INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS
Int. J. Commun. Syst. (in press)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/dac.885

Efficient packet classification on network processors

Koert Vlaeminck*,†, Tim Stevens, Wim Van de Meerssche, Filip De Turck, Bart Dhoedt and Piet Demeester

Department of Information Technology—IMEC—IBBT, Ghent University, Gaston Crommenlaan 8 bus 201, Gent B-9050, Belgium

SUMMARY

Always-on networking and a growing interest in multimedia- and conversational-IP services offer an opportunity to network providers to participate in the service layer, if they increase functional intelligence in their networks. An important prerequisite to providing advanced services in IP access networks is the availability of a high-speed packet classification module in the network nodes, necessary for supporting any IP service imaginable. Often, access nodes are installed in remote offices, where they terminate a large number of subscriber lines. As such, technology adding processing power in this environment should be energy-efficient, whilst maintaining the flexibility to cope with changing service requirements. Network processor units (NPUs) are designed to overcome these operational restrictions, and in this context this paper investigates their suitability for wireline and robust packet classification in a firewalling application. State-of-the-art packet classification algorithms are examined, whereafter the performance and memory requirements are compared for a Binary Decision Diagram (BDD) and sequential search approach. Several space optimizations for implementing BDD classifiers on NPU hardware are discussed and it is shown that the optimized BDD classifier is able to operate at gigabit wirespeed, independent of the ruleset size, which is a major advantage over a sequential search classifier. Copyright © 2007 John Wiley & Sons, Ltd.

Received 23 May 2006; Revised 1 February 2007; Accepted 9 February 2007

KEY WORDS: packet classification; firewall; binary decision diagram; network processor

1. INTRODUCTION

Offering a myriad of new IP multimedia and communicational services to an increasing number of subscribers has triggered an evolution of the architectural model of access networks towards multi-service and multi-provider networks. Ethernet as well as full IP alternatives have been investigated as viable connectionless successors for the legacy ATM-based platforms. While the introduction of Ethernet up to the edge solves some of the problems of existing access networks, new ones are

*Correspondence to: Koert Vlaeminck, Department of Information Technology—IMEC—IBBT, Ghent University, Gaston Crommenlaan 8 bus 201, Gent B-9050, Belgium.
†E-mail: koert.vlaeminck@intec.ugent.be

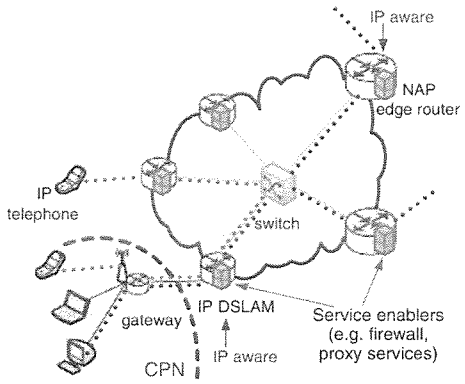


Figure 1. Evolution towards an IP-aware multi-service, multi-provider access network (NAP: network access provider; CPN: customer premises network; DSLAM: DSL access multiplexer).

created. Per subscriber traffic segregation and the lack of Quality of Service (QoS) support are the main issues of standard Ethernet. Although VLANs could alleviate these shortcomings, it can be questioned whether this approach is sufficiently scalable for larger access network deployments. Therefore an IP-aware network model, as depicted in Figure 1, is often considered a valuable alternative [1].

Deployment of next generation network services in such multi-provider, multi-service access networks depends on the ability of the access and aggregation nodes to perform packet classification at wirespeed. Packet classification is a key enabler for a wide variety of services, such as QoS support, network accounting, policy enforcement and network security (e.g. firewall, intrusion detection), and is often the primary bottleneck [2, 3]. This paper explores several packet classification techniques and proposes and evaluates a Binary Decision Diagram (BDD) based classifier, comparing it to the straightforward approach of sequentially searching through the classification ruleset.

Next to an efficient classification algorithm, deploying wirespeed packet classifiers in IP access and aggregation nodes requires additional processing power. Traditionally, telecom equipment vendors have used fixed-function application-specific integrated circuits (ASIC) to cope with the huge performance requirements of today's network systems. Such ASICs provide little or no flexibility to introduce new protocols or services on existing hardware. At present, the ever-changing requirements of network equipment ask for flexible solutions with assured time to market. As opposed to ASICs, general-purpose processors certainly meet the flexibility criteria for implementing modern network services. However, they often fail to meet the performance requirements of these services or consume too much power—hence generate too much heat—for integration in large telecom systems. For this reason, a hybrid hardware device, called network processor unit (NPU), has emerged over the last few years. Network processors are highly parallel, programmable hardware, combining the low cost and flexibility of a RISC processor with the speed and scalability of custom silicon (i.e. ASIC chips) [4].

The remainder of this paper is structured as follows. First, network processor technology is discussed in Section 2. Then, Section 3 provides an overview of state-of-the-art packet classification

DOI: 10.1002/dac

algorithms and highlights opportunities for NPU implementations. After a discussion of the firewall application architecture, Section 4 integrates the BDD classifier and presents several BDD space optimizations. Section 5 discusses the performance evaluation of the sequential search and BDD packet classifiers. Finally, in Section 6, concluding remarks and possible directions for future research are presented.

2. NETWORK PROCESSOR TECHNOLOGY

Although vendors share the general concept of a network processor, the specifics vary considerably. There is no agreement about which hardware building blocks to include in a network processor, exactly what hardware functions to replicate or how to organize the components on a network processor chip [4].

A single network processor includes many physical processors that must work together to perform the necessary network services, e.g. packet filtering and network address translation. Network processors can integrate embedded-RISC processors for handling higher protocol layers and control processing, specialized co-processors optimized for a particular packet processing task, I/O processors, switching fabric interfaces, memory interfaces, etc. This processor hierarchy is complemented by a memory hierarchy, where each memory type offers a specific trade-off between size and speed. The choice of which data to store in which type of memory is left to the programmer [4].

The Intel IXP2400 [5] was selected as a target platform, since its hardware architecture is highly compatible with a modular firewall design, as will be demonstrated in Section 4.1. In the literature, the Intel IXP network processor family is a popular platform for evaluating a wide range of network applications [6–10] and it is even used as a reference platform for evaluating new hardware designs [11]. Hardware details of the IXP2400, together with its Radisys ENP-2611 evaluation board [12], are described in Section 2.1. An overview of the programming model is provided in Section 2.2.

2.1. The Intel IXP2400

The Radisys ENP-2611, depicted in Figure 2, is a 64-bit PCI board, equipped with an Intel IXP2400 network processor. The major IXP2400 processing units are the XScale control processor and eight microengines (MEs), organized into two clusters of four.

The Intel XScale core is a general-purpose 32-bit RISC processor, running at 600 MHz. The MEs are small RISC processors, also running at 600 MHz, with an instruction set specifically tuned for processing network data. Each ME has support for eight hardware threads. All MEs have an independent instruction store large enough for 4K, 40-bit instructions, which is initialized by the XScale core before the ME starts running. The instructions are executed in a six-stage pipeline. Each ME also has 640 long words (or a total amount of 5 kB) of low-latency local memory.

The IXP2400 also has a Hash unit and support for three types of memory, shared between the different processing units: 16 kB of on-chip scratchpad memory, SRAM, and DRAM. The Radisys ENP-2611 board provides 8 MB quad data rate (QDR) SRAM and 256 MB of double data rate (DDR) DRAM. Furthermore, it connects three gigabit Ethernet (GbE) interfaces to the IXP's Media and switch fabric (MSF) interface.

DOI: 10.1002/dac

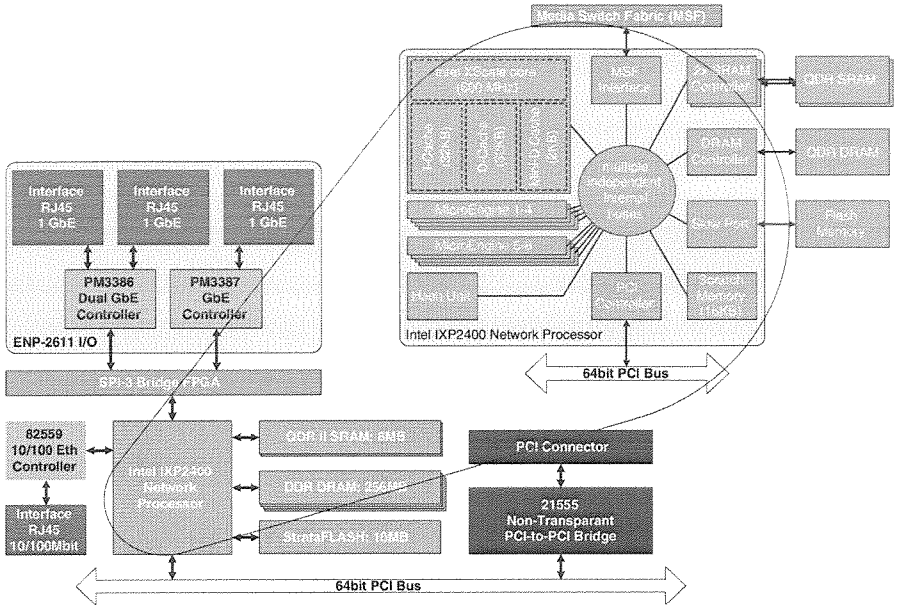


Figure 2. The Radisys ENP-2611 evaluation board, equipped with an Intel IXP2400 network processor [4, 5, 12].

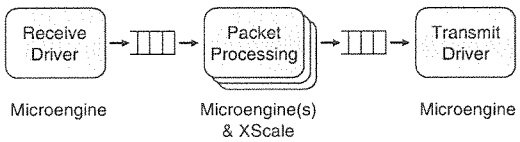


Figure 3. The basic structure of receiving, processing and transmitting packets on an Intel second generation network processor [4].

2.2. *Programming an Intel IXP2400*

Intel second generation network processors are programmed using the receive—process—transmit paradigm [13] as depicted in Figure 3: receive, process, and transmit tasks run on different processing units (MEs and/or XScale core) with queues between them. The MEs can run a series of sequential processing tasks (pipelined approach), a pool of parallel processing tasks or a mixture of both.

Typically, the MEs handle all (or most) of the data plane processing, while the Intel XScale core handles exception packets that require more complex processing (e.g. control and configuration packets). Furthermore, the XScale core is used to recover from erroneous conditions and provides configuration and management access for the entire application.

DOI: 10.1002/dac

3. PACKET CLASSIFICATION, A VERSATILE SERVICE ENABLER

As motivated in the introduction, line rate packet classification is a prerequisite for the emergence of advanced network services in IP access, aggregation and transit networks. The remainder of this paper focuses on a firewall service to evaluate different packet classification mechanisms in a network processor context. This means two possible actions are considered for each rule, either ACCEPT or DROP. Furthermore, classification fields of interest are: IP source and destination addresses, IP protocol, TCP or UDP source and destination ports, TCP flags, and ingress and egress interface.

Efficient packet classification based on multiple header fields is a hard problem. The simplest approach, a sequential search through a rule set containing N rules, has $O(N)$ time and space complexity. More complex algorithms improve the search time, but require advanced and larger data structures. The taxonomy depicted in Figure 4 roughly relates time and space complexity for state-of-the-art algorithms described in the literature. These algorithms—together with optimizations—are: sequential (linear) search, Bit Vector (BV) and Aggregated BV (ABV) [14], Hierarchical Cuts (HiCuts) [15] and HyperCuts [16], Recursive Flow Classification (RFC) [17] and Hierarchical Space Mapping (HSM) [18], and finally BDDs [19]. In order not to overload the figure, the optimized versions of the algorithms are not shown. The following paragraphs briefly touch upon the key properties of each algorithm and explore opportunities for network processor implementations. The remainder of this paper then focuses on network processor implementation approaches and optimizations for the two algorithms on the opposite ends of the taxonomy: *sequential search* and *BDD* classification. Sequential search offers the lowest space complexity but is also the slowest, while BDD classification is the most time-efficient, requiring a constant execution time, provided the classifier fits into memory.

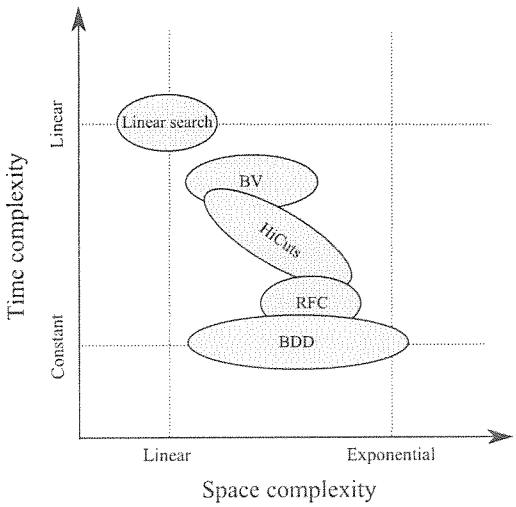


Figure 4. Taxonomy relating the time and space complexity for state-of-the-art packet classification algorithms.

DOI: 10.1002/dac

3.1. Sequential search

A straightforward solution to packet classification is a sequential traversal of the rule database. As mentioned, both search time and memory requirements scale linearly with the number of rules in the classifier. In practice, the sequential search method may be effective if the number of rules is small or when the classifier is preceded by a stateful component (where only the initialization packets of a session need sequential classification). A clear administrative advantage of this approach is that the classifier can be altered easily at runtime by adding or deleting rules.

For any target platform, the performance of the sequential search will be bounded by CPU and memory speed and degree of parallelism. Later in this paper, the bottlenecks related to memory access time (for different types of memory) and CPU power for the Intel IXP2400 network processor are identified.

3.2. BV and ABV

The BV algorithm constructs separate tries for each of the L packet classification fields. In each of the trie nodes, an N-bit vector indicates which rules match. By performing a longest prefix match lookup in each trie, L N-bit vector are retrieved. Next, combining these vectors by a bitwise AND operation yields the first rule matching the current packet. ABV further optimizes this algorithm by rearranging the rule order and aggregating a number of bits in the N-bit vector. This way, unnecessary bit operations can be eliminated and the number of memory accesses can be reduced. Despite this optimization, BV and ABV are less suitable for NPU-based implementations of medium size or large classifiers, because their memory word size is too small (for the IXP2400, the word size is 32 bits) to accommodate the algorithm requirements, resulting in a large number of memory accesses per packet. For relatively small classifiers with a small number of classification fields L, this is a viable approach, however.

3.3. Hicuts and hypercuts

The HiCuts algorithm constructs a decision tree where each node (except the leaves) represents a cut of the search space in one out of L classification dimensions, such that the total number of rules in each leaf node does not exceed a threshold T. As such, packet classification comprises a descent of the decision tree and a sequential search through at most T rules. HyperCuts optimizes the HiCuts algorithm by allowing multi-dimensional cuts. The worst case space complexity of these approaches is $O(N^L)$, but fortunately this behaviour is not noticed for real ISP classifiers, due to their structural properties [17]. By balancing the size of the decision tree and the threshold T, a successful mapping to an NPU architecture can be achieved. For the IXP2400, the decision tree could be stored entirely in the microcode store (thereby limiting the number of memory accesses), whereafter $O(T)$ memory accesses suffice to find the (first) matching rule.

3.4. RFC and HSM

Similar to BV and ABV, RFC is a decomposition algorithm searching for matches for each of the classification fields L prior to joining the results together. The execution time of this algorithm is constant, requiring a fixed number of memory accesses for each packet. Unfortunately, the algorithm is relatively space-inefficient and the number of (parallel) memory accesses cannot be reduced by incorporating (part of) the data structures in the NPU instruction store. For the IXP2400, an implementation in memory requires too many memory accesses per packet, making

DOI: 10.1002/dac

Table I. Number of boolean variables necessary for each classification criterion.

Property	Number of variables
IP source address	32
IP destination address	32
IP protocol	8
TCP/UDP source port	16
TCP/UDP destination port	16
TCP flags	6
Ingress interface	8
Egress interface	8
Total	126

the algorithm less suitable for this kind of architecture. HSM is an optimized version of RFC, but focuses on space optimization rather than reducing the number of memory accesses.

3.5. BDD

When a ruleset is mapped onto a BDD, the (total) aggregated number of bits K from the classification header fields instead of the ruleset size determines packet classification time. Each bit corresponds to a boolean variable and at most K comparisons are needed to classify a packet. Table I summarizes the number of bits (hence variables) necessary for the classification fields used in this paper. Since each variable can take two values, the worst case space complexity of a BDD is $O(2^K)$. Fortunately, this is a theoretical upper bound and in practice many BDD applications exhibit linear memory utilization [20]. Typical characteristics of real ISP classifiers [17] strengthen this statement for our application field. Nevertheless, implementing a BDD classifier on an NPU requires considerable space optimization. K sequential memory accesses per packet would clearly disallow using this approach for wire speed packet classification, even for the fastest type of memory available. Therefore, memory accesses are avoided by implementing static, hardcoded BDDs in the—limited—network processor MEs' instruction stores. In Section 4, space optimization techniques and BDD parallelization are presented. This section briefly discusses the basic BDD concepts.

The process of transforming a traditional ruleset into a BDD can be clarified by detailing each of the following subtasks:

(i) creating a logical expression representing a single rule;
(ii) representing this boolean function by means of a BDD;
(iii) tying together all the single-rule BDDs into a global BDD reflecting the entire ruleset.

Task (i) is accomplished by representing each classification field by a number of boolean variables, equal to the corresponding number of bits in Table I. This way, the IP source address would be represented by 32 variables, for example. Figure 5 illustrates how a single criterion can be mapped to an equivalent boolean representation. For several criteria it makes sense to define a *range* of

DOI: 10.1002/dac

The criterion expressing the protocol in the IP header field of a packet should match TCP (protocol number 6), can be converted to the following boolean formula:

$$\neg \mathtt{pr}_0 \wedge \neg \mathtt{pr}_1 \wedge \neg \mathtt{pr}_2 \wedge \neg \mathtt{pr}_3 \wedge \neg \mathtt{pr}_4 \wedge \mathtt{pr}_5 \wedge \mathtt{pr}_6 \wedge \neg \mathtt{pr}_7$$

The decimal value 6 (boolean 110) is represented using eight variables, because this equals the number of bits reserved for the protocol field in the IP header.

Figure 5. Boolean expression generation.

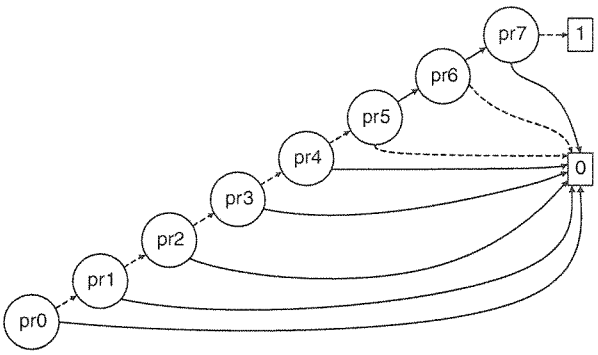


Figure 6. A BDD representing the boolean expression in Figure 5: a dashed arrow points to the 'false path' of a variable and a solid arrow to the 'true path.' This BDD maps TCP packets to terminal 1, while IP datagrams carrying another protocol are mapped to terminal 0.

matching values, instead of a single matching value as in Figure 5. When this range is expressed using a *bit mask*, translation into a boolean expression is relatively straightforward because variables corresponding with zero bits in the mask can simply be ignored, whereafter a translation similar to the one in Figure 5 can take place. Arbitrary ranges are more difficult to express, as they require a boolean representation for *greater than*. Equation (1) presents a recursive formula evaluating the expression *x greater than y*:

$$gt(x, y, r) = \begin{cases} x[r] \vee gt(x, y, r-1) & \texttt{if } \neg y[r] \\ x[r] \wedge gt(x, y, r-1) & \texttt{if } y[r] \end{cases} \tag{1}$$

In Equation (1), $x[r]$ denotes the most significant bit of the *r—bit* value x.

For item (ii), mapping a boolean expression to a BDD, the approach described in the work of Bryant [21] was followed. An example BDD is depicted in Figure 6.

Until now, the action (DROP or ACCEPT) associated with a rule and tying rules together has not been taken into account. For task (iii), tying together the single-rule BDDs, the algorithm presented in [19] is applied, but adapted to cope (explicitly) with a firewall default policy.

DOI: 10.1002/dac

Given a set of rules $R = \{r_1, r_2, \ldots, r_n\}$, a boolean expression τ_0 representing the entire ruleset can be constructed using the following rules:

$$\tau_n = \begin{cases} \text{false} & \texttt{if default policy is DROP} \\ \text{true} & \texttt{if default policy is ACCEPT} \end{cases} \tag{2}$$

$$\tau_{i-1} = \begin{cases} r_i \vee \tau_i & \texttt{if } r_i \cdot \text{action} = \texttt{ACCEPT} \\ \neg r_i \wedge \tau_i & \texttt{if } r_i \cdot \text{action} = \texttt{DROP} \end{cases} \tag{3}$$

If $\tau_0 = \text{false}$ the packet should be dropped, otherwise ($\tau_0 = \text{true}$) the packet should be accepted.

4. A FIREWALL SERVICE WITH OPTIMIZED NPU CLASSIFIER

In order to evaluate the sequential search and BDD packet classification mechanisms discussed in this paper, a firewall service is implemented on an Intel IXP2400 network processor. The implementation is based on the netfilter forwarding chain. Netfilter and iptables are building blocks of a framework that enables stateful packet filtering, network address (and port) translation and other packet mangling inside the Linux kernel [22]. A stateful firewall not only increases security, but packet processing speed may also increase, since the actual packet classification can be avoided for packets belonging to a registered connection. Stateful packet inspection is enabled by a connection-tracking component. The operation of the IXP2400 firewall is described by the following pseudo-code:

```
1.  if conntrack status == INVALID then
2.     drop packet and stop
3.  else if conntrack status == ESTABLISHED, RELATED then
4.     forward packet and stop
5.  else if conntrack status == NEW then
6.     classify packet
7.     if DROP then
8.        conntrack status ⟵ INVALID
9.        drop packet and stop
10.    else if ACCEPT then
11.       conntrack status ⟵ ESTABLISHED
12.       forward packet and stop
13.    end if
14. end if
```

Implementation choices are described in the remainder of this section. First, the sequential search packet classification mechanism is considered in Section 4.1. In Section 4.2, the integration of the BDD packet classifier is discussed.

4.1. An IXP2400 firewall service

Next to connection tracking and packet filtering, a firewall implementation requires a routing component and ingress and egress processing. Mapping these components to the IXP2400's processing

DOI: 10.1002/dac

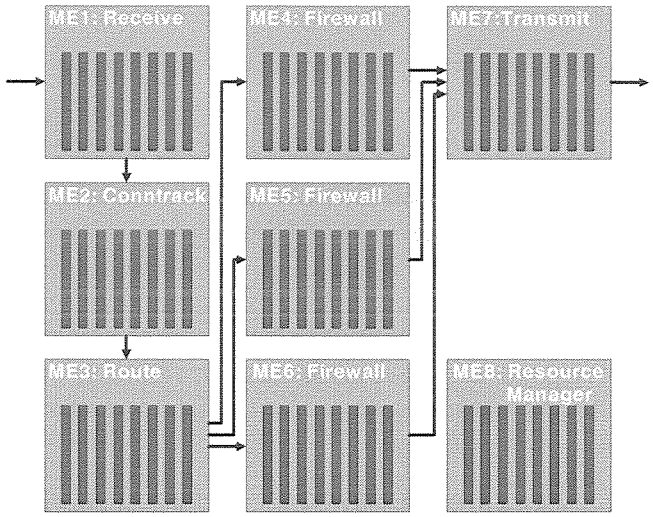


Figure 7. Mapping the firewall software components on IXP2400 hardware.

units is relatively straightforward, as depicted in Figure 7. Receive and transmit code each occupy one ME (ME1 and ME7, respectively). A third ME (ME8) is used for resource management. Running the resource management code on a separate ME has a number of advantages. First, the amount of code on the other MEs is reduced, putting less strain on the limited instruction store per ME. Furthermore, a synchronization between the different MEs is simplified and does not influence the data plane packet forwarding performance. The resource management ME performs tasks like memory allocation for new conntrack entries, searching for expired conntrack entries (a conntrack expires when no packets belonging to its connection arrive for a certain amount of time) and freeing memory.

The five remaining MEs can be used for packet processing. Connection tracking occupies ME2: connection tracking identifies the flow a packet belongs to (based on source/destination address and port) and searches this flow in the conntrack hash table. If the flow is unknown, a new entry will be added to the table. After that the packet will be passed to ME3 for routing. The packet filtering code (which contains the classifier) can run on the remaining three MEs (ME4, ME5, and ME6) in parallel. Each of the available packet filtering threads (up to $3 \times 8 = 24$ threads) processes a different packet.

4.2. Integrating the BDD packet classifier

When a ruleset is transformed into a BDD (see Section 3.5), this BDD packet classifier can replace the sequential search packet classification from the IXP2400 firewall implementation described above. This BDD classifier is again preceded by a stateful block and can be deployed on three MEs.

Two operational firewall rulesets were selected for the BDD packet classifier study: a regular ruleset consisting of 87 handcrafted rules and a complex ruleset with 322 very specific handcrafted

DOI: 10.1002/dac

```
label#:   BR_BSET[reg,bitnr,truelabel#]
          BR_BCLR[reg,bitnr,falselabel#] (or BR[falselabel#])
```

Figure 8. Microcode representation of a BDD node. If bit *bitnr* in register *reg* is set, the instruction at *truelabel* is executed, if not the instruction at *falselabel* is executed.

rules. The regular ruleset was taken from the operational firewall of a medium-scale company, serving a network with approximately 240 workstations (accessing typical Internet applications such as Web browsing, FTP, e-mail, Skype, etc.) and 10 servers (used for e-mail, mailinglists, web, firewall, proxy, routing, and file sharing purposes). The complex ruleset was taken from our research network firewall, serving a regular subnet with workstations (also running typical Internet applications and some additional applications such as CVS) and some servers (including a VPN server), a student network with limited access, a demilitarized zone (DMZ) and the actual test network, consisting of several tens of test beds (over 200 machines), each test bed requiring its own access profile. For privacy reasons, no further details on the rulesets can be published. However, a comparison of the selected rulesets to Gupta and McKeown's packet classifier characteristics indicates they are certainly representative and actually quite large. After analysing almost 800 classifiers from over 100 different ISP and enterprise networks, Gupta and McKeown found that only a small percentage of classifiers contained more than a few hundred rules, with a mean of only 50 rules [17].

Details on how to implement a BDD classifier in microcode in order for it to run on an IXP ME are presented in Section 4.2.1. Then, Section 4.2.2 discusses how the BDD classifier can be split over multiple MEs and evaluates some heuristics for reducing a BDD's size. Finally, Section 4.2.3 introduces and evaluates an algorithm for optimizing the microcode generation process.

4.2.1. Mapping the BDD to the IXP architecture. Each node of the BDD can be represented by two conditional branch instructions: either the *true* or *false* branch of the node is followed, based on the value of the particular boolean variable. As depicted in Figure 8, this can be mapped to two microcode instructions per BDD node. In the leaf nodes (either ACCEPT or DROP), additional instructions take care of the actual processing of each packet.

A BDD for a boolean expression in n variables yields a final result after at most n branches (each variable is evaluated at most once on each path from the root to a leaf node). Hence, *the packet classification time is independent of the number of rules*. Table I indicates $n = 126$ for the firewall discussed in this section.

The limitation encountered when adopting the BDD approach for packet classification is memory related. In a worst-case scenario, a boolean function in n variables has a BDD space complexity of $O(2^n)$. For $n = 126$, such a BDD is obviously too large to be of any value for the embedded application in mind. Fortunately, many realistic boolean functions have a BDD representation with a number of nodes that is linear in the number of variables if a 'good' variable order is chosen [20]. Unfortunately, finding the optimal variable order is also known to be an NP-complete problem [21]. Nevertheless, several optimization heuristics [23] often yield an acceptable to near-optimal reordering solution.

Returning to the target platform, the IXP2400, the *ME instruction store memory size* is the single most important hardware parameter when investigating BDD deployment feasibility. On an

DOI: 10.1002/dac

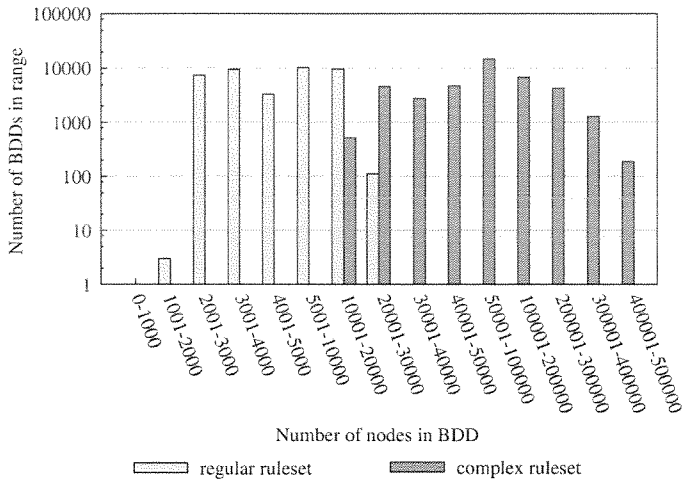


Figure 9. Distribution of the BDD sizes for all 8! variable-group permutations for a regular and complex ruleset.

IXP2400, each ME can store up to 4K instructions. By consequence, the maximum size of a BDD is restricted to 2000 nodes, provided that each node requires two microcode instructions. In this paper, in order to find a near-optimal variable order, boolean variables are grouped per property (see Table I) whereafter all possible inter-variable-group permutations are computed. This can be considered as a *reduced brute force* heuristic in which $8! = 40\,320$ (out of $126! \approx 2 \times 10^{211}$) possible orders are examined. Results for both selected rulesets are depicted in Figure 9.

As shown in Figure 9, a BDD for the regular ruleset—with near-optimal variable order—fits in the instruction store. Unfortunately, for the complex ruleset even an optimal variable order cannot reduce the BDD to such an extent it fits within the ME instruction store.

In the remainder of this section, this issue is tackled from two totally different—but complementary—angles: first two methods to divide the BDD in smaller parts are investigated, then the BDD-independent microcode generation process is optimized.

4.2.2. Multi-processing binary decision diagrams. In the discussion above, a BDD representing an entire ruleset is mapped into the instruction store of a single ME. However, as depicted in Figure 7, the application architecture provides three MEs that can be employed for packet classification. This time, it is the ME instruction store size limitation rather than the packet classification execution time that leads to the exploitation of the IXP2400 parallelism features.

Smaller BDDs can be obtained in two fundamentally different ways:

(a) by decomposing an existing BDD in several smaller parts;
(b) by partitioning the ruleset (or boolean expression), whereafter distinct BDDs are generated for each part.

The first approach requires no *a priori* knowledge of the underlying boolean expression and is application independent. This path is investigated in several research publications, including [24],

DOI: 10.1002/dac

and is not further explored here. For the indirect BDD decomposition strategy (b), again two options arise:

(i) each part can be represented by a regular BDD;
(ii) each part can be represented by a multi-terminal BDD (MTBDD).

Evidently, both representations require—minor—extensions to the BDD generation process outlined in Section 3.5. For technique (i), Equations (2) and (3) need further investigation. ACCEPT rules are merged with the remainder of the ruleset by applying boolean OR ($\vee$), whereas deny rules are merged using boolean AND ($\wedge$). Therefore, because of operator precedence rules, deny rules are propagated to all rules following the deny rule, while ACCEPT rules are not. For partitioning the ruleset, it therefore suffices to copy the deny rules from all previous fragments and insert them *before* the rules of the current partition. Once the ruleset is properly partitioned, a separate BDD can be constructed for each part. Additionally, each partition except the last one should apply the *default DROP* policy. The last fragment should adopt the default policy of the initial ruleset. This way, intermediate results F_i—from all fragments except the last one—yield 'false' for packets that match either an explicit DROP rule or the default DROP policy. Since DROP rules are copied to all subsequent fragments, the global result F_{BDD} for k fragments can be computed as follows:

$$F_{\mathrm{BDD}} = F_1 \vee F_2 \vee \cdots \vee F_k$$

Technique (ii) is very similar to (i), but generates a ternary—instead of a binary—result in each fragment but the last. The ternary set is defined as follows:

$$S = \{\mathrm{DROP}(=0),\ \mathrm{CONTINUE}(=1),\ \mathrm{ACCEPT}(=2)\}$$

Here, each regular rule yields DROP or ACCEPT and the default policy of all fragments but the last is set to CONTINUE. Again, the last fragment adopts the default policy of the initial ruleset. Since this type of decision tree should be considered as a function $f_{\mathrm{MTBDD}} : S^n \rightarrow S$ instead of $f_{\mathrm{BDD}} : \mathbb{B}^n \rightarrow \mathbb{B}$ as for a regular BDD, the operators *and* ($\wedge$) and *or* ($\vee$) need to be redefined as follows:

$$a \wedge b \longrightarrow \mathrm{Min}(a, b)$$

$$a \vee b \longrightarrow \mathrm{Max}(a, b)$$

The global result F_{MTBDD} equals the result of the *first* partition yielding a *decisive* result, either DROP or ACCEPT. Contrary to technique (i), DROP rules are not transferred to subsequent fragments. Note that for a default DROP policy, the number of DENY rules is usually relatively small (they define exceptions to exception rules) and no further BDD reduction over technique (i) is achieved.

Figure 10 depicts results for both techniques, applied to the complex ruleset. There are two remarkable observations: (i) both splitting approaches generate approximately the same number of nodes and (ii) the aggregated number of nodes decreases drastically as the number of fragments increases.[‡] Since the number of nodes per fragment is still greater than 2000, the (MT)BDD code

[‡]Therefore, splitting a BDD into multiple smaller BDDs could also be used to fit a large ruleset in a single ME's instruction store. However, the execution time of the classifier increases linearly with the number of parts. Performance with multiple BDD classifiers on a single ME converges to the performance of the sequential search classifier as the number of parts approaches the number of rules in the set.

DOI: 10.1002/dac

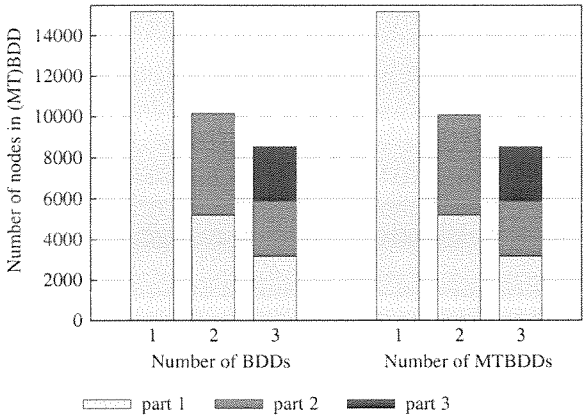


Figure 10. Approach (i) and (ii) results for the complex ruleset. Results are shown for the ruleset divided in two and three (MT)BDDs, respectively. Histogram bars show the average sizes of the best 16 results from the reduced brute force approach.

size still exceeds the capacity of the MEs' instruction store, even for a fragmentation in three parts. Therefore, further optimization by means of standard CUDD [23] heuristics (i.e. simulated annealing, symmetric sifting, and group sifting) is performed, starting from the results obtained by the reduced brute force approach. By unbundling the variable groups, a local optimum can be found. Results for the best performing heuristics are illustrated in Figure 11. Detailed descriptions and references for the applied CUDD heuristics can be found in the CUDD documentation [23].

4.2.3. Microcode optimization. The various optimizations and splitting efforts presented previously are necessary in an attempt to fit the global BDD classifier into three MEs' instruction store memory. Figure 11 shows that the outcome is satisfactory for parts 2 and 3, but part 1 of the ruleset remains too large to fit on one ME. Therefore, this section optimizes the microcode generation process itself by reducing the number of branch instructions per node.

As described in Section 4.2.1, one non-optimized node consists of two explicit branch instructions. However, listing nodes in a particular order can render several explicit branch instructions superfluous, since they can be replaced by an implicit *fall through*. Minimizing the number of instructions is again an NP-complete problem with a worst-case complexity of $O(2^n!)$ (n stands for the number of variables in the BDD). Fortunately, by subtracting paths from the BDD in a well-defined way, a near-optimal reduction can be achieved. A detailed approach is outlined in the following algorithm:

1. $v \longleftarrow$ empty vector
2. $n \longleftarrow$ NULL
3. **while** not all non-terminal nodes visited **do**
4. $\quad n \longleftarrow$ breadth-first first unvisited non-terminal
5. $\quad$ **while** n is non-terminal node && n is unvisited **do**
6. $\quad\quad$ add n to v behind the last element

DOI: 10.1002/dac

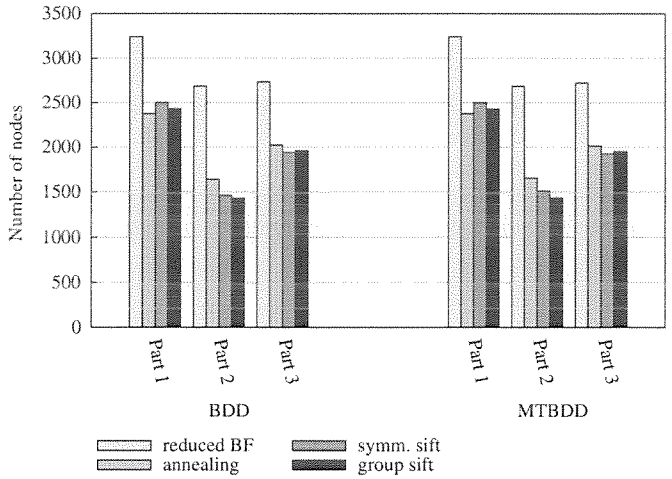


Figure 11. Complex ruleset fragmented in three parts according to the BDD and MTBDD technique, followed by an optimization according to the best performing heuristics from the CUDD package (for this scenario). These heuristics are: simulated annealing, symmetric sifting, and group sifting. Again, averages of the best 16 results are shown.

```
7.        mark n as visited
8.        if left child of n is non-terminal and unvisited or right child is terminal then
9.          n ⟵ left child of n
10.       else
11.         n ⟵ right child of n
12.       end if
13.     end while
14.   end while
15.   add terminals to v
16.   for i = 0 to v.size do
17.     if v[i + 1] != left child of v[i] then
18.       print branch instruction to left child of v[i]
19.     end if
20.     if v[i + 1] != right child of v[i] then
21.       print branch instruction to right child of v[i]
22.     end if
23.   end for
```

Figure 12 presents the results of the algorithm presented above for the BDDs computed by means of the three selected CUDD heuristics. For the optimized BDDs, the average gain in terms of code size is 45%, thereby reducing all fragments of the splitted BDD to such an extent they can be mapped in a ME's instruction store. Note that the maximum gain from the fall through strategy for an ideal BDD is 50%, since each node requires at least one branch instruction.

DOI: 10.1002/dac

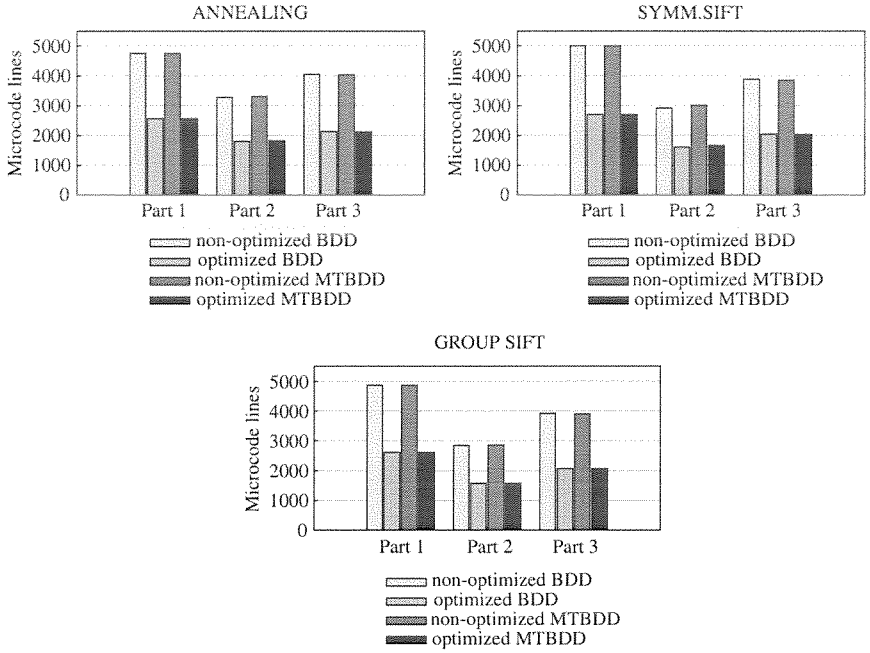


Figure 12. Code optimization results per heuristic.

5. PERFORMANCE

In this section, the performance of the Intel IXP2400 firewall implementation with both sequential search and BDD classifier is analysed. First, Section 5.1 presents a thorough performance evaluation of the sequential search classifier. Bottlenecks are identified and the limitations of the sequential search algorithm are exposed. Then, Section 5.2 presents a theoretical analysis of the worst-case behaviour of the BDD classifier, as the implementation could handle any stream of (minimum sized) packets at gigabit wirespeed for both selected rulesets.

5.1. Sequential search classification

For evaluating the firewall with sequential search packet classification, the firewall ruleset and traffic flows were chosen in such a way that each packet matches the last rule of the set. With the stateful block disabled, forcing all packets to be classified, this effectively stresses the IXP2400 evaluation board to its maximum. Using 2 GbE interfaces (one connected to the protected network, the other to an untrusted network), the IXP's throughput was measured for an increasing number of firewall rules.

A Spirent Smartbits 6000 network performance analysis system [25] was used to measure the IXP's performance. Each test was done using minimum-sized IP packets (64 bytes), implying a packet rate of 1 488 095 packets per second at GbE speed. This translates to a packet inter-arrival

DOI: 10.1002/dac

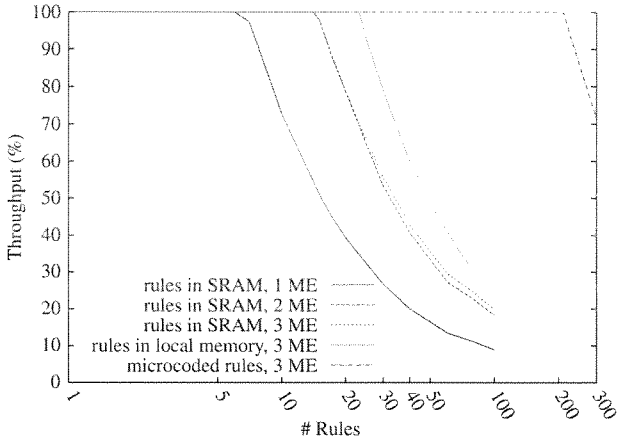


Figure 13. Throughput at GbE speed for minimum-sized packets as a function of the size of the ruleset.

time of 672 ns, which means each ME, running at 600 MHz, has 403 clock cycles available per packet. One hundred per cent throughput means all 1.488 million packets/s can be processed. Results are plotted in Figure 13.

A first implementation of the firewall with sequential search classifier reads the filter rules directly from SRAM, which takes about 90 clock cycles before the first longword is available. While a firewall rule is being read, the reading thread swaps out and allows another thread to process a different packet. With only one out of three packet filtering MEs active, the IXP2400 evaluation board is able to handle six to seven firewall rules at gigabit speed. This is clearly insufficient. Activating a second packet filtering ME doubles this performance (up to 14 firewall rules can be handled at gigabit speed). This shows the sequential search packet classification is indeed the IXP's performance bottleneck. Surprisingly, adding a third packet filtering ME no longer increases performance: the bottleneck has shifted from processing the firewall rules to fetching those rules from SRAM memory.

A second implementation reads the firewall rules from the MEs' local memory. The local memory is initialized by reading all the rules at once from SRAM when a packet filtering ME is started. Local memory has a much lower latency than SRAM (only three clock cycles) and hence there is little use in swapping threads while the consecutive rules are read from memory. When the filter rules are read from local memory, one packet filtering ME is able to handle seven to eight rules at gigabit speed (in order not to overload the graph, this measurement has not been plotted in Figure 13). Although data can be read from local memory much faster than from SRAM (in only three clock cycles *vs* 90 cycles for SRAM access), this implementation is actually only slightly faster. This shows the multithreaded hardware design of the IXP2400 MEs is very efficient in hiding memory latency. Reading the filter rules from local memory does have an advantage, however. Since each ME has its own local memory, adding a third packet filtering ME now effectively triples performance. Up to 23 rules per packet can be processed at gigabit speed. On the downside, an ME's local memory has limited capacity and can only store up to 80 filter rules.

DOI: 10.1002/dac

Even when using the fast local memory for fetching filter rules and three packet filtering MEs, performance of the IXP firewall with sequential search classifier is clearly insufficient. In order to remove any memory-related bottlenecks, a third and last implementation was made, manually translating the ruleset into microcode. This implementation consists of a large linear block of microcoded (static) firewall rules. Since no memory accesses are needed for fetching the firewall rules, no thread swapping needs to be done. Using the microcoded sequential search packet classifier, the IXP evaluation board running three packet filtering MEs could handle up to 210 rules per packet at gigabit speed. While totally inflexible (updating the ruleset implies re-implementing the firewall code), this implementation gives a good indication of the limitations of a sequential search packet classifier: one IXP2400 ME can handle up to 70 firewall rules at gigabit speed. If larger rulesets or higher packet rates have to be processed, a different packet classification algorithm is needed.

5.2. BDD-based classification

In Section 4.2 was shown how a BDD can be implemented by a sequence of branch instructions. A BDD for a boolean expression in n variables yields a final result after at most n branches (i.e. at most 126 branches for the BDD classifier presented in this paper). Consequently, the time necessary to classify a packet is independent of the number of rules in the ruleset, which is a major advantage over the sequential rule processing evaluated in the previous section.

From a performance viewpoint, branch instructions that are effectively taken, have the negative side effect of reducing the speedup caused by a pipelined processor architecture. For the IXP2400, this leads to a branch penalty of four processor cycles per branch instruction (for the specific branch instructions depicted in Figure 8). In a worst-case scenario, 504 (4×126) cycles are needed to evaluate the BDD representation of the ruleset, while only 403 processor cycles are available per packet / per ME for minimum-sized packets at gigabit link speed. Theoretically, this implies the network processor can classify IPv4 packets at 80% of the GbE wirespeed with the BDD classifier, as illustrated in Figure 14. For a realistic ruleset, however, including the regular and complex rulesets in this paper, the implementation can classify any stream of minimum sized packets at gigabit *wirespeed*: First, the number of evaluation paths containing 126 edges is limited or even non-existent, since a rule corresponding to a path with depth 126 specifies single values (no ranges) for *all* classification fields listed in Table I. For the selected regular ruleset, for instance, the longest evaluation path contains only 106 edges, resulting in a worst-case throughput of 95% of the GbE wirespeed. The complex ruleset, on the other hand, has evaluation paths of up to 125 edges in its first and third part (the longest evaluation path in its second part contains 109 edges). However, even in the presence of long evaluation paths, in practice only a minority of packets is classified by a path containing more than 100 branches.

Furthermore, the microcode optimization process outlined in Section 4.2.3 effectively avoids explicit branch instructions where possible by listing the BDD nodes in an optimized order. Replacing explicit branch instructions by implicit fall throughs not only reduces code size, but also reduces the number of cycles required for an evaluation path since the branch penalty associated with an explicit branch is removed. If only 34 out of the 126 branches (27%) of a maximum size evaluation path can be replaced by implicit fall throughs, minimum-sized IPv4 packets can be classified at GbE wirespeed ($(126 - 34) \times 4 + 34 = 402$). Since the algorithm presented in Section 4.2.3 was able to replace 45% of the branch instruction by fall throughs for the selected complex ruleset, this is not unlikely. Although the algorithm was designed to optimize code size

DOI: 10.1002/dac

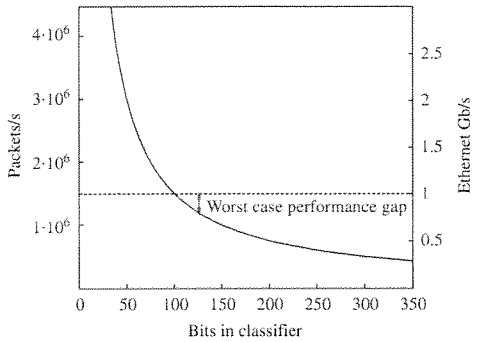


Figure 14. Worst-case throughput of a BDD classifier on an IXP2400 microengine for an increasing number of bits in the classifier.

instead of execution time, it tends to promote longer paths for explicit branch removal. However, some paths requiring >403 cycles could remain.

On the downside, the complete process of transforming a regular, sequential ruleset into a set of reduced BDDs requires a considerable amount of processing time (2–3 h on modern PC hardware[§]). First, all 8! brute force variable orders are computed for each part of the ruleset, whereafter additional—computationally intensive—heuristics search for a local optimum order. This makes the BDD approach less flexible than a sequential search packet classifier, where rules can be updated on the fly. Nevertheless, the ability to classify packets at gigabit speed with a single IXP2400 processor is appealing, especially for applications requiring infrequent updates to the classification ruleset (e.g. assigning different QoS classes to different services). Using a two-stage approach, applications with a more dynamic ruleset, such as a frequently updated firewall, can also benefit from a BDD classifier. In the first stage, the bulk of the packets is processed by a static BDD classifier, whereafter a sequential search classifier applies the dynamic part of the ruleset.

6. CONCLUSION AND FUTURE WORK

Efficient packet classification on multiple criteria is a hard problem that requires platform-specific trade-offs between execution time and memory size. In this paper, the hardware platform of choice is a network processor. These units exhibit low power consumption and great programming flexibility, making them a perfect fit for the expansion of existing network equipment, especially access and aggregation nodes residing in remote offices and operating in a restricted environment. In this context, state-of-the-art packet classification algorithms were explored and their suitability with respect to an NPU implementation was investigated. This has led to an implementation of two approaches lying at opposite ends of the time/space performance

[§]The transformations were executed on an AMD64 3000+ Linux system with 512 MB of memory.

DOI: 10.1002/dac

spectrum: sequential search and BDD packet classification. For the sequential search classifier, several hardware-specific optimizations to reduce the classification time were carried out. Despite the optimization efforts, measurements demonstrate that this approach was only successful for small- to medium-size classifiers. In contrast, the execution time of the BDD classifier is independent of the ruleset size and it is shown that packets can be classified at gigabit line rate on an Intel IXP2400 NPU. Space optimization is necessary to fit the BDDs in the limited ME instruction stores, however. This is accomplished by BDD variable reordering, ruleset-based BDD splitting techniques to distribute the BDD over multiple MEs, and an effective microcode compression algorithm.

Of course, a persisting hunger for more bandwidth or a network technology change—including the introduction of IPv6—will necessitate a proportional increase in processing power in NPUs. At the same time, memory access latency might further restricts the number of accesses that can be issued per packet, which would eliminate the use of most existing packet classification algorithms. For IPv6 classifiers specifically, the ruleset database size for a similar ruleset and the number of bits in a BDD classifier roughly quadruple, resulting in more memory accesses for the sequential search approach and an increased classification time for the BDD classifier. At present, migrating the IPv6 BDD classifier to a high-end IXP2800 network processor, would suffice to achieve performance results similar to the IPv4 implementation on the IXP2400 network processor, since its higher clock speed allows more processing cycles per packet. However, it is unclear how the BDD size will evolve as the number of bits in the classifier increases and it is possible increased parallelism—the IXP2800 offers 16 MEs instead of 8—would be necessary to fit the BDD classifiers on the MEs. If the BDD classification trees become too large, a hybrid approach—such as HiCuts—that combines a tree lookup with a sequential search through a small number of rules might open up new opportunities.

REFERENCES

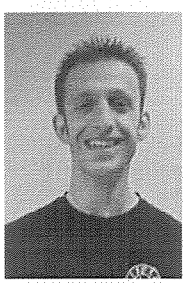
1. Stevens T, Vlaeminck K, Van de Meerssche W, De Turck F, Dhoedt B, Demeester P. Deployment of service aware access networks through IPv6, *Eighth International Conference on Telecommunications*, vol. 1. ConTEL 2005, Zagreb, Croatia, 15–17 June 2005; 7–14.
2. Taylor DE. Survey and taxonomy of packet classification techniques. *Technical Report WUCSE-2004-24*, Department of Computer Science and Engineering, Washington University in Saint Louis, May 2004.
3. Taylor DE, Turner JS. *Classbench: A Packet Classification Benchmark*, vol. 3, IEEE INFOCOM 2005, Miami, U.S.A., 13–17 March 2005; 2068–2079.
4. Comer DE. *Network System Design using Network Processors*. Pearson Education Inc.: New Jersey, 2004. ISBN: 0-1314179-4-2.
5. Carson B. *Intel Internet Exchange Architecture and Applications, A Practical Guide to IXP2XXX Network Processors*. Intel Press: Hillsboro, 2003. ISBN: 0-9702846-3-2.
6. Lin Y-D, Lin Y-N, Yang S-C, Lin Y-S. DiffServ edge routers over network processors: implementation and evaluation. *IEEE Network* 2003; **17**(4):28–34.
7. Grosse E, Lakshman YN. Network processors applied to IPv4/IPv6 transition. *IEEE Network* 2003; **17**(4):35–39.
8. Neogi R, Lee K, Panesar K, Zhou J. Design and performance of a network-processor-based intelligent DSLAM. *IEEE Network* 2003; **17**(4):56–62.
9. Yan L, Yang J, Bhuyan LN, Zhao L. NePSim: a network processor simulator with a power evaluation framework. *IEEE Micro* 2004; **4**(5):34–44.
10. Tan Z, Lin C, Yin H. Optimization and benchmark of cryptographic algorithms on network processors. *IEEE Micro* 2004; **4**(5):55–69.
11. Papaefstathiou I *et al.* PRO3: a hybrid NPU architecture. *IEEE Micro* 2004; **4**(5):20–33.
12. Radisys Corporation. *ENP-2611 Hardware Reference*, August 2003.

DOI: 10.1002/dac

13. Johnson EJ, Kunze AR. *IXP2400/2800 Programming, The Complete Microengine Coding Guide*. Intel Press: Hillsboro, 2003. ISBN: 0-9717861-6-X.
14. Baboescu F, Varghese G. Scalable packet classification. *Proceedings of ACM SIGCOMM*, San Diego, California, U.S.A., August 2001; 199–210.
15. Gupta P, McKeown N. Packet classification using hierarchical intelligent cuttings. *Proceedings of HOT INTERCONNECTS 7*, Stanford, California, U.S.A., August 1999.
16. Singh S, Baboescu F, Varghese G, Wang J. Packet classification using multidimensional cutting. *Proceedings of ACM SIGCOMM*, Karlsruhe, Germany, August 2003; 213–224.
17. Gupta P, McKeown N. Packet classification on multiple fields. *Proceedings of ACM SIGCOMM*, Cambridge, Massachusetts, U.S.A., August 1999; 147–160.
18. Xu B, Jiang D. HSM: a fast packet classification algorithm. *Proceedings of the 19th International Conference on Advanced Information Networking and Applications* (AINA), Taipei, Taiwan, March 2005; 987–992.
19. Hazelhurst S, Attar A, Sinnappan R. Algorithms for improving the dependability of firewall and filter rule lists. *Proceedings of the 2000 International Conference on Dependable Systems and Networks* (*DSN-2000*), New York, U.S.A., June 2000; 576–585.
20. Bryant RE. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys* (*CSUR*) 1992; **24**(3):293–318.
21. Bryant RE. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* 1986; **35**(8):677–691.
22. Russell P. *Netfilter Tutorial*. LinuxWorld, San Jose, August 2002.
23. Somenzi F. *CUDD: CU Decision Diagram Package*. Department of Electrical and Computer Engineering, University of Colorado, Boulder.
24. Ravi K, McMillan KL, Shiple TR, Somenzi F. Approximation and decomposition of binary decision diagrams. *Proceedings of the IEEE/ATM 35th Annual Conference on Design Automation*, San Francisco, U.S.A., June 1998; 445–450.
25. Spirent Communications. *Spirent Smartbits 6000 Product Overview*.

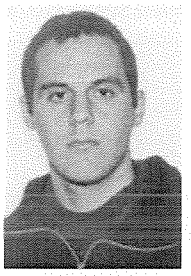
AUTHORS' BIOGRAPHIES

Koert Vlaeminck is working as a PhD student in the Broadband Communication Networks Lab of the Department of Information Technology at Ghent University, Belgium, where he received the MSc degree in Computer Science Engineering in June 2002. The main focus of his research is on next-generation multi-service access networks. Topics of interest include the deployment of network processors to increase functional intelligence of access nodes and the migration towards IPv6. In this respect, he has been active in the EU IST-FP6 project MUSE. He is currently working on distributed filesystems and data replication.



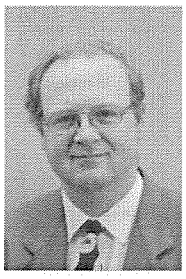
Tim Stevens received his Masters degree in Computer Science from Ghent University, Belgium, in June 2001. Until August 2003, he was a database administrator for the VRT (Flemish semi-governmental radio and television broadcasting company). He is now a research assistant and PhD student affiliated with the Department of Information Technology of Ghent University. His research interests include service-aware access network architectures and quality of service. He participated in the Flemish IWT MOVE and European IST Muse projects, focusing on next-generation access networks.

DOI: 10.1002/dac

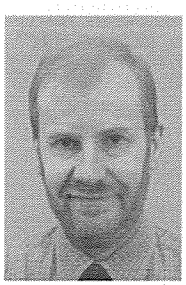


Wim Van de Meerssche received his MSc degree in software development in 2004 from the University of Ghent, Belgium. In August 2004, he started working on software technologies for access networks in the Department of Information Technology (INTEC), at the same university. His work has been published in several scientific publications in international conferences.

Filip De Turck received his MSc degree in Electronic Engineering from the Ghent University, Belgium, in June 1997. In May 2002, he obtained the PhD degree in Electronic Engineering from the same university. From October 1997 to September 2001, Filip De Turck was research assistant with the Fund for Scientific Research-Flanders, Belgium (F.W.O.-V.). At the moment, he is a part-time professor and a post-doctoral fellow of the F.W.O.-V., affiliated with the Department of Information Technology of the Ghent University. Filip De Turck is author or co-author of approximately 120 papers published in international journals or in the proceedings of international conferences. His main research interests include scalable software architectures for telecommunication network and service management, performance evaluation and optimization of routing, admission control and traffic management in telecommunication systems.



Bart Dhoedt received a degree in Engineering from the Ghent University in 1990. In September 1990, he joined the Department of Information Technology of the Faculty of Applied Sciences, University of Ghent. His research, addressing the use of micro-optics to realize parallel free space optical interconnects, resulted in a PhD degree in 1995. After a 2 year *post doc* in opto-electronics, he became professor at the Faculty of Applied Sciences, Department of Information Technology. Since then, he is responsible for several courses on algorithms, programming and software development. His research interests are software engineering and mobile and wireless communications. Bart Dhoedt is author or co-author of approximately 140 papers published in international journals or in the proceedings of international conferences. His current research addresses software technologies for communication networks, peer-to-peer networks, mobile networks and active networks.



Piet Demeester received the Masters degree in Electrotechnical Engineering and the PhD degree from the Ghent University, Gent, Belgium in 1984 and 1988, respectively. In 1992, he started a new research activity on broadband communication networks resulting in the IBCN-group (INTEC Broadband communications network research group). Since 1993, he became professor at the Ghent University where he is responsible for the research and education on communication networks. The research activities cover various communication networks (IP, ATM, SDH, WDM, access, active, mobile), including network planning, network and service management, telecom software, internetworking, network protocols for QoS support, etc. Piet Demeester is author of more than 450 publications in the area of network design, optimization and management. He is member of the editorial board of several international journals and has been member of several technical program committees (ECOC, OFC, DRCN, ICCCN, IZS, etc.).

DOI: 10.1002/dac

Efficient packet classification on network processors

Author(s): Vlaeminck K (Vlaeminck, Koert), Stevens T (Stevens, Tim), de Meerssche WV (de Meerssche, Wim Van), De Turck F (De Turck, Filip), Dhoedt B (Dhoedt, Bart), Demeester P (Demeester, Piet)

Source: INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS **Volume:** 21 **Issue:** 1 **Pages:** 51-72 **Published:** JAN 2008

Times Cited: 0 **References:** 25

Abstract: Always-on networking and a growing interest in multimedia- and conversational-IP services offer an opportunity to network providers to participate in the service layer, if they increase functional intelligence in their networks. An important prerequisite to providing advanced services in IP access networks is the availability of a high-speed packet classification module in the network nodes, necessary for supporting any IP service imaginable. Often, access nodes are installed in remote offices, where they terminate a large number of subscriber lines. As such, technology adding processing power in this environment should be energy-efficient, whilst maintaining the flexibility to cope with changing service requirements. Network processor units (NPUs) are designed to overcome these operational restrictions, and in this context this paper investigates their suitability for wireline and robust packet classification in a firewalling application. State-of-the-art packet classification algorithms are examined, whereafter the performance and memory requirements are compared for a Binary Decision Diagram (BDD) and sequential search approach. Several space optimizations for implementing BDD classifiers on NPU hardware are discussed and it is shown that the optimized BDD classifier is able to operate at gigabit wirespeed, independent of the ruleset size, which is a major advantage over a sequential search classifier. Copyright (c) 2007 John Wiley & Sons, Ltd.

Document Type: Article

Language: English

Author Keywords: packet classification; firewall; binary decision diagram; network processor

Addresses: Vlaeminck, K (reprint author), Univ Ghent, IBBT, IMEC, Dept Informat Technol, Gaston Crommenlaan 8 Bus 201, B-9050 Ghent, Belgium
Univ Ghent, IBBT, IMEC, Dept Informat Technol, B-9050 Ghent, Belgium

E-mail Addresses: koert.vlaeminck@intec.ugent.be

Publisher: JOHN WILEY & SONS LTD, THE ATRIUM, SOUTHERN GATE, CHICHESTER PO19 8SQ, W SUSSEX, ENGLAND

Subject Category: Engineering, Electrical & Electronic; Telecommunications

IDS Number: 252OQ

ISSN: 1074-5351

<< Back to results list ◄ Record 1 of 3 ► Record from **Web of Science®**

Output Record

Step 1:

- (•) Authors, Title, Source
  - [ ] plus Abstract
- ( ) Full Record
  - [ ] plus Cited Reference

Step 2: [How do I export to bibliographic management software?]

Print | E-mail | Add to Marked List | Save to EndNote Web | Save to EndNote®, RefMan, or other reference software

Save To... Save