INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

This work has been supported by the TeamIt project, which is co-financed by the European Social Fund and the government of the Czech Republic via grant ESF CZ.1.07/2.3.00/09.0067.

Building competitive research reams for IT

ANT@FIT research group:

Netbench: Framework for Evaluation of Packet Processing Algorithms

Technical report

March 10, 2012

Faculty of Information Technology, Brno University of Technology
Božetěchova 2, 612 66 Brno
tel.: +420 541 141 144, +420 541 212 219, fax: +420 541 141 170, 270
http://www.fit.vutbr.cz, http://teamit.fit.vutbr.cz

Netbench: Framework for Evaluation of Packet Processing Algorithms

ANT@FIT research group

March 10, 2012

Contents

Abstract

With the growing speed of computer networks, many algorithms and hardware architectures are proposed to increase processing speed of time-critical operations especially in the field of longest prefix matching, packet classification and regular expression matching. Despite many proposed algorithms and hardware architectures, there is still no free and easily extensible platform for evaluation and comparison with existing approaches.

We propose new framework for easy evaluation and experiments with packet processing algorithms. The aim of the framework is to serve as an independent platform for researchers seeking the easiest way to implement their algorithms, as well as a comparison of their algorithms with reference implementations of other approaches. The framework is provided as an open source and can be easily extended to support new algorithms or new comparison methodology. Netbench is publicly available at
`http://www.fit.vutbr.cz/netbench`.

1 Introduction

With the rapid increase of number of Internet users, improvements of the Internet infrastructure are constantly required. As the speed of network lines increases, there is a need for new algorithms and architectures for acceleration of time-critical operations used in network devices. In current networks, the IP lookup, packet classification and regular expression matching are highly important and ubiquitous operations.

The IP lookup or more generally the longest prefix match (LPM) operation is typically performed by routers, but it is also used by firewalls and other devices. It is used to select the most specific table entry (prefix) which matches a given input value. Usually the table is a routing table and the input value is a destination IP address, but the operation can be performed on any packet header field. In recent years, many hardware architectures were introduced to increase the processing speed [15, 21] or to reduce memory requirements [12, 30].

Packet classification is the task mainly used in firewalls. It matches packets to a set of rules, which are usually defined by values, ranges or prefixes of packet header fields. Generally, packet classification is a mathematical problem of multidimensional range search. Due to the rule set size and complexity of rules, it is very difficult to check all rules in constant time with low memory requirements which is needed to achieve throughput of nowadays networks. Many hardware approaches have been presented to cope with the trade off between memory and time requirements [11, 25, 33].

Regular expression (RE) matching is used mainly in IDS and IPS systems to detect malicious network traffic by a set of patterns, which are often described by strings or regular expressions. The presence of patterns is checked in a packet payload or TCP streams. As every byte in the packet payload has to be inspected, high computational power is required and many hardware architectures were introduced to increase the processing speed [7, 16, 27, 31].

All three listed operations (IP lookup, packet classification, RE matching) are often required to work at wire-speed. Therefore they may become the bottleneck, or may incur high costs of high-performance devices. Complexity of the IP lookup and packet classification rises with an adoption of IPv6 addressing scheme. All three operations are from the same domain and share many common ideas. Some existing packet classification approaches even use IP lookup as the first step.

There are many approaches to IP lookup, packet classification and RE matching. They vary in the technology used. The list of common technologies includes software implementation for general-purpose processors, network processors, implementations for graphics processing units (GPUs), FPGA and ASIC designs, TCAMs. While all of these technologies are rapidly innovated, *algorithms* for IP lookup, packet classification and RE matching are often independent on the technology, and may only benefit from technology improvements.

We argue that research of algorithms is highly needed to keep pace with the industry requirements. New algorithms from the discussed fields are often published in renowned proceedings and journals. In this situation, it may be surprising that there exists no common platform for researchers to evaluate and compare their algorithms. Moreover, access to real data sets (routing tables, firewall rule sets, patterns of malicious data) is often limited due to security and confidentiality issues. Some researchers have gained access to data sets, but are not allowed to distribute those data. The lack of standard data sets makes it hard to compare the properties of algorithms, especially their memory requirements. Implementations of published algorithms are not often available, so running own tests includes also reimplementation of previous approaches presented by other authors. Such reimplementation may be imperfect by having some differences to the original implementation, so it can not be perfectly trusted.

These facts contribute to lower quality of published research results. The Netbench Framework aims at addressing the stated issues by providing common platform for implementation and evaluation of packet processing algorithms, specifically IP lookup, packet classification and RE matching algorithms.

The rest of the paper is organized as follows: We extend our motivation and provide criticism of the current state of the field in section 2. The Netbench Framework is presented in detail in section 3. We provide several use cases in section 4 to illustrate the value that Netbench can provide to researchers. The paper is concluded in section 5.

2 Related Work

At the time of writing, there is no platform for early experiments with packet processing algorithms. The most related work to Netbench is the Click modular router [23]. It is a software implementation of a router with modular and configurable architecture. The Click router consists of a number of elements connected in a packet processing pipeline. Each element is a C++ class and implements single router function such as packet classification, queuing, and others.

Similarly to Click, NetFPGA [22] provides possibility to build a configurable router, but with hardware processing pipeline. This pipeline is assembled as well from simple processing elements which are written in Verilog. Each element performs a single packet processing task.

In the viewpoint of previous works, Netbench aims at experimenting with inner function of these elements – leaving aside issues related to performance and complexity of the whole pipeline. This renders it very simple to prototype and investigate new packet processing algorithms.

The lack of framework for early experiments is also related to the absence of common data sets used by all (or most of) researchers. Data sets for the IP lookup operation are most commonly the routing tables. There are some large public tables available, for example [1]. However, these data are not designed for experiments, and are changing over the time.

For packet classification, excellent work was done by Taylor and Turner in the ClassBench tool-set [34]. It contains the Filter Set Generator tool that generates pseudo-random rule sets with some predefined statistical properties. These properties may be gathered from real-life firewalls, so that generated rule sets have the similar structure and do not contain any confidential data. The random character of generated rule sets can become a disadvantage, because ClassBench does not provide exactly same data to all researchers. The second ClassBench tool is the Trace Generator that produces a sequence of packet headers to exercise packet classification algorithms with respect to a given rule set.

For RE matching, Snort [29] database is de facto standard for most researchers. Snort uses PCRE [14] to describe regular expressions. PCRE contain rich set of features that makes the task of writing new patterns easier. However, some of these features have more descriptive power than regular languages [9]. Due to these features, commonly used matching tools do not transform PCRE into the simple Finite Automata, but build more complex structures in the ad hoc manner even if given expression is regular.

According to our knowledge, there is no publicly available parser supporting conversion of all regular expressions described in PCRE into the Finite Automata. Therefore, researchers are forced to implement their own parsers which often support only a subset of the PCRE. Every parser implements different subset, which affect comparison of the methods. Even the smallest difference in the used parser can lead to the large difference between generated automata. For example, parser which ignores the case sensitivity switch may produce automata with only about half of necessary transitions. One of the commonly used parsers was implemented by Michela Becchi [4], but even this parser does not support all used regular features of the PCRE.

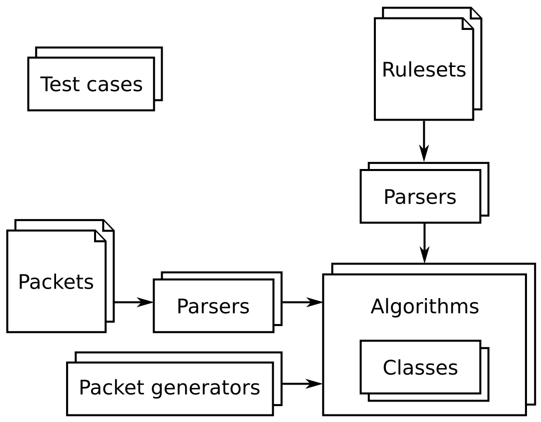

Figure 1: The Netbench Framework overall structure.

3 Netbench Framework

We present goals of the framework, together with explanation of how these goals are fulfilled. The Netbench Framework is designed with several objectives in mind:

- Serve as a uniform independent platform for researchers, without being tightly associated to one research group or algorithm.
- Enable rapid prototyping of algorithms and their software models.
- Simplify tasks which are done frequently in many experiments (e.g. loading classification rules from the input file etc.).
- Provide comprehensive data sets for IP lookup, packet classification and RE matching.
- Serve for education purposes.

We make the framework independent by making it publicly available under open-source license and by inviting researchers to submit their contributions.

We choose the Python 2.6 programming language to implement the algorithms, because of its popularity, sharp learning curve, and its feasibility for rapid prototyping of complex systems. Python is also considered to be the language with clear and easily readable syntax. If the performance of the Python interpreter is too slow for some particular computation, it is still possible to implement time-critical part of the algorithm in the C language. However, Netbench is intended to be used for rapid prototyping and experimental work, not for real deployment in the network.

To simplify common tasks such as loading data sets from files, Netbench provides set of classes in the form of library together with the documentation generated by the Sphinx [3] tool. There are also classes representing basic data structures, such as prefix, classification rule, regular expression, automaton, packet header etc.

Overall structure of the Netbench Framework is in Figure 1. Comprehensive selection of algorithms for IP lookup, packet classification and RE matching is already available in the Netbench Framework (see Sections 3.2, 3.3 and 3.4).

3.1 Data sets

We gather and add to Netbench several data sets. For IP lookup, we focus on three target data sets – IPv4 routing tables, IPv4 firewall tables and IPv6 routing tables. After further deployment of IPv6 to end user networks the IPv6 firewall tables will be added.

In case of LPM algorithms, most of the authors use BGP tables from Potaroo [1], but unfortunately the AS numbers, table release dates and numbers of currently available prefixes change in time. Therefore

Rule set name	Rules	Unique fields (Source IP/Destination IP /Protocol/Source Port /Destination Port)
fw1_05_05	822	138/123/1/22/52
fw1_m05_05	827	133/146/1/22/52
fw1_05_m05	882	156/171/1/22/52
fw1_m05_m05	852	121/143/1/22/50
fw2_05_05	956	735/274/1/13/1
fw2_m05_05	984	802/412/1/13/1
fw2_05_m05_100	100	75/37/1/12/1
fw2_05_m05_250	240	55/53/1/13/1
fw2_05_m05_500	455	65/57/1/13/1
fw2_05_m05	962	746/307/1/13/1
fw2_m05_m05	992	826/425/1/13/1
fw3_05_05_500	427	103/52/1/18/48
fw4_05_05_500	482	54/82/1/47/61
fw5_05_05_500	481	105/84/1/20/46
simple1	3	2/3/1/1/1
simple2	6	4/4/1/1/1

Table 1: Packet classification rule sets in Netbench

we download available IPv4 and IPv6 BGP tables and store them on regular time basis. As a result, any author can use the same data sets for comparison of introduced approach to other algorithms. Some more BGP table sources like [2] are also included.

As IP lookup is an important step of decomposition based classification methods, we use the same data sets for firewall tables as in the packet classification section. Both source and destination IP addresses are extracted from the rules and obtained sets form a benchmark for firewall IPv4 tables.

For packet classification, 14 rule sets generated by ClassBench with different settings are used. There are also two very simple rule sets (only 3 and 6 rules) for early experiments and debugging. Table 1 shows sizes and numbers of distinct conditions in each dimension of the rule sets.

Data sets for the pattern matching can be freely downloaded from the Internet [29] [6] [19]. Every tool has different format of the pattern language and supports different features. Sometimes, it is not clear from the documentation which features are supported. Snort's [29] pattern sets are used as a standard for evaluation of regular expression matching algorithms for IDS. However, Snort supports complete PCRE, which is extremely complex and has more descriptive power than regular expressions [9]. Due to the large complexity of the PCRE it is almost impossible to write a parser supporting all features of the PCRE that can be successfully transformed into regular expressions. However, every feature has its effect on the resulting automaton. Therefore, processing the same patterns with two different parsers may lead to different automata and ultimately to unfair comparison of algorithms.

To deal with this problem, Netbench contains its own parser which can be used with all implemented algorithms. The generated automaton is used as an input for all pattern matching algorithms in Netbench. If a new parser is implemented, it is possible to exchange it without disrupting the functionality.

Netbench also contains several sets of rules originating from Snort [29], Bro [6] and Netfilter [19] project. Table 2 shows the number of rule set modules, total number of regular expressions and the total number of states and transitions for Nondeterministic Finite Automata(NFA) with char classes for the 3 main rule sets groups.

3.2 Algorithms for IP Lookup

Algorithms for IP lookup are usually tree-based, because prefix directly corresponds to the descent path in binary tree. However, Netbench implements also approaches based on range searching or hashing.

Rule set name	Modules	Reg Exps	NFA states	NFA transitions
L7	30	30	791	949
Snort IDS	42	4 374	170 077	188 390
Bro IDS	1	1 400	64 493	77 110

Table 2: Data sets in Netbench for pattern matching

Unibit Trie [13] is a basic memory structure which includes all prefixes directly in its construction. Each node may contain prefix and up to two pointers to the child nodes. One input value bit is processed in each cycle during the Trie algorithm run. According to this bit the next node is the left (with bit $=$ 0) or right (with bit $= 1$) successor until there is no continuation possible. The longest prefix is the last matched node in the Trie. The main disadvantage of this algorithm is low matching speed given by the high number of processing steps. Therefore many following algorithms focus on speed (and memory) optimizations.

Controlled Prefix Expansion (CPE) [32] algorithm introduces multiple bits processing per clock cycle and therefore higher algorithm speed. Processing of more input bits is enabled by prefixes expansion to requested processing width, which leads to higher memory requirements. The leaf pushing optimization decreases the memory requirements typically to one half of the original memory structure.

Lulea [10] starts with a conceptual leaf pushed expanded trie and replaces all consecutive elements in a trie node that have the same value with a single value. This can greatly reduce the number of elements in a trie node. To allow trie indexing to take place even with the compressed nodes, a bitmap with 0's corresponding to the removed positions is associated with each compressed node.

LC-Tries (LCT) [24] are another form of compressed multibit tries. The main idea behind LC-Tries is to recursively pick strides (and therefore multibit tries) that result in having real prefixes at all of the leaves. Having fully populated leaves means that none of the internal prefixes need to be pushed to the leaves of the multibit tree or to lower trees (no leaf pushing). In addition to this strategy, LC-Tries use path compression to save space on non-branching paths.

Tree Bitmap (TBM) structure [12] is focused on hardware suitable implementation. Therefore it doesn't use LC-Tries scheme with variable processing steps, but it further optimizes memory usage of Lulea structure. Each compressed node of the structure represents 2^n unibit trie elements and stores only pointers to the first successor node and the first prefix. Next successors and prefixes follow in the memory, so their positions are given by the bitmaps.

Hash-Tree Bitmap algorithm (HTBM) [35] is focused on longer IPv6 addresses processing and reduction of number of Tree Bitmap steps. This is done by hash functions which serve as shortcuts in the trie memory structure. The resulting algorithm run consists of three stages: parallel multiple hash functions, longest matched prefix selection and consequent matching using the Tree Bitmap algorithm.

Shape Shifting Trie (SST) [30] starts with the Tree Bitmap structure and introduces memory optimization for long non-branched segments. Each node can represent the standard balanced tree or any other "shape" which is more suitable for given prefix set and allows to move faster and with better memory utilization through the prefix space.

Binary Search on Ranges (BSR) [20] treats a database of prefix addresses as a set of intervals. It expands all prefixes to full length and builds a sorted table of intervals limits where more specific intervals have higher priority then less specific ones. Binary search or B-Tree search is subsequently used to find the match for specific input IP address.

Binary Search on Prefix Lengths (BSP) [36] is based on the idea of modifying a prefix table to make it possible to use hashing. The hashing is used to search among all entries of a given prefix length. Instead of searching every possible prefix length and picking the longest prefix length with a match, binary search is used to reduce the number of searches.

MultiMatch (MM) [28] is trivial hardware algorithm which splits input prefixes to n subsets with the same length. Shorter prefixes are expanded to get requested length. After that, parallel hash searches are used to find the LPM in the group and the longest positive search is the operation result.

3.3 Packet Classification Algorithms

From the wide choice of packet classification algorithms, Netbench currently focuses on the family of decomposition methods. These algorithms divide the problem of classification into several independent parts. The first part is usually the longest prefix match operation, performed on all examined packet header fields. The advantage of such division is the strong potential for parallel computation.

Distributed Crossproducing of Field Labels (DCFL) [33] by Taylor and Turner modifies the LPM to return all valid prefixes (not only the longest one) for the given field value. What follows is the hierarchical structure of small crossproduct engines. Inputs of each engine are two sets of prefixes (or Labels, in general). Engine then performs set membership query for each possible pair of Labels. Result of the engine is another set of Labels. The result of the last engine is in fact a set of rules, from which the one with the highest priority is selected.

Multi Subset Crossproduct Algorithm (MSCA) [11] by Dharmapurikar et al. provides heuristics on how to break rule set into several subsets, eliminating the memory requirements significantly. The paper also identifies rules that generate excessive amount of memory. These rules are called *spoilers* and are treated in a separate algorithm branch to further reduce the memory.

Perfect Hashing Crossproduct Algorithm (PHCA) [25] improves MSCA by using specifically constructed hash function to map all possible results of the LPM stage directly onto the correct rule. Considerable amount of memory is saved this way. Perfect (collision-free) hash construction algorithm is used, but in this case, many collisions are intentionally introduced to create many-to-one mapping of LPM results to rules.

Prefix Coloring Classification Algorithm (PCCA) [26] further improves PHCA. The LPM stage is extended by adding abstract *color* property to prefixes. Prefixes now contain bitmaps of allowed and suppressed colors of prefixes from other dimensions. Simple logic is then used to filter out most of unwanted combinations of LPM results. This lowers the perfect hash table size significantly, while the throughput is not affected.

3.4 Regular Expression Matching Algorithms

Regular expressions are excellent tools for describing patterns. However, high succinctness of regular expression makes the matching process complicated operation. Therefore, finite automata are used to match input data against the pattern set. This section describes several finite automata based approaches implemented in Netbench.

Sindhu et al. in [27] were the first to map NFA into FPGA. Every state of finite automaton is implemented as one Flip Flop (FF) in FPGA. Transitions are realized by logic functions connecting registers.

Previous architecture has to implement character comparison for every transition in the automaton. However, many transitions in the automaton are labeled by the same character. Clark et al. [7] has improved Sidhu's architecture by adding shared decoder to reduce the amount of comparators and routing resources. Shared decoder transforms every character into one-bit signal and this signal is then used instead of the 8-bit character information.

In [8], Clark proposed usage of automaton which accepts multiple characters per transition by improving the shared decoder. Authors were able to achieve up to 100 Gb/s throughput on Virtex2 Pro FPGA for the strings. Main disadvantage of the solution was high amount of required resources.

Sourdis et al in [31] introduced new elements into NFA to increase efficiency of RE mapping. Constrained repetitions are one of the most common extensions of the regular expression in PCRE. Constrained repetition may generate hundreds or thousands states in NFA. Authors propose to implement constrained repetitions by combination of shift registers and counters to reduce resource utilization. Second improvements described by Sourdis is extraction of static pattern, such as strings. DCAM algorithm is used for string matching and transition in the automata can be described by one string.

Every described NFA based method requires direct implementation to the FPGA, which can take long time and requires special synthesis tool. Another disadvantage of NFA based method is the large state information.

Deterministic Finite Automata can store all transition table in the memory and achieve constant time complexity in processing of every character. The main disadvantage of DFA is the exponentially larger transition table in the worst case. Many algorithms were introduced to solve this problem. We describe only methods currently implemented in Netbench.

Kumar [18] noticed that many states in the DFA have very similar sets of outgoing transitions. Kumar reduces number of transitions to reduce memory requirements of the automaton by introducing transitions, that do not accept any character and connect states with similar outgoing transitions. This approach is called Delay DFA (DDFA) and is able to reduce size of the transition table by more than 90%. Since the default transition does not accept any symbol, the throughput of the DDFA may be slower than DFA but the time complexity of accepting one symbol remains constant.

The DDFA [18] reduces the number of transitions, but the number of states does not change. In [17] Kumar identifies one of the problems of DFA based methods as forgetting history of the matching. Indeed, DFA remembers only the active state and not the path it took. To solve this problem, Kumar introduces additional memory to store history of the matching. Content of this memory can be used as a condition on the transition of the automaton.

Many other improvements of the DFA were introduced in the literature to reduce memory requirements, however, DFA still have exponentially larger memory requirements than NFA. To solve this problem Becchi introduced Hybrid Finite Automaton [5]. Construction algorithm stops as soon as state blow-up is recognized. The remaining parts of the automaton are left as NFA or transformed into new DFAs. This method provides speed and memory consumption trade-off.

NFA Split algorithm [16] exploits the fact that only small number of states are concurrently active. It can compute which states can be concurrently active and use it for reduction. The states which can not be concurrently active are called states without collision. More than one set of collision free states can exist. The states without collision are implemented as DFA and the remaining states with collisions are implemented as NFA. This splitting operation results in decrease of ASIC/FPGA logic utilization to one half.

3.5 Modular pattern matching design

Regular expression matching is highly active area of research with many new approaches being presented. These new approaches often differ only in one part of the algorithm, while remaining parts are implemented according to previous papers. Netbench simplifies implementation of new approaches by allowing reusability of the already implemented algorithms.

Rapid prototyping in Netbench allow user to quickly evaluate new approaches and their combination with the previously implemented algorithms. For example, let's look at the splitting algorithms [5] or [16]. The contribution of these approaches is the algorithm for dividing one large NFA into several smaller NFA or DFA. The Netbench library allows user to experiment with different implementations of smaller automata to select the best one for the splitting algorithm.

Figure 2 shows the principle described in previous paragraph. The splitting algorithm is used to create several smaller automata in common format and to call another approaches to implement these small automata. Every of these approaches reports its results (such as resources consumption) through standardized interface back to the main splitting algorithm. Splitting algorithm sums the required resources and implements the same interface as all other methods. Therefore it is possible that even this splitting algorithm is a part of another, more complex regular expression matching approach. Algorithms can be set at runtime.

Many algorithms achieve high throughput by accepting several character in one step. Automata supporting this feature is called multistrided or multichar automata. Every algorithm implemented in the Netbench can be easily converted into multistrided form due to Netbench modular structure.

4 Use Cases

Netbench is an excellent tool for comparing various aspects of algorithms. It is designed to help researchers answer questions like:

- How much memory does the algorithm need?
- How (and why) does the memory size change for different input data set?

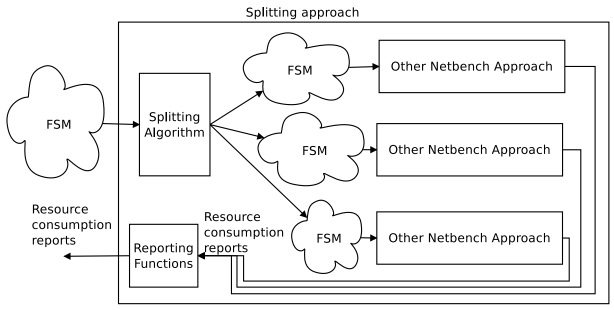

Figure 2: Modular structure of pattern match

- How should the algorithm parameters be set for the best performance on the given data set?

We provide examples how Netbench can be used in IP lookup, packet classification and RE matching experiments in following three sections.

4.1 IP Lookup

Each new LPM algorithm is typically evaluated by its author in terms of speed, scalability (IPv4, IPv6), update complexity and memory usage. While speed, scalability and update complexity can be measured independently (speed for example in clock cycles and number of memory accesses), the memory usage evaluation is always dependent on an input prefix set structure.

Therefore, Netbench constructs the complete algorithm's memory structure and follows the algorithm flow during the lookup operation. Using the unified input data sets, results are the precise number of nodes or table rows and the memory usage for the given algorithm. The user gets full insight in the algorithm memory structure and the lookup operation.

This allows the user to easily compare different algorithms and their suitability for a given task. The following tables and graph evaluate memory usages of selected algorithms for classification (firewalling) and routing. Table 3 compares memory usages for small classification rule sets, Table 4 compares memory usages for large IPv4 routing tables and Figure 3 gives comparison of memory usages for current IPv6 routing table.

Rule set	Trie	TBM3	SST32	MM8
fw1_05_m05	6 406	2 244	1 704	905
fw1_m05_m05	5 280	1 788	1 476	508
fw2_05_05	59 119	18 907	10 822	2 145
fw2_05_m05_100	1 538	615	657	427
fw2_m05_05	86 838	26 998	14 907	2 734
fw3_05_05_500	1 073	402	1 170	672
fw4_05_05_500	998	387	929	656
fw5_05_05_500	1 437	547	1 350	672

Table 3: Comparison of memory usage for small IPv4 classification rule sets (bytes)

Ruleset	CPE4	Lulea4	BSI	BSP
router_1009	23 630	17 130	14 630	16 070
router_10084	290 330	143 060	156 310	188 570
router_53781	1 719 490	632 620	860 490	814 940

Table 4: Comparison of memory usage for IPv4 routing tables (bytes)

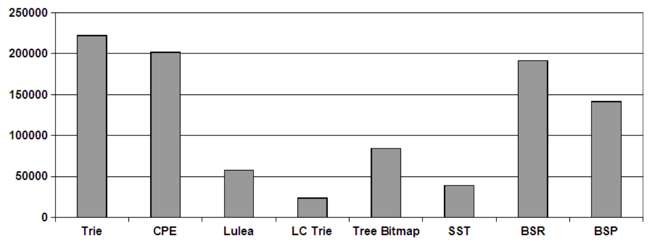

Figure 3: Comparison of memory usage for IPv6 routing table (bytes)

The Netbench is further suitable for algorithm's parameter settings evaluation. Many algorithms have parameters like number of bits processed in one step. These parameters have essential influence on both speed and memory usage. The rising processing speed typically causes memory requirements expansion. Detailed study of parameters is therefore important if the user looks for optimal settings of IPv4 or IPv6 match, or for the suitable hardware architecture with the given constraints. The example is given in Table 5 which presents memory expansion of Tree Bitmap algorithm with number of processed bits per clock cycle.

Ruleset	$n = 3$	$n = 4$	$n = 5$	$n = 8$
fw1_05_m05	2 244	2 487	3 524	13 728
fw1_m05_m05	1 788	2 184	2 860	12 451
fw2_05_05	18 907	21 094	26 159	109 858
fw2_05_m05_100	615	675	905	4 061
fw2_m05_05	26 998	30 271	36 325	153 125
fw3_05_05_500	402	489	683	2 560
fw4_05_05_500	387	449	597	2 297
fw5_05_05_500	547	662	907	3 544

Table 5: Comparison of memory required by Tree Bitmap algorithm with different strides (bytes)

4.2 Packet Classification

The most important properties of packet classification algorithms are (similarly to IP lookup) throughput and memory. The throughput is measured by number of packets classified per second, while the measure for memory consumption is often number of bytes per rule.

This is however somewhat vague because of different memory technologies. For example, it is of great importance whether the data structure can be stored in inexpensive and slow DRAM, or if more expensive SRAM is required. Special types of memories like TCAM or on-chip FPGA memory are also often utilized in packet classification algorithms. Netbench exposes all algorithm data structures to user, so that the memory requirements may be examined in detail.

The throughput highly depends on the utilized technology, working frequency, etc. Therefore, it is insufficient to provide a single measure describing number of processed packets per second. It rather makes more sense to count the number of memory accesses and computational steps. With this information, the algorithm can be better scheduled and partitioned to the particular technology.

We provide a table comparing memory requirements of all four algorithms in Table 6. Memory for the LPM operation is not acounted for in these experiments, because the LPM is common to all four algorithms. Instead, Tables 3 and 5 can be used to select the suitable LPM algorithm. For MSCA, PHCA and PCCA, eight spoilers are removed from the rule set. For PCCA, eight colors are used, and the extra memory that is required for LPM extension is included in the results.

The second use case for the packet classification is the measurement of advanced rule set properties. In [11], Dharmapurikar et al. introduced the idea of *pseudorules*. Pseudorules can be obtained by creating

Ruleset	DCFL	MSCA	PHCA	PCCA
fw2_05_m05_100	15	200	62	23
fw2_05_m05_250	15	1 026	265	169
fw2_05_m05_500	25	1 928	480	332
fw2_05_m05	85	44 846	63 108	14 590

Table 6: Comparison of memory required by packet classification algorithms (kbits)

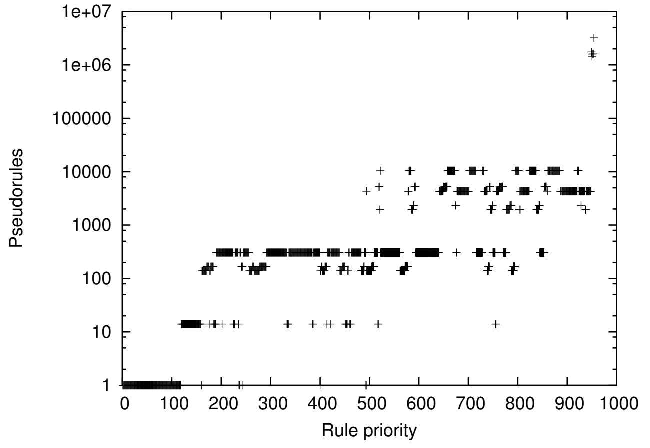

Figure 4: Distribution of pseudorules in the rule set.

a Cartesian product of all values found in the rule set. Each member of this Cartesian product is either a special case of some rule, or it does not match any rule. Only special cases of some rule are pseudorules.

This concept is fundamental for MSCA, PHCA and PCCA algorithms. Figure 4 shows the example of how many pseudorules are associated to each rule for one particular rule set (sorted by priority). Note the logarithmic scale of Y axis. The information about pseudorules distribution is crucial in understanding the difficulties of packet classification, and is also used to select spoilers to be removed from the rule set.

4.3 RE Matching

Evaluation of the pattern matching algorithms can be very tedious work. Often, it is next to impossible to obtain implementation of all approaches needed for the comparison. The straightforward solution would be to use results published together with the algorithm. However, the new algorithm is often implemented on new hardware and tested on new data sets which makes the comparison unfair. Netbench may help with the fair and correct comparison, since it contains data sets together with implementations. Therefore it is possible to generate results of the previous approaches on the current data sets and for up-to-date hardware.

Regular expression matching algorithms are often evaluated by the resource utilization and the throughput. To demonstrate ability of Netbench to compare implemented approaches, we compare several NFA based methods described in section 3.4. Table 7 contains Netbench estimation of the resource utilization of algorithms without using multistriding capability. The results are generated for thirteen data sets distributed with Netbench.

Table 8 presents resource utilization for multistriding implementation of Sidhu's and Clark's work. It is important to note that Clark presented multistriding approach without support of character classes. According to our experiments, automata without character classes cannot be built for modern data sets due to their prohibitive sizes. Therefore, we switch the alphabet decoder to support character classes. The Sidhu's approach did not use multistriding. To produce comparative results we used the same alphabet transformation as in Clark but without shared decoder. This experiment also demonstrates

	Sindhu		**Clark**		**Clark with character classes**	
Ruleset	**LUT**	**FF**	**LUT**	**FF**	**LUT**	**FF**
L7	86 308	821	15 126	789	1 491	789
backdoor	169 328	3 955	28 977	3 680	4 317	3 680
exploit	11 286 017	19 453	1 865 660	19 326	21 408	19 326
ex.web-rules	1 108 910	4 186	185 912	4 186	4 574	4 186
imap	2 075 800	5 810	346 954	5 810	6 530	5 810
smtp	3 724 403	9 884	619 653	9 782	10 826	9 782
spyware-put	756 950	12 964	125 835	10 582	11 050	10 582
voip	745 015	1 890	123 494	1 787	3 044	1 787
web-client	8 121 216	14 717	1 346 964	14 670	16 305	14 670
web-misc	5 231 762	9 602	845 226	9 239	10 979	9 239
web-iis	1 156 224	2 394	192 080	2 394	3 234	2 394
snort-default	9 966 132	26 139	1 664 896	26 139	26 143	26 139
web-activex	1 461 133	44 768	257 843	44 768	46 816	44 768
	Sourdis without repetition		**Sourdis without strings**		**Sourdis**	
L7	1 424	320	1 491	788	1 424	320
backdoor	4 309	3 496	4 273	3 635	4 273	3 462
exploit	21 418	19 261	2 760	1 198	2 168	802
ex.web-rules	4 345	3 216	4 188	3 787	3 949	2 724
imap	21 582	3 763	1 291	558	1 291	558
smtp	10 829	9 772	2 508	1 524	2 224	1 295
spyware-put	11 039	10 392	11 054	10 519	10 976	10 318
voip	3 045	1 684	1 804	604	1 728	455
web-client	16 298	14 521	2 738	1 405	2 286	1 027
web-misc	10 982	9 232	2 343	1 163	2 123	967
web-iis	3 238	2 378	1 209	411	1 214	391
snort-default	39 021	19 299	14 881	13 889	11 029	7 815
web-activex	36 823	32 784	46 816	44 767	36 823	32 784

Table 7: Estimation of utilization of FPGA resources for Virtex-5 FPGA for NFA approaches

ability of Netbench to combine several approaches into new one. It can be seen from the table 8 that advantage of the shared decoder rises with the size of the input alphabet.

We use Netbench to evaluate Sourdis [31] work in more detail. This approach presents two new ideas. The first is to use the exact string matching algorithm (DCAM) to simplify the resulting automaton, and the second is to implement constrained repetition by the special block in FPGA fabric. The fourth column in Table 8 contains resource utilization of Sourdis's algorithm without the constrained repetition block, while the fifth column is without the string matching improvement. It can be seen that effects of these improvements depend on the used data sets. For example the imap data set has the same results for complete approach and for constrained repetition only, while web-activex data set does not contain any constrained repetition and therefore has the same results for complete algorithm and the algorithm only with string extension. However, most of the data sets utilize both extensions presented by [31].

5 Conclusion

We present new framework for evaluation and rapid prototyping of packet processing algorithms. While previous works in this field focused more on the feasibility for deployment, Netbench targets the early

	Sindhu 2 − stride		Clark 2 − stride		Sindhu 4 − stride		Clark 4 − stride	
Ruleset	**LUT**	**FF**	**LUT**	**FF**	**LUT**	**FF**	**LUT**	**FF**
L7	223 886	680	2 695	670	741 728	669	4 613	666
backdoor	571 326	3 610	6 411	3 504	2 447 400	3 720	11 732	3 677
exploit	17 347 330	16 189	21 260	16 135	32 922 927	14 655	25 360	14 629
ex.web-rules.reg	2 630 614	4 449	7 011	4 449	7 628 936	4 972	10 184	4 972
imap	3 868 025	5 806	7 507	5 806	8 192 129	5 900	9 454	5 900
smtp	6 538 509	8 856	12 447	8 808	20 748 056	8 452	19 070	8 431
spyware-put	2 063 781	11 937	14 600	10 915	8 378 857	12 269	25 293	11 805
voip	1 282 586	1 676	4 592	1 631	2 882 458	1 686	7 409	1 665
web-client	14 886 004	14 470	18 619	14 448	31 001 497	14 445	23 858	14 433
web-misc	9 206 847	8 637	13 205	8 457	32 137 808	8 386	27 878	8 301
web-iis	2 154 012	2 397	4 446	2 397	4 462 920	2 414	6 450	2 414
snort-default	20 219 020	26 969	31 334	26 969	48 107 080	28 633	38 537	28 633
web-activex	4 508 950	45 242	53 134	45 242	19 826 264	46 190	86 637	46 190

Table 8: Estimation of Utilization of FPGA resources for Virtex-5 FPGA for Multicharacter NFA approaches

stages of algorithm development when the ease of use and rapid prototyping are highly appreciated. Comprehensive set of algorithms was implemented to the framework in order to provide reference implementations and allow comparison of new approaches to existing algorithms. Moreover, the framework is designed to easily combine features from different algorithms. For example multistriding can be supported by regular expression matching algorithm even if this feature was not mentioned in the original paper.

Netbench also aims at defining a baseline standard in data sets for evaluating new and existing algorithms, while the possibility to use own data sets is not limited. This should contribute to better quality of the published results in the field.

We already found Netbench very useful in our own research [16, 25, 26, 35]. It is our ambition that every newly published algorithm for IP lookup, packet classification, and regular expression matching is included to Netbench.

Researchers are invited to submit new algorithms, patches, data sets, suggestions etc. to email address
`netbench@fit.vutbr.cz`. After review, these patches will be added to the framework.

Acknowledgment

This research has been partially supported by the Research Plan No. MSM, 0021630528 – Security-Oriented Research in Information Technology and the grant BUT FIT-S-11-1.

References

[1] Bgp table data.
`http://bgp.potaroo.net/`, 2011.

[2] Ipv6 dfp visibility.
`http://www.sixxs.net/tools/grh/dfp/`, 2011.

[3] Sphinx: Python Documentation Generator.
`http://sphinx.pocoo.org/`, 2011.

[4] M. Becchi. regex tool.
`http://regex.wustl.edu/`, 2011.

[5] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, CoNEXT '07, pages 1:1–1:12, New York, NY, USA, 2007. ACM.

[6] Bro IDS. Project WWW Page.
`http://http://www.bro-ids.org/`, 2011.

[7] C. Clark and D. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *Field Programmable Logic and Application, 13th International Conference*, pages 956–959, Lisbon, Portugal, 2003.

[8] C. R. Clark and D. E. Schimmel. Scalable Pattern Matching for High-Speed Networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 249–257, Napa, California, 2004.

[9] C. Câmpeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003.

[10] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *in ACM Sigcomm*, pages 3–14, 1997.

[11] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. Fast packet classification using Bloom filters. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 61–70, New York, NY, USA, 2006. ACM.

[12] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software ip lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, 2004.

[13] E. Fredkin. Trie memory. *Commun. ACM*, 3:490–499, September 1960.

[14] P. Hazel. PCRE WWW Pages.
`http://www.pcre.org/`, 2011.

[15] W. Jiang and V. K. Prasanna. Sequence-preserving parallel ip lookup using multiple sram-based pipelines. *J. Parallel Distrib. Comput.*, 69:778–789, September 2009.

[16] J. Kořenek and V. Košař. Efficient mapping of nondeterministic automata to fpga for fast regular expression matching. In *Proceedings of the 13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems DDECS 2010*, page 6. IEEE Computer Society, 2010.

[17] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pages 155–164, New York, NY, USA, 2007. ACM.

[18] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 339–350, New York, NY, USA, 2006. ACM.

[19] L7 Filter. Project WWW Page.
`http://l7-filter.sourceforge.net/`, 2011.

[20] B. Lampson, V. Srinivasan, G. Varghese, and A. Member. Ip lookups using multiway and multicolumn search. In *IEEE/ACM Transactions on Networking*, pages 324–334, 1998.

[21] H. Le and V. K. Prasanna. Scalable high throughput and power efficient ip-lookup on fpga. In *Field-Programmable Custom Computing Machines*, pages 167–174, 2009.

[22] J. W. Lockwood, N. Mckeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. Netfpga - an open platform for gigabit-rate network switching and routing. In *In IEEE Microelectronic Systems Education (MSE*, pages 160–161, 2007.

[23] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 217–231, New York, NY, USA, 1999. ACM.

[24] S. Nilsson and G. Karlsson. Fast address lookup for internet routers. In *IEEE Broadband Communications*, pages 11–22, 1998.

[25] V. Puš and J. Kořenek. Fast and scalable packet classification using perfect hash functions. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 229–236, New York, NY, USA, 2009. ACM.

[26] V. Puš, M. Kajan, and J. Kořenek. Hardware architecture for packet classification with prefix coloring. In *IEEE Design and Diagnostics of Electronic Circuits and Systems DDECS'2011*, pages 231–236. IEEE Computer Society, 2011.

[27] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)*, pages 227–238, April 2001.

[28] M. Skačan. Longest prefix match algorithms, FIT BUT, Brno, Czech Republic, 2010. Bachelor thesis.

[29] Snort. Project WWW Pages.
`http://www.snort.org/`, 2011.

[30] H. Song, J. Turner, and J. Lockwood. Shape shifting tries for faster ip route lookup. In *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols*, pages 358–367, Washington, DC, USA, 2005. IEEE Computer Society.

[31] I. Sourdis, J. Bispo, J. Cardoso, and S. Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51:99–121, 2008.

[32] V. Srinivasan and G. Varghese. Faster ip lookups using controlled prefix expansion. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '98/PERFORMANCE '98, pages 1–10, New York, NY, USA, 1998. ACM.

[33] D. Taylor and J. Turner. Scalable packet classification using distributed crossproducing of field labels. In *IEEE INFOCOM 2005, 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, pages 269–280, July 2005.

[34] D. E. Taylor and J. S. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, 2007.

[35] J. Tobola and J. Kořenek. Effective hash-based ipv6 longest prefix match. In *IEEE Design and Diagnostics of Electronic Circuits and Systems DDECS'2011*, pages 325–328. IEEE Computer Society, 2011.

[36] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '97, pages 25–36, New York, NY, USA, 1997. ACM.