PERG-Rx: An FPGA-based Pattern-Matching Engine with Limited Regular Expression Support for Large Pattern Database

Johnny Ho

Supervisor: Guy Lemieux
Date: September 11, 2009

University of British Columbia
The Pattern-Matching Problem

5312c39392c33372c3131372c35312c35332c35352c35322c33372c3131372c34382c35312c3
5352c35362c33372c3131372c35332c35342c3130322c35322c33372c3131372c35352c3534
2c35362c39382c33372c3131372c34382c35312c35302c34382c33372c3131372c35312c3531
2c3130322c35332c33372c3131372c35322c35372c39392c35372c33372c3131372c39372c31
30322c35322c34392c33372c3131372c3130322c39382c35312c35312c33372c3131372c343
82c31530065006e00640020006b5312c39392c33372c3131372c35312c35332c35352c35322
2c33372c3131372c34382c35312c35352c35362c33372c3131372c35332c35342c3130322c353
32c33372c3131372c35352c35342c35362c39382c33372c3131372c34382c35312c35302c343
82c33372c3131372c35312c35312c3130322c35332c33372c3131372c35322c35372c39392c3
5372c33372c3131372c39372c3130322c35322c34392c33372c3131372c3130322c39382c35
The Pattern-Matching Problem

5312c39392c33372c3131372c35312c35332c35352c35322c33372c3131372c34382c35312c3
5352c35362c33372c3131372c35332c35342c3130322c35332c33372c3131372c35352c3534
2c35362c39382c33372c3131372c34382c35312c35302c34382c33372c3131372c35312c3531
2c3130322c35332c33372c3131372c35322c35372c39392c35372c33372c3131372c39372c31
30302c35322c34392c33372c3131372c3130322c39382c35312c35312c33372c3131372c343
82c31530065006e00640020006b5312c39392c33372c3131372c35312c35332c35352c35322
0c33372c3131372c34382c35312c35332c35362c33372c3131372c35332c35342c3130322c353
2c33372c3131372c35352c35342c35362c39382c33372c3131372c34382c35312c35302c343
82c33372c3131372c35312c35332c3130322c35332c33372c3131372c35342c33372c3131372c
5372c33372c3131372c39372c3130302c35322c34392c33372c3131372c3130302c39382c35

Pattern Database

234ab3200000383  Fixed string
The Pattern-Matching Problem

Pattern Database

234ab3200000383  Fixed string
21372{8}ef00{2}17ad  Multiple strings with fixed gaps
The Pattern-Matching Problem

Pattern Database

234ab3200000383  Fixed string
21372\{8\}ef00\{2\}17ad  Multiple strings with fixed gaps
234a*00000*df  Wildcards
The Pattern-Matching Problem

Pattern Database

- **Fixed string**
  - 234ab3200000383
- **Multiple strings with fixed gaps**
  - 21372{8}ef00{2}17ad
- **Wildcards**
  - 234a*00000*df
The Pattern-Matching Problem

Pattern Database

- Fixed string
- Multiple strings with fixed gaps
- Wildcards

# of Patterns

- 234ab3200000383
- 21372{8}ef00{2}17ad
- 234a*00000*df

Pattern Length
The Pattern-Matching Problem

• Applications:
  ▪ Network intrusion detection systems (NIDS)
    • Deep packet inspection
    • Well-studied
    • Several thousands in number of patterns (Snort database)
  ▪ Antivirus
    • Virus signature matching
    • PERG
    • Over 80,000 patterns in ClamAV database used
Motivation

- Antivirus is slow
  - Up to over 500% slowdown in I/O intensive process
  - Bottleneck: Pattern-Matching
  - Virus signature database
    - Large in number and range of lengths
    - Requires frequent update
Motivation

NIDS (Snort)

Antivirus (ClamAV)
## Related Works

- **Existing approaches:**
  - FSM (Aho-Corsaik), Bloom filter, Perfect/Cuckoo hash

<table>
<thead>
<tr>
<th></th>
<th>Regular Expression</th>
<th>Dynamic Update</th>
<th>Resource Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM</td>
<td>Excellent</td>
<td>Poor</td>
<td>Poor</td>
</tr>
<tr>
<td>Bloom filter</td>
<td>Poor</td>
<td>Excellent</td>
<td>Excellent</td>
</tr>
<tr>
<td>Perfect/Cuckoo hash</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
</tbody>
</table>
PERG: A FPGA-based pattern-matching engine for ClamAV
- Support limited regular expression
- 24x better density than the next-best competitor (excluding Bloom filter)
- 15x faster than software antivirus scanner

<table>
<thead>
<tr>
<th></th>
<th>Regular Expression</th>
<th>Dynamic Update</th>
<th>Resource Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM</td>
<td>Excellent</td>
<td>Poor</td>
<td>Poor</td>
</tr>
<tr>
<td>Bloom filter</td>
<td>Poor</td>
<td>Excellent</td>
<td>Excellent</td>
</tr>
<tr>
<td>Perfect/Cuckoo hash</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>PERG</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
</tr>
</tbody>
</table>
Contributions

• **A Novel Hardware Architecture**
  - Handle pattern matching in a multi-staged manner without resorting to high-bandwidth off-chip memory requirement

• **A Novel Filter Consolidation Algorithm**
  - Reduce the hardware resources required by packing filter units into high capacity, thus reducing the number of filter units needed.

• **Circular State Buffer**
  - Support multiple traces of multi-segmented patterns with zero false negative probability

• **Limited Regular Expression Support**
  - Support for wildcard operators to detect polymorphic virus
Contributions

- Published in three conferences:


• Boolean hash table
  - False: Input **MUST** not be a pattern in database
    • Zero false negative probability
  - True: Input **MAY** be a pattern in database
    • False positive probability due to hash collision
    • Do not know which pattern is the potential match
      • Exact matching is needed and complex

• Use multiple hash functions to reduce false positive probability
  - All hash locations returned must be true for a match in a Bloom filter

• One Bloom filter is needed for each input (hash) length
**Background: Bloom Filters**

- **Construction:** Hash patterns in database to the Boolean hash table

<table>
<thead>
<tr>
<th>Pattern 1: 1234567890abc</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Hash}_0$</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table

Pattern 1: 1234567890abc

<table>
<thead>
<tr>
<th>Hash₀</th>
<th>Hash₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table
Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table

Pattern 2: 234567890abcd

| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

$\text{Hash}_0 \quad \text{Hash}_1$
Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table

Pattern 2: 234567890abcd

\[
\begin{array}{cccccccc}
\text{Hash}_0 & & & & & & & \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]
Construction: Hash patterns in database to the Boolean hash table

Pattern 2: 234567890abcd

Hash

1
0
1
0
0
0
1
0
0
0
0
0
Pattern 2: 234567890abcd

• Construction: Hash patterns in database to the Boolean hash table

Hash collision
Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations

Input 1: abc34243432e2

<table>
<thead>
<tr>
<th>Hash₀</th>
<th>Hash₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations

Input 1: abc34243432e2

<table>
<thead>
<tr>
<th>Hash₀</th>
<th>Hash₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations
Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations

Input 1: abc34243432e2

Hash_0

Hash_1

1 0 1 0 0 1 0 0 0 0 0

Logic AND

1 AND 0 = 0 = FALSE!
• Usage: Hash input and logic-AND Boolean values at the hash locations

Input 2: 234567890abcd

Logic AND

1 AND 1 = 1 = TRUE!
Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations

Input 2: 234567890abcd

Hash0
1 0 1 0 0 1

Hash1
1 0 0 0 0 0

Logic AND

1 AND 1 = 1 = TRUE!
But..Pattern 1? Pattern 2?
Background: Bloomier Filters

- Structurally similar to Bloom filter
  - Resource efficient
  - Zero false negative probability
  - False positive probability
- Perfect-hash capability
  - Associate hash location with ONE single pattern
- Use multiple hash functions
  - Higher theoretical setup success rate than traditional perfect hash
Construction: Start off similarly to Bloom filter; hash each pattern in database one by one into a hash table
Instead of storing Boolean membership information, stores two attributes: a hash select and the pattern itself.
Background: Bloomier Filters

- Construction: Hash patterns in database to the hash table

Pattern 1: 1234567890abc

Hash

\[ Hash_0 \]

\[ \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array} \]

Hash\[1\]

\[ \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]
As with Bloom filter and any other hash-based scheme, collision is inevitable.

Hash collision between Pattern 1 and 2
• Identify hash location that is uniquely occupied by a pattern
  • If \( N \) hash functions are used, only one out of the \( N \) hash locations need to be unique

Background: Bloomier Filters

- Unique to Pattern 1 (using Hash\(_0\))
- Unique to Pattern 2 (using Hash\(_1\))
Store the pattern at its uniquely associated location
Store a hash select value to identify which of the $N$ hash function will point to this unique location
A location can be uniquely associated with a pattern even if it exists in multiple pattern neighborhoods.

Pattern 3: 123e342430aaea

Pattern 1 and 3 both map to this location.
Construction may fail if unique association between hash location and input pattern cannot be achieved

Background: Bloomier Filters

Pattern 3: 123e342430aaea

Pattern 1 and 3

Pattern 2 and 3

Pattern 1 and 2
• **Usage:** Similar to Bloom filter, hash the input with the \( N \) hash functions

\[
\begin{align*}
\text{Input: 1234567890abc} \\
\text{Hash}_0 & \quad \text{Hash}_1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
P1 & P2 \\
\end{align*}
\]
Logic XOR hash select values at the \( N \) hash locations to determine which hash location stores the unique pattern.

Background: Bloomier Filters

Input: 1234567890abc

\[\begin{array}{cccccccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
P1 & P2 & & & & & & & & & & \\
\end{array}\]

\[\text{Logic XOR} 0 \text{ XOR } 0 = 0 = \text{Use } Hash_0(\text{Input})\]
• Performs virus pattern matching on hardware
  - Rely on host to perform exact-matching
  - Communicate with host system through PCI bus

• Contains two parts
  - Pattern Compiler
    - Input: pattern database
    - Output: HDL and memory initialization file
    - Breaking up patterns into segments for optimization and regular-expression support purpose
  - Configurable Hardware Architecture
    - Virtex II-Pro FPGA + 4 MB SRAM
Hardware contains three units:
- **Inspection Unit**
  - Contains Bloomier Filter Units (BFU) to filtering input for patterns
- **Metadata Unit**
  - Stores *Metadata* that contains information on how to link segments of patterns back together
- **Fragment Reassembly Unit (FRU)**
  - Keep track of traces of multi-segmented patterns and link them back accordingly
ABCD{4}EFG
1. Split at displacement/wildcard

ABCD\{4\}EFG

ABCD\{4\}  EFG
1. Split at displacement/wildcard
2. Adjust offset

ABCD{4}EFG

ABCD{7} EFG
Patterns in ClamAV come in a wide range of lengths
- Each pattern length would require its own BFU
- Pattern length range is not evenly distributed

Filter consolidation reduces the number of pattern lengths by packing patterns at different length together
- Packing begins at the longest pattern length
- When the utilization threshold of a BFU hash table is met, assign this length as a BFU length
- Segments whose lengths do not match any BFU length are split into two overlapping segments of equal length
  - Length of the new overlapping segments is equal to the length of the nearest shorter BFU length
  - Splitting is done in *filter-mapping* stage
• Assume threshold is set to be 9 patterns
- If the current length is below threshold, decrement the length
If a length is skipped, patterns at the skipped length are divided to two overlapping segments.
Since the # of segments doubled, its cost contribution also doubles.
- The contribution however only needs to be doubled once
• Consolidation completes at user-defined minimum length
Pattern Compiler: Filter Mapping

1. Split at displacement/wildcard
2. Adjust offset

ABCD\{4\}EFG

ABCD\{7\} EFG
1. Split at displacement/wildcard
2. Adjust offset
3. Assume BFU length = 3 character, split the unfit segment into two overlapping segments
1. Split at displacement/wildcard
2. Adjust offset
3. Assume BFU length = 3 character, split the unfit segment into two overlapping segments
4. Assign Link #
Hardware Architecture
Hardware Architecture

[Diagram showing various components of a hardware architecture, including BFU L1, BFU L2, BFU L3, BFU L4, BFU L5, Memory Interface, Binary Tree, Byte Counter, Off-Chip SRAM, Metadata Unit, and FPGA.]
Hardware Architecture
Hardware Architecture
• Purpose
  ▪ Detect patterns spanned across multiple segments and separated by fixed byte lengths

• Advantages
  ▪ Support Multiple Traces
  ▪ Guarantee No False Negative
  ▪ Low Hardware Usage
  • Aliasing allows Design Trade-off between
  • Hardware and False Positive Probability

Circular Speculative Buffer
Circular Speculative Buffer

- Works like a time-wheel
- Operation is divided into *Verification* and *Speculation* phases
- Number of rows = Maximum displacement supported
  - Indexed by lower bits of Byte Count
- Three types of columns
  - Upper bits of Byte Count
  - Data (Rule ID + Link #)
  - Occupancy
- Reset upon a new file stream
Example
- Pattern A: \text{ABC}\{1\} \text{BCD}\{7\}\text{EFG}
- Input: \text{ABCD1234EFG}
At Byte Count = 2, C arrives and Segment ABC is detected

Verification:
- ABC is the first segment of Pattern A, so no previous state is needed to progress
Speculation:
- Record the next segment (BCD) expect to follow ABC at the expected Byte Count location \(\text{(Speculation Pointer)}\)
  - Byte Count + Displacement = 2 + 1 = 3
- Increment \textit{Occupy Column} pointed by Speculation Pointer by 1
• At Byte Count = 3, D arrives and Segment BCD is detected
• Verification:
  - Is BCD expected by an ongoing trace at the current Byte Count row? Yes, as set previously by Segment ABC
Speculation:
- Record the next segment (EFG) expect to follow BCD at the expected Byte Count location
- Speculation Pointer = 3 + 7 =10
- If Speculation Pointer > # of rows, the value wraps over
- Increment Occupy Column by 1
At Byte Count = 10, G arrives and Segment EFG is detected

Verification:
- Returns true as EFG is indeed expected by an ongoing trace as set previously by Segment ABC
Since EFG is the last segment of the pattern

- The full pattern has been reconstructed from the input
- Request for exact-matching is sent

Trace of Pattern A remains in CSB until overwritten or reset when new file stream arrives
Wildcards can be generated to two types
- At-least wildcard
- Within wildcard (Lossy)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Original</th>
<th>After Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>??</td>
<td>Single-Byte Wildcard</td>
<td>Displacement</td>
</tr>
<tr>
<td>*</td>
<td>(Any-Number-of-Byte) Wildcard</td>
<td>At-Least Wildcard</td>
</tr>
<tr>
<td>{n-}</td>
<td>At-Least (n-Byte) Wildcard</td>
<td>At-Least Wildcard</td>
</tr>
<tr>
<td>{-N}</td>
<td>Within (n-Byte) Wildcard</td>
<td>Within Wildcard</td>
</tr>
<tr>
<td>{n-N}</td>
<td>Range wildcard</td>
<td>Within Wildcard</td>
</tr>
</tbody>
</table>
Wildcard Support

- **Wildcard Table**
  - Indexed directly by Rule ID of the pattern
  - Contains a Byte Range attribute in each entry to keep track of within/at-least conditions
  - State (progress of trace) is maintained through Link # similar to CSB
    - Reset at start of a new file stream

- Different traces of the same pattern is mapped to the same table entry to reduce resource usage

- Lossy but resource efficient
  - Increase false positive probability
  - Zero false negative probability
• At-least Wildcard
  ▪ State only progress forwards (Link # only increases)
  ▪ If state remains the same until the expected segment arrives after its Byte Range is satisfied
• For an At-least Wildcard of \( n \) bytes \((\{n\})\), once \( n \) bytes has passed in the file stream, the range condition is always satisfied
Within Wildcard
- State only progress forwards (Link # only increases)
- If state remains the same until the expected segment arrives after its Byte Range is satisfied
  - For an At-least Wildcard of $n$ bytes ({n-}), once $n$ bytes has passed in the file stream, the range condition is always satisfied
- Exception
  - If incoming segment contains the same Link # as the previous segment (which indicate it is followed by a Within Wildcard), Byte Range is refreshed (updated)
Example

- Pattern A: ABC\{3\-\}DEF\{7\}GHI \{8\}JKL
- Input: ABC…DEF….GHI…JKL...DEF…GHI…JKL
• Example
  - Pattern A: ABC\{3\-\}DEF\{-7\}GHI \{-8\}JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL

• Wildcard Table Entry After the first ABC has arrived at Byte Count =0

<table>
<thead>
<tr>
<th>Link #</th>
<th>Byte Range</th>
<th>Wildcard Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>At-Least</td>
</tr>
</tbody>
</table>
• Example
  - Pattern A: ABC{3-7}DEF{-7}GHI {-8}JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL

• Wildcard Table Entry After the first DEF has arrived at Byte Count =4

<table>
<thead>
<tr>
<th>Link #</th>
<th>Byte Range</th>
<th>Wildcard Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>11</td>
<td>Within</td>
</tr>
</tbody>
</table>
• Example
  ▪ Pattern A: ABC\{3\}-DEF\{-7\}GHI \{-8\}JKL
  ▪ Input: ABC…DEF…GHI…JKL…DEF…GHI…JKL
  
• Wildcard Table Entry After the first GHI has arrived at Byte Count =10

<table>
<thead>
<tr>
<th>Link #</th>
<th>Byte Range</th>
<th>Wildcard Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>18</td>
<td>Within</td>
</tr>
</tbody>
</table>
Wildcard Support

• Example
  
  ▪ Pattern A: ABC{3-}DEF{-7}GHI {-8}JKL
  ▪ Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL

• Wildcard Table Entry After the first JKL has arrived at Byte Count =20
  ▪ Byte Range condition is NOT satisfied; no action taken

<table>
<thead>
<tr>
<th>Link #</th>
<th>Byte Range</th>
<th>Wildcard Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>18</td>
<td>Within</td>
</tr>
</tbody>
</table>
Wildcard Support

- Example
  - Pattern A: ABC\{3\}-DEF\{-7\}GHI \{-8\}JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL

- Wildcard Table Entry After the second DEF has arrived at Byte Count = 23
  - Incoming Link # < Link # in Table Entry; no action taken

<table>
<thead>
<tr>
<th>Link #</th>
<th>Byte Range</th>
<th>Wildcard Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>18</td>
<td>Within</td>
</tr>
</tbody>
</table>
• Example
  - Pattern A: ABC{3-}DEF{-7}GHI {-8}JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL

• Wildcard Table Entry After the second GHI has arrived at Byte Count =26
  - Incoming Link # < Link # in Table Entry, BUT
    - Wildcard Type = Within
    - Incoming Link # = Link # - 1
  - Updated Byte Range: 26+8 = 34

<table>
<thead>
<tr>
<th>Link #</th>
<th>Byte Range</th>
<th>Wildcard Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>34</td>
<td>Within</td>
</tr>
</tbody>
</table>
• Example
  - Pattern A: ABC{3-}DEF{-7}GHI {8}JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL

• Wildcard Table Entry After the second JKL has arrived at Byte Count =30
  - Incoming Link # = Link # in Table Entry
  - Metadata indicates JKL is the final segment of the pattern
    - Request of exact-matching is sent
    - Wildcard Table entry unchanged

<table>
<thead>
<tr>
<th>Link #</th>
<th>Byte Range</th>
<th>Wildcard Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>34</td>
<td>Within</td>
</tr>
</tbody>
</table>
Experimental Results

- Resource usage is determined by synthesizable Verilog model

- Performance is determined by cycle-accurate simulator written in C, normalized to the frequency reported by synthesis tool
  - SRAM is assumed to operate at $\frac{1}{4}$ of core frequency

- Based on ClamAV 0.93.1 main
  - # of patterns remained after special-case removal stage = 84,387

- Use Ubuntu-7.10-i386.iso sample input
  - Two tests: iso and extracted
## Performance

<table>
<thead>
<tr>
<th></th>
<th>Single File (Ubuntu7_10_x86.iso)</th>
<th>Extracted Files (274)</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Bytes Scanned</td>
<td>729,608,192</td>
<td>727,677,929</td>
</tr>
<tr>
<td># of False Positives</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>False Positive Probability for Each Byte Scanned</td>
<td>0.0000005%</td>
<td>0.0000005%</td>
</tr>
<tr>
<td># of Off-chip Memory Requests</td>
<td>82,499,591</td>
<td>80,500,329</td>
</tr>
<tr>
<td>Probability of Off-chip Memory Request for Each Byte Scanned</td>
<td>11.31%</td>
<td>11.07%</td>
</tr>
<tr>
<td>Off-chip Memory Throughput</td>
<td>19.4 MB/s</td>
<td>19.0 MB/s</td>
</tr>
<tr>
<td>Average Throughput</td>
<td>166 MB/s (0.922 B/cycle)</td>
<td>168 MB/s (0.933 B/cycle)</td>
</tr>
<tr>
<td>Modeled Frequency</td>
<td>180 MHz</td>
<td>180 MHz</td>
</tr>
</tbody>
</table>
## Comparison

<table>
<thead>
<tr>
<th>System</th>
<th>Patterns mapped</th>
<th>LC per Pattern</th>
<th>Memory per Pattern (kb/pattern)</th>
<th>TLP</th>
<th>TMP</th>
<th>Throughput (Gbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERG</td>
<td>84,387</td>
<td>0.5073</td>
<td>0.0358</td>
<td>2.56</td>
<td>36.28</td>
<td>1.3</td>
</tr>
<tr>
<td>Cuckoo Hashing</td>
<td>5,026</td>
<td>0.5933</td>
<td>0.2220</td>
<td>3.84</td>
<td>10.27</td>
<td>2.28</td>
</tr>
<tr>
<td>HashMem</td>
<td>1,474</td>
<td>1.7436</td>
<td>0.4410</td>
<td>1.55</td>
<td>6.12</td>
<td>2.70</td>
</tr>
<tr>
<td>PH-Mem</td>
<td>2,200</td>
<td>2.8509</td>
<td>0.1309</td>
<td>0.74</td>
<td>16.12</td>
<td>2.11</td>
</tr>
<tr>
<td>ROM+Coproc</td>
<td>2,031</td>
<td>4.1753</td>
<td>0.1359</td>
<td>0.50</td>
<td>15.31</td>
<td>2.08</td>
</tr>
</tbody>
</table>
Comparison

Average Throughput (MB/s)

- PERG: 166 MB/s
- ClamAV 0.93.1 on Intel Core Duo E4500: 10.5 MB/s
## Effectiveness of Filter Consolidation

<table>
<thead>
<tr>
<th></th>
<th>Without Filter Consolidation</th>
<th>With Filter Consolidation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total # of Segments Mapped to BFUs</td>
<td>89,423</td>
<td>141,147</td>
</tr>
<tr>
<td>Total # of BFUs</td>
<td>220</td>
<td>26</td>
</tr>
<tr>
<td>Total # of BRAMs used by BFUs</td>
<td>256</td>
<td>168</td>
</tr>
<tr>
<td># of Cache Entries</td>
<td>132</td>
<td>3823</td>
</tr>
</tbody>
</table>
## Scalability and Dynamic Updatability

<table>
<thead>
<tr>
<th>Number of BFUs</th>
<th>16</th>
<th>16</th>
<th>16</th>
<th>16</th>
<th>16</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of patterns</td>
<td>1440000</td>
<td>1440000</td>
<td>1440000</td>
<td>1440000</td>
<td>1440000</td>
<td>1440000</td>
</tr>
<tr>
<td>Utilization</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
</tr>
<tr>
<td>Change %</td>
<td>10%</td>
<td>25%</td>
<td>50%</td>
<td>75%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>Average number of rehashes</td>
<td>13.98</td>
<td>11.36</td>
<td>15</td>
<td>16.56</td>
<td>15.72</td>
<td></td>
</tr>
<tr>
<td>Number of setup failures (out of 50)</td>
<td>32</td>
<td>36</td>
<td>31</td>
<td>30</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>
### Scalability and Dynamic Updatability

<table>
<thead>
<tr>
<th>Number of BFUs</th>
<th>16</th>
<th>16</th>
<th>16</th>
<th>16</th>
<th>16</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of patterns</td>
<td>80000</td>
<td>96000</td>
<td>112000</td>
<td>128000</td>
<td>144000</td>
<td></td>
</tr>
<tr>
<td>Utilization</td>
<td>50%</td>
<td>60%</td>
<td>70%</td>
<td>80%</td>
<td>90%</td>
<td></td>
</tr>
<tr>
<td>Average number of patterns inserted</td>
<td>37466</td>
<td>27774</td>
<td>17892</td>
<td>3892</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Average number of insertions until failure</td>
<td>749.32</td>
<td>555.48</td>
<td>357.84</td>
<td>77.84</td>
<td>0.96</td>
<td></td>
</tr>
<tr>
<td>% of theoretical max reached</td>
<td>73.41625</td>
<td>77.35875</td>
<td>81.1825</td>
<td>82.4325</td>
<td>90.03</td>
<td></td>
</tr>
</tbody>
</table>
Conclusions

- PERG excels in pattern-per-resource density
- Lags behind in throughput
  - Still significantly faster than software
- Bloomier filters, checksum, and FRU together ensure false positives stay low despite lossy wildcard support
- A highly-utilized BFU is desirable
  - Filter consolidation is necessary
- To allow dynamic update, hash function must become more programmable
Future Works

- Support for interleaving file stream
- Integration with antivirus software
- Alternative database
- Update and Expansion Option
- Eliminate special-case removal stage
Contributions

- **A Novel Hardware Architecture**
  - Handle pattern matching in a multi-staged manner without resorting to high-bandwidth off-chip memory requirement

- **A Novel Filter Consolidation Algorithm**
  - Reduce the hardware resources required by packing filter units into high capacity, thus reducing the number of filter units needed.

- **Circular State Buffer**
  - Support multiple traces of multi-segmented patterns with zero false negative probability

- **Limited Regular Expression Support**
  - Support for wildcard operators to detect polymorphic virus