

# Efficient Statistics Gathering from Tree-Search Methods in Packet Processing Systems

Nils Kammenhuber

Technische Universität München, Institut für Informatik

Boltzmannstraße 3

D-85748 Garching bei München, Germany

kammenhuber@net.in.tum.de

Lukas Kencl

Intel Research

15 JJ Thomson Avenue

Cambridge, CB3 0FD, UK

lukas.kencl@intel.com

**Abstract**—We present a novel algorithm for efficiently gathering statistics about the hit frequencies on the nodes of a search tree in a packet processing system, under limiting space constraints. The Expand and Collapse (EaC) algorithm is a heuristic that periodically adjusts the subset of nodes of the search tree at which statistics are gathered, in order to use the limited space available to collect statistics in preference from the currently most heavily-hit nodes in the search tree. We prove convergence and good node-hit coverage of the algorithm and validate its performance on a set of simulated data.

The information collected can be useful for a variety of reasons, such as inferring traffic properties, discovering failures and attacks or dynamically optimizing the search method itself for locality patterns in the oncoming traffic.

**Index Terms**—Resource allocation, network architecture, traffic monitoring and management, adaptive networks.

## I. INTRODUCTION

A common task in packet processing systems is a lookup or search over a database organized into some form of a search tree. Typically executed per every packet, it must be carried out within a tight time budget. Examples are a next-hop lookup, where a packet destination address is compared to a table of address prefixes using a *longest-prefix match*, or *classification*, where multiple fields in the packet header are compared against rules in the classification table, in order to determine the applying rule with the highest priority.

Efficient implementation of the longest-prefix match [1], [2], [3] and rule-based classification [4], [5], [6], [7] has been a subject of extensive research. Often, a search tree is built over a pre-processed prefix-table or rule-set, to achieve a favorable size vs. search-speed tradeoff. For example, the Hi-Cuts [5] and HyperCuts [6] classification methods construct search trees by a heuristic that cuts the search space into subspaces containing approximately equal amount of rules.

Generally, few assumptions or observations are made about the workload patterns of the search keys—the methods are typically optimized for the worst case scenario, minimizing the depth of the search tree. However, neither the packet flows are distributed uniformly over the address- or rule-space, nor the popularity of flows in terms of packet count is uniform [8], [9]. Clearly, such non-uniform distribution of workload is going to lead to massive inequalities in the number of packets traversing different paths of the search trees, as some paths will be traversed much more frequently than others.

To our best knowledge, little work has been so far dedicated to the ability to monitor, at run-time, the hit rates per different paths traversing the search tree structure. This information can be useful in a number of ways: for inferring traffic properties, determining the most potent traffic flows, discovering failures and attacks or dynamically optimizing the search method itself for locality patterns in the oncoming traffic.

In this work, we study statistics gathering from a tree-search method on a high-speed architecture with a complex memory hierarchy, such as a network processor [10]. We aim to efficiently capture how traffic is distributed over the search tree and to determine the current frequently traversed paths.

This paper is organized as follows: In Section II we briefly review the related research and in Section III we pose the problem of efficient gathering of statistics over the tree search method. Then, in Section IV, we define the notation to describe the environment, and in Section V, we present the algorithmic solution. In Sections VI and VII, we analyze the algorithm properties and performance. Finally, in Section VIII, we discuss the open issues and add concluding remarks.

## II. RELATED WORK

Gathering statistics about current network traffic has become a standard task in both research and in operational networking environments [11], [12], [13]. Usage of such statistics ranges from analysis of flow dynamics on wide-area networks [8], [9] over rules how traffic can be engineered [14], [15] to traffic-adaptive methods within the network node [16], [17].

The principal lessons can be summarized as that many quantities characterizing network performance have long-tail probability distributions, which may have a dramatic effect upon performance [8], [9]. Thus, a common engineering principle is to select a small set of objects (packet flows) that account for a large fraction of the overall traffic (due to the long-tail distribution), to be treated differently so as to achieve a specific performance objective. However, the object dynamics are volatile and the particular objects (packet flows) need to be reselected often [14].

The counter management algorithm in [18] allows to maintain a large number of counters at line rate. We take an orthogonal approach, by restricting the number of counters to only those that convey useful information, to further limit the overhead.

### III. EFFICIENT GATHERING OF STATISTICS

We assume that packet information is processed by performing a tree search using some packet data as the search key. More precisely, that the search *starts from the root of the tree and continues downwards, without any backtracking*, as in HyperCuts [6] (Fig. 1). Our goal is to determine the frequently traversed tree paths and to measure the *hit count* on these paths.

We assume a programmable networking system, such as a network processor (NP) [10], to be the target device for the method's implementation. NPs typically consist of a control processor, multiple forwarding engines and a hierarchy of memories, differing in memory access latency, size and cost. Typically, the memory accesses are a key bottleneck in terms of performance. *Fast memory* is scarce and it is the use of this resource that we aim to optimize in this work.

The tasks executed on a NP belong either to the *data plane*, i. e., tasks that are processed per every packet, generally carried out at the forwarding engines, or to the *control plane*, i. e., not-so-frequent, more computationally intensive tasks carried out on the control processor. We assume the tree search method to belong among the data plane tasks, as well as the counting of hits along the nodes of the search tree. The algorithm to select the monitored nodes would run on the control plane, as the selection process may be relatively complex and would not happen on a per-packet timescale.

Gathering of hit statistics should not impose a large overhead on top of the search algorithm. We thus need to use fast memory for counting the accesses to the nodes. For example, on the Intel® IXP™ 2400 NP, there are  $8 \times 640$  words of high-speed local memory [10] available, yet the rule databases containing several thousands of rules can range from 10k to 100k words [5], [6]. Furthermore, if the search is performed on multiple engines in parallel, simultaneous memory accesses must be dealt with. Thus, reducing the counting overhead is important to prevent potentially expensive access conflicts.

#### A. Counting strategies

It is unwise to maintain a counter for each node of the search tree, as it is desirable to reduce the number of counters located in the fast memory (other counters could be placed into slower but cheaper types of memory, as in [18]). A straightforward approach is to maintain *counters for the tree leaves only*. This allows to reconstruct hit counts up through the nodes in the tree by summing up hits of child nodes. However, the number of leaves in a typical classification search tree can be quite large (a tree of a uniform node degree 4 and depth 10, which is quite realistic [5], [6], has about 1 million leaves), and as most of the leaves would most likely be hit infrequently, monitoring all of them would waste too much memory on “uninteresting” counters.

In classification trees [5], [6], many rules span different leaves of the tree (see Fig. 1). We may keep a *hit counter for each rule*, yet this approach wastes a lot of counters for rules that are hit infrequently. Once determined, the counters for the infrequently-hit rules can be shifted to slower memory, but we

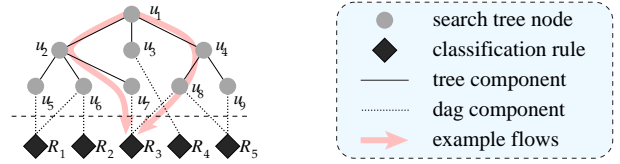


Fig. 1: Example HyperCuts [6] classification search tree and corresponding classification rules. Note that for example rule  $R_3$  can be reached both via the path  $(u_1, u_2, u_6)$ , as well as  $(u_1, u_4, u_8)$ .

would still need a vast amount of fast memory in the initial phase (the number of rules in today's ACLs is in the order of thousands [5], [6], [7], but is expected to grow fast with proliferation of mechanisms like VPNs or traffic engineering). Furthermore, by knowing how often each rule is hit, we cannot precisely infer how traffic traverses the tree, since one rule may be reached via different paths, as in Fig. 1.

The method presented in this work collects data preferably at the nodes that attract a large number of hits. To monitor the heavily-hit nodes, we must determine the position of these nodes, but that requires earlier monitoring of the tree structure! To address this chicken-and-egg problem, we iteratively find out and adjust which nodes are heavily-hit. Given  $x \cdot \max\{\text{height of tree}\}$  counters, in Section VI-C we prove that our algorithm provides the precise hit count for all nodes that are hit at least  $1/x \cdot \#\text{search operations}$  times.

### IV. NOTATIONS AND DEFINITIONS

In the case of the HiCuts [5] and HyperCuts [6] algorithms, there are three layers in the search tree structure: (1) the tree with the search nodes, each representing a set of cuts, (2) lists of rules attached to nodes, and (3) the rules themselves (see Fig. 1). For our purposes, we perceive the lists of rules (2) as leaves of the tree. Coupling the rules (3) with the search tree would turn the tree into a directed acyclic graph: one rule may be reached through different paths in the tree, as in Fig. 1. When referring to a *tree*, we mean the part without the rules (i.e. without (3)).

In this text, variables and symbols have the following meaning:

$u_i$	node $i$
$f(u_i)$	hit count (number of hits) on node $i$
$t_k$	time interval $k$
$\rightarrow(t_k)$	transition to next time interval $t_k$
$\chi_{(t_k)}$	some “ $\chi$ ” during time interval $t_k$ (e. g.: $f_{(t_4)}$ = hit count during interval $t_4$ )
$f_{(t_k)}(0)$	hit count on root, i.e., packets processed by the system during $t_k$
$u_i \supset u_j$	node $i$ is an <i>ancestor</i> of node $j$ in the search tree
$u_i \dot{\supset} u_j$	node $i$ is a <i>direct parent</i> of node $j$
$\mathfrak{M}u_i$	node $i$ is being monitored (predicate)
$\mathfrak{F}u_i$	node $i$ can be followed (predicate)
$\mathfrak{H}u_i$	node $i$ is heavily-hit
$M$	set of monitored nodes
$h$	maximum height of search tree

**Definition IV.1** A node  $u$  is said to be **heavily-hit** relative to a set of nodes  $\{u_1, \dots, u_n\}$  iff, assuming that  $f(u_1) \geq f(u_2) \geq \dots \geq f(u_n)$  holds,  $f(u) \geq f(u_\rho)$ , for some fixed  $\rho \in [1, n]$ .

We use the metric of hit counts for determining whether a node should be monitored or not. Other metrics may be possible, however, the algorithmic rules in Section V use the assumption that the values generated by the metric are *monotonically non-increasing* along any path down the tree, and thus  $(u \subset v \wedge \mathfrak{H}u) \Rightarrow \mathfrak{H}v$ .

Input traffic patterns undergo significant changes over time. We divide time into intervals  $t_1, t_2, \dots$ , of equal length, and associate the property of being heavily hit to a specific time interval.

## V. THE EXPAND AND COLLAPSE (EAC) ALGORITHM

### A. Statistics gathering

The method for efficiently gathering statistics under stringent memory constraints is divided into three parts:

- 1) The actual **statistics gathering** happens during the classification of a packet. Updating of the counters must be integrated into the traversal of the tree during the search. Due to accessing counters, this part of the algorithm needs to operate using fast memory, as the per-packet processing time is potentially prolonged.
- 2) **Selection of monitored nodes**, i.e., the nodes for which we keep counters, is performed after each time interval of a pre-defined length. The selection is performed based on the data read from the counters and may be done off-line. This part constitutes the actual *EaC algorithm*.
- 3) **Bounds computation for other tree nodes**, i.e., inferring from our knowledge about the monitored nodes to other nodes of the tree, can also be performed off-line. This part can be implemented as an on-demand API for the application that requires the statistics being gathered, e.g., a tree optimization algorithm.

As we are looking for the heavily-hit nodes, we start monitoring nodes near the root of the tree first, and then proceed towards the leaves selectively at the heavily-hit nodes. Whenever the traffic characteristics change, we gradually adjust the monitored set to fit the new distribution of node hits—i.e., we refrain from nodes no longer heavily-hit and instead monitor nodes that have recently become heavily-hit.

To keep track of which nodes are monitored, we augment *all the nodes* of the monitored tree by a flag `monitored`. Formally, for a node  $u$  we indicate monitoring by a predicate  $\mathfrak{M}u$ . Furthermore, the *nodes being monitored* carry the following additional fields:

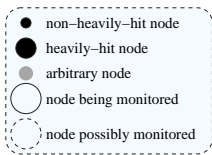


Fig. 2: Legend

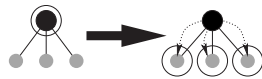


Fig. 3: Expand-to-children rule. If a node is heavily-hit, we start monitoring its children.

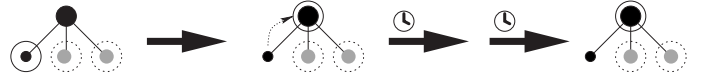


Fig. 4: Collapse rule, with the oscillation prevention rule. If a monitored node is not heavily-hit, we stop monitoring it and monitor its parent instead. The oscillation prevention stops us from re-descending to the non-heavily-hit node for a configured time period.

- A *counter* that counts all the search accesses passing through this node;
- A flag `follow`, indicated by a predicate  $\mathfrak{F}u$ . The purpose of this will be explained further below.

After each monitoring time interval  $t_i$ , the counters of the monitored nodes are read (i.e., the  $f_{(t_i)}(u)$  values for each monitored  $u$ ) and reset, and the monitoring attributes of the nodes are changed, applying the algorithmic rules below. Note that the old values of  $f_{(t_i)}(u)$  can be stored in slow memory once the time interval  $t_i$  has elapsed.

### B. The EaC algorithmic rules

After each time interval, we adjust the choice of nodes to monitor, according to the following rules (a legend for the rule description figures is given in Fig. 2):

a) *Expand to children*: If a node is monitored and found to be heavily-hit, then monitor all of its children. Stop monitoring that node itself. Formally,  $\mathfrak{M}u_p \wedge \mathfrak{H}u_p \rightarrow_{(t_{i+1})} \neg\mathfrak{M}u_p \wedge (\forall u_c \dot{\subset} u_p : \mathfrak{M}u_c)$  (see Fig. 3).

b) *Collapse to parent*: If a node is monitored and found to be non-heavily-hit, stop monitoring it, and instead start monitoring its parent again. If applied to  $n$  siblings, then we free  $n - 1$  counters. Formally,  $\mathfrak{M}u_c \wedge \neg\mathfrak{H}u_c \rightarrow_{(t_{i+1})} \neg\mathfrak{M}u_c \wedge \mathfrak{M}u_p$ , where  $u_p \dot{\supset} u_c$ .

c) *Prevent oscillation*: Suppose that a node is heavily-hit, but all of its children are non-heavily-hit. In this case, the children are examined using the *Expand* rule. Since all the children are non-heavily-hit, the *Collapse* rule is applied to each, and the parent node is monitored again. Obviously, this scenario easily leads to oscillations. Thus, upon applying the *Collapse* rule, we clear the `follow` flag (set true by default) for the parent that is being monitored again:  $\mathfrak{M}u_c \wedge \neg\mathfrak{H}u_c \rightarrow_{(t_{i+1})} \neg\mathfrak{M}u_c \wedge \mathfrak{M}u_p \wedge \neg\mathfrak{F}u_p \wedge \mathfrak{F}u_c$ ; and the *Expand* rule is only applied to nodes that have the `follow` flag set:  $\mathfrak{M}u_p \wedge \mathfrak{F}u_p \wedge \mathfrak{F}u_c \rightarrow_{(t_{i+1})} \neg\mathfrak{M}u_p \wedge (\forall u_c \dot{\subset} u_p : \mathfrak{M}u_c)$ . The `follow` flag is reset in collapsed children, as we might have to re-examine them at a later time. See Fig. 4 for a graphical representation of the *Collapse* rule combined with the oscillation prevention rule.

After a configurable number of time intervals, we re-set the `follow` flag in the parent, allowing to apply the *Expand* rule again (see Fig. 5). Formally,  $\neg\mathfrak{F}_{(t_i)}u \rightarrow \dots \rightarrow_{(t_i + \text{timeout})} \mathfrak{F}u$ . This is to allow for finding children newly heavily-hit due to changes in the network traffic patterns, to whom the *Collapse* rule had previously been applied.

### C. EaC algorithm iterations

The algorithmic rules for selecting the monitored nodes allow the algorithm to work with a given number of counters



**Fig. 5:** The follow flag timeout. The follow flag is reset after a configurable timeout, to allow for finding newly heavily-hit children, to whom the *collapse* rule has previously been applied.

efficiently. The algorithm iteration consists of the following steps: 1. order the currently monitored nodes by their respective hit counts; 2. start at the top of the list (i.e., the heavily-hit nodes) and apply the rule that requires more counters—the *Expand* rule; 3. free memory for new counters applying the *Collapse* rule, starting at the bottom of the list; 4. goto step 2 and repeat, until the *Expand* and the *Collapse* parts meet (i.e., the expansion cannot use any more collapse rules without having to remove nodes that it just expanded during the same iteration). If an *Expand* operation fails because we could not free enough memory positions, then perform a rollback of this operation and all operations related to it, and terminate prematurely. The next EaC iteration will start at the next time interval. An example of monitoring a very small tree with EaC is shown in Fig. 6.

The value of  $\rho$  and the boundary between heavily-hit and non-heavily-hit nodes is thus not fixed, but rather re-calculated by the algorithm during each iteration, based upon the hit counts and the number of counters available.

If interested in obtaining the precise hit count on every node hit at least by a  $\frac{1}{x}$  fraction of traffic, the EaC iteration must stop at a point where we would have to remove the counter from such a node, i.e., we refrain from applying the *Collapse* rule to nodes that are hit more than  $\frac{1}{x}f(0)$  times (the *collapsing threshold*). See section VI-C for further discussion of this particular task.

#### D. Non-monitored nodes

The purpose of the *EaC* algorithm is to select the monitored nodes. We can obtain hit counts on non-monitored nodes too. The *precise* node hit count can be deduced by summing up the hit counts of its children, if known. This can be done recursively. See theorem VI.4 for more details.

As no node can have more hits than its parent, we can use the precise node hit count as an *upper bound* for all of its descendants. A bound for a node can be tightened by including hit counts for its siblings: For example if a node’s parent is hit by  $z$  hits and its siblings are hit  $\geq x + y + \dots$  times, then the node can be hit at most  $z - x - y - \dots$  times.

## VI. ALGORITHM PROPERTIES

### A. Convergence

In this subsection we prove that the EaC algorithm converges to a stable choice of monitoring nodes. Since convergence is difficult to define in the case of a moving target (i.e. the evolving node hit counts), we analyze the case of a tree whose hit patterns remain constant.

**Theorem VI.1** In the case of constant hit patterns and a large enough timeout value, the EaC algorithm converges.

**Proof** by induction over tree potential (as in [19]): We define the tree potential of the tree and then show that it is monotonically decreasing over iterations.

**Definition VI.1** We define the potential  $\Phi(u)$  of a node  $u$  as

$$\Phi(u) := \begin{cases} 2 & \Leftrightarrow u \text{ has never been monitored yet} \\ 1 & \Leftrightarrow u \text{ is currently being monitored} \\ 0 & \Leftrightarrow u \text{ is not monitored any longer} \end{cases}$$

The potential  $\Phi(\mathcal{N})$  of a set of nodes  $\mathcal{N}$  is the sum of the potential of the nodes:  $\Phi(\mathcal{N}) := \sum_{u \in \mathcal{N}} \Phi(u)$ . This means that the potential  $\Phi(T)$  of a search tree  $T$  is the sum of the potentials of the tree nodes.

### Lemma VI.2

- (a) The initial potential  $\Phi_{(t_0)}(T)$  of any tree  $T$  is finite.
- (b) The potential of a tree is always non-negative, i.e.,  $\forall t_i : \Phi_{(t_i)}(T) \geq 0$ .

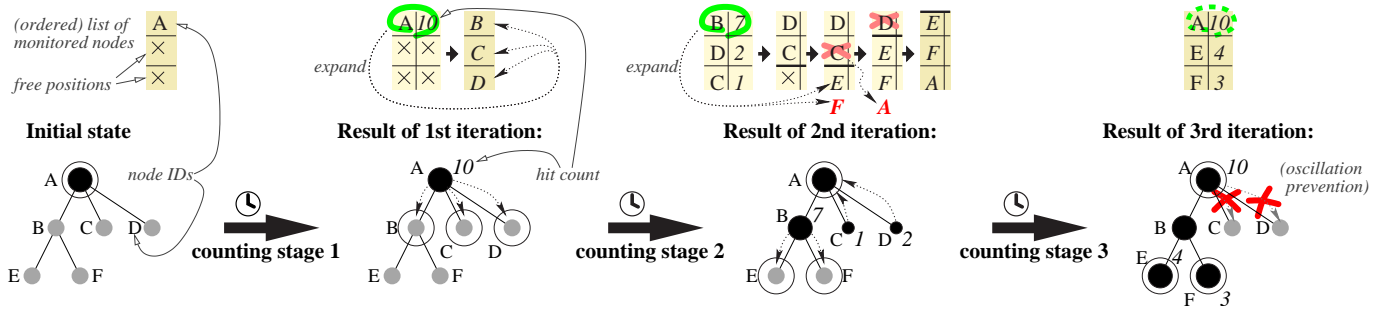
Both statements are obvious, since (a) we operate on finite trees, and (b) the potential is a sum of non-negative numbers.

**Lemma VI.3** The potential  $\Phi(T)$  of any tree  $T$  with constant hit patterns is monotonically decreasing when applying the EaC algorithm to the tree.

**Proof** of lemma VI.3: The potential of the tree only changes at nodes whose monitoring state is changed. This is only possible if these nodes have been affected by one of the rules described in section V. Let  $\Delta\Phi(x) := \Phi_{(t_i)}(x) - \Phi_{(t_{i-1})}(x)$  be the difference in potential of some node  $x$  after an iteration  $t_{i-1} \rightarrow t_i$ , i.e.,  $\Delta\Phi(x) < 0$  means that the potential of  $x$  has decreased. Assume that during one iteration of the algorithm, one application of a single rule has affected the monitoring state of each node in  $\mathcal{N} = \{u, v_1, \dots, v_n\}$  with  $\forall v_i : u \dot{\supset} v_i$ . Now distinguish the following cases of rules being applied:

*Expand:* Monitoring of  $u$  has been expanded to monitor each  $v_i$ . Due to **oscillation prevention**,  $u$  could not have been monitored at an earlier stage, since then we would not have been allowed to expand  $u$  again. This means that none of the  $v_i$  had been previously monitored either. Thus  $\Delta\Phi(\mathcal{N}) = \Delta\Phi(u) + \sum_{i=1}^n \Delta\Phi(v_i) = (0 - 1) + n \cdot (1 - 2) \stackrel{(\text{as } n \geq 2)}{\leq} -3$ .

*Collapse:* Assume w.l.o.g.  $n = 1$ . As  $u$  must have been monitored before, we have  $\Delta\Phi(\mathcal{N}) = \Delta\Phi(u) + \Delta\Phi(v_1) = (1 - 0) + (0 - 1) = 0$ . However, the *collapse* rule is only applied if there is need to free a memory position. This need can only arise in two situations: (a) another node  $\check{u} \dot{\supset} \{\check{v}_1, \dots\}$  has been expanded, or (b) another node  $\hat{v} \dot{\subset} \hat{u}$  has been collapsed, which required freeing the memory position of  $v$  needed by



**Fig. 6:** EaC example iterations with 3 available counters. After the first hit-counting stage, EaC is run on the tree. It expands node A; i.e., nodes B, C, D get monitored. The second hit-counting stage reveals that node B (7 hits) should be expanded, which requires to collapse node C (1 hit). However, this implies monitoring A again, so D (2 hits) needs to be collapsed also. After the fourth iteration, node A has more hits (10) than E (4) and F (3) thus should be expanded; however, this is not allowed if oscillation prevention is in effect.

$\hat{u}$ . In case (b), the *collapse* rule is invoked recursively, but since we are dealing with a finite tree, this only may happen a finite number of times  $k$ . The recursion thus must have been triggered by one *expand* rule (i.e., case (a)). In the end, we have a chain of rule applications, and the potential of all the nodes  $\tilde{\mathcal{N}}$  affected by this rule chain is thus  $\Delta\Phi(\tilde{\mathcal{N}}) = \Delta\Phi(\text{expand rule}) + (k+1) \cdot \Delta\Phi(\text{collapse rule}) \leq -3 + 0$ .

Thus, no matter what rule has been applied, the tree potential  $\Phi(T)$  is always reduced. ■

To summarize,  $\Phi(T)$  starts from a positive finite value, decreases monotonically over the iterations of the EaC algorithm, but remains non-negative. This concludes our proof that the algorithm converges at some point. ■

### B. Precise hit count reconstruction

**Theorem VI.4** If  $v$  is a monitored node, we obtain the precise hit count on  $v$  and all its ancestors.

**Proof:** Assume otherwise. Then there are nodes  $u \supset v$  with  $\neg \mathfrak{M}_{(t_k)} u \wedge \mathfrak{M}_{(t_k)} v$ . Since  $\mathfrak{M}_{(t_k)} v$ , there must have been some previous  $t_i$ ,  $i < k$  such that  $\mathfrak{M}_{(t_i)} u$ . Node  $u$  can only have been ceased being monitored due to either the *Collapse* or the *Expand* rule. The former would result in  $\neg \mathfrak{M}_v$  before  $\neg \mathfrak{M}_u$  (since  $v \subset u$  and thus  $f(v) \leq f(u)$ )—a contradiction. The latter must result in  $u$  being fully covered by its descendants—a contradiction to the assumption that we cannot obtain the precise hit count  $f_{(t_k)}(u)$ . ■

Note that the theorem implies that any path from the root to an arbitrary leaf passes at least one counter, and the monitored nodes thus form a cut across the tree structure.

### C. Hit coverage

A common definition of a heavy-hitter object is one that receives at least  $\frac{1}{x}$  of the total traffic. We now prove a relationship between  $x$  and the number of counters made available to EaC.

Assume that the search pattern remains constant over a number of iterations. Let  $h := \max\{\text{height of the tree}\}$ . Then EaC converges to a state where it provides the precise hit count on all nodes hit by at least  $\frac{1}{x} \cdot f(0)$ , using  $x \cdot h = |M|$  counters.

**Lemma VI.5** The number of nodes hit at least  $\frac{1}{x} \cdot f(0)$  times, and that are deepest down in the tree, is at most  $x$ .

**Proof:** If one node is hit by  $\geq \frac{1}{x} \cdot f(0)$  traffic, then its parent also must be hit by  $\geq \frac{1}{x} \cdot f(0)$  of the traffic. Since the nodes cover disjoint search space areas, there can be at most  $x$  such “deepest” nodes, each attaining  $\geq \frac{1}{x} \cdot f(0)$  hits. ■

**Lemma VI.6** To obtain the *precise* hit count on all nodes that are hit  $\geq \frac{1}{x} \cdot f(0)$  times, we need to monitor at most  $x \cdot h$  counters.

**Proof:** Since we have  $\leq x$  “deepest”  $\frac{1}{x}$ -hit counters, all additional nodes that we have to monitor must be parents of these  $x$  counters. i.e., we have to monitor all nodes along the  $x$  paths from the root to each of these deepest  $\frac{1}{x}$ -hit nodes. The worst case is that these paths already separate immediately below the root, and that  $\text{height}(\text{deepest } \frac{1}{x}\text{-hit nodes}) = h$ . This implies that we need to monitor  $\leq x \cdot h$  nodes in total. ■

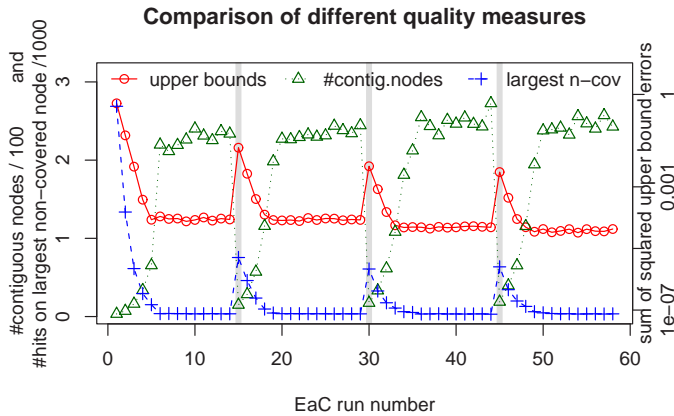
**Theorem VI.7** If  $x \cdot h$  monitors are available, then the EaC algorithm picks all of those nodes that are hit at least  $\frac{1}{x} f(0)$  times, if the hit patterns do not change until the algorithm has converged and if the *collapsing threshold* (V-C) is applied.

**Proof:** Obviously, the *collapsing threshold* guarantees that we never cease monitoring  $\frac{1}{x} \cdot f(0)$ -hits nodes once we have found them; we only may expand them and this way can detect possible  $\frac{1}{x} \cdot f(0)$ -hit children.

EaC converges to a state where it has found *all*  $\frac{1}{x} \cdot f(0)$ -hits nodes, because (a) during each iteration, it always expands those nodes that are hit most frequently, (b) we always obtain precise hit count on any node that is monitored or a parent of a monitored node, (c) EaC does converge. ■

## VII. PERFORMANCE EVALUATION

In this section we analyze the EaC algorithm within a simulator environment. The results yielded by its choice of monitored nodes can then be subject to a statistical evaluation.



**Fig. 7:** The algorithm converges rapidly, and the different quality measures behave consistently with each other.

### A. Evaluation set-up

To analyze its performance, we implemented the EaC algorithm in Java. This implementation is embedded into a simulation framework that generates random requests on the search tree monitored by EaC. At each node, the search requests follow a fixed user-defined probability distribution that changes from time to time.

The trees we use are various uniform trees of fixed out-degree ( $OD$ ) at each node and a fixed depth ( $d$ ), as well as several randomly-built trees. (The latter ones were constructed as follows: minimum depth=4; maximum depth=24;  $OD$  uniformly distributed  $\in 2 \dots 32$  then rounded down to nearest  $2^i$ ; probability for a node to have children=0.75, number of leaves=4096—these numbers were chosen to make the tree roughly resembling a small packet classification tree.) To ease comparison of simulation results on trees that are different in shape but comparable in size, each tree has a fixed number of leaves of either 4096 or 65536. We create 9–20 instances for each tree shape and analyze the average values of the simulation runs in each equivalence class.

After  $5 \cdot 10^5$  requests have been issued, we run EaC on the tree to select a new set of monitors. This is followed by another  $5 \cdot 10^5$  search requests, then another EaC run, etc. We chose a constant number of search requests instead of a random distribution in order to make results from independent EaC iterations easier comparable. Every 15 monitoring intervals, the simulator imposes an entirely new hit pattern distribution at each node. This is done in order to analyze EaC’s reaction to pronounced dynamic changes to the oncoming hit patterns.

### B. Performance measures

Since it provides exact measures for monitored nodes and upper bounds for all other nodes of the tree, one aspect of the performance of the EaC algorithm is the accuracy of these bounds. As a measure, we define the (normalized) sum of squared errors for the upper bounds of each node in a tree  $T$  as  $\mathcal{E}(T) := \frac{\sum_{v \in T} (u(v) - f(v))^2}{|T| \cdot f^2(0)}$ .

Another interesting aspect is EaC’s performance in finding the “heavy-hitters” among the nodes. To this end, we deter-

mine the maximum number of hits on any node for which the algorithm cannot guarantee to yield the precise number of hits. We call this the *largest non-covered node*. As with the error sums, better performance is indicated by smaller values of this measure.

Closely related is the number of nodes that are hit more often than the largest non-covered node. By definition, EaC yields their precise number of hits. We call this measure the *number of contiguously covered nodes*; better EaC performance is indicated by a larger number of this value.

Fig. 7 shows the average of simulation runs on 20 trees ( $OD = 4$ ,  $d = 6$ , 4096 leaves) having different Pareto-distributed hit patterns (EaC had 512 counters available and Oscillation prevention timeout was uniformly distributed with  $\mu = 2$ ). We will refer to this set-up as the *default setup* in the further analyses. This (typical) plot demonstrates that all four performance measures described above show consistent behaviour at the beginning and after each time the simulator installs new hit probabilities. In the following, we thus concentrate on only one performance measure, i.e., the number of contiguously monitored nodes.

### C. Hit patterns, tree topology, tree size

Fig. 8 shows some of the results for different groups of search trees. To compare the effect of different hit distributions on a tree, we simulate different hit patterns on various trees while keeping our default values for all other parameters. A hit pattern is defined as follows: At each node, the probability to continue the search in either child 1 or child 2 or child 3 or (...) is (a) uniformly, (b) exponentially, (c) Pareto-distributed ( $\alpha = 1.3$ ). We notice that the hit distribution has a measurable, but not very pronounced effect on EaC’s performance: The three lines for  $OD = 4$ ,  $d = 6$  are not far apart, with a uniform hit pattern being the worst case (i.e., the line appears more to the left) and exponential the best, Pareto lying in the middle. In the following we thus only investigate Pareto-distributed hit patterns, as flow popularities in the Internet typically are consistent with power laws [9].

Looking at the other lines in Fig. 8, we see that the number of contiguously covered nodes decreases significantly as the branching factor ( $OD$ ) increases: For  $OD = 2$  we get the best coverage, even though a binary tree has more nodes (8191) than, e.g., a tree with  $OD = 16$  (4368 nodes) whose coverage is worse. Consistently, the randomized trees (average  $OD = 7.1$ ) show a better performance than the trees with  $OD = 8$ .

When we increase the size of the tree by a factor of 16 while keeping the same number of available counters, the coverage becomes slightly worse (compare the lines  $OD = 4$ ,  $d = 6$  to  $d = 8$ ; and  $OD = 16$ ,  $d = 3$  to  $d = 4$ ).

### D. Oscillation prevention rule

Next, we analyze the benefits of applying the oscillation prevention rule while keeping all other values at default. In Fig. 9, we analyze EaC oscillation prevention timeout values of 0 (i.e., no prevention), a fixed value of 2, and uniformly distributed timeout values with averages of 2 (as was the

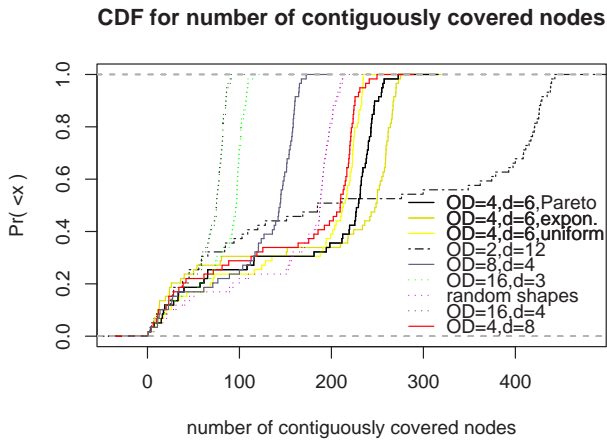


Fig. 8: Influence of tree topology, size, and hit distribution.

setting for all previous simulation runs) and 10. We conclude that a small randomized non-zero oscillation prevention value has a small but measurable positive effect on the efficient use of counting nodes.

#### E. Number of counters

Fig. 9 also shows the results for simulation runs on the default tree, but only with 128 counters available (i.e., only  $\frac{1}{4}$  of those before). As one would expect, the number of contiguously fully-covered nodes remains roughly linear in proportion to the number of available counters.

### VIII. CONCLUSION

The initial validations on simulated data confirm the theoretically derived convergence and coverage properties. It is our intent to further evaluate the EaC performance on search requests resulting from real-life packet traces, applied to a search tree built from an existing rule base, using a state-of-the-art method like HyperCuts [6]. Further understanding is likewise needed in how to adjust seamlessly to changes in the underlying rule-base, or how to employ effectively knowledge from previous algorithmic iterations.

As the restricted search-tree monitoring presents a negligible system overhead, it holds a significant potential for possible applications, one of them being a run-time optimization of the search method itself. However, the gathered data can be useful in a number of ways, for example for identifying heavy flows or detecting rapid changes, failures or attacks in the network.

The presented approach can be generalized to any tree search abiding by the assumptions of downward search without backtracking. As such, this self-monitoring mechanism is a clear step towards developing a fully autonomous packet processing system.

### IX. ACKNOWLEDGMENTS

We thank Prof. Anja Feldmann for her inspiring comments and suggestions. Nils Kammenhuber's work was partly funded by Intel Research and by grant DFG-Schwerpunkt 1126.

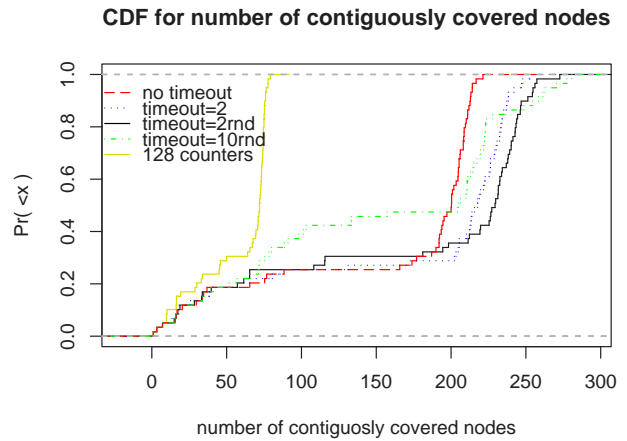


Fig. 9: Influence of number of counters and oscillation prevention.

### REFERENCES

- [1] S. Nilsson and G. Karlsson, "Fast address lookup for Internet router," in *Proceedings IFIP 4th International Conference on Broadband Communications '98*, 1998, pp. 11–22.
- [2] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," *ACM Computer Communication Review*, vol. 27, no. 4, pp. 3–14, October 1997.
- [3] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP route lookups," in *Proceedings of ACM SIGCOMM*, Cannes, France, 1997.
- [4] A. Feldmann and S. Muthukrishnan, "Tradeoffs for packet classification," in *Proceedings of IEEE Infocom*, 2000.
- [5] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Proceedings of Hot Interconnects VII*, 1999.
- [6] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of ACM SIGCOMM*, Karlsruhe, Germany, 2003.
- [7] M. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. Campbell, "Directions in packet classification for network processors," in *Proceedings of the 2nd IEEE Workshop on Network Processors*, 2003.
- [8] A. Feldmann and W. Whitt, "Fitting mixtures of exponentials to long-tail distributions to analyze network performance models," in *Proceedings of IEEE Infocom*, 1997.
- [9] Jörg Wallerich, Holger Dreger, Anja Feldmann, Balachander Krishnamurthy, and Walter Willinger, "A methodology for studying persistency aspects of internet flows," *ACM SIGCOMM CCR*, 2005.
- [10] E. J. Johnson and A. R. Kunze, *IXP2400/2800 Programming*, Intel Press, 2003.
- [11] G. Iannaccone, C. Diot, I. Graham, and N. McKeown, "Monitoring very high speed links," in *ACM SIGCOMM Internet Measurement Workshop*, San Francisco, CA, USA, November 2001.
- [12] A. Moore, J. Hall, E. Harris, C. Kreibech, and I. Pratt, "Architecture of a network monitor," in *Proceedings of the Fourth Passive and Active Measurement (PAM) Workshop*, April 2003.
- [13] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," in *Proceedings of ACM SIGCOMM*, Portland, OR, USA, September 2004.
- [14] K. Papagiannaki, N. Taft, and C. Diot, "Impact of flow dynamics on traffic engineering design principles," in *Proceedings of IEEE Infocom*, Hong Kong, 2004.
- [15] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford, "Netscope: Traffic engineering for ip networks," in *IEEE Network Magazine, special issue on Internet Traffic Engineering*, 2000.
- [16] Anees Shaikh, Jennifer Rexford, and Kang Shin, "Load-sensitive routing of long-lived ip flows," in *Proceedings of ACM SIGCOMM*, 1999.
- [17] L. Kencl and J.-Y. Le Boudec, "Adaptive load sharing for network processors," in *Proceedings of IEEE Infocom*, New York, 2002.
- [18] S. Ramabhadran and G. Varghese, "Efficient implementation of a statistics counter architecture," in *Proceedings of ACM SIGMETRICS*, San Diego, CA, USA, June 2003.
- [19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press, 2nd edition, 2001.