Network of Shortcuts: An Adaptive Data Structure for Tree-Based Search Methods

Andrea Bergamini[1] and Lukas Kencl[2]

[1] Ecole Polytechnique Fédérale de Lausanne (EPFL),
1015 Lausanne, Switzerland
andrea.bergamini@epfl.ch
[2] Intel Research, 15 JJ Thomson Avenue, Cambridge,
CB3 0FD, United Kingdom
lukas.kencl@intel.com

Abstract. In this work we present a novel concept of augmenting a search tree in a packet-processing system with an additional data structure, a *Network of Shortcuts*, in order to adapt the search to current input traffic patterns and significantly speed-up the frequently traversed search-tree paths. The method utilizes node statistics gathered from the tree and periodically adjusts the shortcut positions.

After an overview of tree-search methods used in networking tasks such as lookup or classification, and a discussion of the impact of typical traffic characteristics, we argue that adding a small number of "direct links", or shortcuts, to the few frequently traversed paths can significantly improve performance, at a very low cost. We present a shortcut-placement heuristic, compare our method to a standard caching mechanism and show how the use of different levels of aggregation in a search tree enables to achieve similar results with much fewer entries.

1 Introduction

1.1 Networking and Search Trees

The explosive Internet growth requires complex tasks to be processed ever faster by the intermediate systems such as routers or firewalls. Optimization of these processes is necessary, yet often impossible a-priori, as the optimum often depends on the particular traffic mix. Run-time self-reconfigurable devices offer a solution with several advantages, i.e. optimized resource utilization, improved performance and ease of programming and management.

In networking systems, several targets for dynamic reconfiguration can be identified: resources re-mapping [1], codepath adaptation or *data structure* adaptation. Data structures comprise e.g. forwarding tables [2] and classification rule bases [3], which can be very large (e.g. 100k prefixes or 32k rules [3]) and the search speed is a major performance factor. As the search usually requires a number of memory accesses, and memory latency lags behind processor and link transmission speeds growth, optimization of data structures is critical. Typically

R. Boutaba et al. (Eds.): NETWORKING 2005, LNCS 3462, pp. 523–535, 2005.
© IFIP International Federation for Information Processing 2005

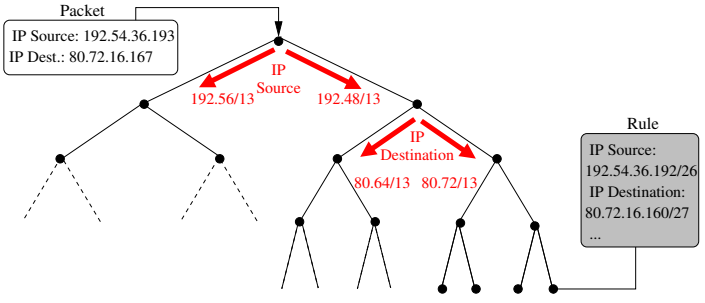


Fig. 1. Example of tree-based packet classification, as in e.g. [3]. Here, the root test is based on the IP source address, and the process is repeated at each node, where the test may be based on different, possibly multiple, packet fields

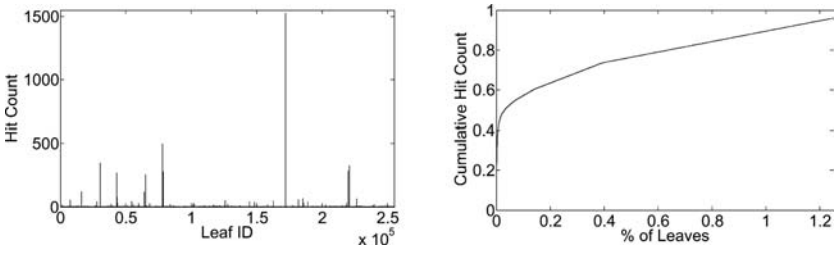


Fig. 2. *Example:* few tree paths are frequently searched. Tree depth 9, degree 4, generated packet flows conform to a measured length distribution (see Subsection 3.1). *Left:* Hit count distribution at tree leaves. *Right:* Cumulative Hit count. 90% of the traffic concentrated in less than 1% of the top searched paths

built based on the prefix tables or rule-bases and optimized for the worst case traffic, the data structures may be grossly sub-optimal for the actual traffic.

Various techniques are used to perform these optimizations [3, 4, 5, 6, 7], e.g. hash tables, compressed bitmaps or tree-like structures. HiCuts [4] and HyperCuts [3] deploy *search trees* to perform packet classification, the process of identifying the highest-priority rule applying to a packet. Both algorithms use heuristics to guide the tree construction, trading-off memory space and speed. At each node, a test is performed on the subset of the packet fields that narrows most the amount of rules that may apply to the packet (see Fig. 1), until a leaf is reached. The rules can be a combination of a fixed match, prefix match or a range match over various fields of the packet header, and may span multiple tree leaves.

The input, Internet traffic, is typically studied as aggregation of *flows*, sequences of packets sharing the same 5-tuple flow ID[1]. Previous studies [8] confirmed that some of the flow parameters are correlated (e.g. rate and size) and

[1] The 5-tuple flow ID is defined as: IP source and destination addresses, IP protocol number and TCP/UDP source and destination port numbers.

that small flows (mice) represent the majority, yet most of the traffic (in bytes) is concentrated in few big flows (elephants). In traffic engineering [9] it is common practice to identify flows that capture most of the traffic at a certain point in time and treat those differently to optimize resources utilization. This relies on such sets being persistent at least on a small time scale (shown to hold for flow volumes [10]). Temporal locality, the likelihood of a recently observed flow being referenced in the near future, is strongly present in the Internet traffic [11]. Critically, in search trees, the bias in flow lengths (many mice vs. few elephants) is reflected as *imbalance in the tree traversal patterns*, as a vast majority of searches typically end in a tiny subset of tree leaves (see example simulation in Fig. 2).

1.2 Objective: Adaptive Tree Search Optimization

In this work, we focus on run-time optimization of *tree-based* search data structures [3, 4]. It is critical to minimize the depth of the search tree to minimize the number of memory accesses. As some parts of the tree are likely to be traversed more frequently, we ask: *"Assuming we knew the traversal pattern, how could we optimize the tree for this pattern?"* The assumption is realistic, as it is possible to gather such statistics at line rate [12, 13]. The above observations about Internet traffic, such as the persistence property, motivate optimizing the data structure for the current time frame using the tree hit statistics from the previous time frame. Splay trees [14] and randomized binary search trees [15], are known examples of adaptive tree data structures, however, not applicable in this environment.

A common way of exploiting temporal locality is a *cache* of search keys, e.g. flow IDs. The Least Recently Used (LRU) cache replacement scheme has been shown to be near-optimal for networking workloads [16]. A cache cannot exploit information from the search data structure itself, so if the data structure is updated, there is a discrepancy between the contents and the cache must be flushed, which negatively affects the hit rate. The cache works with fine-grained fixed match keys (e.g. a 5-tuple flow ID), preventing aggregation of keys classified the same way, which can be exploited by a Denial of Service (DoS) attack by flooding with bogus entries. To achieve an acceptable hit rate, the cache needs to have a considerable size (e.g. 12,000 entries [17]) for a typical networking workload. With the increasing key size, this may become prohibitively large.

We propose a novel method to augment the existing search tree with an additional data structure, a *Network of Shortcuts (NoS)*, along the frequently traversed paths. The objective is to capture a large fraction of traffic and reduce the total number of memory accesses per time frame. The hierarchical aggregation of search keys in subsequent tree nodes is preserved and thus the method scales well with the number of distinct search keys (e.g. flow IDs). Heuristics to determine where to place the shortcuts are presented.

The combination of a search tree and statistics-based shortcuts may recall the *Huffman tree* [18]. But a pure Huffman tree construction would alter the node tests, and thus would not reflect the rule base structure.

Both caching and shortcuts reduce the number of memory accesses in the average case, possibly affecting the worst case scenario. The affects on the worst case can be limited by constraining a number of shortcuts along a single path, as well as by adjusting the method parameters or disabling it altogether, should the current traffic patterns not be favorable. We demonstrate that a controlled adjustment of the worst case (i.e. keeping the worst case bounded) results in massive gain in the average case. The tree-based packet classification is a working example, but the shortcut method can be applied to various tree-search methods.

The paper is organized as follows: in Section 2 we present the shortcut method and in Section 3 the performance results and a LRU cache comparison using generated and real traces. Section 4 describes a possible implementation of the method, and in Section 5 we discuss open problems and add concluding remarks.

2 The Network of Shortcuts (NoS) Method

2.1 Assumptions and Notation

1. We assume a tree-based search method, like HyperCuts [3], to be deployed. Backtracking, or any other mechanism that may alter the way the tree is traversed, must not be used.
2. We assume that statistics (hit count) about *every* node in the tree are periodically available. Efficient gathering of such statistics is possible [12, 13].
3. The decision at each tree node has a *cost* in terms of memory accesses (e.g. pointers to be fetched from the memory), which we assume equal to 1.

Let $F(u)$ be the periodically measured hit-count at node u, $f(u)$ the fraction of traffic at node u, $f(u) = \frac{F(u)}{F(root)}$, d the tree node degree and l the tree depth.

Definition 1. Given a tree $T = (V, E)$, a *shortcut* is an edge $e \notin E$ that connects a non-leaf *Starting Node* $SN \in V$ with its descendant in the tree T (as in Fig. 3(a)).

A shortcut is specified by a list of nodes it bypasses: shortcut $SC = (a, b, c, d, e)$. Fig. 4 presents the search before and after the placement of a single shortcut. A shortcut at a starting node SN enforces the search to test first the path using

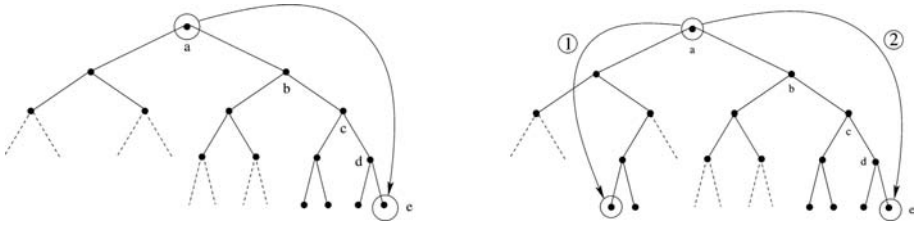
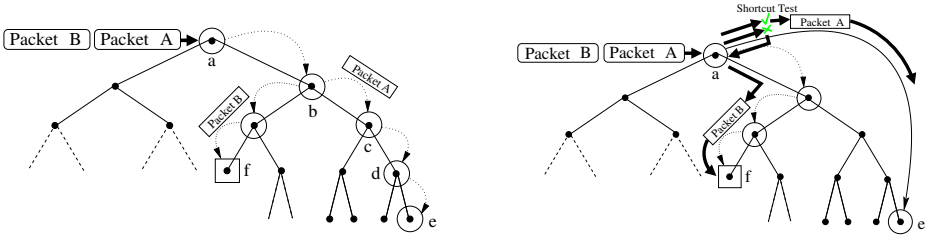


Fig. 3. *Left:* A shortcut is an additional edge between two nodes along a tree path. *Right:* A multi-shortcut. Single-shortcuts (1) and (2) are combined



(a) Without shortcuts. (b) With a shortcut (a, ..., e).

Fig. 4. In 4(a) there are no shortcuts. At each node, packets A and B perform a test which direction to take, ending in leaves e and f respectively. In 4(b), both packets first test the shortcut. Packet A succeeds and shortcuts to e directly, whereas B fails and must retrieve the pointer along the normal path, incurring cost α

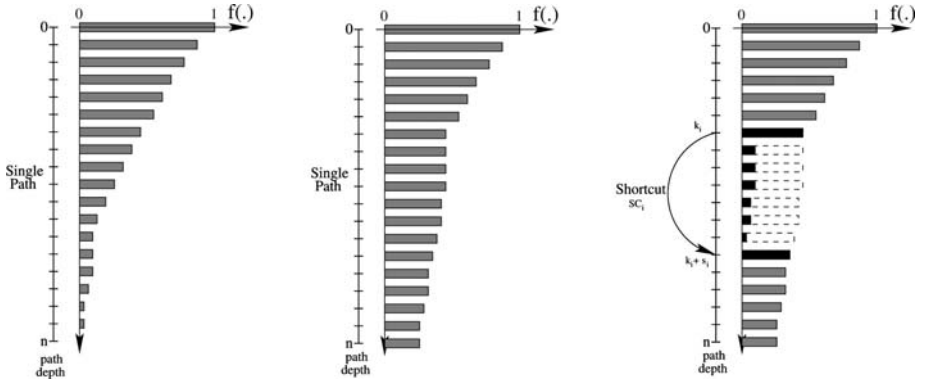


Fig. 5. *Left:* Traffic along a generic path, *center:* Traffic along a heavily-hit path, *Right:* Effect of placing a shortcut: packets skip intermediate nodes

a shortcut, and only in case of failure to continue through the search tree. Note that this requires the ability for a shortcut $SC = (a, b, c, d, e)$ to compress tests at nodes b, c and d into one test stored at node a. When a packet fails to take a shortcut, a *cost* $\alpha \geq 1$ (in terms of memory accesses) is incurred, as an additional read may be required to read the child pointer. Fig. 5 shows the effects on the traffic $f(.)$ of a single-shortcut. The notation $SC_i(m : l)$ means "from the m_{th} node to the l_{th} bypassed by the shortcut". The notion of a shortcut is a logical abstraction, for practical implementation issues please refer to Section 4.

Multi-shortcut A multi-shortcut MSC is a set of single shortcuts with a common starting node (e.g. $MSC = \{SC_1, SC_2\}$, $SC_1 = (a, b, c, \ldots)$ and $SC_2 = (a, f, g, \ldots)$, as in Fig. 3(b)). Introducing multiple shortcuts allows to increase the fraction of traffic making use of the SN. The memory access *cost* is independent from the number of shortcuts if $\forall MSC$, $MSC = \{SC_1, SC_2, \ldots\}$, $SC_i(2) \neq SC_j(2)$, $\forall\ i, j : i \neq j$, i.e. there is at most one shortcut per outgoing

edge to test. Relaxing this constraint would allow more shortcuts to be placed but it could also increase the number of memory accesses required at SN.

2.2 Local Shortcut Gain Functions

Let k_i be the SN and s_i be the length of a shortcut SC_i, placed between nodes k_i and $(k_i + s_i)$ (see Fig. 5). The number of memory accesses along the path $k_i \ldots (k_i + s_i - 1)$ *before* placing the shortcut is:

$$BMA_i = \sum_{j=0}^{s_i-1} f(k_i + j) = f(k_i) + \sum_{j=1}^{s_i-1} f(k_i + j), \tag{1}$$

where BMA stands for "Before Memory Accesses". When the shortcut is added we obtain:

$$AMA_i = \underbrace{f(k_i + s_i) + \alpha[f(k_i) - f(k_i + s_i)]}_{(A)} + \underbrace{\sum_{j=1}^{s_i-1} [f(k_i + j) - f(k_i + s_i)]}_{(B)}, \tag{2}$$

where AMA stands for "After Memory Accesses". The expression (A) refers to the number of memory accesses experienced at the SN. The first term indicates that $f(k_i + s_i)$ (traffic taking the shortcut) traverses the SN at cost 1 , and the remainder $(f(k_i) - f(k_i - s_i))$ at cost α. The expression (B) accounts for the reduced number of memory accesses experienced in the intermediate nodes. We define a *metric* to evaluate the impact of a shortcut:

Single-shortcut Local Gain: Given a single-shortcut SC_i, the local shortcut gain is defined as $G_i = BMA_i - AMA_i$. After some simple computations:

$$G_i = f(k_i)[1 - \alpha + (s_i + \alpha - 2)\gamma_i], \text{ where } \gamma_i = \frac{f(k_i + s_i)}{f(k_i)}. \tag{3}$$

A shortcut is effective if the gain of placing a shortcut is positive: $G_i > 0 \Leftrightarrow s_i > 2 - \alpha + \frac{\alpha-1}{\gamma_i}$.

Multi-shortcut Local Gain: The local gain of a multi-shortcut of degree p is:

$$G_{iMSC} = f(k_i) \left(1 - \alpha + \sum_{l=1}^{p} (s_{il} + \alpha - 2)\gamma_{il}\right); \ \gamma_{il} = \frac{f(k_{il} + s_{il})}{f(k_i)}. \tag{4}$$

If $s_{i1} = s_{i2} = \ldots = s_{ip} = s_i$ and $\gamma_i = \sum_l \gamma_{il}/p$ is the average fraction of traffic taking a single-shortcut, the gain simplifies to:
$G_{iMSC} = f(k_i)\,(1 - \alpha + p(s_i + \alpha - 2)\gamma_i)$ and $G_{iMSC} > 0 \Leftrightarrow s_i > 2 - \alpha + \frac{\alpha-1}{p\gamma_i}$.
Thus, given an average fraction of traffic γ_i, the multi-shortcut achieves p times higher gain than a single-shortcut.

2.3 Shortcut Placement Strategy

The shortcut placement problem is the following optimization problem:

Given: a tree $T = (V, E)$ of variable degree and node statistics $f : V \rightarrow N$.

Find: $SC = \{SC_i\}$, a set of single- and multi-shortcuts.

Constraints (conditions to determine the validity of a set of shortcuts)**:**

$\forall i, j \ : i \neq j;\ SC_i, SC_j$ are valid $\Leftrightarrow$
$(SC_i(1:2) \subset SC_j \wedge SC_i(s_i - 1 : s_i) \subset SC_j) \ \vee$
$(SC_i(1:2) \not\subset SC_j \wedge SC_i(s_i - 1 : s_i) \not\subset SC_j)$.

Objective function: $max \sum_i G_i$, where G_i is the local gain of SC_i.

We maximize the total gain, a sum of the local gains. As the local gains are *not* mutually independent, the optimization problem is non-linear. We conjecture that the placement problem is NP-Complete, although we could not prove it. We present a heuristic that provides good performance with few computations, however, we are unaware how close our results are to the optimum:

Shortcut Placement Algorithm: Let $Node_i$ $(i = 1, \ldots, N)$ be the i_{th} node in the tree (N nodes in total, numbered from the root, top-down, level-by-level); and let $Node_{ij}$ $(j = 1, \ldots, d)$ be the j_{th} child of the i_{th} node. Let γ_{th} and $gain_{th}$ be the thresholds that discriminates the SNs, and BSS_j the "best single-shortcut" in the j_{th} subtree. The pseudocode for the shortcut placement is:

```
1: for (i = 1; i <= N; i++) do
2:   if HitCount(Node_i) ≠ 0 then
3:     for (j = 1; j <= d; j++) do
4:       BSS_j, γ_j, gain_j ← BestSingleShortcut(Node_ij);
5:     end for
6:     γ ← Σ γ_j; gain ← Σ gain_j;
7:     if (γ > γ_th) & (gain > gain_th) then
8:       PlaceShortcut(Node_i, BSS_1, ..., BSS_d);
9:     end if
10:  end if
11: end for
```

BestSingleShortcut(.) identifies in the sub-tree rooted at node $Node_{ij}$ the single-shortcut that gives the highest gain, and returns the shortcut target node, γ_j and $gain_j$. The algorithm places shortcuts in a top-down fashion and reduces the number of nodes under consideration at each step downwards. The complexity is $O(N \log_d N)$, where N is the total number of nodes.

Starting from the root allows to consider the longer shortcuts, along the few paths where the traffic curve $f(.)$ is completely flat, first. Note that changing the threshold values will affect the number of accepted SNs, and that the traffic pattern may not justify placing shortcuts at all.

Adaptation Period: The shortcut placement is re-computed periodically with period TA. Statistics collected in the previous interval are then used to place shortcuts for the following interval. The initial length of the interval is determined by the network workload and the tree size (the NoS method needs "enough" packets in the tree to obtain a meaningful description).

3 Method Validation

In this Section we validate the NoS method through simulations involving both synthetic and real traces and compare the method with a *Least Recently Used* (LRU) Cache. We picked LRU for the comparison because it performed best in our settings (we compared FIFO, RAND, LFU and LRU [16]). To make a fair comparison, since the number of shortcuts placed is variable, we vary the cache size accordingly (1 shortcut = 1 cache entry). We compare the relative gain in memory access reduction, defined as $GAIN = (\sum_i G_i)/(\sum_i BMA_i)$, and the hit rate, $HITRATE = (Traffic\ taking\ the\ shortcuts\ or\ cache)/(Total\ traffic)$.

3.1 Traffic Generator, Real Traces and Simulation Parameters

Synthetic packet traces were generated using MATLAB R7 software. The traces are characterized by a constant number of concurrent flows, Poisson new-flow arrival times and 32 bit *uniformly* distributed flow-IDs. The flow length distribution is obtained combining measured flow lengths with a tail fitted using a Pareto distribution (parameter $a = 0.9163$). The uniformity of flow-IDs is not realistic. However, as this leads to a uniform spread in the search tree, it is a good "worst-case" test for the NoS method.

Throughout this study we also used real data collected by the network monitor described in [19]. The monitored site connection is a full-duplex Gigabit Ethernet. The `dump_*` and the `mon_*` traces contain approximately 3000 and 15000 unique flow-IDs respectively. The packet source/destination addresses and source/destination port numbers (96 bits in total) were mapped into the 32 bit flow-ID used in the simulations[2].

In all the simulations, the parameters are set to: $\gamma_{th} = 0.4$, $gain_{th} = 4000$, $TA = 10000$ packets (synthetic traces), $TA = 100ms$ (real traces), $\alpha = 2$. All trees are $d = 4$, $l = 9$ (no missing nodes), except where stated otherwise. The trees are balanced, to avoid results biased due to the tree structure. At every tree node, the search space is partitioned into consecutive chunks of equal size, starting with the 32-bit space at the root. In each bar graph, the number on top of the bars is the number of shortcuts placed.

3.2 NoS Versus LRU Cache

Fig. 6 presents the results using *synthetic traces*. The NoS method achieves 4 times the cache gain in every condition given the same size and up to 50 times

[2] We combined the 10 most significant bits of the addresses with the 6 least significant bits of the port numbers=2*10+2*6=32 bits.

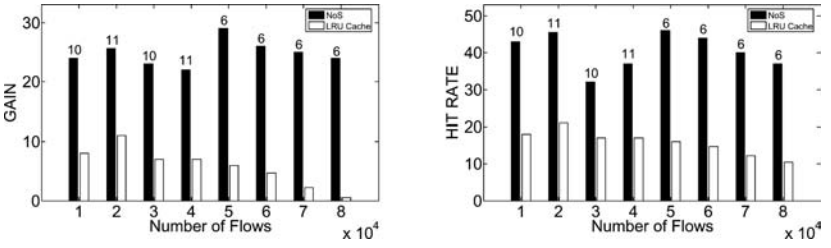


Fig. 6. NoS vs. LRU: Gain and Hitrate comparison using synthetic traces

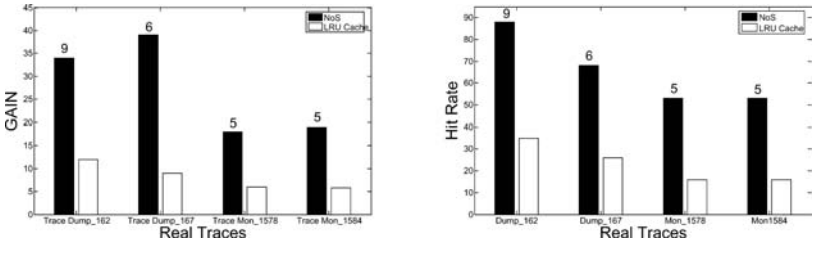


Fig. 7. NoS vs. LRU: Gain and Hitrate comparison using real traces

when the number of flows increases. This is possibly due to several factors: the aggregation of keys across shortcuts (improving scalability in the number of flows), i.e. the cache works at the flow level, whereas shortcuts target intermediate nodes, grouping multiple flows together; in addition, the shortcuts allow to capture all packets from the most populated flows, whose cache hit-rate is hurt by the less populated flows. The NoS method likewise exhibits approximately 4 times higher hit-rate.

Fig. 7 shows the results using *real traces*. For `dump_162` and `dump_167` both gain and hit rate are much higher than those observed in the synthetic case, probably due to non-uniformity in the flow-IDs. The `mon_1578` and `mon_1584` traces contained a higher number of flow-IDs, with more uniform distribution. Still, the NoS method managed to capture more than 50% of the traffic, achieving gains 3 times higher than the cache.

Worst Case and Average Case: Fig. 8 shows detailed worst case and average case statistics from one of the real traces. It illustrates that the NoS method achieves a significant improvement in the average number of memory accesses, while keeping the worst case bounded. The amount of the worst case instances is marginal ($10^{-4}\%$). In spite of the average case improvement, fewer packets are adversely affected by NoS than by the LRU cache.

3.3 Variable Root Degree

Real-life search trees are typically of variable degree; in particular, the root degree tends to be significantly higher (e.g. 64). We present experiments with

	Tree Search	LRU Cache	NoS
Average Case	10	8.5	6.4
Worst Case	10	11	14
% Worst Case	100	75	0.00068
% Worse Off	0	75	23.12

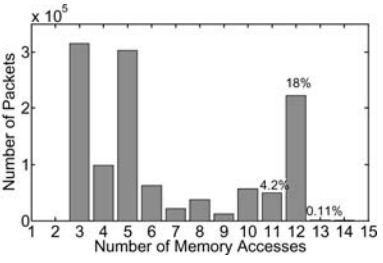


Fig. 8. Detailed results from trace dump_167. *Left:* Per-packet memory accesses. *Right:* Histogram of memory access distribution using the NoS method

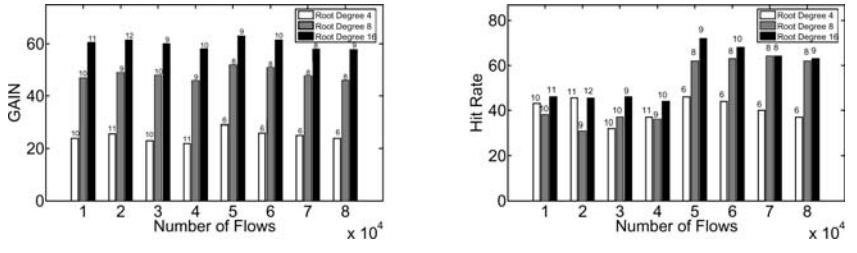


Fig. 9. NoS Gain and Hitrate comparison using synthetic traces: root of variable degree

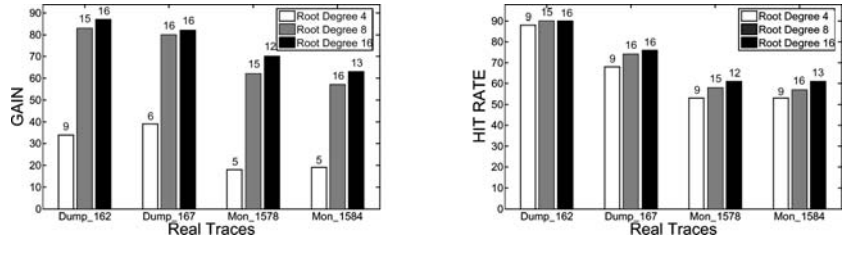


Fig. 10. NoS gain and Hitrate comparison using real traces: root of variable degree

root degrees of 8 and 16, using both synthetic and real traces, in Fig. 9 and 10. The boost in performance between degree 4 and degree 8 is remarkable (120 times the gain of a LRU cache). The higher degree of the root provides more nodes at the top of the tree, with more starting nodes, and enables capturing more traffic at a higher tree-level, increasing the gain. The same reasoning does not apply to hit-rate and the results show little variation from root degree 4.

4 Possible Implementation and Applications

A possible architecture of a NoS implementation is shown in Fig. 11. The data plane performs the per-packet operations, whereas the control plane performs

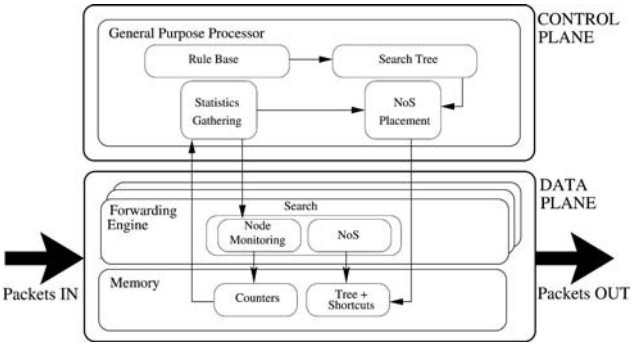


Fig. 11. Relations between building blocks and system resources. The arrows represent the direction of the information exchange. Information is pulled by the statistics gathering engine from the node hit counters and pushed to the search method by the NoS algorithm (shortcuts placement)

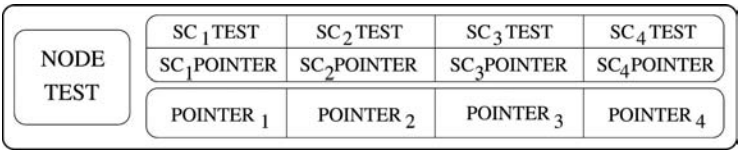


Fig. 12. Shortcut starting node possible memory layout. The entry stores the original tree node test and child pointers and the shortcut compressed tests and target pointers

the more complex but less frequent tasks. The data plane consists of one or more forwarding engines and a fast memory (e.g. SRAM), while the control plane is typically implemented on a general purpose processor. An example of such a system is the Intel® IXP™ Network Processor [20].

In the *data plane*, the actual per-packet tree search, including taking the shortcuts, and the hit statistics collection are performed. The hit counters and the shortcut-augmented search-tree reside in fast memory. The actual shortcut entry in the search tree requires to change the memory layout of the original data structure (see Fig. 12). The *control plane* processes include the creation and update of the actual search tree (independent of NoS), the periodic hit counter read and the periodic re-computation of the NoS shortcut placement. The NoS results are periodically uploaded to the search method, updating the shortcuts (but not the search tree itself). The period TA over which the update is performed is set based on the traffic profile and system characteristics.

One possible NoS method application is a denial of service (DoS) attack prevention. Where a cache can be flooded by random independent entries and thus practically disabled, the aggregation of entries into tree nodes would allow the shortcuts to capture such malicious traffic. On the other hand, malicious attacks like the RoQ (Reduction of Quality) attacks [21] against adaptive methods like

NoS can be prevented by placing a second control loop around the adaptation period TA, adjusting it to the rate of changes in the current patterns, and by introducing randomness into TA selection.

5 Conclusions

We have proposed an adaptive data structure to improve the performance of tree-based search methods, using shortcuts along frequently searched paths. The NoS method has been validated through simulation and the results show that NoS outperforms a LRU cache of comparable size in terms of hit rate and memory access reduction. NoS has shown scalable performance even with high number of concurrent flows (70,000+), as the method does not rely on per-flow information, but rather works with the data structure itself.

Despite the good performance when the number of shortcuts is small, it is an open issue how to design a scalable placement strategy, i.e. one that would increase the overall gain when increasing the number of shortcuts. Likewise, efficient update of the shortcut placement remains an open problem. As for the search database's update, there are generally few changes in the search tree structures. If the NoS method was made aware of the changes, the placement could be adjusted, instead of entirely invalidated as in the cache case.

In summary, the shortcut approach offers a viable mechanism for dynamic tree data structure optimization and represents a step towards a fully autonomous packet processing system.

Acknowledgment

We thank Andrew Moore from the Cambridge University Computer Laboratory for providing the networking traces and for very helpful comments.

References

1. R. Kokku et al., *A Case for Run-Time Adaptation in Packet Processing Systems*, In the 2nd Workshop on Hot Topics in Networks (HOTNETS-II), November 2003.
2. M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, *Survey and Taxonomy of IP Address Lookup Algorithms*, IEEE Network Magazine, March 2001.
3. S. Singh, F. Baboescu, G. Varghese, and J. Wang, *Packet Classification Using Multidimensional Cutting*, Proceeding of SIGCOMM, Karlsruhe, Germany, 2003.
4. P. Gupta and N. McKeown, *Packet Classification using Hierarchical Intelligent Cuttings*, Proceedings of Hot Interconnects, 1999.
5. M. Kounavis, A. Kumar, R. Yavatkar, and H. Vin, *Two Stage Packet Classification Using Most Specific Filter Matching and Transport Level Sharing*, Technical Report, Intel Research and Development, September 2004, in submission.
6. V. Snirivasan, S. Suri, and G. Varghese, *Packet Classification using Tuple Space Search*, Proceedings of SIGCOMM, September 1999.
7. T. Lakshman and D. Stiliadis, *High Speed Policy-based Packet Forwarding using Efficient Multi-Dimensional Range Matching*, Proceedings of SIGCOMM, 1998.

8. Y. Zhang, L. Breslau, V. Paxson and S. Shenker, *On the characteristics and origins of internet flow rates*, Proceedings of SIGCOMM, 2002.
9. K. Papagiannaki, N. Taft and C. Diot, *Impact of Flow Dynamics on Traffic Engineering Design Principles*, Proceedings of INFOCOM, 2004.
10. J. Jo, Y. Kim, H. J. Chao and F. Merat, *Internet Traffic Load Balancing using Dynamic Hashing with Flow Volume*, Proceedings of SPIE ITCom, July 2002.
11. K. Claffy, *Internet Traffic Characterization*, Ph.D. Thesis, 1994.
12. S. Ramabhadran and G. Varghese, *Efficient Implementation of a Statistics Counter Architecture*, Proceedings of SIGMETRICS, June 2003.
13. N. Kammenhuber and L. Kencl, *Efficient Statistics Gathering from Tree-Search Methods in Packet Processing Systems*, in press, May 2005.
14. D.D. Sleator and R.E. Tarjan, *Self-Adjusting Binary Search Trees*, Journal of the ACM, July 1985.
15. C. Martínez and S. Roura, *Randomized Binary Search Trees*, Journal of the ACM, March 1998.
16. R. Jain, *Characteristic of Destination Address Locality in Computer Networks: A Comparison of Caching Schemes*, June 1990.
17. C. Partridge et al., *A 50-Gb/s IP Router*, IEEE/ACM Transactions on Networking, 1998.
18. David A. Huffman, *A Method for The Construction of Minimum Redundancy Codes*, Proceedings of IRE, September 1952.
19. A. Moore, J. Hall, C. Kreibich, E. Harris and I. Pratt, *Architecture of a Network Monitor*, Passive and Active Measurement Workshop (PAM), La Jolla, CA, 2003.
20. E.J. Johnson and A.R. Kunze, *IXP2400/2800 Programming*, Intel Press, 2003.
21. M. Guirguis, A. Bestavros and I. Matta, *Exploiting the Transients of Adaptation for RoQ Attacks on Internet Resources*, 12th IEEE International Conference on Network Protocols (ICNP), Berlin, Germany, 2004.