ACCELERATING REGULAR EXPRESSIONS USING MANY-CORE SYSTEMS

TIM LEATHART

October 2014

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

ABSTRACT

Regular expression matching is an essential component of critical computer systems. As the amount of data online is constantly growing, and the number of applications for data analysis using regular expressions increases, more powerful and efficient regular expression matching systems are required.

This study explores the design space of regular expression matching using two different many-core systems: the Intel Xeon Phi Coprocessor and the Azure System-on-Chip by Wave Semiconductor. A many-core, software based implementation of Thompson's NFA algorithm [29] and B-FSM [34] has been developed for the Xeon Phi, with generally poor throughput compared to using a standard Xeon CPU. However, a hardware-based implementation of B-FSM has also been developed and evaluated for the Azure System-on-Chip, with an estimated throughput per device 11.7x greater than current state-of-the-art ASIC-based implementations. The matching systems have been evaluated using rules from the Snort rule set, a popular network intrusion detection suite.

ACKNOWLEDGEMENTS

I would like to first acknowledge my supervisor, Dr. Anthony Blake. Without his constant support and encouragement throughout, this project would not have been achievable, and I am truly grateful for his guidance during the past year.

Thank you to Henry Gouk, Chris Goodwin, Aaron Storey and all the other members of the Advanced Computation Group for your support, company and entertainment during this project.

Last, but certainly not least, thank you to my friends and family for your support and encouragement throughout my university career.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ASIC Application Specific Integrated Circuit

B-FSM BaRT Based Finite State Machine

BaRT Balanced Routing Table

CGRA Coarse-Grained Reconfigurable Architecture

CPU Central Processing Unit

DFA Deterministic Finite Automata

D^{2}FA Delayed input Deterministic Finite Automata

FPGA Field Programmable Gate Array

FSM Finite State Machine

GPU Graphics Processing Unit

MIC Many Integrated Core

NFA Non-deterministic Finite Automata

RAM Random Access Memory

PE Processing Element

SoC System-on-Chip

XFA Extended Finite Automata

1

INTRODUCTION

The need for greater performance from regular expression matchers is ever present as the amount of data online and line speeds continue to grow. Network security systems rely on regular expressions to perform filtering of data streams, and databases can utilise pattern matching to perform more flexible and powerful search queries. The goal of this project is to explore methods of achieving high-throughput regular expression matching using highly parallel devices such as the Intel Xeon Phi Coprocessor, a many-core device based on Intel's Many Integrated Core (MIC) architecture, and with the Azure System-on-Chip (SoC), a Coarse-Grained Reconfigurable Architecture (CGRA), designed by Wave Semiconductor.

Regular expression matching systems are naturally represented as a Finite State Machine (FSM). In general, they may naively be represented as a Deterministic Finite Automata (DFA) or Non-Deterministic Finite Automata (NFA). Both of these forms have advantages and disadvantages – NFA has low memory requirements but high computational overhead, whereas DFA have constant processing complexity but have high storage costs (Table 1) [17].

Since the development of the original string searching algorithms such as Aho-Corasick [5] and Boyer-Moore [8], most approaches to improve the efficiency of pattern matching have involved attempting to find a balance between these representations that provide both low memory requirements and low

	Storage Complexity	Processing Complexity
NFA	$O(n)$	$O(n^2)$
DFA	$O(\Sigma^2)$	$O(1)$

Table 1: Storage & Processing complexities for an n-state finite automata with alphabet Σ.

computational complexity. This study considers a deterministic hardware based regular expression matching system called BaRT-Based FSM (B-FSM) by Jan van Lunteren [34]. This system is implemented in IBM's "Power Edge of Network Processor", a device designed to enable wire speed computing for networks [9].

In this study, the B-FSM engine has been implemented for the Intel Xeon Phi Coprocessor, and a mapping for the Azure SoC has been designed from which the performance can be accurately estimated. These implementations have then been compared to current state-of-the-art high throughput regular expression matching systems, and the improvements and shortfalls of these devices are discussed.

2

BACKGROUND

This chapter gives an overview of the devices that are used in this study, an outline of the body of past work on the problem, and a more in-depth look into the inner workings of the B-FSM scheme.

2.1 MANY-CORE VS MULTI-CORE

There is no strict definition of "many-core" to differentiate it from "multi-core" – technically speaking, any many-core processor is a multi-core processor. The term has been loosely defined as a "chip with several tens, but more likely hundreds, or even thousands of processor cores" [30].

Both of the devices used in this research fit under the umbrella of many-core systems, although the Azure SoC is more accurately described as a Massively Parallel Processor Array. There are certainly many more different architectures and devices that could be used to broaden the scope of this study.

2.2 DEVICES AND LIBRARIES USED

2.2.1 *Intel Xeon Phi Coprocessor*

The Intel Xeon Phi Coprocessor is a multiprocessor computer architecture developed by Intel, commonly utilised in supercomputers (Figure 1). It is based on Intel's Many Integrated Core (MIC) architecture. Designed to tackle highly parallel applications, they are equipped with up to 61 cores, 244 threads and 512-bit wide vector units. Many groups have achieved significant speed-ups for common applications by taking advantage of its massive potential computational power [11, 13, 16]. However, published results for regular expression performance on the Xeon Phi are notably absent from the literature.

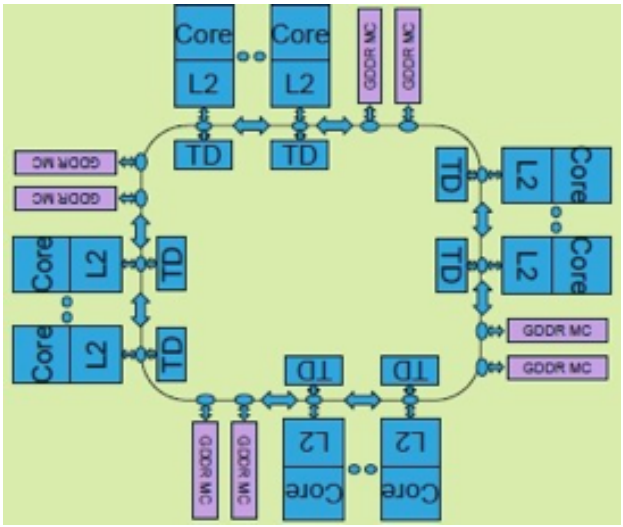

Figure 1: Xeon Phi Architecture[1]

2.2.2 *Azure System-on-Chip*

The Azure System-on-Chip (SoC) is a Coarse-Grained Reconfigurable Architecture (CGRA) currently in development by Wave Semiconductor (Figure 2). CGRAs consist of a large 2D

1 Image source: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

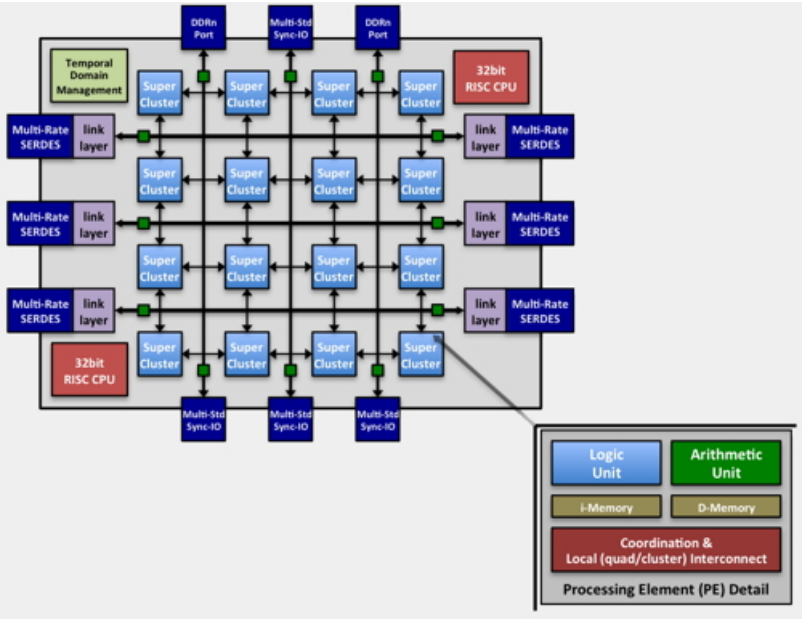

Figure 2: Azure Architecture[2]

array of processing units interconnected with a mesh style network. Each processing unit can execute common word-level instructions such as arithmetic and bitwise operations. In contrast to Field Programmable Gate Arrays (FPGA), CGRAs have short reconfiguration times, low power consumption and low delay characteristics. While gate-level reconfigurability is sacrificed, a large increase in hardware efficiency is achieved [36].

The Azure SoC consists of 16384 8-bit microprocessors running at an effective clock rate of up to 10GHz, arranged into 16 super-clusters. Each super-cluster is composed of 64 clusters. Each cluster contains four quads, each containing four 8-bit microprocessors. Each quad also has access to an adder, multiplier and a bit manipulation unit which are shared between the upper and lower quads. Communication between microprocessors in a quad occurs at very low latencies. Logical instructions can be computed at very low latencies as well, but use of the arithmetic instructions incurs additional latency. Each quad has 2KB of data RAM.

2 Image source: http://www.wavesemi.com/products.html

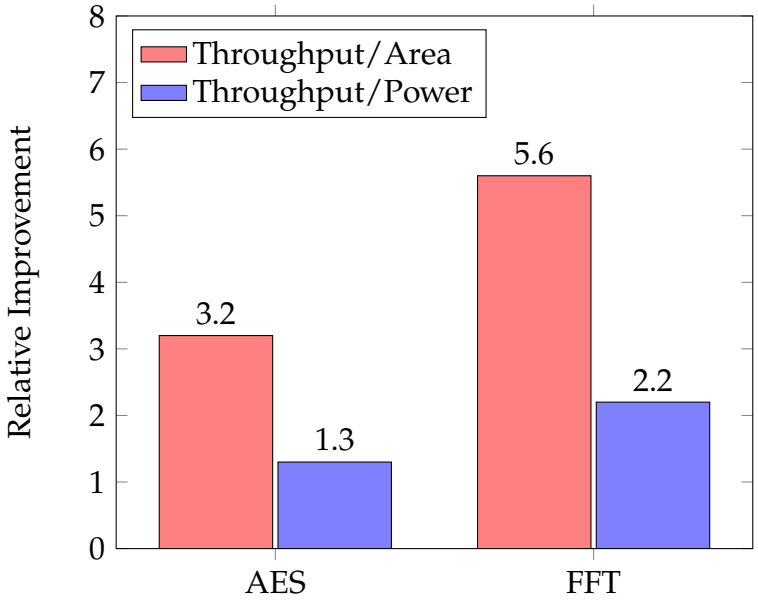

Figure 3: Azure SoC performance for AES and FFT, relative to FPGA implementations.

The Azure SoC has seen vast improvements in throughput per area and throughput per power over FPGA for a number of algorithms. As the chip is still in development, the body of performance benchmarks for common algorithms is not comprehensive. Figure 3 shows examples of performance gains achieved by using the Azure SoC [7, 15].

2.2.3 *OpenMP*

OpenMP is an API that assists programmers in writing multi-threaded applications in C, C++ and Fortran [26]. It consists of a set of compiler directives, environment variables and library routines that affect run-time behaviour. In this study, OpenMP has been used to write programs for the Xeon Phi that utilise all of its cores.

2.3 PAST WORK

Most popular network security software tools such as Snort [24] use regular expressions to apply content-filtering rules. To keep up with trends of increasing numbers and more sophisticated methods of network attacks, enhancing the performance of regular expressions has been a focus of research in recent years, in both the hardware and software spectrums.

2.3.1 *General Algorithm Improvements*

There is a reasonable body of research focusing on improving regular expression performance in software at the algorithm level.

2.3.1.1 *Speculative Matching*

Luchaup *et al.* propose a method of parallelising regular expression matching on a single input text called speculative parallel matching [22]. The general idea behind speculative matching is to split the input text into a number of equally sized chunks, opportunistically scan each chunk with traditional DFA matching in parallel, and if the guesses made are wrong, fix the mistakes in post processing. The algorithm will guess an initial state for each chunk – except for the first chunk which obviously begins in the initial state of the whole DFA – and then ensures that this speculation does not produce inaccurate match information. The method of false speculation correction is best illustrated through an example – see Sec III-A of Luchaup *et al.*'s publication [22]. By using this technique, a speedup of at least 40% is achieved.

2.3.1.2 *D²FA*

Kumar *et al.* propose a method of reducing the space requirements of a DFA by compressing transition tables, but also suffer slightly longer matching times called Delayed Input DFA (D^2FA) [20, 21]. Their approach finds different states that have comparable transition tables, and replaces them with smaller transition tables consisting of the transitions that are not shared between the states. During execution of the D^2FA, default transitions are taken between states until a transition table entry is encountered that matches up with the input character currently being considered. The slight throughput reduction is due to the fact that no input character is read when a default rule is taken.

2.3.1.3 *XFA*

Smith *et al.* observe that certain regular expression sets can result in state space explosion when compiled to DFA, *e.g.*, a set of expressions that are of the form $.*S.*S'$, where S and S' are distinct strings [27, 28]. This state space explosion obviously greatly increases the memory requirements. They propose a solution called Extended Finite Automata (XFA), which includes a small amount of additional memory and instructions to manipulate it to FSAs. Smith *et al.* report that XFAs use 10 times less memory than a DFA-based solution, yet achieve 20 times higher matching speeds. Becchi *et al.* propose improvements to the D^2FA method (Section 2.3.1.2) by bounding the number of default transitions followed [6], which is analogous to this approach [27].

2.3.2 *GPU*

Vasiliadis *et al.* propose a design for regular expression matching for packet inspection on a GPU [35]. This is performed by storing a DFA in texture memory as it is more suited to

random access patterns required by DFA execution than the global memory on the GPU. To overcome the overhead associated with transferring data from the main memory to the GPU, packets are copied to the GPU in batches. This implementation achieves a throughput 9-10 times greater than a CPU implementation [35].

2.3.3 *Reconfigurable Hardware*

Most recent efforts to improve regular expression performance are implemented in reconfigurable hardware, *e.g.*, FPGA (Field Programmable Gate Array) and ASIC (Application-Specific Integrated Circuit). The throughput required for typical applications is more easily met through the use of reconfigurable hardware.

Sidhu and Prasanna developed the first FPGA based regular expression matching implementation in 2001 [25]. They use an NFA to represent the state machine. They also construct the NFA on the FPGA instead of performing compilation in software and moving it to hardware. They report that their implementation outperforms GNU grep on a typical CPU of the era, the Pentium III.

Yu *et al.* developed an FPGA based regular expression matching system using a DFA structure to represent the state machine [37] in 2006. They describe a method of grouping regular expressions strategically into several engines, which can give a vast throughput increase with minimal increase in memory requirements. Yu *et al.* also describe a method of re-writing regular expressions to further reduce memory usage. They report 50-700x improvement over state-of-the-art NFA based implementations at the time of publishing.

Van Lunteren proposes a hardware architecture for matching regular expressions called B-FSM. This is explained in depth in Section 2.4.

While regular expression matchers have been implemented successfully in software, FPGAs and ASICs, there has not been a published CGRA-based regular expression matcher which competes with the existing top-performing implementations.

2.3.4 *Purpose-Built Hardware*

While most hardware based efforts to improve regular expression matching performance use FPGAs and ASICs, Dlugosch *et al.* have recently produced a semiconductor architecture for parallel automata processing [12]. The proposed architecture allows simultaneous parallel exploration of all possible valid paths in an NFA, which gives the reduced computational complexity of a DFA without the state space explosion suffered by such an approach. Dlugosch *et al.* claim that this architecture achieves higher throughput than FPGA implementations, but no comparison could be found to substantiate this claim.

2.4 B-FSM ENGINE

The BaRT-based FSM (B-FSM) engine is a fast, deterministic hardware based pattern matching engine (Figures 4, 5), first described by van Lunteren in 2006 [34]. The main optimisation that B-FSM provides over other engines is the reduced number of memory accesses and overall compression of the state transition table. This is achieved by slightly modifying a method originally used by van Lunteren to compress routing tables called Balanced Routing Table (BaRT) described in 2001 [31].

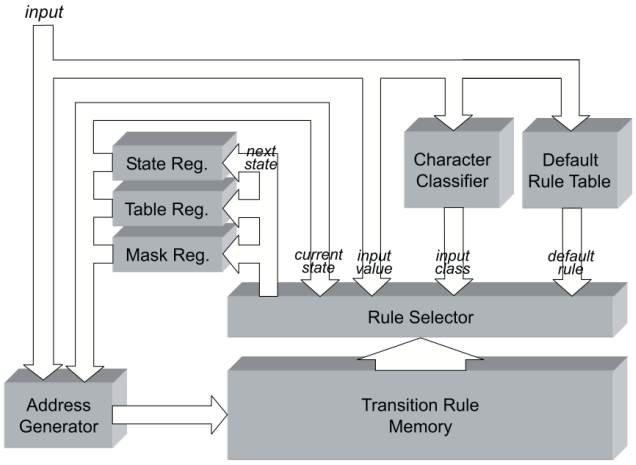

Figure 4: Block diagram of the B-FSM engine

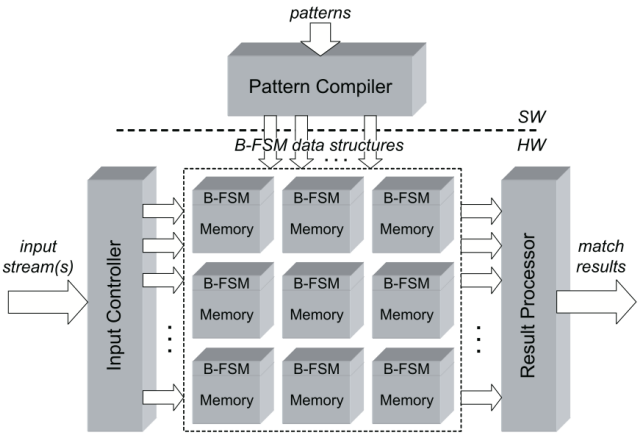

Figure 5: Block diagram of the full pattern matching scheme

B-FSM stores the state machine in hash tables of transition rules, which correspond to the transitions between states in an FSM. The index in the hash table is extracted from specific bits within the input and current state values:

$$index = (state' \textbf{ and not } mask) \textbf{ or } (input' \textbf{ and } mask) \quad (1)$$

where **and**, **not**, and **or** are bitwise operations, $state'$ and $input'$ are fixed subsets of the current state and input character (such as the most or least significant part), and the mask field selects for each bit of the index if it is obtained from the current state value or from the input symbol (where the mask bit is zero or one respectively) [34]. Figure 6 illustrates how the transition rules could be stored for the regular expression `a[bB][^c][0-8]*9`. Underlined bits in the current state and input fields denote the bit positions from which the index is extracted, according to Equation 1. Note that in this example, there are two transition rules per table row – the number of rules per row P is configurable and is usually set to be the number of rules that can be read with a single memory access.

Since its initial publication, van Lunteren *et al.* have published another two papers describing other improvements made to the engine [32, 33]. These improvements introduce common features of modern regular expression matchers such as character class support, as well as changes to the engine to improve performance including the introduction of default rules and changes in the format of transition rules.

Default rules are transition rules that have an "any state" condition for the current state, *i.e.*, the rule is followed no matter what the current state, if there are no other matching rules. Instead of filling the transition rule tables with default rules, they are placed in a separate look-up table where the index in the table is the input character (Figure 7). Default rules with a wildcard for the input character (of which there can only be one) are

rule	current state	input	→	next state
R_2	0000001**0**b	0x100010b	→	S_2
R_3	0000000**1**b	xxxxxxx**x**b	→	S_4
R_4	0000000**1**b	0110000**1**b	→	S_3
R_5	0000000**1**b	0110001**1**b	→	S_0
R_6	000**1**0000b	001**1**xxxxb	→	S_4
R_7	000**1**0000b	001**1**1001b	→	S_5
R_8	000**1**0000b	01x**0**0010b	→	S_2
R_8	0000010**0**b	0011xxxxb	→	S_4
R_{10}	0000010**0**b	00111001b	→	S_5

010000b	rule R_7	rule R_6
...		
000100b	rule R_{10}	rule R_9
000011b	rule R_5	rule R_3
000010b	rule R_2	
000001b	rule R_4	rule R_3
000000b	rule R_8	

Figure 6: Example transition rules and a B-FSM compressed transition rule table containing them.

stored in all entries in the default rule table that are not taken up by a default rule with a specified character. This allows more room in the transition rule memory for normal transition rules, resulting in much higher compression rates.

Another improvement made to the B-FSM scheme is the reduction of the transition rule size by only storing the three most significant bits of the current state in the transition rule, denoted as the current state tag [32]. Hash collisions in the rule selection of states having the same most significant bits do not occur, as the B-FSM compiler ensures that the state vectors are encoded such that states with equal least significant state bits

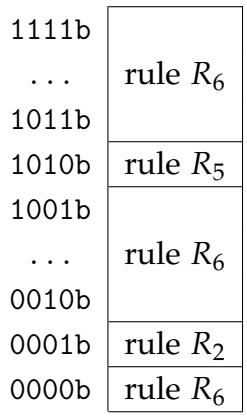

Figure 7: Default rule table

do not appear in the same line of a transition rule table. This reduces the total transition rule size by approximately 15%.

3

IMPLEMENTATION

This chapter describes the implementation of the B-FSM scheme in three parts: B-FSM compiler, Xeon Phi implementation and Azure SoC implementation. The first section explains the process of building compressed transition rule tables from regular expressions. The second section describes the approach taken to implement the B-FSM engine for the Xeon Phi, as well as a baseline implementation for comparison. The third section details the design and approach taken to map the B-FSM engine to the Azure SoC.

3.1 B-FSM COMPILER

The B-FSM compiler is a piece of software which creates compressed transition rule tables from regular expressions. Van Lunteren's original implementation of this algorithm [34] is unfortunately closed-source, so it was implemented from scratch for the purposes of this study.

3.1.1 *Algorithm Details*

Van Lunteren proposes an algorithm to build compressed transition rule tables from regular expressions [33, 34].

The first step is to split up the regular expressions into distinct subsets. As typical implementations of the B-FSM scheme have more than one FSM running on the same input data, the regular expressions must be decomposed and assigned to spe-

cific FSMs. To lower the total number of states, simulated annealing [19] is used to put regular expressions with matching prefixes into the same FSM [34].

Next, a DFA needs to be produced for each subset of regular expressions. This is achieved by first compiling each set of regular expressions into a single NFA, using some existing NFA building algorithm. In this case, Thompson's NFA building algorithm was used [29]. Then, a minimal DFA needs to be produced from the NFA to further reduce the total number of states, and therefore the total memory requriements. Brzozowski describes a method of finding a minimal DFA [10]:

- Dualize the NFA
- Powerset algorithm [23] to convert to DFA
- Dualize the DFA
- Powerset algorithm again

Now that a minimal DFA has been produced, the transition rules need to be extracted and assigned priorities. These priorities are determined by the compiler according to the nature of the rule [34] (Table 2). A default rule has a wildcard for its current state and is a rule which leads to the first state in the DFA. This rule is taken only if there are no other matching rules, *i.e.*, if a pattern has been partially matched but the next character of

Type	Priority
Default Rule	0
First Pattern Character	1
Normal Rule	2

Table 2: Rule types and their priorities

the input does not continue to match. A first pattern character rule is similar to a default rule, but it leads to a state that is an immediate child of the first state. This kind of rule is taken if a pattern has been partially matched; the next character does not match the current pattern but is the first character in another pattern. A normal rule is literally that – a normal rule.

Finally, an attempt is made to compress the rule table to memory constraints. This is achieved by choosing an appropriate mask for transitions from each state to produce a densely packed rule table. To do this, a brute force approach is used to choose a mask for each state individually, starting with states with the most transitions (Algorithm 1). As more states are considered, it is possible that no 8-bit mask will allow a state's rules to fit into the table, at which point another table is tried. If the state's rules cannot fit into any existing tables with any 8-bit mask, then a new table is made.

Once the transition rule tables have been produced, the default rule table is built. For transition rules that start in the initial state and transition on symbol s, a default rule is created at index s that leads to the next state. The rest of the indices of the default rule table simply lead to the initial state.

This compiler, along with the original scheme, has undergone several revisions since its initial publication in 2006 [32, 33, 34]. For this study, the 2010 version of the compiler was implemented except for the character class support.

3.1.2 *Constraints*

In this implementation, all regular expressions used in the rule set must be both prefix- and suffix-closed. A suffix-closed regular expression is an expression that has the property that if it matches a string, it matches the same string followed by any

```
1:  S ← set of states
2:  T ← set of transition rule tables
3:  for s ∈ S, t ∈ T do
4:      for i ← 0 to 255 do
5:          placed ← true
6:          for tr ∈ s do
7:              index ← calculate_index(i, tr, s)
8:              if t[index] is non-empty then
9:                  placed ← false
10:                 break
11:             end if
12:         end for
13:         if placed = true then
14:             t[index] ← s
15:             break
16:         end if
17:     end for
18: end for
```

Algorithm 1: Method for packing transition rules into a compressed transition rule table. `calculate_index` (line 7) refers to Equation 1.

suffix. More formally, their language L over alphabet Σ has the property that $a \in L \iff \forall b \in (\Sigma)^* : ab \in L$. Similarly, a prefix-closed regular expression is an expression that has the property that if it matches a string, it matches the same string preceded by any prefix, *i.e.*, their language L over alphabet Σ has the property that $a \in L \iff \forall b \in (\Sigma)^* : ba \in L$.

This constraint is not particularly restrictive for the most common applications of regular expression matching – all Snort rules are suffix closed, and a large majority of them are prefix-closed as well [22]. In fact, many of the high throughput reg-

ular expression matching engines looked at in this study have similar constraints. However, most existing high throughput regular expression matching engines are targeted towards network intrusion detection and deep packet inspection – in less specific use cases such as database lookups or parsing, these constraints would have more of an effect.

In the original implementation, there is an optional parameter to reduce the time taken to compile the compressed rule tables, at the cost of a compression rate reduction. This feature has not been implemented as it was deemed unnecessary for the purposes of this investigation – the time taken for the maximally-compressed compilation is not excessive at approximately one second per input regular expression [33].

3.2 INTEL XEON PHI IMPLEMENTATION

3.2.1 *Baseline*

Although the main purpose of this part of the investigation is to compare the performance of the B-FSM scheme on the Xeon Phi, it is useful to compare it to a simple software implementation of regular expression matching on the Xeon Phi as well. To this end, an existing implementation of a regular expression matching system [3], based on Thompson's NFA algorithm [29], was extended to utilise the many cores available in the Xeon Phi using OpenMP.

Due to the large amount of memory available to the Xeon Phi, it is usually possible to load an entire input file into memory before testing the throughput of the engine. This is done in this implementation for evaluation purposes.

The first step is to parse the regular expressions. For testing purposes, regular expressions from the Snort community ruleset were used [2, 24]. These regular expressions are then combined into one. This is achieved simply by turning the list of expressions into one large disjunct of expressions. In accordance with the constraints of the B-FSM compiler described in Section 3.1.2, the resulting regular expression was made to be prefix- and suffix-closed by appending a $.^*$ term before and after it.

The input file is now loaded into memory. In testing, each line of an input file is considered to be a separate input stream for the regular expression matching engine. The input file is memory mapped and scanned, taking note of where each new line begins. Pointers to the beginning of new lines are stored in a vector. Lines from the input file are now assigned to threads. The lines are assigned such that every thread has an approx-

imately equal number of bytes, taken from adjacent lines to maximise cache usage.

Now a single regular expression has been obtained and a lot of input data is in memory. The pre-existing implementation from [3], with minor modifications, is now run on each thread. This step includes building the FSM. A data structure was added to encapsulate all of the relevant information about the NFA and DFA execution process, as the original implementation modified FSM memory while executing it. The number of regular expression matches is stored in an array, which is accessed atomically.

3.2.2 *B-FSM Design Strategy*

As previously mentioned, the B-FSM scheme has undergone several revisions since its initial publication. These changes improve performance when the engine is implemented in hardware, but were found to actually cause a slight decrease in performance in software – this is further discussed in Section 4.2.1. Because of this, the original B-FSM scheme proposed in 2006 is the version that was implemented for the Xeon Phi.

OpenMP was used to utilise the many cores and threads available in the Xeon Phi. Listing 1 shows a method used in this implementation to perform pattern matching across all cores in the Xeon Phi. `num_threads` denotes the number of threads that the programmer desires to be executed in parallel. `start_indices` and `lengths` are arrays containing the index of the first input string, and the number of input strings, that should be processed by each thread respectively. `counts` is an array holding the number of regular expression matches for each thread. By adding the pragma before the for loop (line 2), each iteration of the for loop is executed on a new thread. The pragma above the counts array incrementation (line 10) ensures

that threads are not interrupted while executing the counts update.

Listing 1: OpenMP usage

```
...
#pragma omp parallel for private(i,j)
for (i=0; i<num_threads; i++)
{
    for (j=start_indices[i]; j<lengths[i]; j++)
    {
        // match patterns here
        if (match)
        {
            #pragma omp atomic
            counts[i]++;
        }
    }
}
```

type	current state	input	next state	mask	table addr
8	16	8	16	8	8

Figure 8: Fields and their sizes in bits in a 2006 transition rule

The total width of the transition rule is 64-bits (Figure 8). The Xeon Phi reads 64-bytes per memory access [14], so the transition rule table width P can be up to eight rules wide. However, experimental results show that a lower P tends towards greater performance in software (Section **??**).

The B-FSM engine is overall a simple algorithm. As can be seen in Figure 9, most of the logic behind the engine is contained in the rule selection and address generation units. Algorithm 2 shows a method of implementing these units in soft-

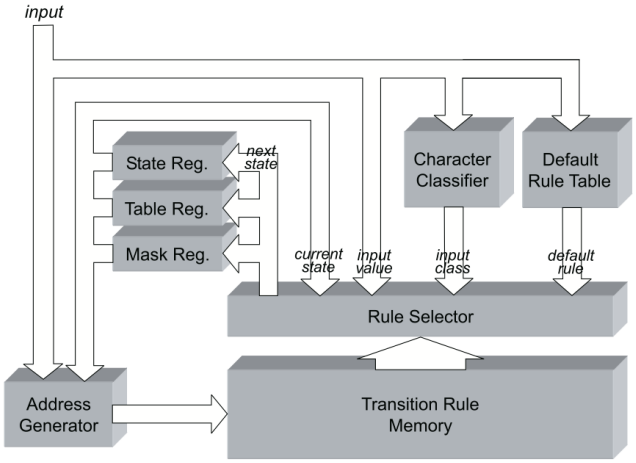

Figure 9: Block diagram of the B-FSM engine

ware. The transition rule memory is stored as a large array of `tr_table` structs (Listing 2). The size of this array is a multiple of 256, where every 256 elements represents one full transition rule table. The `table address` field is stored as an offset and is prepended to the index calculated on line 8 of Algorithm 2.

Listing 2: Structs used

```
1  struct tr_table
   {
       tr_vector rules[P];
   } __attribute__ ((packed));
   struct tr_vector
6  {
       uint16_t curr_state;
       char type;
       char symbol;
       uint16_t next_state;
11     char table_addr;
       char mask;
   } __attribute__ ((packed));
```

```
1: symbol ← input symbol
2: state ← current state
3: mask ← mask for state
4: addr ← table address for state
5: T ← transition rule tables
6: DRT ← default rule table
7:
8: index ← calculate_index(symbol, state, mask) | (addr ≪ 8)
9: for t ∈ T[index] do
10:     if t matches symbol and state info then
11:         return t
12:     end if
13: end for
14: return DRT[symbol]
```

Algorithm 2: Implementing the B-FSM rule selector in software. `calculate_index` (line 8) refers to Equation 1.

3.2.3 *Xeon Phi Optimisation*

There are many common techniques used to optimise software-based implementations. These techniques include exploiting temporal and spatial locality, reducing the amount of common computation, and utilising architecture-specific vector units on the CPU with vector intrinsics. Unfortunately, due to the nature of regular expression matching, most of these optimisation techniques could not be effectively utilised.

Spatial locality is the assumption that when accessing an element in memory, one is likely to access elements around it as well, *i.e.*, if an address in memory has just been accessed, it is likely that the next address that is accessed is a neighbouring address. The likelihood that this address will already be in a cache line is high, so by arranging our data appropriately we

can reduce the number of cache misses, which in turn reduces the number of main memory accesses and increases the performance. There is little that can be done in regular expression matching to improve upon a naive implementation in this area. The text to be matched is already obviously stored sequentially in memory. In the B-FSM scheme, all of the potentially matching transition rules are adjacent in memory, and the number of hash collisions will result in a total size of potentially matching transition rules less than or equal to the width of a cache line, so this is one area in which the B-FSM scheme should see an improvement. Furthermore, the input to be matched is memory mapped in contiguous memory, so the start of the next line is likely to be already in the cache from the final read of the previous line.

Temporal locality is the idea that if the same data gets reused within a short period of time, *e.g.*, in two or more adjacent loops where a section of memory is read, processed and stored, it is better to combine as many loops as possible into one. This is because the latency of main memory is much greater than that of the CPU cache. In the problem of regular expression matching, however, the data being processed is only accessed once – the time that the expressions are matched against it – so this cannot be utilised to improve the performance.

If the same computation is done multiple times, it can be faster to store the results after the first computation in memory and read that instead of re-computing it. This does not lend itself well to regular expression matching, as there is virtually no common computation.

Intrinsics allow programmers to access low-level instructions without having to use inline assembly. This gives a huge amount of power to the programmer to utilise CPU features such as vector units, which allow the same computation to be performed

on multiple pieces of data at the same time (Figure 10). Effectively making use of these vector units can obviously bring forth a huge improvement in performance, although they do come at the cost of reduced portability due to the fact that these instructions are architecture-specific. The reduced portability is not an issue in this case, as the implementation is specific to the Xeon Phi Coprocessor architecture.

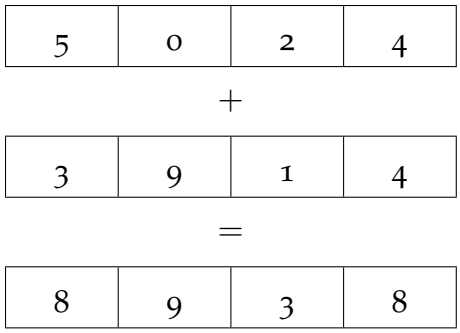

Figure 10: Example of a vector operation

Unfortunately, in regular expression matching, each computation relies completely on the result of the previous computation. The combination of the current state and input symbol defines what to do with the next input symbol, so several adjacent input symbols cannot be considered at once using vector intrinsics.

3.3 AZURE SYSTEM-ON-CHIP IMPLEMENTATION

3.3.1 *Mapping the B-FSM scheme to Azure*

The design of the implementation of the B-FSM engine for Azure is very different to that of the Xeon Phi. This is mostly due to the fact that the amount of memory on the Azure SoC is much less than what is available to the Xeon Phi.

In the implementation, there is one B-FSM engine per cluster. This allows for a total of 1024 state machines on the chip. This is a much greater number than previous implementations of the B-FSM scheme, which typically used 4-32 state machines in total [33]. However, the data RAM available is much lower per state machine at 8KB compared to FPGA implementations which have 512KB or more.

One quad per cluster handles the test parts of two rules from a transition table row. As can be seen in Table 11, the test part of a transition rule in 2012 format is two bytes. This allows the test parts of two rules to be loaded into one word of the data RAM of a single quad. This quad will be doing most of the work of the engine as it is here that all of the testing for input conditions and handling the rule type, *e.g.*, case sensitive rules.

Two quads per cluster handle the result parts of two rules from a transition table row. The table will be partitioned over the RAMs such that one row element will be available to each of the two quads using the same index. Each element in the table will be 48-bits (discussed below), so 1024 entries can be stored across the three quads' RAM. However, it is unlikely that the transition rule table will be completely full for a typical regular expression set, so the actual number of entries in a given state machine will be lower.

The last quad per cluster handles the default rule table and character classifier. The default rule table and character classifier (Figure 9) are simply lookup tables with one entry for each possible input symbol. The data contained in the lookup tables is only the result part of the transition rules, so the entries for both lookup tables can be retrieved in the same time as one lookup for the transition rule tables. In any case, the amount of processing for the default rule and character classification is minimal so it is easy to perform twice the memory accesses while the rest of the B-FSM engine is performing the test part.

The transition rules are 48-bits in length – 16-bits for the test part and 32-bits for the result part. The test parts for both rules are stored in one quad's RAM, while each result part is stored in one of the other two quads RAMs. Note that the actual amount of memory required for the rules is slightly less than what has been stated – both parts of the rules are padded to a number of bits equal to a power of two for ease of indexing. Figures 11 and 12 illustrate this.

padding	rule type	state tag	result flag	input
1	3	3	1	8

Figure 11: Fields and their sizes in bits in the test part of a 2012 transition rule

padding	next state	mask	table addr
5	11	8	8

Figure 12: Fields and their sizes in bits in the result part of a 2012 transition rule

Figure 13: Azure code for test part of B-FSM engine.

The type field contains flags such as case-insensitive or case-sensitive match, wildcards and negation. The mask field states which bits are extracted from the state and which bits are extracted from the input symbol for address generation (Equation 1). For the default rules and character classification unit, there is no current state information or conditions, so only the test part is required.

3.3.2 *Code Overview*

Figure 13 shows the code for the test part of the B-FSM engine. Note that the code for two processing elements (PEs) is shown – the code for the other two PEs in the quad is almost identical. Sections of the code have been colour coded – the green section is the address generation unit, the red section loads the test parts of the transition rules from memory, the blue part performs the testing required, and the yellow section loads the new transition rules and updates the registers. Unfortunately this code is not completely working, but the basis of the algorithm is there.

The address generation unit (in green) applies Equation 1 to compute an index for transition rule table lookup. It computes each side of the **or** operation in parallel on adjacent PEs, then performs the **or** to generate the index. Also included in this section is retrieving the input symbol from the switch element.

Testing the transition rules for a potential match on the Azure SoC (in blue) can be done with a series of `tests` and `skips`. Most of the instructions in this section test the type field for certain conditions such as case insensitive matching, wildcard matching and whether the rule considers a character class or

an exact match. This is achieved by extracting relevant bits in the test part of a transition rule using bitwise operations and testing for zero or non-zero values accordingly. This section determines which of the two rules tested, if any, match the current state and input information.

Retrieving the matching transition rules from other quads is performed in the final section (in yellow). All potentially matching rules and the default rule are loaded by the other quads, and placed on the link which connects the quads together. The quad which handles the test part loads one of these transition rules/default rule from the appropriate link and uses the information to update the registers.

3.3.3 *Constraints*

The Azure SoC implementation will only allow 8KB of memory per state machine. 2KB of this is used for default rules and character classification purposes, leaving 6KB for the transition rule table. Each transition rule is 48-bits, so there is a maximum of 1024 rules per state machine. This means that a regular expression that contains more than 1024 transition rules will not be able to be used in this implementation. In effect, the actual transition rule limit will be lower as the B-FSM compiler is unlikely to completely fill a transition rule table.

4

RESULTS AND DISCUSSION

This chapter presents the performance of the B-FSM compiler, the baseline and B-FSM implementations for the Xeon CPU and the Xeon Phi Coprocessor, and the estimated performance of the Azure SoC. Performance as the number of patterns increases, as well as the number of threads for the software implementations, is presented and discussed. All performance results only consider the computation time; I/O from disk is not included.

4.1 B-FSM COMPILER RESULTS

The B-FSM compiler produced for this study achieved excellent compression ratios and transition rule table densities. These results are similar to those reported by van Lunteren [34].

If the number of patterns is sufficiently high, it is likely that all but one of the transition rule tables will be completely full. It was also found that the value P, the number of allowed hash collisions, had minimal effect on the transition rule table density and on the total size of the transition rule memory. Table 3 shows the total size of the transition rule memory for a range of values of P. In this example, 589 patterns from Snort made up the rule set, and the number of B-FSMs in the array was only one. As P rises, the number of tables decrease, although the sizes of the tables increase resulting in similar storage requirements.

P	tables	memory
2	18	73728B
3	12	73728B
4	9	73728B
6	6	73728B
8	5	81920B

Table 3: Effect of P on the transition rule memory compression

Table 4 shows the effect of increasing the number of B-FSMs on the storage requirements. As has been discussed, the patterns are split between the B-FSMs using a simulated annealing strategy (Section 3.1.1) to keep patterns with common prefixes in the same B-FSMs. The table shows that increasing the number of B-FSMs only slightly increases the storage requirements.

B-FSMs	rules	rules/char	memory
4	3723	0.89	29784B
8	3857	0.93	30856B
12	3892	0.94	31136B
16	3948	0.95	31584B

Table 4: Effect of number of B-FSMs on transition rule memory compression

These results are all comparable to the results reported by van Lunteren [34], which indicates that the compiler was implemented correctly, and therefore potential mistakes in implementing it are not a factor in the performances of the implementations of the B-FSM engine.

4.1.1 *Storage Requirements vs Number of Patterns*

Figure 14 shows the total size of FSM-related data structures as the pattern count increases. The red line "B-FSM (allocated)" shows the total size of the transition rule tables (which may include empty, unused space), whereas the blue line "B-FSM" only shows the size of the actual transition rules. It is clear that the storage requirements for both B-FSM and the baseline implementation are linear in the number of patterns, but that the baseline implementation requires a significantly larger amount of memory to store the state machine. For comparison, the total size of the transition rule memory for 500 patterns in the B-FSM implementation is similar to the amount of memory required to store a single pattern in the baseline implementation.

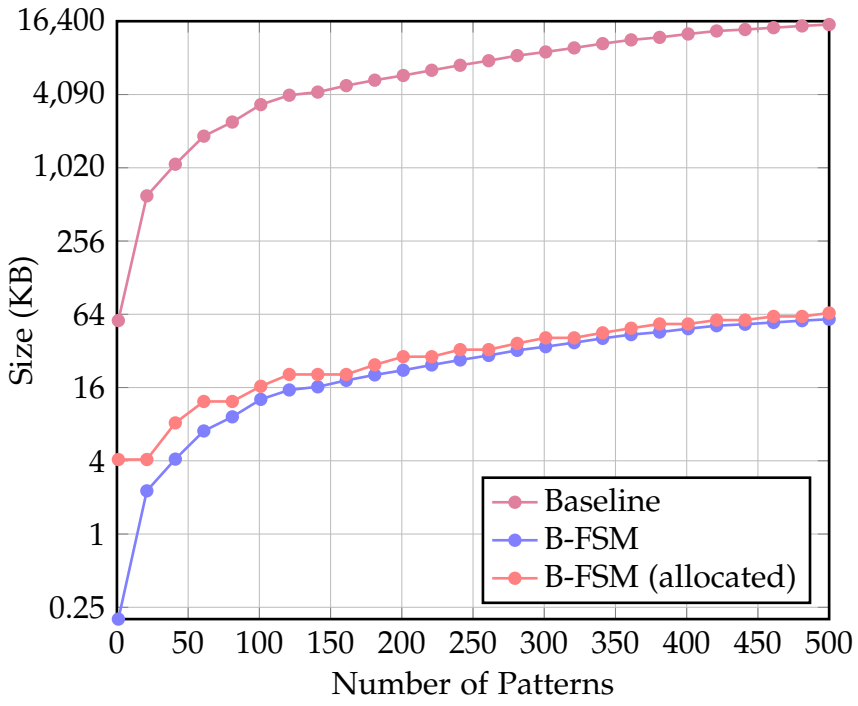

Figure 14: Memory requirements to store the state machine

4.2 XEON PHI RESULTS AND DISCUSSION

To test the Xeon Phi implementation, regular expressions from the Snort rule set have been extracted and built into DFA and B-FSM structures. These regular expressions were tested against Wikipedia dumps of varying sizes [4]. In testing, an Intel Xeon Phi 5110P Coprocessor was used for the Xeon Phi tests, and an Intel Xeon E5-2695 was used for the CPU tests.

4.2.1 *Throughput per Device*

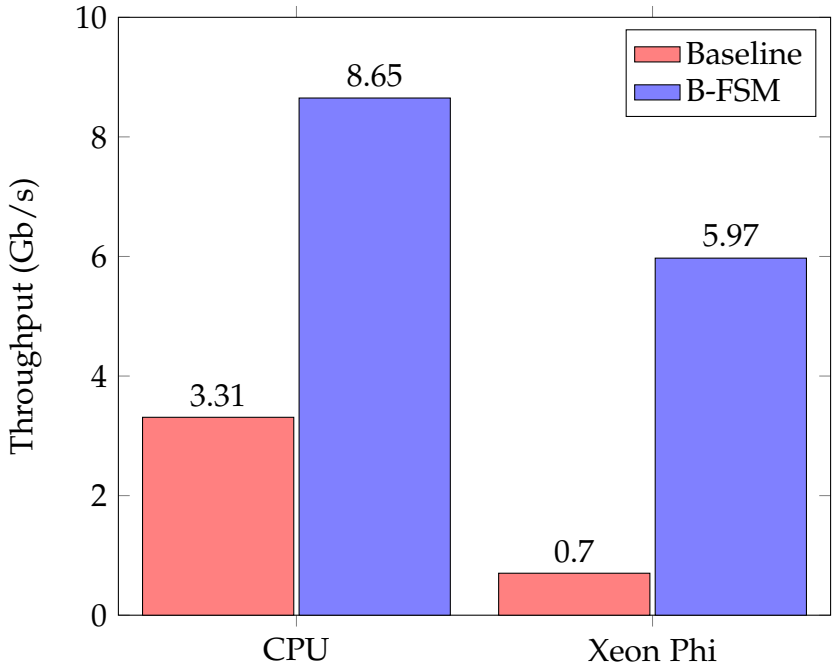

Figure 15: Throughputs of the B-FSM scheme and baseline implementation. For this graph, a pattern count of 100 was used.

As can be seen from Figure 15, the performance of B-FSM and a Thompson NFA based implementation on the Xeon Phi is poor compared to a traditional CPU. This result may be surprising, considering the huge potential performance of the Xeon Phi. The main reason for this result is that the vast majority of the potential computing power of the Xeon Phi is in the vector units. To make full use of the vector units, exactly the

same computation must be performed on multiple sets of data (Figure 10, Section 3.2.3). As the evaluation of a DFA requires both the current state and input character for each input byte, and the calculation of the next state cannot be completed by a single CPU instruction, it is very difficult to utilise the vector units to check several patterns at once, or several streams at once. By not using the vector units, the maximum computational throughput of the Xeon Phi is cut to one sixteenth of its actual capabilities.

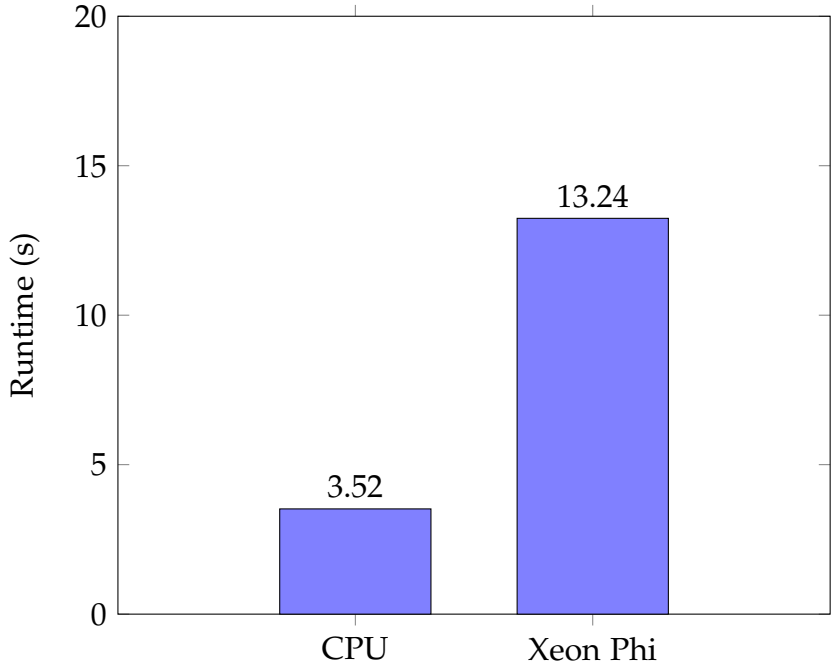

Figure 16: Time taken for a Sobel filter (no vectorisation) on CPU and Xeon Phi for a 6000x6000 pixel image.

It was initially hypothesised that even without use of the vector units, the Xeon Phi should outperform a standard Xeon CPU based on the total number of cores, and that regular expression matching could simply be a pathological problem for these devices. To test this idea, programs were developed for which it is known should see a performance boost just by increasing the number of threads. These implementations used all cores available in the Xeon Phi but did not utilise the vector

units. These programs were tested on the Xeon Phi and the standard Xeon CPU. The results obtained were similar to the regular expression results (Figure 16).

The conclusion from these tests is that the cores in the current generation of the Xeon Phi are very heavily optimised for vector computation and in general have very poor scalar performance. This conclusion is supported by the fact that the Xeon Phi cores are essentially P54C cores – the design which was used in the original Pentium – with a larger cache and 512-bit vector units added [18]. Unless the algorithm can be effectively vectorised, it is usually not worth using the current generation of the Xeon Phi. However, future generations of the Xeon Phi may make them feasible for use with heavily scalar applications like regular expression matching.

4.2.2 *Throughput vs Number of Patterns*

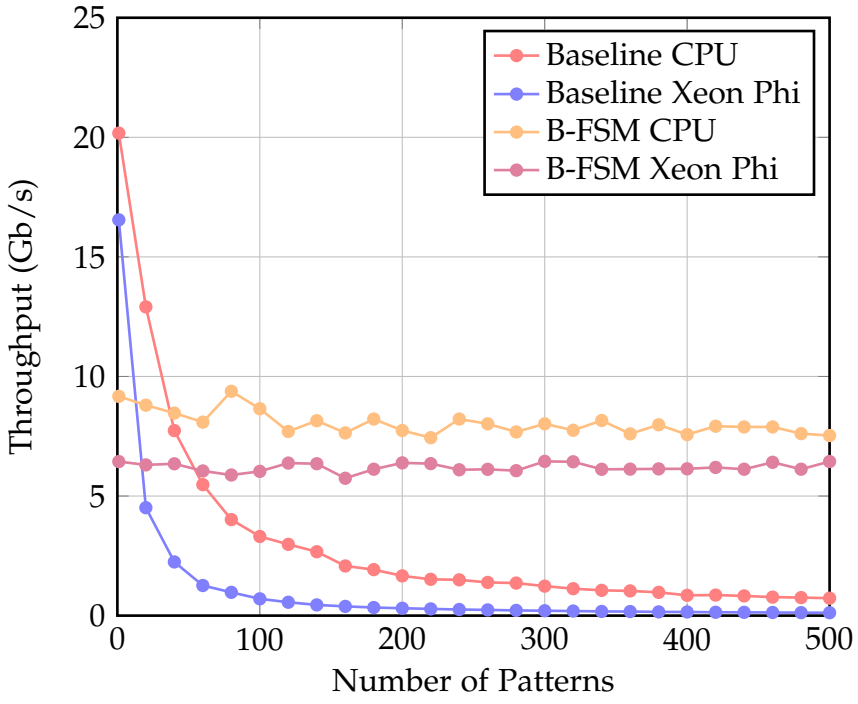

Figure 17: Throughput vs number of patterns of the baseline implementation on Xeon and Xeon Phi

The throughput of the baseline implementation quickly declines as the number of patterns increases; however, the throughput of the B-FSM implementation stays approximately constant (Figure 17). This is due to the total size of the FSM structure in the baseline implementation growing too quickly, becoming too large to be effectively cached. Even though the throughput at low pattern counts is much higher for the baseline implementation, it is quickly overtaken by the B-FSM implementation as the number of patterns in the rule set becomes sufficiently high.

Figure 17 shows that if the number of patterns is sufficiently low, the baseline implementation outperforms the B-FSM implementation on both the CPU and the Xeon Phi. There are two factors that contribute to this observation: the overhead associated with implementing a hardware design in software, and the reduced benefits of the B-FSM engine when implemented in software.

Hardware designs often employ certain parallelism which is difficult to implement in software efficiently. The B-FSM engine is no exception – the original implementation can check multiple rules at a time whereas the software implementation for the Xeon Phi must iteratively loop over these rules (Algorithm 2). It may be possible to utilise the vector units of the Xeon Phi for rule checking in parallel, and this is a possible topic of future research. However, this problem is not specific to the Xeon Phi – even with these improvements, it is likely that the performance of the B-FSM scheme on the Xeon Phi would be unimpressive compared with a standard Xeon CPU.

The main advantages of the B-FSM engine – the reduced number of memory accesses during execution and smaller total storage requirements – are not as important in software when the number of patterns is low due to CPU caching. The Xeon Phi boasts a very large cache (30MB L2) [1] which can be ac-

cessed at low latencies. The transition rule tables will be stored in the cache for the majority of the runtime, meaning that the total benefit of reduced memory accesses is much less compared to embedded memory in an FPGA or ASIC when the pattern count is low.

4.2.3 *Throughput vs Number of Threads*

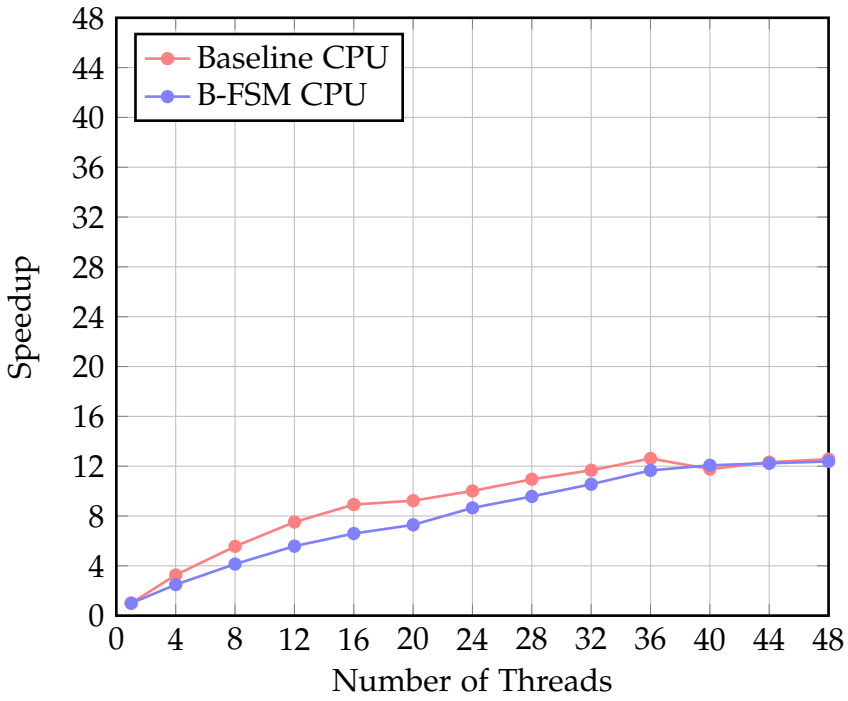

Figure 18: Speedup vs Number of Threads on a CPU

The performance of the baseline and B-FSM implementations on CPU scale similarly well as the number of threads increases. However, on the Xeon Phi, the B-FSM implementation scales much better than the baseline. Figures 18 and 19 illustrate this result, showing the average relative speedup for several different numbers of patterns.

4.2.4 *Throughput vs P*

In the Xeon Phi implementation, a transition table rule width of 8 was possible as the memory width is 64-bytes. However,

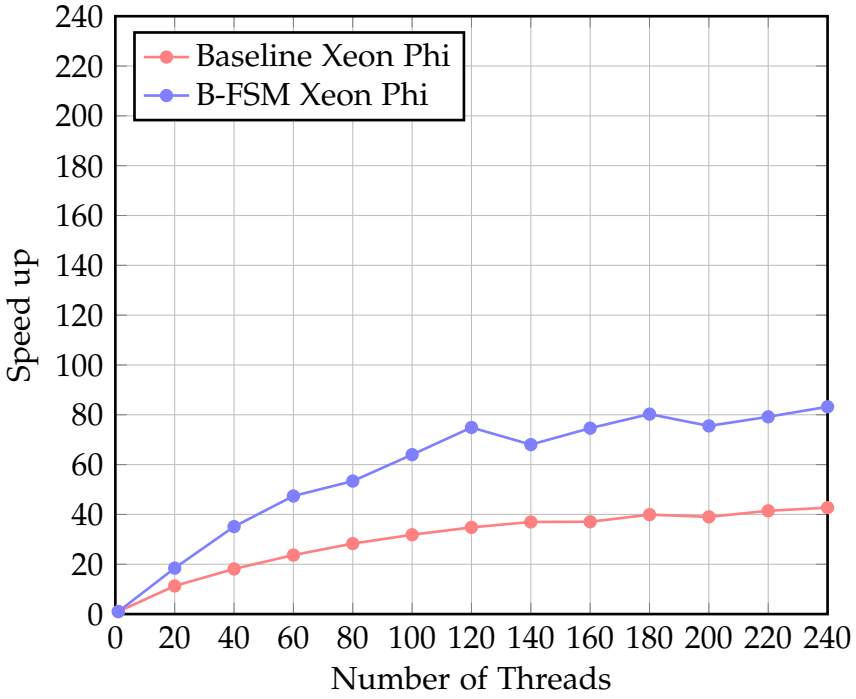

Figure 19: Speedup vs Number of Threads on a Xeon Phi

experimental results showed that a lower value of P produced higher throughputs (Table 5). This is due to the limitations in software regarding the need to iteratively test each rule in a transition rule row.

P	throughput (Gb/s)
2	7.58
4	7.25
8	6.17

Table 5: Effect of P on the B-FSM implementation

4.3 AZURE SOC RESULTS AND DISCUSSION

As stated in Section 3.3, the Azure implementation was not fully completed. However, performance estimates can still be made accurately with high confidence.

4.3.1 *Throughput Per Device*

Clusters in the Azure SoC have a circular instruction buffer with a size of 128 instructions. The complete, but not fully working, code for the B-FSM engine is 48 instructions long, meaning that two copies of this code can be stored in the instruction memory. It is unlikely that more than 17 instructions are needed in order to fix this code, which would only allow a single run through of the engine per 128 instructions. It is therefore possible to say with high confidence that a complete implementation of the B-FSM engine would consume two characters of an input stream per cluster for every 128 instructions.

The chip runs at 10GHz, so it is estimated that each cluster will be able to process an input stream at a maximum throughput of $\frac{10GHz}{128} \times 2 = 156MB/s$. This gives an estimated peak aggregate throughput for the entire Azure SoC of $156MB/s \times 1024 = 160GB/s$, although this figure only holds for a small regular expression set. There would also need to be 1024 different input streams for this peak to be hit, which is unlikely to occur in a real-world scenario.

4.3.2 *Throughput vs Number of Patterns*

The number of patterns is estimated to have a large effect on the throughput of the Azure SoC (Figure 20). This is because the memory available to each cluster is relatively small. As the set of regular expressions increases in size, the transition rule tables are likely to exceed the 6KB available for storing them.

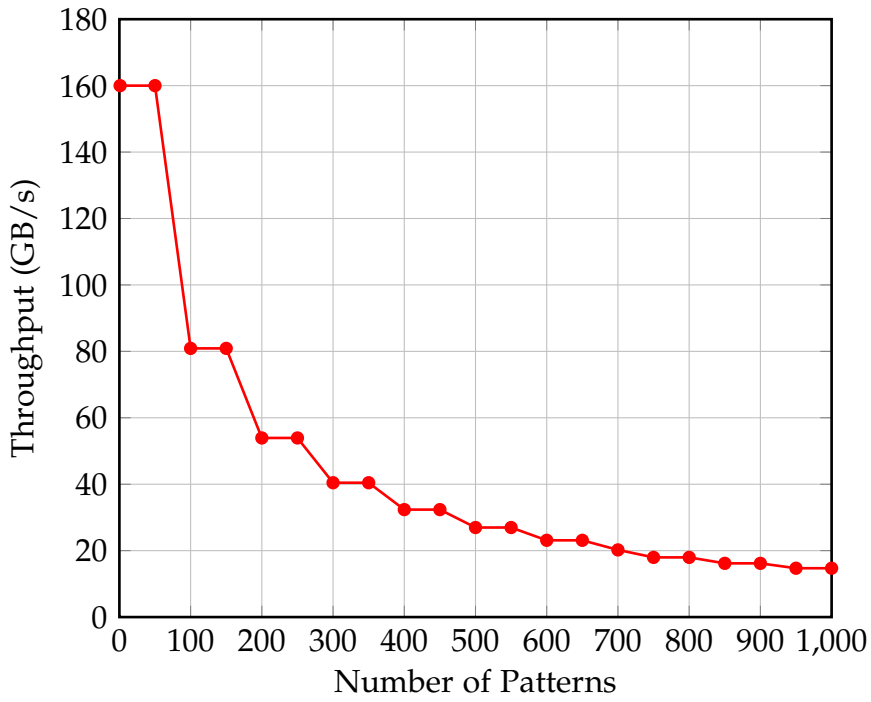

Figure 20: Number of Patterns vs Estimated Throughput

Van Lunteren reports that the B-FSM scheme achieves average compression rates of 4-6 bytes of transition rule memory per character for sets of case sensitive patterns [34] such as the ones tested in this study, and that the average pattern length is 13.2 characters. Using this information, an estimation for the number of FSMs required to cover the whole rule set can be calculated:

$$FSMs = \left\lceil \frac{n \times 5 \times 13.2}{6144} \right\rceil$$

From here, the throughput of the device can be estimated as

$$throughput = \frac{1024}{FSMs} \times 158$$

4.4 COMPARISON TO OTHER DEVICES

Figure 21 shows the throughputs of the B-FSM implementations produced as part of this study, as well as the B-FSM scheme implemented in IBM's Power Edge of Network (PowerEn) processor [33]. It is clear from the graph that the Azure SoC has excellent throughput potential for the problem of regular expression matching. The Xeon Phi and CPU implementations do not compare favourably to existing implementations of the scheme. It is also highly likely that the power consumption of the software implementations will be much greater than the Azure and PowerEn implementations.

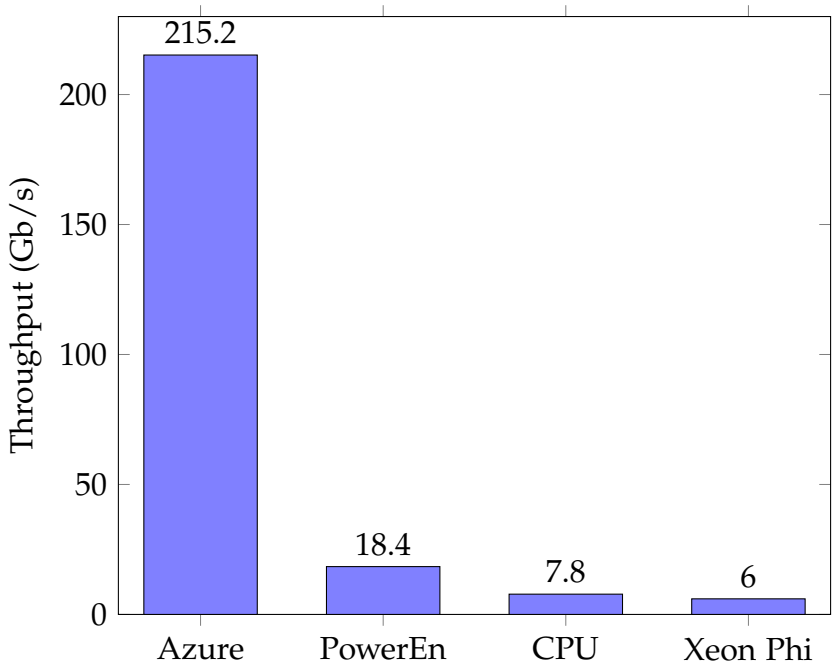

Figure 21: Throughputs of the B-FSM scheme

5

CONCLUSION

The goal of this study was to evaluate the performance of the Intel Xeon Phi Coprocessor and the Azure System-on-Chip for regular expression matching. The B-FSM engine has been implemented and thoroughly tested for the Xeon Phi, which resulted in poor performance in terms of throughput compared to an ordinary CPU. A mapping of the B-FSM engine has been designed for the Azure SoC, and unlike the Xeon Phi, this is estimated to have much greater throughput per device than current state-of-the-art hardware based implementations of the B-FSM scheme.

5.1 CONTRIBUTIONS AND FUTURE WORK

As mentioned in Section 4.2.2, the vector units may be able to be applied to test multiple transition rules in parallel, much like the hardware implementation of the B-FSM engine. This has the potential to improve the overall efficiency of the Xeon Phi implementation, as well as reduce the overhead regarding increasing the transition rule table width discussed in Section 4.2.4. This would involve refactoring the data structure of the transition rule memory (Listing 2) to align fields of adjacent rules to allow for minimal data shuffling.

Results from this study show that it is not worthwhile in general to apply heavily scalar problems to the current generation of Xeon Phi. However, it would be interesting to compare the results from this study with the performance of the Xeon Phi

implementation on the next generation of Xeon Phi, *Knights Landing*, which is currently in development. *Knights Landing* will feature Airmont cores, a 14-nanometer Atom architecture. These cores will be more suitable for predominantly scalar algorithms such as regular expression matching. This has the potential to vastly increase the performance of the B-FSM engine on the Xeon Phi.

Completing the Azure implementation and testing it on real sets of expressions to confirm the estimated results presented would be useful to show that CGRA based devices are superior to other reconfigurable hardware such as FPGAs for regular expression matching. The estimated result of 11.7x greater throughput per device than existing state-of-the-art implementations of B-FSM is significant. A complete Azure implementation would allow recording of throughput per power estimates which is a useful statistic as well. The implementation of algorithms for Azure will be easier in future. Useful developer tools, such as a tool which allows programming the device in C, are currently in development.

5.2 FINAL REMARKS

This thesis shows that the Xeon Phi, while appearing attractive, is not always useful in speeding up arbitrary applications. It also gives support to the growing idea that CGRA based devices can give excellent gains in performance over other types of reconfigurable hardware for certain applications.

BIBLIOGRAPHY

[1] Intel Xeon Phi 5110P Coprocessor specifications. http://ark.intel.com/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core. Accessed: 5-10-2014.

[2] Snort.org rules downloads page. https://www.snort.org/downloads. Accessed: 15-7-2014.

[3] Regular expression matching can be simple and fast. http://swtch.com/~rsc/regexp/regexp1.html. Accessed: 21-4-2014.

[4] Wikimedia downloads. http://dumps.wikimedia.org/backup-index.html. Accessed: 9-7-2014.

[5] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[6] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, pages 145–154, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-945-6. doi: 10.1145/1323548.1323573. URL http://doi.acm.org/10.1145/1323548.1323573.

[7] Anthony M. Blake and Chris Nicol. Computing the FFT on a hybrid coarse grained reconfigurable architecture. Technical report, Wave Semiconductor, 2013.

[8] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[9] Jeffrey D Brown, Sandra Woodward, Brian M Bass, and Charles L Johnson. Ibm power edge of network processor: A wire-speed system on a chip. *IEEE Micro*, 31(2):0076–85, 2011.

[10] Janusz A Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical theory of Automata*, 12(6):529–561, 1962.

[11] Tim Cramer, Dirk Schmidl, Michael Klemm, and Dieter an Mey. OpenMP programming on Intel ® Xeon Phi coprocessors: An early performance comparison. 2012.

[12] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *Parallel and Distributed Systems, IEEE Transactions on*, 2014.

[13] Jiri Dokulil, Enes Bajrovic, Siegfried Benkner, Sabri Pllana, Martin Sandrieser, and Beverly Bachmayer. Efficient hybrid execution of C++ applications using Intel ® Xeon Phi coprocessors. *arXiv preprint arXiv:1211.5530*, 2012.

[14] Jianbin Fang, Ana Lucia Varbanescu, Henk Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. An empirical study of intel xeon phi. *arXiv preprint arXiv:1310.5842*, 2013.

[15] Henry Gouk and Anthony M. Blake. AES implementation for the Azure SoC. Technical report, Department of Computer Science, University of Waikato, 2013.

[16] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G Shet, George Chrysos, and Pradeep Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on Intel ® Xeon Phi coprocessor. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 126–137. IEEE, 2013.

[17] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[18] Joel Hruska. Intel's 50-core champion In-depth on Xeon Phi. http://www.extremetech.com/extreme/133541-intels-64-core-champion-in-depth-on-xeon-phi. Accessed: 17-10-2014.

[19] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simmulated annealing. *science*, 220(4598): 671–680, 1983.

[20] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM Computer Communication Review*, 36(4):339–350, 2006.

[21] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 81–92. ACM, 2006.

[22] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Speculative parallel pattern matching. *Information Forensics and Security, IEEE Transactions on*, 6(2):438–451, 2011.

[23] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[24] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[25] Reetinder Sidhu and Viktor K Prasanna. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 227–238. IEEE, 2001.

[26] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts*. Wiley, 2013.

[27] Randy Smith, Cristian Estan, and Somesh Jha. Xfa: Faster signature matching with extended automata. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 187–201. IEEE, 2008.

[28] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *ACM SIGCOMM Computer Communication Review*, 38(4):207–218, 2008.

[29] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6): 419–422, 1968.

[30] A. Vajda. *Programming Many-Core Chips*. Springer, 2011. ISBN 9781441997395. URL http://books.google.co.nz/books?id=pSxa_anfiG0C.

[31] Jan van Lunteren. Searching very large routing tables in wide embedded memory. In *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE*, volume 3, pages 1615–1619. IEEE, 2001.

[32] Jan van Lunteren and Alexis Guanella. Hardware-accelerated regular expression matching at multiple tens of Gb/s. In *INFOCOM, 2012 Proceedings IEEE*, pages 1737–1745. IEEE, 2012.

[33] Jan Van Lunteren, Jon Rohrer, Kubilay Atasu, and Christoph Hagleitner. Regular expression acceleration at multiple tens of gb/s. In *Workshop on accelerators for high performance architectures, international conference on supercomputing*, 2009.

[34] Jan van Lunteren et al. High-performance pattern-matching for intrusion detection. In *Infocom*, volume 6, pages 1–13. Citeseer, 2006.

[35] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Recent Advances in Intrusion Detection*, pages 265–283. Springer, 2009.

[36] Alfred KW Yeung and Jan M Rabaey. A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput DSP algorithms. In *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, volume 1, pages 169–178. IEEE, 1993.

[37] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102. ACM, 2006.