



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

Lukas Kencl; Nils Kammenhuber

IRC-TR-04-034

**Research at Intel**

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © Intel Corporation 2003

\* Other names and brands may be claimed as the property of others.

# 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  
+49-89-289-18035  
kammenhuber@net.in.tum.de

Lukas Kencl

Intel Research  
15 JJ Thomson Avenue  
Cambridge, CB3 0FD, UK  
+44-1223-7-67072  
lukas.kencl@intel.com

\* Author's work partly funded by grant DFG-Schwerpunkt 1126

**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 other favorable properties 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, determining the most potent traffic flows, 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, such a search procedure is executed per every packet and thus needs to be executed within a tight time budget. Examples are a next-hop lookup, a task in which a destination address in the packet is compared to a table of address prefixes using a *longest-prefix match* criterion, or *classification*, where multiple fields in the packet header are compared against rules in the classification table (which may span various ranges in the respective header fields), in order to determine the applying rule with the highest priority.

Efficient implementation of the longest-prefix match [1], [2], [3] and of the rule-based classification [4], [5], [6], [7] tasks has been a subject of extensive research in recent years. It is a common approach in solving these tasks to construct some form of a search tree structure over a pre-processed prefix-table or rule-set, in order to achieve a favorable size vs. search-speed tradeoff. For example, the Hi-Cuts [5] and HyperCuts [6] classification methods construct search trees over the rule base by a heuristic that cuts the search space into subspaces containing approximately equal amount of rules. Traversing the search tree thus corresponds to reaching subsequently smaller subspaces, represented by the nodes of the tree.

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]. Hence, 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. However, this information can be potentially useful in a number of ways, such as 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.

Contrariwise, gathering statistics about the current network traffic has become a standard task in both research and in operational networking environments [9], [10], [11]. Usage of such statistics ranges from analysis of packet flow dynamics on wide-area networks [8] over rules how traffic can be engineered on these networks [12], [13] to traffic-adaptive methods within the network node itself [14], [15].

The principal lessons taken can be summarized as that many quantities characterizing network performance have long-tail probability distributions, which may have a dramatic effect upon performance [8]. 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 thus to treat a large fraction of traffic consistently over time, the particular objects (packet flows) need to be reselected often [12].

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 [16]. It is our goal to efficiently capture the bias of how the oncoming traffic

is distributed over the various paths in the search tree and to determine the current frequently traversed paths. We work with a restricted number of counters to limit the overhead.

The rest of this work is organized as follows: In Section II we pose the problem of efficient gathering of statistics over the tree search method. Then, in Section III, we define the notation to describe the environment, and in Section IV, we present the algorithmic solution. In Sections V and VI, we analyze the algorithm properties and performance, using theoretical tools as well as a simulated environment. Finally, in Section VII, we discuss the open issues and add concluding remarks.

## II. EFFICIENT GATHERING OF STATISTICS

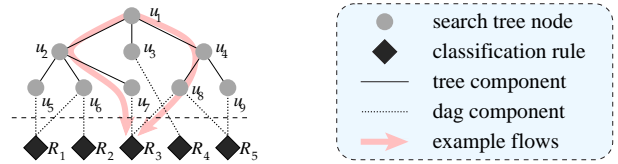
We assume that packet information is processed by performing a tree search using some packet data as the search key, as in the case of a tree-based packet classification algorithm. More precisely, we assume that the search starts from the root of the tree and continues downwards, without any backtracking. HyperCuts [6] (see Fig. 1) is a fine example of such a method, however, our approach can be generalized to any other tree search method abiding by the above criteria. Our *goal* is to determine which paths of the tree structure are traversed frequently, and to measure the *hit count* on nodes belonging to these paths.

We assume a programmable networking system, such as a network processor (NP) [16], 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.

### A. Memory constraints

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 [16] available, yet the rule databases containing several thousands of rules can range from 10k to 100k words [6], [5]. Furthermore, in case the search is performed in parallel on multiple engines, simultaneous memory accesses must be dealt with. Thus, reducing the counting overhead is important to prevent potentially expensive access conflicts.



nodes to compute upper bounds. See Section IV-D for more details.

Given  $x \cdot \max\{\text{height of tree}\}$  counters, in Section V-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. As these are the frequently hit nodes of the tree, it is consistent with the method's objective to obtain just bounds for the remaining nodes.

### III. 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 with the search tree turns the tree into a dag (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, whereas *dag* denotes the graph including the rules.

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

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

**Definition III.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,  $f(u) \geq f(u_p)$ , for some fixed  $p \in [1, n]$ .

In this work, we use the metric of hit counts for determining whether a node should be monitored or not. It may be possible to use other metrics, however, the algorithmic rules in section IV are based upon the assumption that the values generated by the metric are *monotonically decreasing* along any path through the tree, and thus  $(u \subset v \wedge \mathfrak{H}u) \Rightarrow \mathfrak{H}v$ .

Input traffic patterns undergo significant changes over time. The monitoring process must be able to reflect and adapt to these changes. We divide time into intervals  $t_1, t_2, \dots$ , of equal length. As the frequency at which a node is hit changes over

time, we implicitly link any classification of a node being heavily hit to a specific time interval.

## IV. 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. The updating of the counters must be integrated into the traversal of the tree during the search. Due to accessing counters, this is the only part of the algorithm which 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. We 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 recently have become heavily-hit.

To keep track of which nodes are monitored and which not, we augment *all the nodes* of the monitored tree structure by the following fields:

- Each node carries 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:

- A *counter* that counts all the search accesses passing through this node (in fast memory). We denote the number of search operations passing through a node  $u$  during time interval  $t_i$  with  $f_{(t_i)}(u)$ .
- 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

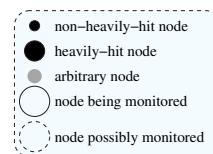


Fig. 2: Legend

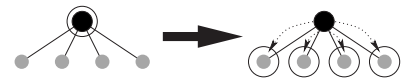
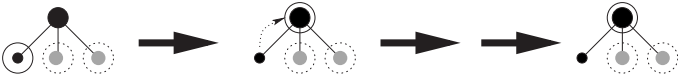


Fig. 3: Expand-to-children rule. If a node is heavily-hit, we start monitoring its children.



**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.

monitored  $u$ ) and reset to zero, 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,  $\mathcal{M}u_p \wedge \mathfrak{H}u_p \rightarrow_{(t_{i+1})} \neg\mathcal{M}u_p \wedge (\forall u_c \dot{\subset} u_p : \mathcal{M}u_c)$  (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,  $\mathcal{M}u_c \wedge \neg\mathfrak{H}u_c \rightarrow_{(t_{i+1})} \neg\mathcal{M}u_c \wedge \mathcal{M}u_p$ , 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, if only the two aforementioned rules are applied. Thus, upon applying the *Collapse* rule, we clear the `follow` flag (set true by default) for the parent that is being monitored again:  $\mathcal{M}u_c \wedge \neg\mathfrak{H}u_c \rightarrow_{(t_{i+1})} \neg\mathcal{M}u_c \wedge \mathcal{M}u_p \wedge \neg\mathfrak{F}u_p \wedge \mathfrak{F}u_c$ ; and the *Expand* rule is only applied to nodes that have the `follow` flag set:  $\mathcal{M}u_p \wedge \mathfrak{F}u_p \wedge \mathfrak{F}u_p \rightarrow_{(t_{i+1})} \neg\mathcal{M}u_p \wedge (\forall u_c \dot{\subset} u_p : \mathcal{M}u_c)$ .

The `follow` flag is re-set in collapsed children, to cater for the case that we might have to re-examine them at a later time when the traffic has shifted. 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 newly heavily-hit children, to whom the *Collapse* rule has previously been applied. Let us assume that a heavily-hit parent has non-heavily-hit children that once were monitored but now are not monitored any longer due to the *Collapse* rule, and the parent does not have the `follow` flag set. Suppose, due to changes in the network traffic patterns, a non-heavily-hit child turns heavily-hit. As the parent is heavily-hit, but does not have the `follow` flag set, without resetting the flag, we would never monitor the child again.

### C. EaC algorithm iterations

The algorithmic rules for selecting the monitored nodes allow the algorithm to work with a given number of counters 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 among the monitored 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 V-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 other, non-monitored nodes too, as follows:

d) *Precise values and upper bounds*: We may obtain the precise hit count on a node by summing up the number of hits on all of its children. This procedure can be continued recursively, as the node is monitored itself or an ancestor of a monitored node. See theorem V.4 for more details.

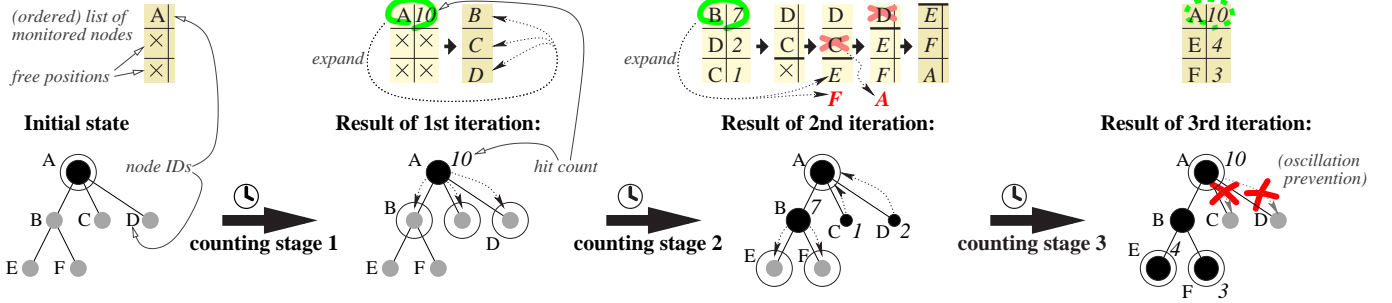
Since no node can have more hits than its parent, we can use the precise number of hits on a node as an upper bound for all of its descendants and compute upper bounds for all descendants of monitored nodes.

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.

e) *Monitoring all nodes*: If desired, we can extend the monitoring to all nodes. Under the assumption that there is only a limited amount of fast memory, we will have to place the counters for the infrequently-hit nodes into slow memory. Naturally, this puts packets passing through non-heavily-hit nodes at a disadvantage, as accessing the counter takes longer.



**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.



**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.

## V. ALGORITHM PROPERTIES

### A. Convergence

In this subsection we prove that the EaC algorithm actually converges to a stable choice of monitoring nodes. Since convergence is difficult to define in the case of a moving target, such as with changing frequencies on the number of hits on the nodes, we analyze the case of a tree whose hit patterns remain constant.

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

**Proof** by induction over the potential of the tree: First we define the potential of the tree and then show that the potential is monotonically decreasing over the iterations. The idea of using a “potential” is due to [17].

**Definition V.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 V.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 V.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 V.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 IV. 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:* The monitoring of  $u$  has been expanded to monitor each  $v_i$ . Note that, 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 in turn means that none of the  $v_i$  had been previously monitored either. Thus  $\Delta\Phi(\mathcal{N}) = \Delta\Phi(u) + \sum_{i=1}^n \Delta\Phi(v_i) = (0 - 1) + n \cdot (1 - 2) \leq -3$  (since  $n \geq 2$ ).

*Collapse:* Assume w.l.o.g.  $n = 1$ . Since  $u$  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  $\hat{v} \dot{\supset} \hat{u}$  has been collapsed, which required freeing the memory position of  $v$  needed by  $\hat{u}$ , or (b) another node  $\check{u} \dot{\supset} \{\check{v}_1, \dots\}$  has been expanded. In case (a) 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 (b)). 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 V.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 V.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 V.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 V.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* (IV-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. ■

## VI. 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.

### A. Performance measures

The algorithm provides exact measures for monitored nodes and upper bounds for other nodes. One aspect of the performance of the EaC algorithm is the accuracy of the bounds. As a measure, we define the (normalized) sum of *squared errors* for the *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 determine the number of hits on the most-hit node for which the algorithm does not yield the precise number of hits. We call this the *largest non-covered node*. A better EaC 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 these nodes.

### B. Evaluation set-up

To analyze its performance, we implemented the EaC algorithm in Java. This implementation is embedded into a simulation framework. It allows us to generate search requests on the search tree monitored by EaC. The *collapsing threshold* (IV-C) is derived automatically from the tree’s maximum depth and node outdegree (OD). Search requests are generated randomly, with fixed probabilities for the branches at each node.

The trees we use are a binary tree with a fixed depth of 16 (65536 leaves, 131071 nodes), a tree with outdegree (OD) 4 and a fixed depth of 8 (65536 leaves, 87381 nodes), a tree with OD=16 and a fixed depth of 4 (65536 leaves, 69905 nodes) and a randomly-built tree (66412 leaves, 132823 nodes) which was constructed as follows: minimum depth=4; maximum depth=24; branching factor  $\in 2^i$ , exponentially distributed between 2 and 32 (mean 4); probability for a node to have children=0.75.

After  $10^6$  requests have been issued, we run EaC on the tree to select a new set of monitors. This is followed by another

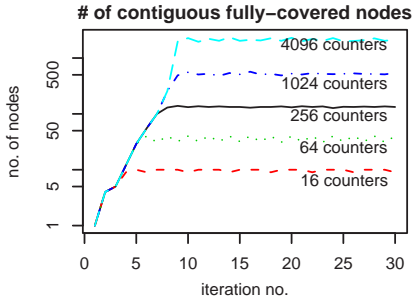


Fig. 7: Number of counters (1).

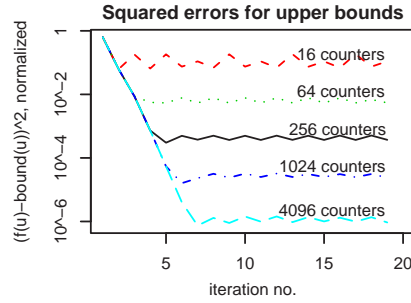


Fig. 8: Number of counters (2).

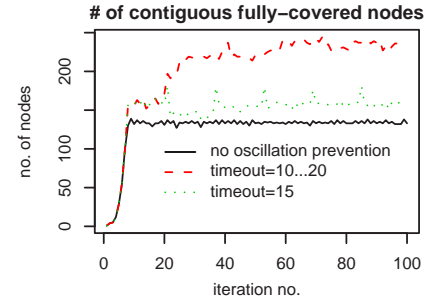


Fig. 9: Influence of oscillation prevention.

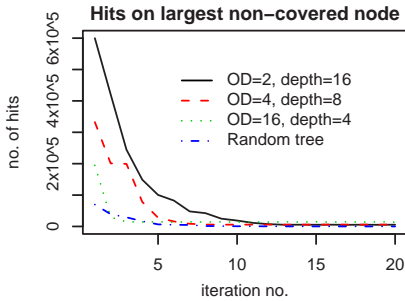


Fig. 10: Effect of different topologies (1).

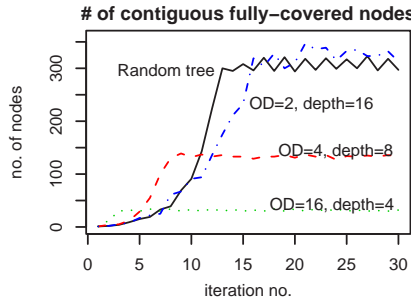


Fig. 11: Effect of different topologies (2).

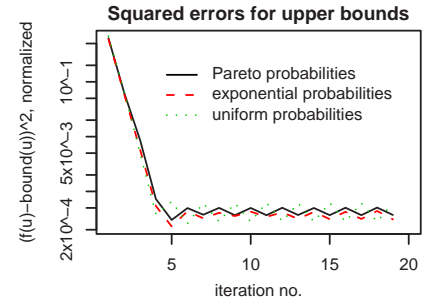


Fig. 12: Effect of different hit distributions.

$10^6$  search requests, then another EaC iteration, etc. We chose a constant number of search requests instead of a random distribution in order to make the results from subsequent EaC iterations easier comparable. Each plot shows results from individual EaC simulations; however there is almost no variation across different simulation runs, even with variations in the parameter sets.

### C. Results

1) *Number of counters*: First, we investigate the benefit from increasing the number of counters available to EaC. Fig. 7 shows results for a simulation run on a tree with 87381 nodes (fixed branching factor of 4, fixed depth of 8, Pareto-distributed hit patterns). We see that the number of contiguous fully-covered nodes is roughly linear with the number of available counters. Fig. 8 shows that the same holds for the quality of the upper bounds. Other simulation runs show that these characteristics are invariant under the choice of other parameters (plots not shown).

2) *Oscillation prevention rule*: Next, we analyze the benefits of applying the oscillation prevention rule. Fig. 9 shows the results for simulation runs with the same simulation setup as above; the number of available counters is 256. We analyze EaC oscillation prevention timeout values of 0 (i.e., no prevention), a fixed value of 15, and uniformly distributed values between 10 and 20. We conclude that the oscillation prevention has a positive effect on the efficient use of counting nodes, at least in the case of non-changing or slowly-changing hit patterns: With randomized timeouts, the number of fully-covered nodes grows to values very near the number of available counters. The recurring spikes in the graph for the fixed-value timeout indicate that synchronization takes place, which explains the bad performance when comparing to random timeouts of the same mean.

3) *Tree topology*: Fig. 10 shows some of the results for test runs on our different trees. We can see that the largest non-covered node receives about the same number of hits in all four different topologies. In contrast, Fig. 11 reveals that the actual number of contiguously covered nodes varies greatly with the topology of the tree. This is to be expected due to the different branching factors.

4) *Hit distribution*: To compare the effect of different hit distributions on a tree, we simulated different hit patterns on our various trees. 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 ( $\alpha = 1.3$ ).

Results for one example tree (again outdegree 4, depth 8, 256 counters) are shown in Fig. 12. We observe that the distribution does neither greatly affect the convergence time of the EaC algorithm, nor the quality of the results (plots only shown for quality of upper bounds).

## VII. CONCLUSION

We have described the problem of obtaining information on the number of hits in a search tree, under the constraint of having only a fixed amount of memory available for holding counters. The presented algorithm iteratively picks a subset of nodes at which accesses are counted. Precise hit counts and upper bounds can be computed for other nodes in the tree, and thus a good view of frequently traversed tree paths can be obtained. An alternate algorithm formulation allows to find all nodes hit by certain fraction of traffic, using a minimal amount of counters.

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.

As such, this self-monitoring mechanism is a clear step towards developing a fully autonomous packet processing system.

#### VIII. ACKNOWLEDGMENTS

We thank Professor Anja Feldmann for her inspiring comments and suggestions..

#### 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 '97*, Cannes.
- [4] A. Feldmann and S. Muthukrishnan, "Tradeoffs for packet classification," in *Proceedings of the Conference on Computer Communications (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 the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. 2003, pp. 213–224, ACM Press.
- [7] Kounavis, Kumar, Vin, Yavatkar, and Campbell, "Directions in packet classification for network processors," in *Proceedings of the 2nd IEEE Workshop on Network Processors*, 2003.
- [8] Anja Feldmann and Ward Whitt, "Fitting mixtures of exponentials to long-tail distributions to analyze network performance models," in *Proceedings of IEEE Infocom 1997*, 1997.
- [9] 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.
- [10] Andrew Moore, James Hall, Euan Harris, Christian Kreibech, and Ian Pratt, "Architecture of a network monitor," in *Proceedings of the Fourth Passive and Active Measurement Workshop (PAM 2003)*, April 2003.
- [11] Cristian Estan, Ken Keys, David Moore, and George Varghese, "Building a better netflow," in *in press*, September 2004.
- [12] K. Papagiannaki, N. Taft, and C. Diot, "Impact of flow dynamics on traffic engineering design principles," in *Proceedings of IEEE Infocom*.
- [13] 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.
- [14] Anees Shaikh, Jennifer Rexford, and Kang Shin, "Load-sensitive routing of long-lived ip flows," in *Proceedings ACM SIGCOMM*, 1999.
- [15] L. Kencl and J.-Y. Le Boudec, "Adaptive load sharing for network processors," in *Proceedings of IEEE Infocom*, New York, 2002.
- [16] Erik J. Johnson and Aaron R. Kunze, *IXP2400/2800 Programming*, Intel Press, 2003.
- [17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press, 2nd edition, 2001.