# **Reconfigurable Message Traffic Filters** MEng Individual Project

Deryck Arnold (daa07) Imperial College London CID: 00469268

Supervisors: Professor Alexander L. Wolf Professor Wayne Luk

#### Abstract

In this report we discuss the problem of string matching performance in content-based routing. We go on to describe and evaluate designs for reconfigurable hardware such as *field-programmable gate arrays* (FPGAs) that can be applied to solving this problem.

The hardware designs presented in this project are estimated to give a 37 times speed up and 370 times power efficiency increase when running on a Xilinx Virtex6 FPGA over equivalent C implementations on a 2.40GHz dual-core Intel Core i3 processor.

We present a highly-extensible compiler tool that can convert abstract content-based forwarding rules into such designs for processing by further tools into working FPGA images.

Finally we discuss the potential applications and future developments of the work covered in this project.

## Acknowledgements

I would like to thank my supervisors Professor Alexander Wolf and Professor Wayne Luk for providing guidance, enthusiasm and the expertise to encourage me through this project.

I would also like to thank my friends who have offered me help and assisted in proof-reading this report.

# Contents

1	Intr	roduction	3
<b>2</b>	Rela	ated Work	5
	2.1	Content-Based Networking and Forwarding	6
		2.1.1 Siena Fast Forwarding (SFF)	6
		2.1.2 Content-Based Matching using GPUs	7
	2.2	String Matching - Software to Hardware	8
		2.2.1 Brute-Force (BF) Approach	8
		2.2.2 Knuth-Morris-Pratt (KMP)	8
	2.3	Content-Addressable Memory (CAM)	10
		2.3.1 Simple CAM Comparator Architecture	10
		2.3.2 Decoded CAM (DCAM)	10
		2.3.3 Decoded Partial CAM (DpCAM)	11
		2.3.4 Ternary CAM (TCAM)	11
	2.4	Reconfigurable Hardware	12
•			
3	Acc	elerating Matching	14
	3.1	Node and Message Model	15
	3.2	Predicate Language	16
		3.2.1 Operators	16
		3.2.2 Representation of Forwarding Rules	17
	3.3	Hardware Design Implementations	19
		3.3.1 CAM Design	19
		3.3.2 DCAM Design	20
	~ .	3.3.3 Addition of the $lt$ and $gt$ Operators $\ldots \ldots \ldots$	21
	3.4	C Implementation	23
	3.5	Automating Design Generation	24
		3.5.1 Compiler Overview	24
		3.5.2 Rule Format	26
		3.5.3 Validation of Compiler Output	26
		3.5.4 The Users Perspective	27
		3.5.5 The Developers Perspective	28
<b>4</b>	Eva	luation	31
	4.1	Test Setup	32
		4.1.1 C Implementation	32
		4.1.2 Verilog Implementation	33
	4.2	Results	34

CONTENTS

Reconfigurable Message Traffic Filters

	4.2.1	Per-character 1 mings
	4.2.2	Per-message Timings
	4.2.3	Speedup
4.3	Resou	rce Utilisation
	4.3.1	Power Requirements
	4.3.2	Executable Size
	4.3.3	FPGA Utilisation
4.4	Summ	lary
Cor	lclusio	ns and Future Work
	Applie	cations and Future Work
5.1	- ippm	
5.1	5.1.1	Generalised Design

# 1 Introduction

Since the conception of the internet, networked traffic over various protocols has been increasing dramatically, increasing the pressure on the systems that process and route this traffic to do so in a timely manner.

One such class of systems would be those involved in filtering out unwanted traffic, called *network intrusion detection systems*, or NIDS for short. An analysis of the popular SNORT NIDS showed that string matching was one of the major contributors to overall execution time, with 31% for 8.6 million packets totalling more than one gigabyte of network data spent on string matching [12]. It is for this reason that there have been many studies in increasing string matching efficiency, with many focussing on exploiting the unique nature of hardware over software [22, 19, 16, 4, 5, 12].

Hardware implementations are known to be faster than equivalent software implementations for a variety of applications. This is due to the differing methods of operation [10]. One one side of the scale there are *application specific integrated circuits* (ASICs) that are designed specifically to solve a specific problem, however should that problem change, the chip needs to be redesigned and refabricated. A more flexible approach on the other side of the scale is a standard microprocessor that executes software instructions. By changing the instructions, a wide variety of functionalities can be supported without requiring changes to the hardware itself. However, because this microprocessor is required to look up instructions and decode them before ultimately executing them, operation can be much slower than a dedicated ASIC for the same task. In the gap between the two types exists the area of reconfigurable computing. Devices such as *field-programmable gate arrays* (FPGAs) can be configured to exhibit different behaviours by programming the logic blocks that they consist of [3]. The flexibility and performance of reconfigurable computing makes it attractive for string searching, and in this paper we will be looking to apply such techniques to a different, but similar area: *content-based routing*.

Content-based communication is a relatively new area, where messages sent over a contentbased network aren't routed based on a destination address, but rather by their content [8, 6]. It is typically implemented as an overlay network in the application layer, and like NIDS, there is emphasis on increasing the efficiency of routing through this network.

One such implementation that focussed on increasing the efficiency of forwarding messages through a content-based network would be the Siena Fast Forwarding (SFF) algorithm. It presented a novel approach to forwarding by preprocessing messages to reduce overall processing

1. INTRODUCTION

time of the message, as well as other optimisations in areas such as string matching [7].

In this report we focus on possible hardware designs that can increase the performance of contentbased routing of messages with arbitrary string tags. We move onto the automatic generation of these designs, both in software and hardware. This moves the responsibility of converting rules from the user of the system to a compiler. In doing so, this reduces the number of errors that can be introduced by building a design by hand and opens the rule synthesising process up to optimisation.

Results from FPGA synthesis tools of outputted hardware designs estimate speedups to be, on average, 30 times faster than equivalent software implementations. While static designs are the focus of this project, through the automation of design production, prototypes may be generated quickly, providing the framework for future development in the field, capitalising on the substantial performance gains hardware can provide over software.

One possible application identified by this project involves using the design processes and software devised here for just-in-time (JIT) extensions of existing algorithm implementations such as SFF. Another extension to this project discusses exploring the use of the software compiler to generate configuration data for a self-reconfiguring hardware implementation. A self-reconfiguring design would fully exploit the highly reconfigurable nature of FPGAs, giving such an implementation an edge over an equivalent, but possibly faster ASIC design.

In summary, the contributions of this work are:

- High-performance hardware designs and techniques for content-based multi-operator string matching in messages. Estimates put the speedup at 37 times and power-efficiency of 370 times using a Xilinx Virtex6 FPGA over equivalent software running on a 2.40GHz dual-core Intel Core i3. [Section 3.3]
- Abstraction of intent from implementation, utilizing the following two methods:
  - An easy-to-use language, simple-string forwarding rules (SSFR), to specify tag-based message filters. [Section 3.5.2]
  - An extensible compilation framework that can translate SSFR into targeted hardware and software designs that are self-validating. [Section 3.5.5]

We will begin by setting the scene in part 2 with related background research. In part 3, we will lay the groundwork of the model and assumptions before describing the work undertaken. Part 4 covers the test setup and results of benchmarking equivalent software implementations against hardware designs. Finally, we discuss the benefits and disadvantages of our implementation, as well as future paths that could be explored in part 5.

# Related Work

 $\mathbf{2}$ 

Much work has been done in the fields of content-based networking, string matching and hardware acceleration of string matching. In order to put our work into context, we will summarise related work in these fields in this chapter.

We will touch on the concept of content-based networking, before moving on to string matching in software and how equivalent hardware representations have been implemented. Finally, we will cover the exciting topic of reconfigurable hardware and what advances have been made in the field.

## 2.1 Content-Based Networking and Forwarding

A content-based network differs to the traditional IP network (though it may be built on top of one) in that nodes aren't assigned unique network addresses. Instead, each node advertises a receiver predicate, or r-predicate, which specifies what messages sent over the network the node is interested in receiving [8]. An r-predicate doesn't necessarily correspond to what the node itself is interested in, but rather the node, may be part of a path to a node that is ultimately interested in messages matching the r-predicate. This allows messages to be routed through a network from various publishers to interested subscribers without requiring knowledge about the individual subscribers themselves, just their interests which are propagated through the network. The following is an example of an r-predicate:

 $first = john \land last = smith \lor first = mary \land title = mrs. \lor title = sir$ 

Each routing node would maintain a set, or *forwarding table* of these r-predicates associated with each interface it can forward messages to. If an incoming message matches an r-predicate stored in this table, it is forwarded to the interface the predicate is associated with. Thus, this forwarding function can be represented as the following [6]:

 $forward(m) = \{i \in I : m \text{ matches } FwdTable(i)\} \text{ where } FwdTable : I \to P$ 

With large numbers of subscribers, these forwarding tables can get quite large, which can increase filtering and forwarding speed [8]. Therefore emphasis is placed on reducing the size of these forwarding tables, increasing the efficiency of the forwarding function and reducing the amount of uncessary traffic flowing through routers [6]. This is where we move onto implementations such as SFF, that will be discussed in 2.1.1.

#### 2.1.1 Siena Fast Forwarding (SFF)

SFF was designed for fast forwarding of messages in a content based network. It was in particular designed to scale well to situations where there are large numbers of complex predicate filters and high volumes of messages being generated by content publishers [7].

For a particular message, not all the forwarding rules needed to be applied, as some can be filtered out in an initial pre-processing step. Interfaces whose r-predicate attributes are not found in a particular message can be ignored, therefore reducing the additional computation that would have otherwise be spent on filtering on the interface's constraints. This preprocessing step can be performed a number of rounds, with the potential to remove an interface each round, however the number of rounds performed is dependent on the benefit attained against the time cost of the preprocessing itself. The higher the number of constraints, the more benefit is gained from a large number of preprocessing rounds [7].

SFF uses a modified version of a *ternary search tree* (TST) to store search patterns and operators. Each node in a TST is comprised of a partitioning value, such as a character in a search string, as well as three pointers to further nodes in the TST. These pointers correspond to the less-than, equals and greater-than operators on this partitioning value [2].

The SFF implementation uses the same structure as the standard TST with the following extensions:

- Multi-operator index for matching multiple operators.
- 2. RELATED WORK

- Capability of matching partial strings for the prefix and substring operators.
- Greater than and less than constraints at TST leaf nodes are linked.
- Backtrack functions to transition between a partial match to the next closest complete match.

To match the substring and suffix operators, the TST search is performed starting at each character of an input string, with the prefix, equality, greater than and less than operator matches disabled past the first character [7].

#### 2.1.2 Content-Based Matching using GPUs

A recent implementation of a content-based forwarder involved using CUDA to exploit the highly parallel nature of GPUs [15]. In a similar way to SFF, filters for the CUDA Content-based Matcher (CCM) are composed of constraints. In this particular implementation the constraints are not stored in CPU memory but instead in the GPU memory. The CPU selects constraints to filter on for a particular message and copies this data to the GPU, which spawns thousands of threads to perform the matching process. After this process, the vector of interface matches is copied back into the CPU memory space.

The results from this showed that the CCM implementation achieved up to a thirteen times speed up from a software SFF implementation running on a six-core CPU. The results give a good idea of the potential benefits that can be obtained from parallelising the matching process in content-based forwarding.

## 2.2 String Matching - Software to Hardware

Much work has been done in the field of string matching in both software and hardware. Because our filters are built from predicates over arbitrary string tags, we will cover a few methods of string searching in this section.

We will first cover simple methods of string searching in software, and how they have been implemented in hardware in order to give some background into the translation of software algorithms into hardware.

#### 2.2.1 Brute-Force (BF) Approach

The simplest and possibly the most obvious way of matching a string pattern P of length m in search text S of length n would be to use the brute-force method.

#### **BF** in Software

Searching for P involves starting a fresh search at every index of S until a full match is found. This is shown in Listing 1.

Listing 1 BF algorithm

```
for i := 0 to n - 1 do

for j := 0 to m do

if j = m then

return match;

end if

if (i + j) = n or S[i + j] \neq P[j] then

break;

end if

end for

return no match;
```

Due to the excess of character comparisons, for an arbitrary P of length m and arbitrary S of length n the worst-case time complexity is O(mn), which is quite slow [9].

#### **BF** in Hardware

The BF method can be implemented on hardware as simple discrete comparators [11, 22] as shown in Figure 2.2. This method allows matching of the full search string each cycle, with character comparisons done in parallel. This results in high-performance, but with an increased area cost on-chip [11]. We will further discuss the topic in 2.3 as it warrants more attention.

#### 2.2.2 Knuth-Morris-Pratt (KMP)

The Knuth-Morris-Pratt algorithm is one of the most well known string-matching algorithms and was devised for fast matching of single-pattern strings. The worst case time complexity searching for an arbitrary P of length m and arbitrary S of length n becomes O(m + n) when using KMP [9]. Its string matching procedure is represented as pseudo code [14] in Listing 2.

8

2. RELATED WORK

Reconfigurable Message Traffic Filters

Listing 2 KMP pattern matching algorithm

j := 1; k := 0;while j < m and k < n do while j > 0 and  $S[k] \neq P[j-1]$  do j := next[j];end while end while if j = m then return match; else return no match; end if

This procedure relies on a precompiled table called *next* that references the next state to jump to at j if a character does not match. If we take an example where P = "abac" we can compile P into a set of j and next[j] states as shown in Table 2.1.

j	To match	next[j]
0	-	-
1	a	0
2	b	1
3	a	0
4	c	2

Table 2.1: KMP table for "abac"

This can be represented as a finite state machine (FSM) as shown in Figure 2.1.



Figure 2.1: KMP Finite State Machine for "abac"

There have been several implementations of KMP in hardware in the past, one of which is particularly interesting to us, as it takes advantage of the reconfigurable nature of the FPGA considerably. It will be discussed in 2.4.

#### 2. RELATED WORK

# 2.3 Content-Addressable Memory (CAM)

In typical RAM, data is mapped with addresses, requiring the user to supply an address to retrieve data. This is in contrast to *content-addressable memory*, or CAM, where the data is supplied, resulting in an address [16]. This CAM search is typically performed in one clock cycle, which makes it an ideal solution to bandwidth-heavy applications such as network intrusion detection systems (NIDS) [4].

One of the downsides of the CAM design is its high power consumption. This is due to its highly parallel nature, requiring a large amount of circuitry to be active every cycle, however pipelining techniques can reduce this [17, 18].

#### 2.3.1 Simple CAM Comparator Architecture

The simple CAM architecture as shown in Figure 2.2(a) shifts the input data through registers, the outputs of which are distributed amongst pattern comparators. This results in a regular pattern that can achieve high operating frequencies [11].

However, one of the main downsides of this design is the area cost, but this can be improved by combining shared characters at the same index into one comparator, as illustrated in Figure 2.2(b).



Figure 2.2: Simple CAM comparator

#### 2.3.2 Decoded CAM (DCAM)

Another CAM comparator design is the DCAM comparator [22], where the input shift registers are removed, and comparators are fully shared by the AND gates. Extra registers are placed between the comparators and the AND gates to ensure partial matches are delayed appropriately. This is illustrated in Figure 2.3.

The new registers used to delay the match signals are only one bit wide, in contrast to the original input shift registers which were eight bits wide. This can result in massive area savings [22, 11] when coupled with the fact that fewer comparators are required to match the same patterns if they share characters.

2. RELATED WORK

Reconfigurable Message Traffic Filters



Figure 2.3: DCAM comparator

However one drawback is that the number of one-bit registers is proportional to the length and number of the match patterns. This means should either grow too large, the area of the one-bit registers may exceed the area of the eight-bit registers used in the simple design [22, 11].

#### 2.3.3 Decoded Partial CAM (DpCAM)

Another variant of DCAM is *decoded partial CAM*, or DpCAM. This is where characters are decoded in parallel to increase the processing throughput [11]. In a DpCAM with a parallelism factor of two, two characters would be processed per cycle, requiring two decoders, one for each character. Due to the extra decoder, a balance must be found between parallelism and area cost.

#### 2.3.4 Ternary CAM (TCAM)

Up to this point, we have only been discussing binary CAM, where each character is represented as a sequence of 0/1 bits. There is another variant, called *ternary CAM* where a third bit (we will use X to represent it) is added that stands for *don't care* [18, 5, 16]. Table 2.2 is a truth table for the three types of values.

CAM Data	Input Data	Match
0	0	1
0	1	0
1	0	0
1	1	1
Х	0	1
Х	1	1

Table 2.2: Truth table for TCAM bit states

This would effectively allow wildcard characters to be implemented if we were to use TCAM for string searching. The downside to this, however, is the the extra logic required to represent the *don't care* state. Another potential downside is that there can potentially be multiple matches for a given input, which would require some sort of prioritisation by an encoder the TCAM comparator would be fed into [16].

## 2.4 Reconfigurable Hardware

Reconfigurable hardware is consists of configurable logic that is determined by data written into configuration memory, which is typically SRAM. This allows the device logic to be changed by modifying the data stored in the configuration memory.

Devices such as *field-programmable gate arrays* (FPGAs) are examples of reconfigurable hardware, and it is this reconfigurability that gives them an advantage over *application-specifc integrated circuits* (ASICs) for some applications.

This reconfigurability can be exploited at compile-time, i.e. before the device is operational, but to gain the full effectiveness of this paradigm, reconfiguration can be done at run-time. This allows the device to be configured with logic specific to the particular problem at hand. This is known as *instance-specific logic* [21].

In order to take advantage of the increased execution performance from problem-specific logic, reconfiguration must be performed in a timely fashion. Typically reconfiguration is performed by an external device both at compile-time and run-time, which is a relatively slow process [20, 21]. In [21], Sidhu et al. discusses reconfiguration via an external device (Figure 2.4(a)) and self-reconfiguration by the FPGA itself (Figure 2.4(b)).



a) External reconfiguration
 b) Self-reconfiguration

Figure 2.4: Process flow for FPGA reconfiguration

The total time from receiving a problem instance to outputting a solution  $(T_I)$  is the sum of the time it takes to map the problem to logic  $(T_I)$ , the time it takes to reconfigure the device  $(T_M)$  and the time it takes to execute the logic  $(T_E)$ .

 $T_M$  varies depending on the application and can range from minutes to several days, however it is usually quite large, which can limit the effectiveness of reconfigurable computing [21]. Meth-

ods have been tried to reduce this problem, such as using CAD tools offline to create a generic template skeleton that is partially reconfigured at run time [13]. There is still the issue of  $T_{ME}$  in this case if the partial reconfiguration is performed off-chip, as this is generally associated with the speed of the connection between the host and the FPGA, as the delay may be too long for most real-time applications. This can be overcome through self-reconfiguration on chip.

Following on from 2.2.2, there have been several implementations of the KMP algorithm utilising self-reconfiguration. One implementation was based on calculating the KMP pattern FSM and storing the state table on internal block RAM [19]. It used an embedded soft-core processor to perform calculation of the various FSM states for string matching, and dedicated KMP logic to process text based on the FSM states stored in memory.

Another implementation exploited using multiple contexts to achieve self-reconfiguration [21]. The FPGA would switch between contexts in order to calculate the KMP FSM and dedicated logic and to perform the actual string searching.

# Accelerating Matching

3

The initial implementations focussed primarily on attaining a working Verilog prototype implementation. These are covered in section 3.3. Alongside the Verilog, an equivalent software implementation written in C was devised, which is described in section 3.4. The implementations were developed on the assumptions and model outlined in section 3.1, and the predicate language defined in section 3.2.

The choice of C as the software implementation language was primarily down to the low-level nature of the language. Since C does not come with any built-in garbage collection, or any other unpredictable sources of overhead, benchmarks can be more representative of the capabilities of software.

For small sets of forwarding rules, this was adequate, as code can be translated easily by hand from a rule specification. However for larger sets, writing the source code for forwarding rules can be potentially time-consuming and error-prone. The solution was to implement a compiler that could generate a software or hardware representation from a forwarding rule specification input. The design of the compiler written is detailed in 3.5.

# 3.1 Node and Message Model

In order to focus on the problem of implementing forwarding logic on hardware, it was important to have a well-defined model with which we could base our solution on.

A general content based network is made up of many interconnected nodes. Each participant node maintains a set of routing, or forwarding rules to forward messages from one point of the network to another. The network may not be fully connected, so paths from any two nodes may run through intermediate nodes.

For the purpose of this project, we have ignored the structure of the network, the process of distributing forwarding rules between nodes and methods used to pass messages between nodes. This left us with the local forwarding processes occurring at an individual, arbitrary node in an arbitrary content-based network to form our model upon.

As described earlier, our model was based on an individual node within a network. A node within this model is described to be made up of forwarding logic and a series of interfaces with which the node communicates with the external network.



Figure 3.1: A simplified representation of a node

Messages are received on these interfaces and passed to the forwarding logic, which determines the interface or interfaces to pass the message on to. We have assumed that messages received are received whole, with clear distinctions between the components of a message.

The components that make up a particular message are simply modelled as string tags, which are operated on by the forwarding logic using predicate rules described in the next section.

# 3.2 Predicate Language

In our model, an *interface* is defined as a route to which a message may be forwarded to a node. The routing logic for each interface is described as a disjunction of *filter* predicates. Each *filter* is a conjunction of predicate *terms*, which are pairs of patterns and string operators. This is similar to the SFF model, but with the omission of attribute keys and other datatypes. In order to represent this model, we developed a predicate language that describes forwarding rules in this environment in a concise manner.

To further describe this language, we will cover the different aspects in more detail in the following subsections.

#### 3.2.1 Operators

Table 3.1 describes the set of string operators we implemented in our predicate language. Here, P stands for the corresponding pattern in a term, and T describes a tag in the message being filtered. One important point of note is that though operators are represented as unary functions in the forwarding rules, they operate over the term pattern P and *every* tag in a message in the form of P is-op T.

<b>Operator on</b> $P$	True for string $T$ if:
eq	P is exactly equal to $P$
ss	P is a substring of $T$
pf	P is a prefix of $T$
sf	P is a suffix of $T$
lt	P is less than $T$
gt	P is greater than $T$

Table 3.1: String operators implemented

The lt and gt operators stand for lexicographically less than and lexicographically greater than. This ordering is defined as the difference between the characters at the first offset into both strings where characters are different. This means "aaaad" is less than "aaaba", since the first two different characters are "a" and "b", and "a" is less than "b". If one string is a prefix of another, the shorter string is determined to be less than the longer string. Listing 3 shows the operators in pseudocode format.

Listing 3 > and < algorithm

```
 \begin{array}{ll} \mbox{for } i:=0 \ \mbox{to } min(length(str1), length(str2) \ \mbox{do } if \ (str1[i]-str2[i])>0 \ \mbox{then} \\ & \mbox{return } str1>str2; \\ \mbox{else } if \ (str1[i]-str2[i])<0 \ \mbox{then} \\ & \mbox{return } str1<str2; \\ \mbox{end } if \\ \mbox{end } if \\ \mbox{end } for \\ & \mbox{if } (length(str1)>length(str2)) \ \mbox{then} \\ & \mbox{return } str1>str2; \\ \mbox{else } if \ (length(str1)<length(str2)) \ \mbox{then} \\ & \mbox{return } str1>str2; \\ \mbox{else } if \ (length(str1)<length(str2)) \ \mbox{then} \\ & \mbox{return } str1<str2; \\ \mbox{else } \\ & \mbox{return } str1<str2; \\ \mbox{else } \\ & \mbox{return } str1=str2; \\ \mbox{end } if \end{array}
```

#### 3.2.2 Representation of Forwarding Rules

In Listing 4, a set of forwarding rules is described in the predicate language. The interfaces are labelled as  $I_{1-5}$ , with their filter definitions separated by logical ORs ( $\lor$ ), made up of terms as (operator, pattern) tuples, separated by logical ANDs ( $\land$ ).

Listing 4 A set of ru	<b>Listing 4</b> A set of rules in the predicate language			
	$I_1 \leftarrow (\langle eq, "john" \rangle \land \langle eq, "smith" \rangle) \lor \langle eq, "sir" \rangle$			
	$I_2 \leftarrow (\langle eq, "smith" \rangle \land \langle eq, "mrs." \rangle) \lor \langle eq, "jane" \rangle$			
	$I_3 \leftarrow \langle eq, "mr." \rangle \lor \langle eq, "sir" \rangle$			
	$I_4 \leftarrow \langle ss, ``james" \rangle \lor (\langle pf, ``mary" \rangle \land \langle sf, ``bob" \rangle)$			
	$I_5 \leftarrow \langle lt, ``james" \rangle \lor \langle gt, ``mary" \rangle$			

This may be made clearer as a table of forwarding rules in the form used by SFF. A forwarding table for this set of rules is shown in Table 3.2.

Interface	Filter	Operator	Pattern
	f	eq	"john"
$I_1$	$J_{1,1}$	eq	" $smith$ "
	$f_{1,2}$	eq	"sir"
	fai	eq	"smith"
$I_2$	J2,1	eq	" $mrs$ ."
	$f_{2,2}$	eq	"jane"
L	$f_{3,1}$	eq	<i>"mr."</i>
13	$f_{3,2}$	eq	"sir"
I.	$f_{4,1}$	ss	"james"
14	f	pf	"mary"
	$J_{4,2}$	sf	"bob"
	$f_{5,1}$	lt	"james"
15	$f_{5,2}$	gt	"mary"

Table 3.2: Forwarding table

Here, it is clear that interface  $I_2$  is composed of a disjunction between two filters,  $f_{2,1}$ , which is a conjunction of  $\langle eq, "smith" \rangle \land \langle eq, "mrs." \rangle$  and  $f_{2,2}$ , which is described as  $\langle eq, "jane" \rangle$ .

# 3.3 Hardware Design Implementations

Several iterations of hardware design were tried to find one that could implement all the desired operators in our predicate language. Emphasis was also placed on a regular design that would be simple enough to be generated from a forwarding rule specification.

### 3.3.1 CAM Design

The initial approach was a simple CAM design of a set of forwarding rules. The eq, pf operators could easily be represented by a series of 8-bit comparators on each character of a pattern. The lt and gt operators were not considered at this stage as the mechanics of these operators are different to the other operators.

Characters from an input message tag would be shifted into a buffer the length of the largest pattern in the forwarding rule set plus one in bytes. If the tag was shorter than the shift buffer, the additional length would be padded with zero bytes.

The eq operator was implemented as an AND logic gate over the series of comparators for the pattern it was operating on. To prevent false-positives from partial matches, the positions in the input shift buffer after the length of the pattern were compared against zero bytes, to ensure that the input tag had ended. Figure 3.2 illustrates how this was done.



Figure 3.2: Simple CAM comparator implementation

The pf operator was implemented in the same manner as the eq operator, however no characters in the input buffer past the length of the pattern were compared.

The results from these comparisons were saved in 1-bit registers that were only reset per message. This tracked the state of which terms had matched in a particular filter, which were represented as AND logic gates over their corresponding registers. Interfaces were represented as OR logic gates over their corresponding filters.

Because of the simplicity of the design, it was very regular. Additional interfaces could be added with extra OR gates over AND gate filters, with additional terms as extra 8-bit comparators.

However for larger or more diverse sets of rules, many comparators would need to be used. This is because each term used its own set of comparators, which could potentially increase the total area cost. The sf and ss operators weren't very well suited to this design, as comparisons only occurred once the whole tag was shifted into the buffer.

#### 3.3.2 DCAM Design

To solve the problem inherent to the simple CAM design, a decoded CAM (DCAM) design was pursued. This reduced the number of 8-bit comparators in the design to the number of distinct characters found in all patterns. Figure 3.3 shows the comparators and shift registers for the DCAM version of the simple CAM design in Figure 3.2.



Figure 3.3: DCAM comparators and shift registers

Each input character from a tag is shifted into a single register that forms the input to the multiple 8-bit character comparators. Results from the comparators are passed through 1-bit shift registers. The start and end of a message tag are indicated via separate 1-bit inputs into the design. Terms are matched using AND gates on the outputs of the comparator shift registers and on the tag start and tag end input wires.

Since comparisons occur while the input is shifted in, rather than once all the input has been shifted in, which was the case with the simple CAM design, the *ss* operator is much easier to implement with just a single AND logic gate.

The pf and sf operators are built on top of the ss operator output ANDed with, respectively, the tag start input and tag end input. The eq operator is implemented by an AND gate on the pf and sf outputs. Similar to the simple CAM design, the outputs of the operators are put into registers that are only reset every message. Figure 3.4 shows the four operators on the word "and.



Figure 3.4: DCAM ss, pf, sf and eq implementation

#### 3.3.3 Addition of the lt and gt Operators

The designs discussed previously have all omitted the lt and gt operators. This is because while the pf, ss, sf and eq operators just compare characters for equality, lt and gt require less-than and greater-than character comparisons respectively.

The lexicographical less-than and greater-than operators only take into account the first letter difference between two strings. This requires the state to be tracked throughout matching, which wasnt originally required for the earlier operators that were based purely on equality. With equality, either all characters are equal to the comparison string or they arent, which can be modelled with an AND logic gate and shift registers.

To overcome this problem, the greater-than and less-than state for a particular message tag must be tracked for either operator. The DCAM design is extended with extra character comparators for the less-than and greater-than character operations. If either the lt or gt operator is present for a particular pattern, for each distinct character in the pattern both less-than and greater-than character comparators are added to the design.

Registers are then introduced to track whether the current tag is found to be less than or greater than a search pattern. These introduce a delay of one clock cycle and allow the output of a register to be controlled by the output of the opposite register. The less-than tag register is set to high if any character in the incoming tag is found to be greater than (please note the operator definition in 3.2.1) the corresponding character in the search pattern and the greater-than tag register is not high. The converse occurs for the greater-than tag register. This ensures that only the first different character is used for comparison, as once a difference is found, the corresponding tag register blocks the opposite register from going high on subsequent differences. These registers are reset for every new tag in a message, but their results are saved per message in the same way the eq, ss, pf and sf results are saved.

To deal with the case where either the search pattern is a prefix of a message tag or a message tag is a prefix of the search pattern, the tag end flag was incorporated into the greater than operator logic. Should the tag end be encountered at an offset shorter than the search pattern offset, and the less-than tag register isnt high, the greater-than tag register is brought high. Similarly, if the tag end register isnt encountered by the end of the search pattern, the less-than tag register is brought high, on the same condition that the greater-than tag register isnt already high.

3. ACCELERATING MATCHING

Because the *lt* and *gt* operators result in the addition of both less-than and greater-than character comparators, the equals character comparator for an included character can be eliminated. Instead, the output of the equals comparator can be simulated using the less-than and greater-than character comparators with not gates at their output. This models the condition  $\neg(a < b) \land \neg(a > b) \leftrightarrow a = b$ .

# 3.4 C Implementation

So we could contrast the difference between a software implementation and a hardware implementation on equal ground, we developed filtering software in C alongside the Verilog designs.

The C implementation involves building a ternary search tree (TST) to represent all the search patterns in the rule set specified. The operators on each search pattern are then linked into the tree before the tree is flattened into a finite state machine (FSM) representation.

The FSM is transitioned through the use of goto statements. To match the ss and sf operators, the FSM is wrapped inside a loop, starting the search at every character of an input tag. After the first character, the pf, eq, lt and gt operators are ignored to prevent false positives occurring.

Filters are implemented as integers. Each term in a filter is assigned a bit offset into the filter integer, which is toggled should the term be satisfied. Once each tag of a message has been processed, interfaces are selected based on the filters that are true.

The forwarding function is inherently thread-safe, allowing it to be utilised by all the cores on a multi-core CPU independently. This is down to the following reasons:

- The memory for each filter is allocated on the stack by the forwarding function.
- The responsibility for the allocation of memory for the interface match vector is delegated to the thread using the forwarding function.
- No shared memory is required by the forwarding function to represent search strings as all the required data is serialised into the FSM code.

# 3.5 Automating Design Generation

To fully evaluate the capabilities of a hardware implementation of a forwarding rule specification, it is necessary to have many sets of Verilog designs and equivalent C implementations. This allowed the collection of enough data to compare the benefits and drawbacks of a hardware implementation over a software implementation.

However, creating Verilog designs can be time consuming and possibly error-prone, due to the low-level nature of the language and the inherent difficulty of debugging hardware. In order to address this problem, an automated method of generating designs from a set of forwarding rules was required.

The solution was a compiler that could take a set of rules as input and output both Verilog designs and equivalent designs in C. This section will cover the various aspects of this software.

#### 3.5.1 Compiler Overview

The compiler was designed to be modular, decoupling components into two main sections; the frontend and backend. A frontend parses rules and passes them in an intermediate representation to a backend to be converted into a usable output. Input can be taken from either the standard input stream or from the filesystem. Output can also either be directed to the standard output stream or to a file on the filesystem. This allows the compiler to be used interactively.



Figure 3.5: Overview of the compiler

The language chosen for the compiler was D, developed by Digtal Mars. The reasons behind the choice is because D is:

- A multi-paradigm language, so supports object-oriented, functional and imperative programming methods.
- A system programming language, designed for standalone applications.
- 3. ACCELERATING MATCHING

#### Reconfigurable Message Traffic Filters

- Compiled, offering good performance, but still platform-agnostic.
- High-level, providing advanced features such as automatic garbage collection, delegates and closure, allowing for concise intent-driven code.

The following subsections will discuss the key aspects of the compiler shown in Figure 3.5 in further detail.

#### Intermediate Representation

To pass forwarding rule specifications from frontend to backend, a suitable data structure was needed, both to ease the processing burden on backend modules and to reduce the memory footprint. To do this, forwarding rules are split into several sets:

- A map of all string patterns in the rules to all the operators that affect the patterns.
- A set of terms used throughout the forwarding rule specification.
- A set of filters used throughout the forwarding rule specification as collections of terms.
- A map of interface names to collections of filters that comprise the interface rules.

Each element in a set is identified by their properties. For example, a filter would be deemed as equal to another if their collections of terms were equal. This was a simple way of reducing the amount of potential code duplication that could occur in the backend modules should two identical filters be treated as separate.

#### Frontend

Two frontend modules were implemented. The first module can parse simple string forwarding rules (SSFR) described in 3.5.2 into a forwarding rule specification. The second module can parse a word list and generate a forwarding rule specification based on user input parameters. Parameters cover the number of interfaces, the number of filters per interface, the number of terms per filter and the word lengths used, as well as distinct filter, term and word counts.

#### Backend

Four backends were implemented to convert the forwarding rule specifications passed to them into the following outputs:

- C source code.
- Verilog source code.
- SSFR data.
- Test message data in comma-separated value (CSV) format.

The test message data frontend can generate test message data to target filters in the forwarding rule specification passed to it, or it can generate a set of random messages to check for false-positives.

Both the Verilog and C source code frontends can take a test message file generated previously and include test bed data in their output to validate the designs against a set of generated test cases.

**T** • • •

The SSFR backend was included to provide an output for the rule-generator frontend, however rules can be generated straight to source code if desired.

#### 3.5.2 Rule Format

- DMD

The simple string forwarding rule (SSFR) language was designed to be as close to the predicate language described in 3.2, but also to be simple to parse. The Backus-Naur Form (BNF) syntax of the rules are given in Listing 5.

Listing 5 BNF syntax o	ISFR
	$\langle rule \rangle ::= \langle ident \rangle$ ':' $\langle filter \rangle$ {' ' $\langle filter \rangle$ } ';'
4	$filter angle ::= \langle term angle \ \{`,` \ \langle term angle \}$
	$\langle term \rangle ::= \langle op \rangle \ \langle string \rangle$
	$(ident) ::= (`a``z`   `A``Z`   `0``9`   `_`)+$
4	$  string\rangle ::=    '  (/* all letters except    ' */) +   ' '$

The forwarding rules presented as a set of predicate rules in Listing 4 in 3.2.2 can be represented in SSFR as shown in Listing 6. There is no need to include any brackets in SSFR since rules can only be disjunctions of filters and filters can only be conjunctions of terms.

Listing (	3 SSFR	representation	of Listing 4
-----------	--------	----------------	--------------

( aapp

```
i1 : eq"john", eq"smith" | eq"sir";
i2 : eq"smith", eq"mrs." | eq"jane";
i3 : eq"mr." | eq"sir";
i4 : ss"james" | pf"mary", sf"bob";
i5 : lt"james" | gt"mary";
```

#### 3.5.3 Validation of Compiler Output

One of the most important processes in any implementation of a compiler is to ensure that the output of the compiler is correct with regards to the input. This was achieved initially with the introduction of static test harnesses that imported the output code of small hand-written rule sets. Each filter in a rule was tested on a case-by-case basis, with hand-written sample messages.

While this approach was acceptable for small examples, it was impractical as rule set sizes increased. In particular, it could not evaluate whether larger rule sizes could cause problems in the generated code due to possible interactions between the portions of generated code for each term.

The design of the automated compiler presented a solution to the problem. By leveraging the automatic code generation facilities provided by the compiler, test cases can be developed by a specialised backend. The intermediate representation of rules sets are parsed into message data targeted towards specific filters.

The message data is then included in the code generation processes of the C and Verilog source code frontends. The frontends evaluate each test message and produce a set of test cases with expected results. These test cases are then packaged into either a main function for the C implementation, or a separate test bench module for the Verilog implementation, and included with the forwarding logic.

This automatic test case generation, coupled with the forwarding rule generator frontend detailed in 3.5.1, allowed multiple tests to be run with many diverse sets of test cases to ensure the validity of both the C and Verilog forwarding logic.

#### 3.5.4 The Users Perspective

Care was given to providing an easy-to-use user interface. The user is expected to provide input as program arguments when invoking the compiler. The types of options required depend on the frontend and backend used, but the basic syntax is:

#### [./]compiler[.exe] <input file> <output file> [OPTIONS]

The <input file> and <output file> arguments are required to be in the form of a filename, with a valid extension supported by the compiler, or in the form of -<extension>. If the latter form is presented in place of <input file>, then the standard input stream will be read, with the frontend that handles <extension> file types parsing the input. If -<extension> is used for the output file, then output will be written to the standard out stream, in the format selected by <extension>. Valid file extensions are listed in Table 3.3.

Extension	Input/Ouput	Туре
с	Output	C source code.
data	Output	Test message data in CSV format, one message per line.
dict	Input	Dictionary word list, one word per line.
$\operatorname{ssfr}$	Both	SSFR rules.
v	Output	Verilog source code.

Table 3.3: Options for the Verilog and C source code backends

The following invokes the compiler to process the SSFR file rules.ssfr as input and to output Verilog source code to the standard out stream.

#### compiler rules.ssfr -v

When generating source code, an instance name is required to prefix function or variable names in the code. This is usually inferred by the <output file>, but when outputting to the standard out stream, such as in the previous case, an instance name must be provided manually as an option. Table 3.4 lists the possible options for the C and Verilog code generator backends.

Flag	Req.	Args	Description
-iname	YES*	1 arg	Instance name to use (* may be be inferred).
-tb	NO	1 arg	Generate test bed code using file.
-nh	NO	0 args	Do not generate header file (C only).

Table 3.4: Options for the Verilog and C source code backends

3. ACCELERATING MATCHING

As mentioned previously in section 3.5.1, the compiler comes with a forwarding rule generator frontend. This reads a word list as its input and generates a set of forwarding rules, based on the parameters the user provides. The available options are specified in Table 3.5.

Flag	Req.	Args	Description
-fc	NO	1 arg	Maximum unique filter count (default: no bound).
-fpr	YES	2 args	Minimum/maximum filters per rule/interface.
-op	YES	1 arg	Use given operator in SSFR format (may be used multiple times).
-rc	YES	1 arg	Rule/interface count.
-rgs	NO	1 arg	Random seed to use for rule generation.
-tc	NO	1 arg	Maximum unique term count (default: max unique words).
-tpf	YES	2 args	Minimum/maximum terms per filter.
-wc	NO	1 arg	Maximum unique word count (default: number of words available).
-wl	YES	2 args	Minimum/maximum word length.

Table 3.5: Options for the dictionary rule generator frontend

#### 3.5.5 The Developers Perspective

The extensibility of the compiler is one of the major features of the software. Custom input or output languages, formats or generators can be added easily, so long as they are based around the concept of forwarding rules. While the compiler software is documented with in-line comments, to further ease the process of understanding, we will cover portions of the software in more detail.

#### Specifying User Input

The compiler framework was structured to be data-driven. Each frontend or backend can be configured via the command-line by defining a set of possible configuration options, which may be empty. These cover:

- Whether or not the configuration option is required.
- The number of arguments required for the option.
- How many times the option can be specified on the command line.
- A user-friendly description of the option.
- A delegate to handle the option if it is encountered. This can exploit closure to set instancevariables within the frontend or backend.

A frontend or backend must also provide the file type they operate on, which allows for automatic selection of the frontend or backend based on the file types encountered by the framework. The compiler framework takes the option specifications of the selected frontend and backend and processes command-line options using these specifications automatically, alerting the user to any problems found.

#### Internal Structure

Listing 7 describes the elements used in the intermediate data structure to represent forwarding rules. The Filter and Rule data structures are parameterised templates that contain a arrays

of elements they are concerned with. The Filter structure contains pointers to all the Term objects comprising the forwarding filter, whereas the Rule structure contains all the filters that comprise the filtering rule.

Listing 7 Intermediate data structure members				
<pre>string -&gt; Operators</pre>	: patterns			
HashSet <term></term>	: terms			
HashSet <filter></filter>	: filters			
HashSet <rules></rules>	: rules			
string -> Rule	: interfaces			

One aspect that may raise questions is the separation between Rules and interfaces. As mentioned in section 3.5.1, this serves a dual purpose. The use of HashSets to hold the interface rule definitions means any duplicate interfaces are not replicated in memory. The interfaces that do share the same rule will point to the same interface in this intermediate representation structure. This approach is taken with the filters in the forwarding rule set. Only one copy of each distinct feature, no matter which interface(s) they belong to, will be contained in memory.

The HashSet structure is a template type that provides methods to add, remove and check for the occurrence of an element. HashSet elements are checked for equality through the use of an overridable boolean method, which in the case of Term, Filter and Rule structures, is their semantic forwarding rule definitions.

Through the use of HashSet structures to save memory, the compiler can be fairly scalable for large rule sets. It has been tested with rule files exceeding 230,000 lines or 30MB with no problems occurring, apart from needing to be terminated early as over 20GB of C source code was produced. Compilation of a 25MB C file with the popular compilation suite, the GNU Compiler Collection (GCC), caused an out-of-memory error.

#### Extending the Compiler

To implement a new frontend, the Parser interface must be inherited. The main method from this interface that must be overridden is the parse(Scanner s) method. This takes an input scanner that can read from multiple sources, but most importantly implements the readLine() method to obtain input line-by-line.

The only requirement for the frontend implementation is that it either returns a ForwardingTable, which is the intermediate data structure, or throw a ParserException if anything goes wrong. This allows for much flexibility in terms of how the parser operates, or what the input format is, whilst still providing a concise and safe API to use.

On the other side, a backend may be implemented in one of two ways. The first is to inherit the Backend interface and implement the generate(ForwardingTable t, OutputTarget output) method and auxiliary configuration methods. The second is to inherit the CodeGenerator abstract class, which implements the Backend interface, but adds utility methods to read test bed data and provides default configuration options.

The second method simplifies the work required to implement a code-generating backend, but for data-generation purposes, for example, the plain Backend interface should suffice. Again, like implementing a frontend, the API has been designed to be concise, but easy to use. Output is controlled by calling open() on the OutputTarget passed to the generate method, which returns a Writer that performs automatic indentation and allows output to be written to various sources.

As an addition to being able to input and output to files and the standard input or output stream, additional Scanner and OutputTarget types allow input or output to text buffers. If the framework was modified, or if a new framework was written, different stages could be pipelined through the use of text buffers. This is a prime example of the utility of the overall codebase.

# 4 Evaluation

To benchmark the performance of the hardware designs we covered in the previous chapter, we set up a series of test cases. We leveraged the rule and test data generation facilities of the compiler to generate equivalent C and Verilog implementations for increasing numbers of interfaces over different sets of filters per interface and terms per filter. The test setup is described in section 4.1.

The C source code was compiled into executable format and run to obtain readings, but the Verilog code was timed by specialised FPGA software provided by Xilinx. Per-message and percharacter timings were obtained. The factor of speedup was calculated, and auxiliary data such as executable size and power requirements was collected. The analysis and graphs of this data are presented in section 4.2.

# 4.1 Test Setup

To benchmark both the software and hardware implementations, a series of test cases were set up. Test cases were generated automatically using the compiler described in 3.5. The parameters used with the generator are described in Table 4.1.

Name	Description	Range	Range Step
I	The number of interfaces.	5-100	5
FPI	The number of filters per interface.	5-20	5
TPF	The number of terms per filter.	5-20	5
WL	The length of words used in term patterns.	5	N/A

Table 4.1: Test case constraints

The total number of cases is the power set of all the values within *range* for *range step* of (I  $\times$  FPI  $\times$  TPF  $\times$  WL). The number of test messages for each test case is the product of (I  $\times$  FPI), as each message is targeted to each filter in the set of rules.

#### 4.1.1 C Implementation

The C source code was compiled with the  $x86\_amd64$  version of the Microsoft Visual C/C++ compiler and linker (cl.exe) from the Microsoft Visual Studio 2010 (version 10.0) Ultimate development suite. The default optimisation level was used, as increasing the optimisation level dramatically increased compile time for the larger source code sizes.

The executables were run from a bash 2.03 for Windows session with all non-essential applications and power saving features disabled on the following machine setup:

- Asus U35JC Notebook.
- Microsoft Windows 7 Professional 64bit SP1 (Version 6.1.7601).
- Dual Core Intel(R) Core(TM) i3 CPU M 370 with Hyperthreading.
  - 2.40 GHz clock speed.
  - 2 x 256 KB L2 cache.
  - 3 MB L3 cache.
- 4.00 GB 1066 MHz DDR3 SDRAM.
- 2.5" Seagate Momentus 5400 RPM 320 GB hard drive.

Timing was performed using the QueryPerformanceCounter and QueryPerformanceFrequency Win32 API calls for high-resolution timing. The granularity of the timings for this particular case was to the closest  $\frac{1}{2408271}$  of a second. The timing was performed over one thousand runs on each set of messages. For each test message, the forwarding function was called, followed by I × if statements to check the result of each interface.

4. EVALUATION

#### 4.1.2 Verilog Implementation

The Xilinx Virtex6 xc6vlx75t-ff484-3 FPGA was used as a target for all compilation and routing of the source code. The Verilog source code was compiled, mapped, placed and routed using the Xilinx 64bit ISE Design Tools 13.1 suite on Microsoft Windows 7 Professional 64bit. The steps followed to obtain a fully placed and routed design for timing were:

- xst with the -opt\_mode Speed -opt\_level 1 flags passed to the run command.
- ngdbuild with timing constraints of 12ns HIGH 50%.
- map with options -ol std -t 1 -xt 0 -register\_duplication off -r 4 -global\_opt off -mt 4 -detail -ir off -pr off -lc off -power off.
- par with options -mt 4 -ol std.

Minimum clock period timings and resource utilisation counts were then obtained from the final report given by the **par** command.

In addition to placing and routing, power estimates were obtained by using the Xilinx XPower command-line tool, xpwr, with the flag -ol std.

# 4.2 Results

#### 4.2.1 Per-character Timings

The C experimental readings were based on one thousand runs of multiple messages each. To obtain the per-character timings, the readings were first divided by one thousand, and then divided by the number of total characters in the set of messages. The result was then adjusted from millisecond units to nanosecond units to be on par with the Verilog readings.

The top four graphs in Figure 4.1 give the per-character times of the C implementation. Contrasting the C timings with the Verilog timings in the bottom four graphis in Figure 4.1, one obvious feature to note is the different scales used. While the C timings range from 75.2ns to 421ns, the Verilog timings range from 2.21ns to 12.3ns.

The huge difference in the times can be mainly attributed down to the properties of the DCAM architecture used in the Verilog implementation. The C implementation reduced the number of possible character comparisons through the use of trie structures, but due to its single-threaded nature, only one character comparison can be performed at a time. In addition to the multiple character comparisons, the C implementation compares each tag in a message multiple times starting at each character of a tag to match the sf and ss operators. In contrast, the DCAM architecture uses multiple character comparators in parallel. This results in multiple character comparisons per clock cycle. As a consequence, each character in an input message is only required to be processed once, as shift registers deal with offsets of characters in a tag.

#### 4.2.2 Per-message Timings

For the C implementation, there are fixed costs involved with calling functions and initialising variables. While the unit of processing for the Verilog forwarding function is an input character, the unit of processing for the C forwarding function is a whole message. This means for every message, these fixed costs are applied to the C implementations execution time, so there is a need to provide timings per message.

To adjust the Verilog per-character cycle timings to per-message timings, each cycle time was multiplied by the number of characters in a potential test message with the FPI and TPF constraints specified. The top four graphs in Figure 4.2 give the per-message times for the C implementation, which can be compared with the bottom four graphs of the Verilog per-message times.

The Verilog per-message timings start off quite constant, before increasing after around 50 interfaces. This could be due to the placer choosing less-than optimal designs to reduce placement time in large search spaces. This would explain why the results for the lower FPI and TPF counts are flatter and have less variation. Further testing over larger data sets would be required to confirm this. The placer would also need to be set to high effort level when selecting routings.

The C per-message timings appear to be more or less linear, with equal spacing between the TPF tiers. There are several spikes, most notably where FPI is 15 and interface count is 35. These spikes could be caused by the selection of terms used. Again, the linearity would need to be confirmed by using larger rule sets.

4. EVALUATION



Figure 4.1: Per-character times with C above the line and Verilog below

4. EVALUATION

Reconfigurable Message Traffic Filters



Figure 4.2: Per-message times with C above the line and Verilog below

4. EVALUATION

Reconfigurable Message Traffic Filters

#### 4.2.3 Speedup

The graphs in Figure 4.3 describe the speedups from the C implementations to the Verilog implementations. The speedup was calculated using the per-character times for the Verilog and C benchmarks. The average speed up across all the ranges is 37.7 times, but higher numbers of filters yield more speedup.

The data becomes slightly erratic past 50 interfaces. This is possibly due to the Verilog timings becoming inconsistent as the design sizes increase. Spikes can also be caused by the choice of terms used in filters, due to the slight differences in the hardware designs compared to software.

Overall, the hardware implementation is estimated to be much faster than the software implementation. Though the C source code was not compiled with high optimisations due to time constraints, it would be unlikely to reach the speed the hardware designs would run at.



Figure 4.3: Speedup per character from C to Verilog

#### 4. EVALUATION

# 4.3 Resource Utilisation

#### 4.3.1 Power Requirements

The estimated power requirements for the Verilog designs are shown in Figure 4.4. The power requirements vary from around 1350mW to over 1800mW.



Figure 4.4: Average total (quiescent and dynamic) power consumption against the number of interfaces

Power requirements increase steadily until past about fifty interfaces, before starting to fluctuate. This is similar to the cycle times recorded in Figures 4.1 and 4.2. As mentioned in section 4.2.3, the placer may not have chosen the best routings for larger rule sets due to the larger problem space it has to search. These effects are further exacerbated due to the placer searching for routings optimised more for speed than power consumption.

One interesting point to note is that, though the software implementations ran on a low-power notebook PC, the idle power requirements were around 20W, or 20,000mW, over ten times the power requirements of a Virtex6 FPGA. If we assume an average speedup of 37.7 times from software to hardware, the hardware implementation is over 370 times more efficient than a software implementation on a per-unit-of-power basis.

#### 4.3.2 Executable Size

The size of the executables increases exponentially as the number of interfaces increase. It is clear from the graph in Figure 4.5 that with larger numbers of interfaces in the thousands, executable file sizes will be extremely large. This is primarily due to the forwarding rules being compiled into static code, which means an increase in data results in an increase of code and therefore executable machine instructions.



Figure 4.5: Average size of executables against the number of interfaces

The increased size can possibly be reduced by optimisation in the C code generation step by reducing the FSM representation of the search pattern TST. This would reduce the number of nodes that would need to be outputted as source code, indirectly reducing the executable size. The C compiler that parses the code could then be optimised for size, possibly reducing the size of the executable further.

#### 4.3.3 FPGA Utilisation

Figure 4.6 show the percentage of slice registers and look-up tables (LUTs) used on the Xilinx Virtex6 FPGA for the different rule sets. The results are percentages based on the following measures:

- Slice registers used out of 93,120 available.
- Slice LUTs used out of 46,560 available.

One important aspect of the data is the relatively low utilisation of both slice registers versus slice LUTs. The reason for this would be the number of logic functions used in the filters. These are translated into LUTs when synthesised, whereas registers are used as flip-flops and latches. A possible method to lower LUT utilisation could be to analyse shared sets of terms between filters and share the logic for those sets between the parent filters.

Another interesting point to note in this data is that for lower FPIs, the utilisation is roughly linear, but as both FPI and TPF increase, the plot becomes more logarithmic. The reason for this is possibly down to the DCAM architecture of the hardware designs. Increasing the number of terms increases the chance that character-comparison comparators are shared. This means LUTs will be more likely to be shared in this architecture. The logarithmic curve of the slice registers utilised could be explained by the sharing of shift registers on the outputs of comparators. The more search patterns are introduced, the greater the likelihood that letters will be found at the same offset or less than other offsets in different search patterns.

4. EVALUATION





Figure 4.6: FPGA slice register and look-up table (LUT) utilisation

4. EVALUATION

Reconfigurable Message Traffic Filters

## 4.4 Summary

The evaluations of both the Verilog and C implementations show that the Verilog implementations are much faster than the C implementations by a factor over 37 times. The estimated power requirements also showed that the Verilog implementations consume much less power than a traditional notebook PC. The findings highlight the potentially massive gains in both performance and power-efficiency directed FPGA solutions have over conventional software solutions.

One point to note, however, is that the Verilog results are estimates by the Xilinx FPGA tools, rather than experimental run-times recorded. We are confident that the timings will be accurate, as unlike a traditional software set up, there are no additional processes to consider, which would normally be found on a multi-tasking operating system.

It would nevertheless be preferable to validate the Verilog results on a physical FPGA in future work, particularly in terms of power requirements, as environment can play a huge role in overall results. Due to time constraints, larger rule sets could not be covered, but this could be an area of future research to further expand the breadth of the data.

# $\mathbf{5}$

# **Conclusions and Future Work**

From the results it is clear that a hardware implementation on an FPGA is much faster than an equivalent software implementation. The estimated average speed of a hardware solution running in a Xilinx Virtex6 FPGA is around 37 times faster than an equivalent solution running on a 2.40GHz dual-core Intel Core i3. The power efficiency statistics are even more exciting, as with the higher performance and lower power consumption, the Xilinx Virtex6 is estimated to be around 370 times more power efficient. However, one important part that was not covered was the time taken to produce a working implementation from a set of rules. While the time taken to compile a set of rules into either C or Verilog code was minimal, the time taken to compile the C or to place and route the Verilog code increased substantially for larger problem sizes.

The main cause of the long compilation and routing times is down to how the source code produced is a static schematic of the forwarding logic. The larger the set of rules, the larger the overall source code will be. For the C code, the addition of more filters requires the addition of more stack variables to hold the state of matching on those filters. The addition of more matching terms requires additional states in the search pattern TST FSM. This requires more time for a C compiler to parse, analyse and optimise the code and produce valid machine code. The addition of more stack variables increases the likelihood that the stack size allocated to the executable wont be large enough. Stack overflows occurred for the larger designs involving greater than thirty thousand terms, or over two thousand filters, due to the large numbers of filter stack variables used.

On the Verilog side, increases in rule sizes translate to increases in logic elements used. This results in highly dense and difficult-to-route designs, which require extra time for placer software to process. Large numbers of logic elements may also result in much higher timings, due to the delays introduced by routing signals between logic elements. Routing some of the larger designs occasionally took many hours for this reason.

Because of this drawback, both the concept of static hardware designs and the compiler software are suited more towards smaller rule sets, or rule sets that are rarely updated. This presents two viable avenues for future work. The first would be to investigate a more generalised solution. The second would be to explore the potential speedups that can be obtained from using a static design process as an extension to an existing solution such as Siena Fast Forwarding. These alternatives will be discussed in the following sections of this chapter.

Continuing on with the same design, there are optimisations that can possibly further increase the overall speed of the hardware designs. One such optimisation is pipelining, where delay registers are inserted between logic elements to break up long paths. This can potentially increase clock speed by reducing the amount of routing delay in the design, thereby increasing the potential overall speed of the design.

The C designs outputted by the compiler may be optimised further by exploring techniques to reduce the FSM produced by the TST. Filter variables could also be moved off the stack into a structure that is passed to the forwarding function as a parameter by the user. This would allow the user to choose whether the temporary filter variables are allocated on the stack or in the heap. Situations where this may be beneficial would be where there are many filters, requiring a large chunk of stack space that may not be provided by the runtime environment. This enhances the scalability of the system and provides the user finer-grained control over the operation of the forwarding logic.

# 5.1 Applications and Future Work

In this section we will two main possible applications and future developments of this project. While it would have been desirable to have investigated these paths in this project, there was insufficient time. Instead, this project hopes to have laid the groundwork by highlighting the benefits of FPGA designs over software designs, and producing tools to assist with future development of the ideas laid out here.

#### 5.1.1 Generalised Design

This project focussed on producing static designs from rule sets. The cost of this method is scalability and flexibility; however the results showed that FPGAs can achieve massive speedups over software. A potential course that could be taken would be to diverge from static designs on to dynamic designs.

A general design that would take configuration data to determine its operation can be written by hand in Verilog. This could then be mapped, placed and routed to an FPGA chip with maximum optimisations selected, as the design would not need to be changed once the routing is complete. The compiler written for this project could then be adapted with a new backend to produce configuration data for this hardware skeleton from a set of rules.

Whenever a rule set needs to be changed, the time taken from receipt of the new rules to a hardware implementation of those rules would be greatly minimised from a static deployment. Rules would just need to be translated to configuration data, which would then be sent to the hardware, replacing the old configuration data.

A further expansion onto this would utilise the processes of self-reconfiguration explained in 2.4. This would make the most of the reconfigurability of FPGA hardware as the hardware would truly be self-reconfiguring instead of merely reading a new set of configuration data as instructions like a desktop CPU.

This approach could be combined with the technique detailed in the next section, just-in-time compilation.

## 5.1.2 As a Just-in-Time (JIT) Extension

Substantial performance increases can be obtained by synthesising forwarding logic into hardware. It was pointed out earlier that larger rule sets can be a problem with regards to the time taken to synthesise a design. On its own, the designs do not take into account any concept of key/value tags, so attributes that are a common feature of messages cannot be dealt with in a conventional way. This means as a standalone system, a static design, such as one produced by the rule compiler, may not be the best idea.

One avenue that could be explored would be to integrate both the compiler and the static design process with a larger system such as Siena Fast Forwarding. Potential speed ups can be obtained, as the static nature of both the software and hardware output of the compiler reduces the memory lookups attributed with data-driven implementations such as SFF. However, since the designs outputted by the compiler do not account for attribute keys, this process can be offloaded onto the SFF implementation.

One of the key features of SFF is the minimisation of the filtering workload through pre-processing message attributes and discarding the filtering constraints that do not apply to the current message. If one were to harness this feature and record which attributes are commonly selected for filtering, the most commonly selected attributes may be identified.

With this list of commonly selected attributes, the search patterns and operators for these values could be passed to the compiler which could produce either an optimised static C representation, or a Verilog representation. This output could then either be compiled into software or synthesised into hardware and used in place of the SFF filtering.

This is similar to Just-In-Time (JIT) compilation used by the Java Virtual Machine (JVM), where *directly interpretable representation* (DIR) code is translated into *directly executable representation* (DER) code. The Java HotSpot Server VM uses a mixed-mode JIT strategy that involved compiling only the most-used portions of byte-code into native code, while still interpreting byte-code until it has completed [1].

With the SFF/static design software, a mixed-mode strategy could be pursued, where rules are still processed by the default SFF forwarding logic until a hardware implementation is loaded onto an FPGA, or a static software forwarding function is linked into the main executable. Due to the modular nature of the compiler, it can be easily extended to a runtime compiler. Instead of producing C output for software designs, new backends could be implemented to output machine code specific to the machine architecture the SFF router is running on. This could bypass the steps normally required and reduce the dependency on external compilers.

Since the constraints selected for JIT would be the forwarding rule "hotspots", it is highly likely that they would be the weakest link in terms of forwarding speed for the SFF algorithm or any other this technique is applied to. By targeting these key areas, the overall throughput for such an algorithm could be increased, enhancing the scalability of an implementation.

In this capacity, there could be infinitely many possible applications for this work. Good examples would be where there is a high volume of message traffic, such as in the financial industry for routing market data, or e-commerce datacentres.

Reconfigurable Message Traffic Filters

45

# Bibliography

- ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P. F. A survey of adaptive optimization in virtual machines. In *PROCEEDINGS OF THE IEEE*, 93(2), 2005. SPECIAL ISSUE ON PROGRAM GENERATION, OPTIMIZATION, AND ADAP-TATION (2004).
- [2] BENTLEY, J. L., AND SEDGEWICK, R. Fast algorithms for sorting and searching strings, 1997.
- [3] BROWN, S., AND ROSE, J. FPGA and CPLD architectures: a tutorial. Design Test of Computers, IEEE 13, 2 (1996), 42 –57.
- [4] BU, L., AND CHANDY, J. A. FPGA based network intrusion detection using content addressable memories. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 316–317.
- [5] CAI, Z., WANG, Z., AND ZHENG, K. A distributed TCAM coprocessor architecture for integrated policy filtering and content filtering. In *Communications (ICC)*, 2010 IEEE International Conference on (May 2010), pp. 1–5.
- [6] CARZANIGA, A., AND WOLF, A. L. Content-based networking: A new communication infrastructure. In *Revised Papers from the NSF Workshop on Developing an Infrastructure* for Mobile and Wireless Systems (London, UK, 2002), IMWS '01, Springer-Verlag, pp. 59– 68.
- [7] CARZANIGA, A., AND WOLF, A. L. Forwarding in a content-based network. In IN SIG-COMM (2003), pp. 163–174.
- [8] CHAND, R., AND FELBER, P. A scalable protocol for content-based routing in overlay networks. In Network Computing and Applications, 2003. NCA 2003. Second IEEE International Symposium on (2003), pp. 123 – 130.
- [9] CHANG, Y.-K., TSAI, M.-L., AND CHUNG, Y.-R. Multi-character processor array for pattern matching in network intrusion detection system. In Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on (2008), pp. 991 -996.
- [10] COMPTON, K., AND HAUCK, S. Reconfigurable computing: a survey of systems and software. ACM Comput. Surv. 34 (June 2002), 171–210.

BIBLIOGRAPHY

- [11] DHANAPRIYA, M., AND VASANTHANAYAKI, C. Hardware based pattern matching technique for packet inspection of high speed network. In *Control, Automation, Communication and Energy Conservation, 2009. INCACEC 2009. 2009 International Conference on* (2009), pp. 1–6.
- [12] FISK, M., AND VARGHESE, G. Fast content-based packet handling for intrusion detection. Tech. rep., La Jolla, CA, USA, 2001.
- [13] GUNTHER, B., MILNE, G., AND NARASIMHAN, L. Assessing document relevance with run-time reconfigurable machines. In FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on (Apr. 1996), pp. 10–17.
- [14] KNUTH, D. E., JAMES H. MORRIS, J., AND PRATT, V. R. Fast pattern matching in strings. SIAM Journal on Computing 6, 2 (1977), 323–350.
- [15] MARGARA, A., AND CUGOLA, G. High performance content-based matching using GPUs.
- [16] MCLAUGHLIN, K., O'CONNOR, N., AND SEZER, S. Exploring CAM design for network processing using FPGA technology. Advanced International Conference on Telecommunications / Internet and Web Applications and Services, International Conference on 0 (2006), 84.
- [17] PAGIAMTZIS, K., AND SHEIKHOLESLAMI, A. A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme. *Solid-State Circuits, IEEE Journal of* 39, 9 (2004), 1512 – 1519.
- [18] PAGIAMTZIS, K., AND SHEIKHOLESLAMI, A. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *Solid-State Circuits, IEEE Journal of 41*, 3 (2006), 712 – 727.
- [19] RAFLA, N., AND GAUBA, I. A reconfigurable pattern matching hardware implementation using on-chip RAM-based FSM. In *Circuits and Systems (MWSCAS)*, 2010 53rd IEEE International Midwest Symposium on (2010), pp. 49–52.
- [20] SIDHU, R., WADHWA, S., MEI, A., AND PRASANNA, V. K. A self-reconfigurable gate array architecture, 2000.
- [21] SIDHU, R. P. S., MEI, A., AND PRASANNA, V. K. String matching on multicontext FPGAs using self-reconfiguration. In ACM/SIGDA International Symposium on Field Programmable Gate Arrays (1999), pp. 217–226.
- [22] SOURDIS, I., AND PNEVMATIKATOS, D. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines* (2004), IEEE, pp. 258–267.

BIBLIOGRAPHY