （*memory gap* ＞ 遷移計算）の場合

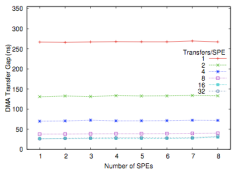



## Using Larger Dictionaries

- *memory gap* の最小化
  - ブロックサイズ（左下グラフ）
    - 最小：64 bytes
  - 同時データ転送数（右下グラフ）
    - 最小：16 transfers/SPE

・64 bytes でSTTアクセス
・1SPEで16オートマトン




## Using Larger Dictionaries

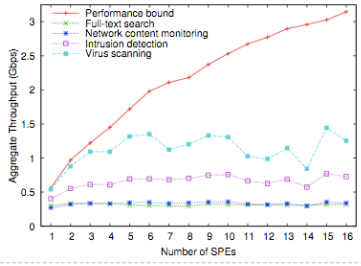
- 理論性能
  - 下記の表を参照
  - メモリコンジェクションにより実験では性能低下

| | 8 SPEs | 16 SPEs |
|---|---|---|
| Memory gap | 29.34 ns | 40.68 ns |
| Symbol Frequency per SPE | 34.07 MHz | 24.58 MHz |
| Throughput per SPE | 272.56 Mbps | 196.64 Mbps |
| Aggregate Throughput | 2.18 Gbps | 3.15 Gbps |

- メモリコンジェクション（memory congestion）
  - *memory gap* を増大させる要因
    - *Memory pressure*
    - *Layout issue*
    - *Hot spot*

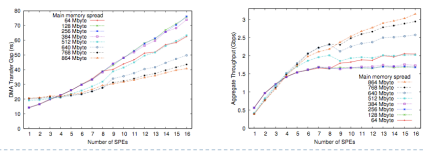
## Using Larger Dictionaries

- 性能評価（First Implementation）
  - 現実的なベンチマークによる性能評価



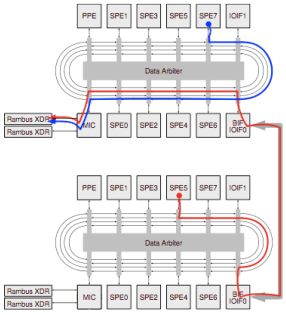

## Using Larger Dictionaries

- *Memory pressure*
  - 単位時間あたりの各メモリブロックへのアクセス数
  - 高くなる（性能低下）要因
    - 多くのSPE（オートマトン）が動作
    - STTが小さい領域に割当て
  - 改善方法
    - 大きなメモリ領域を利用してSTTを拡散させる



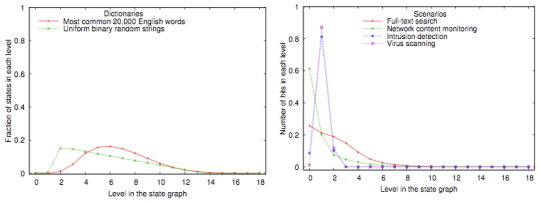
## Using Larger Dictionaries

- *Layout issue*
  - 2Cell/B.E.システムのデータレイアウト
  - トラフィック増大要因
    - データの偏り




## Using Larger Dictionaries

- *Hot spot*
  - 同時に高頻度のアクセスが起きるロケーション（状態）
  - 実世界のシナリオではよくある（下グラフ参照）




## Using Larger Dictionaries

- メモリコンジェクションの改善テクニック
  - STTを再構成することによりメモリコンジェクションをなくす
    - *state caching*
    - *state shuffling*
    - *state replication*
    - *alphabet shuffling*
- *state caching*
  - 高頻度でアクセスされる状態（STTライン）を常にLSに配置
    - プログラム開始時にメモリからデータ転送
    - LS（最大容量256KB）のため，配置領域は小さい（180KB ぐらい）

## Using Larger Dictionaries

- *state shuffling*
  - 状態（STTライン）をリナンバリング・リオーダリング
    - ただし，初期状態は「0」で固定
  - STTの収納スペースを増加
    - 使用可能なメインメモリ量に応じた収納スペース

  - *memory pressure* の減少
    - しかし，これだけでは均一性を保証にするには不十分
      - 混雑している状態（STTライン）中の隣り合うロケーションで発生

## Using Larger Dictionaries

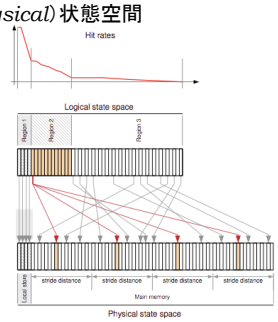
- *alphabet shuffling*
  - STTラインの入力シンボル（エントリ）をシャッフリング
  - シャッフリング関数

$$symbol' = (symbol + state) \bmod A$$

    - 簡単で効果的（加算とビット演算）
- *state replication*
  - 高頻度被アクセスの状態を複製
    - 1つの状態を4つに複製 → *memory pressure* は1/4
  - *Hot spot* の改善
    - *state shuffling* と*alphabet shuffling* では解決できない

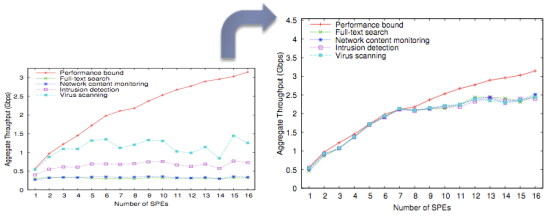
## Using Larger Dictionaries

- 論理（*logical*）と物理（*physical*）状態空間
  - Region1
    - *state caching*
      - 初期状態＋浅い階層の状態
  - Region2
    - *state replication*
    - *state shuffling*
    - *alphabet shuffling*
      - そこそこヒット
  - Region3
    - *state shuffling*
    - *alphabet shuffling*
      - その他



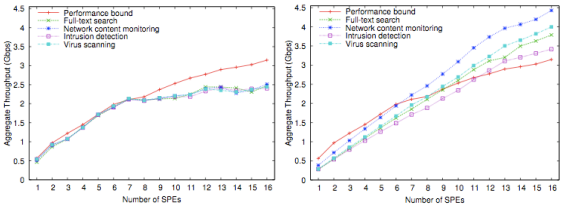
## Using Larger Dictionaries

- 性能評価
  - 初期実装との比較（*state caching* なし）
    - メモリコンジェクションの改善




## Using Larger Dictionaries

- 性能評価
  - *state caching* の有無
  - LSの有効活用による性能向上




## Conclusions And Future Works

- Cell/B.E.プロセッサによる文字列探索の性能評価
  - Aho-Corasick アルゴリズムの利用
  - Cell/B.E.向けに最適化を実行
  - 性能と辞書サイズとのトレードオフ
    - 豊富なバリエーション
- Local Store ベースの最適化アルゴリズム
  - LSに置けるほど辞書が小さい場合の高速な解法
  - 約5Gbps，約200パターンの辞書（1SPE）
  - 複数SPEの活用
    - スループット増加 or 辞書サイズ増加 or Both.

## Conclusions And Future Works

- メインメモリベースの最適化アルゴリズム
  - 大きな辞書をサポートし，一定の性能を発揮する解法
  - 1.5～2.2Gbps / Cell/B.E. プロセッサ
    - 辞書サイズに依存しない性能
- Cell/B.E.は性能と辞書サイズに対し柔軟といえる評価
  - Cell/B.E.では目的に応じた最適化が可能
    - FPGAベースの解法　・・・ 高性能・辞書サイズ小
    - 一般プロセッサの解法　・・・ 低性能・辞書サイズ大

## Throughput and Dictionary size