A Practical Bytecode Interpreter for Programmable Routers on a Network Processor

S. Martin[a,*], G. Leduc[a]

[a]Research Unit in Networking, Institut Montefiore, 4000 Liège, Belgium

Abstract

WASP is a programmable router platform that allows end-hosts to store ephemeral state in routers along the path of IP flows and to execute packet-attached bytecode that processes this data. We exploit lessons from past active network research and our knowledge of network processors to design a minimal interpreter that favours language restrictions over run-time checks. WASP provides safety with limited performance penalty through predictable execution time and bounded usage of memory and network resources. WASP is expressive enough to enable several applications including statistics collection and service discovery. It can also detect common trunk of two Internet paths and exchange local measurements about these paths.

We propose a robust implementation on the IXP2400 network processor, and evaluate its performance through short benchmark programs against native functions hard-coded in the router. We achieve latencies below 7 μs, i.e. less than the reference IPv4 forwarding latency, and throughputs approaching 800 kpps per core, which competes with, and sometimes even outperforms, native programs. We further exploit our results to give hints on further improving resource usage and guidelines on management of ephemeral stores in high-speed networks.

Key words: bytecode programmable router performance analysis network processor

1. Introduction

On today's Internet, multimedia applications have become a daily reality for the average user. Be it socially-shared videos, crawling about massive amount of annotated pictures, video calls or online gaming where you can vocally tease your opponent, the fact is that Internet is going away from its mostly text-based structure and now focuses on other delay-sensitive forms of media. In the meantime the progression of wireless access networks offers a growing variety of connectivity, and potentially multi-homing possibilities. Since most services on Internet are already replicated at multiple locations, an application is likely to get even more potential paths to connect to a replica. Each path will have its own properties, and could be preferred or not for a given kind of transfer, which increases the need for sensing the state of the network.

The currently deployed architecture, however, provides little support for such investigations of the network's "health", and the simple task of figuring out the common trunk between a host and two of its partners involves several probing packets to unveil a part of the network topology. The current trend is to overcome those limitations of the network by more sophisticated interactions between end-systems, collaborating for acquiring network statistics and sharing observations in a peer-to-peer fashion. While it is a perfectly valid approach under many aspects, it certainly leads to additional burden on the network as each application performs redundant active measurements to compensate for the lack of support from the network.

During the last decade the active network research activities have fundamentally revised the paradigms on which we build computer networks to offer more flexibility, such as the ability to perform application-specific forwarding decisions based on network status (congestion, queue levels, available bandwidth, etc.). In a typical active router, each packet carries or identifies a piece of code that defines either the complete forwarding procedure, or some custom code to be applied in pre-defined *hooks* of an otherwise static forwarding procedure. In WASP (World-friendly Active packets for ephemeral State Processing), we explore a way to extend router functionality in order to provide support for network measurements and properties discovery. Unlike the vast majority of previously proposed active platforms, WASP does not attempt to provide a fully expressive programmable network, but rather restricts the programming model to achieve efficient and safe processing.

Given those restrictions, WASP offers a flexible service that can be used in conjunction with orthogonal services such as multicast or multi-path forwarding, rather than a catch-all solution. For instance, WASP cannot deflect traffic around a congested link, but it enables us to scalably publish the observation of congestion along a path so that other flows considering a possible switch to that path would know in advance whether their performance could be improved.

Related Work

Most of the research activities to run active code on network processors so far have focused on building custom services by chaining pre-installed and operator-approved modules [1, 2, 3]. In [4], the authors present a prototype of the Ephemeral State Processing (ESP) router on IXP1200 network processors, where

*corresponding author.
Email addresses: martin@run.montefiore.ulg.ac.be (S. Martin)

end-systems may only use a restricted number of *operations* that are performed on the router state store. Our work is clearly a derivative of this work where we replace pre-defined operations by a bytecode interpreter.

Comparatively an implementation of SNAP [5] on the IBM PowerNP [6] has explored the feasibility of a just-in-time compiler for SNAP bytecode. While the results speak in favour of compilation (rather than interpretation), this only holds for programs using loops or if the code generated for one packet can be reused for subsequent packets. We argue that, given the limited size of the instruction store in network processors (and especially in the Intel IXP family), compilation should be a way to optimise most frequent packets rather than the default behaviour.

We also need to mention StreamCode [7], which also proposed programmability through a dedicated processor on FPGA suited to packet forwarding routines, using 'co-processors' for data copy and routing table lookup. A significant design choice in StreamCode is that the code in packets is in charge of allocating buffers and decides the next hop for each packet. This level of control over the router and network resources implies that StreamCode should not originate from the end-systems, but would rather be used by a smart border gateway to enforce specific QoS behaviours [8].

Document Structure

The paper is organised as follows: Section 2 presents the principles that have guided the design of the WASP platform. Section 3 discusses the platform itself, including the trade-off between expressiveness and safety of operations. We implemented a prototype of our WASP interpreter on the IXP2400 network processor and present a performance evaluation against the ESP filter prototype in section 4. We finally propose scalability improvements and guidelines for deployment on high-grade routers in section 5.

2. Design Principles

2.1. Ephemeral Store on a Router

The Ephemeral State Store (ESS) is a corner stone in the design of WASP, which has been inherited from former ESP (Ephemeral State Processing [9]) filter. It is a $(key, value)$ repository associated with a network interface where packets can drop and inspect state. Because all entries in the ESS have the same size (64-bit key and value) and are allocated for a fixed time period τ (namely 10 seconds), it is possible to engineer the state store so that it can always satisfy a request for a new slot, even at peak rate.

In [4] the authors illustrate this principle on an IXP1200 network processor. With at most 10^5 packets per second, each allowed to create 2 ESS entries that last for 10 seconds, a 46MB ephemeral store never gets full. Access to the ephemeral store can thus be made available to any end-system, without prior authentication or specific accounting, just like IP forwarding is offered to every machine connected to the network.

Similarly, the ESS offers only best-effort privacy, and it is up to the end-systems to agree on the 64-bit key(s) they will use so that no adversary could eavesdrop it. Even without that/those key(s), one could still generate random keys in a brute-force way to try and corrupt existing state. However, since the state is kept only for 10 seconds, an attacker that manages to send 1.4 million WASP packets per second (i.e. saturating a 1Gbps link with his attack) would only alter existing state with probability lower than 2^{-40}.

Furthermore the ESS can ease several monitoring tasks such as evaluating the jitter of a given packet flow. The local time t_0 observed when packet P_0 crosses the router is written in the ephemeral store. The next packet P_1 can compare this with the observed local time t_1 and build average, maximum and minimum values over a few packets. A special packet then collects those values in each router. Other statistics such as the depth of output buffers, transmission errors or access to the network packet header will of course enable more applications.

Proper operation of virtually any ESS-based protocol relies on the assumption that the network can be trusted to deliver packets only to their intended recipients. In some access or local networks this may require link-level encryption between hosts and access routers to avoid blatant information harvesting and impersonation.

2.2. World-Friendliness

Our aim with WASP is to find the right balance between expressiveness, efficiency and safety in processing of application-specific code in the network. We use the term "world-friendly" to refer to this triple objective of (1) *user-friendliness*: WASP should provide enough flexibility and a clear model of the offered service where the end-system keeps control on the service their packet should receive; (2) *router-friendliness*: WASP makes sure that router resources are correctly used, and that (possibly malicious) bytecode have predictable execution time and cannot exhaust router memory; and (3) *network-friendliness*: WASP sticks to the network transparency model of IP, disabling any "surprising" behaviour such as cloning of packets or altering source/destination fields. The transport protocol and application data are also out of reach and the only part of the packet WASP can alter is its own "scratchpad" to record state observed in the network.

2.3. The Case for Cooperation

Even if we perfectly fulfil the world-friendliness objectives, any addition of functionality to the network is only of interest for a network operator if it can bring some added-value to his business. WASP mainly allows to trade-off processing power and temporary storage against bandwidth, in the same way a transparent Web proxy does, which – depending on an operator – may reduce operational expenses or not.

Unlike transparent proxies however, WASP requires cooperation from end-systems, where applications will attach WASP programs and extract results, but it offers a richer set of interactions. It is possible for instance to efficiently locate third-party services along a path, control one's flow rate or locate peer systems that own a replica of a wanted piece of data.

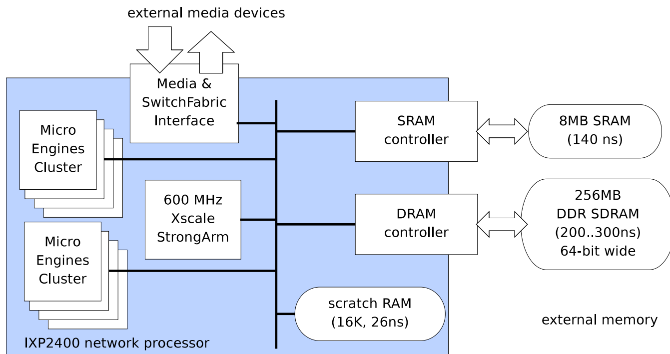

Figure 1: IXP2400 network processor structure diagram, annotated with timings reported by [14]

Locating available members of a peer-to-peer application (e.g. grid computing, file sharing, maintaining virtual coordinates) in a decentralised fashion for instance, typically requires scanning large parts of the Internet (e.g. using randomly generated addresses [10]) before a contact node can be found. We have shown previously [11] that having 10% of a community connected through routers offering ephemeral storage is enough to find a contact node with only a small number of probes. Moreover peers that are attached through a WASP router will almost immediately discover peers running in the same ISP, potentially reducing long-distance traffic.

We also previously illustrated the use of WASP and ephemeral store to allow an operator deploying an application-specific service (such as a proxy/depot, a media transcoder or a system for merging results of grid computations) to make the service visible to end-systems establishing a connection through the network [12].

We believe that in the context of an increasingly flattening Internet [13], where major content providers are pushing their WAN closer to the connectivity providers, a generic mechanism for enabling such interactions could be both technically viable and economically sound, especially as it allows one to decouple hardware upgrades from application-level innovations and improvements. It may also be useful for an operator that has special interest e.g. in grid computing and wants to differentiate his offer with additional services, but has no direct way to remotely configure customers' software (e.g. a national research & education network).

2.4. IXP Network Processors

Network processors (NPUs) are programmable chips designed to replace the dedicated circuits that used to equip line-cards in routing equipments. The term covers a large diversity of designs, but on most of them, we can highlight the presence of a *control core* (typically an embedded RISC processor), and a series of *data-processing cores* to perform custom functions on every packet. The chip usually also features co-processors that assist the other units in specific tasks such as checksums, data transfers, lookups, encryption, etc.

In this paper we will focus on the Intel IXP2400 depicted on Fig. 1, which features eight micro-programmed cores (the *micro-engines* or MEs) for fast-path functions and a StrongArm for control and management. We can also note the presence of memory units of different technologies, each with its own access characteristics (such as commodity DRAM with high access latency but efficient burst transfers) and size limits. Programs on the micro-engines are directly exposed to those hardware details, and optimising the placement of data structures on the right kind of memory is usually required to achieve good performance.

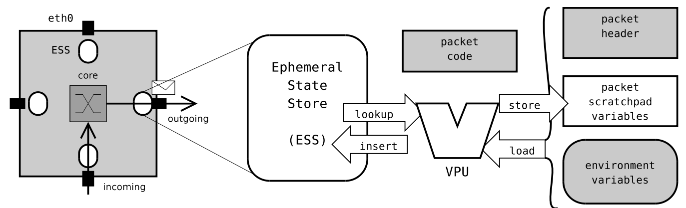

Figure 2: A WASP router and WASP execution environment. Gray items mean the VPU has read-only access to the resource

A determining aspect of NPU programming is the pursuit of *wire speed* processing. In other words we want to ensure that our network device is capable of fully utilising the output links when it receives enough traffic on its input links, even when all packets have minimal size [1]. If a router fails to meet this requirement, an attacker can easily deny routing to other flows with a traffic volume that the router should handle without problems. In order to keep the output link busy even when the processing time exceeds the transmission time of a minimal packet, NPUs typically feature massively parallel architectures that allows for pipelining of sub-tasks and parallel execution on different packets. On the IXP2xxx, each micro-engine has 8 independent hardware contexts and an arbiter that allow another context to be executed on the ME while the previous context is waiting for a memory or I/O access to complete.

3. The WASP Platform

The WASP router combines an unmodified forwarding core, surrounded by WASP filters, each associated with a network interface. A packet crossing the router is thus first processed by the WASP filter associated with the receiving interface, then handled by the forwarding core (which will e.g. lookup the forwarding table and dispatch packet to the proper output card), and finally processed by the WASP filter associated with the output interface, as shown on Fig. 2.

This overall design is inherited from the Ephemeral Store Processing (ESP) router [9] and has strongly helped in reaching network-friendliness. As the core of the router still operates like a regular router forwarding engine, the only notable difference, as far as forwarding is concerned, is that filters introduce the possibility for a packet to be dropped pro-actively (just as if a queue was full) or anticipatively be returned to its source (just as if the TTL limit was reached).

Each WASP filter consists of a dedicated ephemeral state store (ESS) and a virtual processor (VPU) executing WASP

programs embedded in packets passing through the filter. During its execution, a WASP program only has access to its own *scratchpad* S_P (packet-carried variables) and a set of ESS entries ($E_1, E_2, \ldots$) identified by the *keys* carried with the packet. The sole effect of the WASP (or ESP) filter is to transform these entries into $S'_P, E'_1, E'_2, \ldots$ and to decide on the packet's fate. All other state on the router (e.g. FIB, MIB, queues) and in the packet (forwarding headers and payload) remain unchanged.

WASP departs from the ESP design by expressing the function that performs this transformation *in the packet itself*, via a bytecode program that will be interpreted on a virtual processor associated with the ESS. Comparatively, in the original ESP packets just contained operands and an opcode selecting one of the pre-compiled transformation function – some of which requiring over 20 lines of pseudo-code for their description.

We also decided to drop ESP's "central" store, which could theoretically process every packet crossing the router, to ensure a scalable architecture where extra functionalities are solely provided on line cards. We compensate this limitation through "admin packets" that may originate only from management stations of a given network domain and that are given the ability to commit a write in *all* the state stores of a node.

Compared to most interpreters featured in active routers, the VPU found in WASP is extremely simple and lightweight. We refined memory access opcodes and ALU workflow to allow both compact encoding of programs and efficient execution[1], while keeping the interpreter compact enough to fit in a single microengine of an IXP processor. Unlike most virtual machines, the VPU has no heap to manage, only performs one-cycle arithmetic operations on integers (e.g. no float, no divisions) and its whole state is reset every time a new packet is processed.

3.1. Improving Utility

The original ESP router allowed interesting interactions via the ephemeral state store, mainly illustrated in the context of reliable multicast transmission [15]. Yet, the programming model remained tedious to use and master. For a given problem where the designer has the feeling Ephemeral State Store could be helpful, it is at best unclear how to combine available functions (in a sequence of packets) to achieve that goal. In many other situations we just experience the frustration that the operations defined are too specialised to be of any use.

In contrast a bytecode-based encoding of the functions allows the application designer to express the exact operation to be performed by each WASP packet. We also provided new "micro-operations" through the bytecode interpreter that were not found in any previous ESP function, such as the ability for a packet to return to its source.

To further extend the utility of WASP compared to ESP, we allow the bytecode to access per-interface and basic router setup information (depicted on Fig. 2 as "environment variables" memory bank) and network-layer packet header. Such variables will indicate e.g. the IP address of the node, a node-local timestamp, rough state of the output queues of an interface (to allow packet "sensing" congestions). They could also indicate whether a given link is wireless, its error rate and peak bandwidth. We tried to keep such information minimal, as network operators are typically reluctant to disclose any information about the internals of their network. As a rule of thumb we suggest that information is eligible for environment variables only if it is already possible for end-systems to infer that information through active measurements.

Finally we extended the semantics of the ESS itself. While entries in ESP always have public visibility, WASP also supports *protected* entries that can be modified only by the network operators and optional *protocol-private* entries that are accessible only to packets using a specific program. As we shown previously [11, 12] this enables applications operating on more sensitive data such as locations where traffic could be redirected.

3.2. Building for Safety

The challenge of safety in active networks is essentially twofold: execute third-party code without putting the router at risk (e.g. ensure there will be CPU and memory to sustain submitted packets) and allow applications to control their flow without raising threats on the network (e.g. avoid those programs to flood hosts/links). Though we exploited the results and guidelines of previous active network research (e.g. [16]), most of the safety issues were fortunately simplified by VPU design, removing the need for run-time checks.

Network resources usage is at worst similar to the regular use in IP networks, thanks to the filter-based organisation: either the filter drops the packet, or it lets the legacy core handle it. If we ensure that `return` operations can happen only once for each packet, WASP trivially never introduces loops in the network, nor does it flood (or help flood) hosts and links. We should stress however that if we attempt to extend the functionalities of WASP, we need to ensure that we do not break this property. While we initially planned to allow WASP programs to alter the destination of their packets in a restricted way [17], it turned out that this could not be implemented safely unless forwarding infrastructure guarantees that packets originate from their respective source address.

Similarly the simplicity of the VPU and its minimal set of opcodes allow us to guarantee that processing time of any packet P is $O(|P|)$, where $|P|$ is the length of the WASP program attached to P in bytes. This is possible thanks to the fact that we forbid *backward jumps* as in SNAP [5], allowing only *if-else* blocks, but no (wild) loops. By also forbidding complex arithmetic instructions, we ensure that all microbytes[2] can be processed in a similar time, which prevents attacks using an abnormally high amount of "heavy" opcodes (such as `div`). If we enforce a restriction on the number of ESS references a program can issue, the code length (not its content) is thus sufficient to predict execution time.

[1] Mainly through optimising for sequential access to packet variables

[2] With the notable exception of ESS access instructions

The most challenging checks to ensure safety of an active router come from memory protection and the need to avoid router state corruption or information disclosure. This is a specific issue that should not be confused with access control to application-maintained state. In most active networks, the virtual machine processing application-specific code has access to sensitive router state such as forwarding tables or packet queues. This access can either be direct (e.g. the data sits in the VM's address space) or indirect (the VM relays objects between trusted address spaces), and designers mostly rely on strongly typed languages to avoid abuses through wild pointers.

In contrast the VPU has an extremely small address space (256 bytes), which only contains data that WASP bytecode is allowed to access. Packet-bound variable are not allowed to grow beyond the room initially allocated by the sender, and node-resident data are only accessed through the key/value interface of the ESS. The absence of any type-checking in the VPU is possible because the VPU ignores any type except integers, and because we leave the organisation of ESS entries and scratchpad entirely up to the application. Unlike scripting languages designed to be used as glue between sensible services, we do not need more typing in the case of WASP.

Rather than issuing *references*, WASP programs look for randomly generated 64-bit *keys*, and access control is defined on a per-entry basis: either the whole entry is publicly available (and anyone knowing its key can alter any bit of it) or it is an operator-installed, read-only entry. This gives us the extra advantage that packet-bound variables can be accessed directly, without the need for any unmarshalling operation – which may account for e.g. up to 42% of the processing time for simple packets in Java-based active networks [18].

3.3. Example of Use

We insist on the fact that the WASP programming model allows the network to mix WASP-capable and legacy routers. Applications using WASP will thus have to take this into consideration and degrade gracefully when we have less "cooperating" routers. This could mean less information available to the end-systems, sub-optimal placement of helper proxies, etc.

Scanning path The most obvious WASP application is a traceroute-like packet that records information about routers it crosses. However, while a regular traceroute program requires at least one packet per hop, WASP can easily store 30 hops in a single packet. It is also possible to program the WASP byte code so that only a portion of the path is recorded, or that the packet returns after collecting n addresses.

Common trunk Given three nodes S, A and B, a very simple use of the ephemeral state allows S to *tag* routers on path $S - A$ with one packet while a second packet will travel along path $B - S$ and record the addresses of tagged routers[3]. This can identify the common part $S - R$ of the two paths where traffic towards A and B may interfere, or directly return the address of R (the last common router) where deployment of a duplication/merging function could be interesting.

[3] This mechanism was initially described in [15], though none of the ESP operations provide support for it

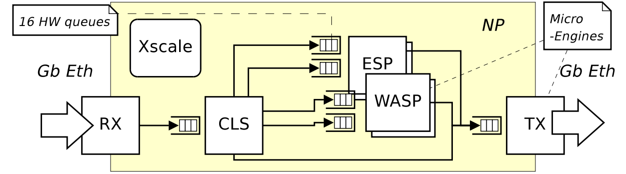

Figure 3: Internal structure of our WASP/ESP packet filter. 4 MEs do WASP/ESP processing and we have one "spare" ME.

Rate Control Given that routers export a hint on the depth of output queues, we can program packets that drop themselves if the router appears congested, making room for "more important" data in the same flow[4]. WASP also offers the flexibility to detect losses between any successive WASP routers and to prepare state so that such losses are reported back to the source by e.g. acknowledgements. The positive impact of such decisions on the quality of a video stream has been demonstrated in many former works on active networks [19].

Service Advertisement WASP packets coming from management stations in an autonomous system drop the IP address of a service provider using a well-known *protected* key (e.g. a hash of the service name), so that a modified version of the path-scanning program can report to the end-system where the wanted service can be found [12]. At each router R where the advertisement is stored, the TTL of the advertisement packet can be used to keep only the closest provider from R.

Most of these programs can of course be extended or modified to better match application needs. Scanning, for instance, is not limited to IP addresses of routers, but can be extended to any piece of information available in the ESS or in the environment variables memory bank.

More sophisticated versions of the common trunk identification could use a bitmask in the ESS in order to quickly get a snapshot of a full sink tree. Another possible extension is to make use of a shared secret key so that members of a community can mark path on the Internet that are under measurement (e.g. bandwidth-wise in a peer-to-peer distribution network) to avoid oscillations due to simultaneous probing of a path by two pairs of nodes.

3.4. WASP on Network Processor

An application on the IXP is typically split up into several components that will run on the different processing elements. Pipe-line processing is typically assisted by hardware *scratch rings* programmed to relay packet handles and metadata between MEs. Note that the packet content is transmitted directly

[4] E.g. dropping B-frames in favour of I-frames in an MPEG flow [17]

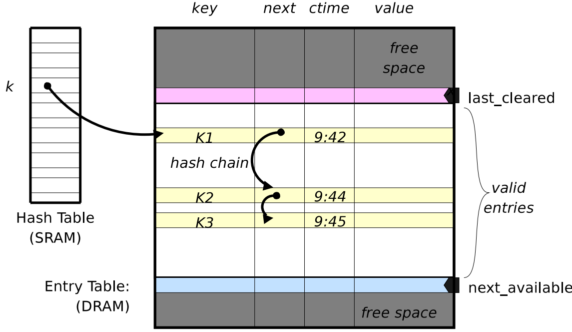

Figure 4: Layout of the Ephemeral Store, showing 3 keys (K_1, K_2 and K_3) hashed to the same value k and chained in the entry table, with their respective creation time ($ctime$).

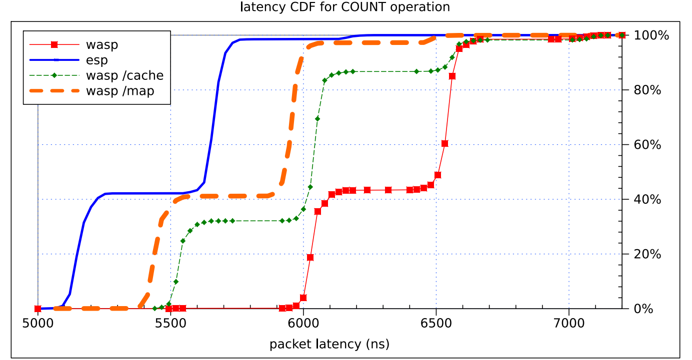

Figure 5: Cumulative Distribution of packet forwarding latency at low throughput for `count` instruction

from I/O buffers into DRAM (by the RX microblock) and only meaningful parts will be fetched on demand, e.g. by the classifier microblock CLS.

The setup of our proof-of-concept implementation is depicted on Fig. 3. We limited ourselves to layer 2 forwarding (plus WASP processing) and the XScale's role is limited to microcode loading, monitoring and debugging. We kept ESP and WASP apart, on separate MEs, to ease individual testing of each implementation. Each of the 4 MEs doing WASP or ESP processing is connected to the classifier with 16 independent queues and operates its own ESS.

Out of the 256 MB of DRAM available on our Radisys ENP-2611 card, 192 are under the control of the Linux kernel running on the XScale, 32 are used as packet buffers by the microengines, and the remaining 32 MB hosts the 4 ephemeral stores. As depicted on Fig. 4, one ESS is made of a *hash table* in SRAM and an *entry table* hosted in DRAM, where each entry consists of a 64-bit key, its associated 64-bit value and metadata for a total of 24 bytes. Every slot k in the hash table points towards the oldest entry in the store where $hash(K_1) = k$. Other colliding entries (e.g. K_2 and K_3 on Fig. 4) that hash to the same value k are simply chained in their creation order. Thanks to this organisation, periodic cleanup of the ESS is greatly simplified. We just sequentially scan entries in DRAM starting from `last_cleared` mark until we hit an entry e with $e.ctime + \tau > now$. When e.g. K_1 expires, we replace pointer in SRAM slot k by a pointer to K_2 ($e.next$) and we advance `last_cleared`.

The initial implementation of ESS on IXP was borrowed from [4] and was accessed through two opcodes (`lookup` and `insert`) that mimics statements $val = get(key)$ and $put(val, key)$ found in the pseudo-code of ESP functions. In previous work on x86 architecture [17], we explored various ways to optimise ESS accesses to compete with the performance of ESP. A first approach, later referred to as "*/cache*", simply adds a one-entry cache storing the last key looked up and the address of the corresponding entry in DRAM. The effect is that we immediately know where to write back a value with the `insert` microbyte. This saves one hash table lookup in SRAM and chain walking in DRAM. We observed that maintaining a larger cache is not worth the effort for WASP programs.

Most our operations will not manipulate just a single 64-bit value, but rather operate on *tuples* in the ESS that need to be manipulated atomically. We allowed WASP to use two consecutive entries of the ESS to offer a 32-byte memory bank where elements of a tuple can be grouped together under a single key. The `map` microbyte allows such an entry to be accessed as a second bank by the VPU. WASP programs rewritten to take advantage of this alternate access mode are referred to as "*/map*" in the following section.

4. Performance on IXP2400

For our performance analysis of WASP on IXP platform, we mostly focused on two "reference" functions that were already available on the ESP implementation. This allows us to evaluate the overhead introduced by bytecode interpretation against a pre-compiled version of the same function. The `count` function only needs one variable in the ESS that is used as a counter. Each packet increments the counter and drops itself if the counter is above a given threshold. The `collect` function is used to *aggregate* samples from n sources, using one ESS variable to store the temporary aggregate for the k sources that have already been processed, and a second variable to count the remaining $n - k$ sources that should still give their sample. A `collect` packet is forwarded only when the n samples have been aggregated and then continues towards its destination with the aggregated sample.

The pseudo-code for `count` and `collect` functions, as well as their translation in WASP bytecode and an example of how they can be used on a merging tree is detailed[5] in [20]. Through the rest of this document, `W:*` prefix will be used to refer to a WASP packet (e.g. `W:count`) while `E:*` prefix refers to ESP packets.

4.1. Latency Measurements

All latencies in this section have been measured directly on the IXP, through instrumentation of the "receive" (RX) and

[5] A lighter version can also be found at *http://www.run.montefiore.ulg.ac.be/~martin/resources/wasp-prm.pdf*

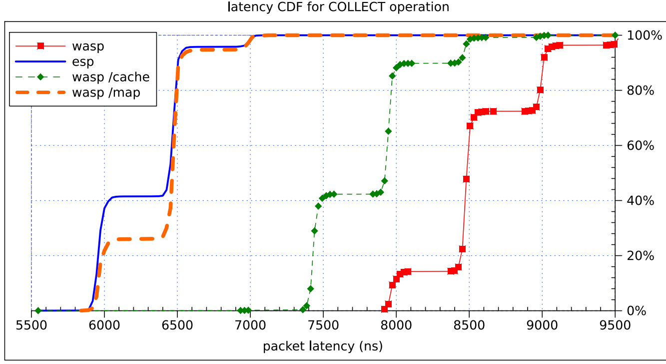

Figure 6: Cumulative Distribution of packet forwarding latency at low throughput for `collect` instruction

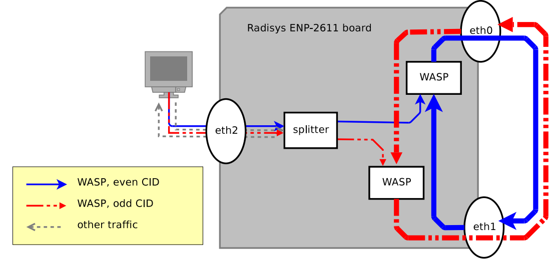

Figure 7: Packet flow in our testbed, highlighting the full-duplex use of the loop-back link.

"transmit" (TX) microblocks provided by Intel as part of the IXA SDK, after proper synchronisation of ME timestamp counters. The 1Gbps interface cards of the PCs in our lab indeed proved to introduce load-dependent delays[6] that interfere with all trace-based measurements – not mentioning NTP offsets seen as high as 10 μs. Taking advantage of the internal structure of the RX and TX code, we further separated latency measurements depending on whether packet size was below or above 128 bytes[7]. This later allowed us to isolate measurements for a specific class of packets by artificially "inflating" all other packets we submitted to the IXP with padding bytes.

In the conditions described above, our WASP filter forwards `W:count` packets with an average latency of 6.34 μs. Dumb packets of identical size have an average latency of 2.45 μs and `W:count` packets that are not processed (i.e. going through classifier tests, but without bytecode interpretation) take on average 2.91 μs. The same experience repeated with `E:count` packets showed an average latency of 5.47 μs. Comparatively, the IPv4 forwarder demonstration application [21], takes on average 6.92 μs to forward a 64-byte packet under 25% throughput. The results for `collect` (Fig. 6), on the other side, are less impressive: 136% of the native version. With 8.58 μs for WASP against 6.29 μs for ESP, we clearly see the impact of a longer WASP program here.

We repeated the experiment with the one-entry cache enabled, which only led to a slight improvement of 0.37 μs and 0.78 μs respectively (see *WASP /cache* series on Fig. 5). However, as reported through the *WASP /map* series on Fig. 6, making use of `map` strongly improved the latency of `W:collect` packets which is now almost equal (101%) of native ESP counterpart. Another good thing is that */map* also achieves good performance (106% of `E:count` latency) although it was *already* using only one key. It even performs better than */cache* although it now transfers twice as much memory, thanks to DRAM burst transfers.

We can also observe on Fig. 5 that the latency distribution is mostly split into two (or sometimes three) steps and that 95% of the samples are located no further than 52 ns from the closest step centre – which corresponds to variance in latency of DRAM accesses. Those steps are intriguingly spaced by 516 ns on average, with very small deviation, independently of the protocol or function considered. By deeper inspection of the chains stored in the ESS, we can state that this is not an artifact of some hash collision, nor an extra delay incurred by the creation or cleaning of entries.

Actually, if we look at latencies until packets are queued to TX microblock, we get a smooth, Gaussian-like distribution. Together with the fact that the delay of 516 ns almost exactly corresponds to the time required to transmit a minimal-sized packet on the 1Gbps medium, it sounds reasonable to consider that the "stepping" is introduced by the transmit hardware depending on whether it is found idle or busy when our transmit request is submitted.

4.2. Throughput Measurement

4.2.1. Methodology

The maximum capacity of our traffic generator (on a dual-core Xeon, 3Ghz) was 907 Mbps with all packets having the maximum size and 170 kpps (98 Mbps) when using only packets of minimum size, which is way below the theoretical maximum throughput of a single port-pair Gigabit Ethernet (1.488 Mpps). In order to keep our tests independent of additional equipment (e.g. aggregating hubs) and to minimise the amount of code to be modified on the network processor, we opted for a half-software, half-hardware "traffic accelerator" testbed.

As depicted on Fig. 7, we connected port 2 of the ENP-2611 board to a single test machine and wire the two remaining ports together. We have then rewritten the classifier's rules to enforce the following policy:

1. regular packets coming on port 2 are returned to port 2;
2. WASP/ESP packets coming on port 2 are delivered either to port 0 or 1 depending on a bit of the Computation ID (CID) field;
3. packets received on port 0 are delivered to port 1 and vice-versa.

The result is the creation of a two-way loop involving one WASP/ESP processing and one transmitting delay that keeps packets until they drop themselves. We can "load" this loop by sending incremental traffic from the test machine and watch the

[6] Likely due to interrupt moderation mechanisms on Broadcom NetXtreme BCM5701 and on-board Intel 82541GI/PI Gigabit Ethernet controller

[7] Which is the size of an *m-packet*: the atomic data unit between the switch fabric interface and the processing unit of the IXP

# queues	1	2	4	6	8	16
WASP (1ME)	315	528	653	746	764	788
ESP (1ME)	390	672	1096	1258	1430	1546
WASP (2ME)	315	632	1026	1190	1293	1552
ESP (2ME)	390	779	1298	1603	1744	1744

Table 1: throughput (kpps) of WASP and ESP microblocks, processing `count` packets with single entry per hash chain and varying number of active queues and hardware contexts.

impact on the system. We of course adjusted ESP functions and WASP microbytes to allow unlimited thresholds.

Rule #1 ensures that the loop remains noise-free during our measurements. We have indeed observed that the test machine automatically exchanges a few packets every time we reload the test software, which may quickly lead to an extra 200 kpps stress on the loop. Temporarily disabling rule #1, we can measure a maximum throughput of 2 972 kpps in the loop for "regular" packets, which is 99.86% of the theoretical maximum throughput of a full-duplex Gigabit Ethernet link. Any throughput limitation that we will measure with ESP and WASP packets will then be attributed either to WASP/ESP microblock, or to WASP/ESP specific part of the classifier.

We also observed that packets that simply start with a `FWD` microbyte can be processed at a maximum throughput of 2 966 kpps by a single ME. The same program padded to 16 microbytes and carrying one bank of unused data (thus with a similar fetch/checksum cost than a `W:count`) will grow to 100 bytes on wire and will limit the throughput to 2 046 kpps – 98.22% of the theoretical maximum for packets of that size.

4.2.2. `Count` Performance with 1 Entry per Chain

Using the "transmitted packets" counter of the TX microblock, we estimated the throughput of flows of `count` packets using both WASP and ESP while varying the amount of processing resources activated. Since the classifier uses the CID field to select the queue a packet should take to reach WASP microblock, and since only one hardware thread can operate on one queue at a time, we can indeed decide how many threads (and MEs) can be active at a time by simply restricting the values allowed for CID in the traffic generator. The best performance of ESP and WASP on a single ME correspond to 58 and 38% of the maximal throughput, respectively.

By translating throughput measurement $T(n)$ (where n is the number of active threads) into inter-packet delays $D(n) = n/T(n)$, we can observe that each new active thread on the ME increases that delay by δ_n (almost constant and in the range 800-900 ns), making $D(n) \approx D(1) + (n-1)\overline{\delta}$ almost linear with n in the case of WASP. We cannot apply a similar model to ESP, as it uses a special read-and-modify bus cycle that reduces the available DRAM bandwidth as the packet rate increases.

While there are only 8 hardware threads on a ME, we can see on Table 1 that all the 16 queues have been required to achieve the highest throughput. This can be charged to the time required to probe 8 (empty) queues and has been avoided in further tests by properly balancing the k active CIDs among the 16 available queues (rather than using queues 1..k).

length:#ME	2:1	6:1	10:1	2:2	6:2	10:2
ESP	1502	1296	1124	1733	1470	1314
W /cache	946	902	826	1858	1686	1476
W /map	774	730	680	1526	1396	1225

Table 2: Throughput (kpps) with 16 queues, depending on the average hash chain length, with one (left) or two (right) microengines.

We then reproduced the experiment with 1 to 8 threads balanced on two different MEs to estimate how increased CPU power improves the performance. A "4 threads" setup thus means that we will have 2 active queues served on each ME. Comparing rows 1 and 3 in Table 1 confirms that the WASP interpreter is CPU-bound. Indeed, while balancing the load on two different MEs leads to throughput increased by 20% with ESP, WASP sees its throughput improved by 57 (4 queues) to 70% (8 queues), and the maximum throughput when all 16 queues are used has almost doubled.

4.2.3. Increasing Hash Chain Length

In order to estimate performance of a saturated store, we extracted chains of colliding keys observed in the store and generated for each individual "computation" a series of k packets that will reference one of the k keys that belong to the same chain, therefore experiencing chain traversal of length from 1 to k.

When processing on a single ME, we can note that the gap between ESP and WASP performance is reduced (from 50 to 40%) as the average chain length increases. We can also observe that the relative throughput reduction is less important in the case of WASP (the worst observed throughput is 87% of the best one with WASP, against 74% in the case of ESP). Table 2 gives the observed throughputs for both `E:count` and the two flavours of `W:count` already discussed in section 4.1. Surprisingly, when we fully load the two MEs, the WASP packet using `lookup` *outperforms* the native implementation offered by ESP, and this regardless of the ESS state. The final explanation has been found in the code: the ESP microblock – in its current state – will re-generate the CRC checksum and update packet header and operands in DRAM regardless of the computation performed. The WASP VPU comparatively keeps track of data and state "dirtiness" and will only issue a DRAM update when the content of the packet has been modified – which never happens in the case of `count`.

4.3. Throughput of Collect function

We repeated the experience with `collect` packets using only the kth entry in chains. Such chains are built with `count` packets that drop themselves immediately after execution. A vertical bar in Fig. 8 reports the throughput of one scenario (i.e., either ESP or WASP, and chain length) for increasing amount of processing power. We can observe here again how ESP takes advantage of additional threads and how WASP rather benefits from an additional ME, even with the same number of threads.

We have seen in section 4.1 that we could come up with a similar latency for the `collect` operation for a single thread, and as expected, on a single ME, WASP remains way behind in

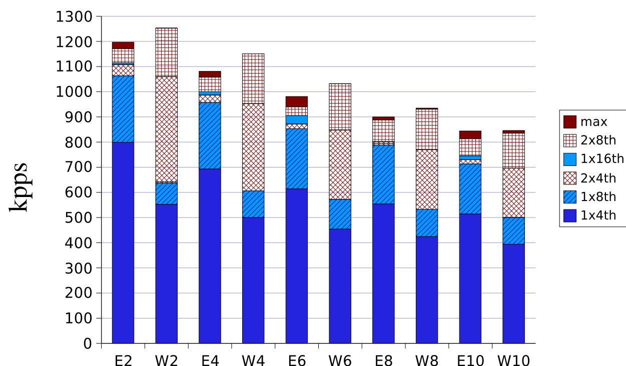

Figure 8: Measured throughput (kpps) for the `collect` operation with chain length varying from 2 to 10 entries, for both ESP and WASP (e.g. E6 is ESP walking a 6-entries chain), varying the amount of threads and ME used for processing (e.g. 2x8 is 2 MEs, each processing 8 queues).

terms of throughput (see e.g., the 1x8 series). With two MEs doing WASP processing however, we can now slightly outperform ESP. In this case, both ESP and WASP have to commit the packet variables into DRAM. The performance improvement can thus be fully attributed to the reduction of memory accesses during the ESS entries lookup. This is true as long as WASP does not hit the memory bandwidth limit, which happens with $W10$. Indeed, while WASP requires less memory accesses, each memory access transfers twice the amount of bytes per access, theoretically requiring more DRAM bandwidth than ESP to sustain the same number of packets per second.

Rather than reading a full bank (48 bytes, including metadata) while walking the chain, we repeated the experiment with a modified `map` implementation that only fetches 24 bytes during chain walking, and then issues an extra access to get the remaining 24 bytes once the correct entry is found (the "max" series for $W8$ and $W10$ on Fig. 8). This indeed slightly improved the throughput for $W8$ and $W10$ (from 924 to 928 kpps and 829 to 840 kpps resp.), but actually degrades the performance for chains of 6 entries and below. As a potential alternative, we could let the extra 24-byte transfer happen while the VPU continues to process the next instructions, and suspend execution only if the data are still missing when we further advance in the memory bank. This is typically an efficient programming technique on the MEs, and our first estimations suggest that we could achieve up to 877 kpps for $W10$. It would require, however, a significant revision of our code since we need to detect and update the partially mapped bank transparently.

In other words, if we were to implement both WASP and another function that is more memory-bound on an IXP network processor (such as IP table lookup or execution of pre-compiled operations on the ESS such as ESP), it would be preferable to balance the amount of threads we are willing to dedicate to the WASP interpreter on the n available microengines (thus sharing microcode and local store with the other functions) rather than grouping them on a single microengine.

5. Towards Deployment

With latencies below 7 μs and throughput up to 1.5 Mpps, WASP can offer an interesting trade-off between flexibility and performance from an end-user point of view. From an operator's perspective, however, the amount of resources required for a guaranteed service level are prohibitive.

While the WASP service model allows partial deployment, and even deployment only on some interfaces through sidekick filter boxes, it still lacks some level of fine-tuning that would allow the operator to decide what amount of resources he's willing to dedicate to WASP traffic, and protect his router against unusually high demand for WASP processing. This section explores possible adjustments to the proposed implementation that can take advantage from the knowledge of "normal" WASP load to lower the required resources. Any such approach, unless properly protected, becomes vulnerable to attacks where a group of hosts craft a flow of WASP packets exceeding the available resource, therefore degrading or denying WASP service to other packets.

5.1. Alternate Structures for the ESS

The organisation of the ESS as a hash table, as presented in section 3.4, has clearly the drawback that excessive chain length may degrade performance up to the point that service will be denied. As a first defensive measure, we can salt the hashing function with a local random value and enforce a maximum chain length, so that attackers cannot craft a collection of packet that ends up in the same chain. Still, the total amount of keys we could store in the ESS is limited by the amount of expensive SRAM which defines the number of chains the ESS can have.

We considered the alternative of balanced trees for the ESS, as these have better worst case. The constraints would be that the tree can store 30 million items[8] with a maximal depth of 8 levels[9]. As a first estimation, this is only possible if nodes have at least a degree of 9, but this could not be arranged so that the node (9 pointers and keys) fits the transfer registers of a single ME thread.

Another possible option would be to use a forest of N trees that satisfy the constraints mentioned above and to map each key to a single tree without memory lookup (e.g. through a hash function). As a first estimation, 2^{16} B+tree holding at most 4 keys per node would meet the constraints. However this approach would involve a memory overhead of 90% of the values stored[10], and we expect difficult implementation of insertions and deletions in a way that keeps the periodic store cleanup lightweight enough (in terms of additional DRAM bandwidth requirements).

The 'forest' approach would not completely solve the case where one of the trees receives significantly more keys than the

[8] Actually 29,660,000 items, assuming a top rate of 2,966 Kpps, and all WASP packets creating an entry for 10 seconds

[9] Which we experimentally determined as the maximal number of DRAM accesses we can afford per ESS lookup.

[10] Assuming 32 bytes of value per key, keys alone accounting for 25% overhead

rest of the forest either, and is thus just a way to allow "longer chains" with the same amount of DRAM requests. It could thus be preferable to stick to the hash table approach, but to operate it fully in DRAM, and to dedicate enough memory (28MB per store) to hold the hash pointers in order to keep all chains short enough, though this still has to be confirmed by additional performance measurements.

5.2. Best-Effort ESS

Rather than engineering the size of the state store to allow every packet to create new state, which requires up to 1.3 GB for a full-duplex Gigabit Ethernet interface, an operator might prefer to estimate how much memory is sufficient to sustain daily traffic through statistical analysis. Users would then expect the operator to ensure fairness when load is increased beyond the level the installed amount of memory can support. When deciding whether a WASP packet should be allowed to create a new entry, it would then be interesting not only to count how many entries have been created by this *aggregate* in the last τ seconds, but also how long the current packet is.

However, the potential denial of state creation in a router independently of other routers' decision may be inconvenient for applications deriving e.g. from common trunk identification. Indeed, when creating state in nearby routers, WASP programs have no guarantee that further routers will also accept new state. Still, if she optimistically creates state in routers close to her, the user consumes quota for her aggregate and might see further request denied later on – just when she gets the chance to create state near the core.

In case two neighbour domains support WASP and have negotiated a "fair rate" of WASP entries created per second, it would be preferable for applications to be notified (e.g. through node environment variables) whether their packet are "in profile" for upstream entry allocation. A WASP program could then avoid creating state nearby unless it is "blessed" by the router and receives the guarantee that it can install state in the next domain as well.

5.3. Optimising through compilation

Although our interpreter is capable of latencies approaching those of the ESP prototype and throughput slightly outperforming ESP, we must not forget that WASP would keep the ALU of microengines almost fully busy, potentially leading to higher power consumption. Moreover, the `count` and `collect` programs used in our tests remain relatively short (16 bytes) compared to the longest program allowed in WASP (64 bytes).

We thus repeated throughput measurements with "benchmark" packets of variable length, mixing ALU microbytes and access to packet scratchpad variables. It revealed that, although WASP processing time increases linearly with bytecode size, the slope ranges from 7 to 10 times higher than the one of packet forwarding times. In other words, on our IXP2400 setup, a 24-byte program should be placed in a 206-bytes packet to ensure wire speed processing, and a 64-byte program should not be found in a packet smaller than 470 bytes. Unfortunately, for another implementation, the ratio between code size and packet size might differ, and thus this cannot be enforced as a rule for "fair" packets globally.

Compiling bytecode into native code is a well-known technique used to speed up execution of bytecode languages which, if applied here, could allow us to sustain a given traffic with reduced CPU power. In the IXP network processor, each microengine has its own control store, capable of storing 4096 instructions (called *μwords*), and only the XScale core can alter the contents of those control stores.

Our measurement shows that, below 1000 μwords, the control store needs $0.25\,\mu s$ per μword written. It must also be noted that reprogramming is only feasible when the ME is halted, which leads to an extra delay of $30\,\mu s$ in our setup. The handcrafted `collect` function of the ESP filter, for instance, requires 60 μwords of unique code, and we estimate that a direct translation of WASP microbytes into IXP native code could take up to 100 μwords. Assuming that both the XScale and the ME are ready for the reprogramming of a filter function similar to `collect` would thus cost between 45 and $55\,\mu s$. If we suppose that this allows a `W:collect` packet ($3.45\,\mu s$) to have the processing time of `E:count` ($2.57\,\mu s$), it still takes 63 packets to amortise the cost of micro-store reprogramming.

In these circumstances, only very specific traffic patterns could benefit from just-in-time compilation approach with IXP network processors. Yet, if the node can identify the k most used WASP programs whose sizes do not exceed the free space on the micro-store after compilation, it would be theoretically possible to strongly improve the performance of the node while keeping the ability to process *any* program and to dynamically adapt to a new set of popular functions.

6. Conclusions

We designed WASP after lessons learnt from former active routers, taking into account constraints in network processor programming. The result is a scalable *virtual processor* that can safely interpret user-emitted bytecode on router linecards. This has been achieved at the cost of a restricted programming model that does not allow most of the constructs found in a general-purpose programming language. Yet, WASP bytecode is expressive enough to implement several application-specific network measurement and control protocols.

We have implemented and tested WASP VPU on the IXP 2400 network processor in a "filter box" setup. Under low load, the interpreter is competitive with pre-compiled operations as seen in ESP and the advantage of larger entries for the state store has been confirmed. The VPU processing makes however more intensive use of the microengines ALU and throughput will not scale with the number of active threads as well as it does with ESP. It does however scale well with the number of *microengines*. This advocates for integration of both code (ESP and WASP), as well as run-time-compiled optimisations of frequent WASP programs if any, on the same microengine, which would better balance the ALU usage.

We also observed that the increase of the average chain length in the state store may have an important impact on the

forwarding latency of WASP and ESP packets. A mechanism limiting the longest chain will be mandatory to support applications that measure network performance or that try to enforce a given quality of service. The size of the ESS is thus no longer the only key parameter for proper ESS behaviour: the amount of SRAM holding the hash table will define the average chain length and the average latency of ESS accesses.

We initially opted for a interpreter-based solution because IXP2xxx series, unlike other NPUs [6], appear poorly suited to Just-in-Time compilation. It is clear however that an IXP 2400 barely has enough resources to handle a couple of Gbps flows full of WASP packets. In a production version of WASP, performance (and number of MEs available for other components) could be strongly improved by a control component that would compile native code chunks for the most frequent functions present in the bytecode. How we can efficiently identify whether we have or not a native code chunk for a given packet is still ongoing work.

Acknowledgements

Sylvain Martin was a Research Fellow of the Belgian National Fund for Scientific Research (FNRS) and also partially funded by the EU under the ANA FET project (FP6-IST-27489).

References

[1] A. Campbell, S. Chou et al. : *"NetBind: A Binding Tool for Constructing Data Paths in Network Processor-Based Routers"*, in Proc. of OPENARCH'02, June 2002, pp. 91-103.

[2] L. Ruf, K. Farkas, H. Hug and B. Plattner : "Network Services on Service Extensible Routers", in Proc. of IWAN'05, Sophia Antipolis, LNCS 4388, pp. 53-64.

[3] K. Lee and G. Coulson : *"Supporting Runtime Reconfiguration on Network Processors"*, in Proc. of Advanced Information Networking and Applications, Vienna, Austria, April 2006. pp. 721 - 726.

[4] K. Calvert, J. Griffioen, N. Imam, J. Li : *"Challenges in Implementing an ESP Service"*, in Proc. of IWAN'03, Kyoto, LNCS 2982, pp. 3-19.

[5] J. Moore : *"Practical Active Packets"*, PhD Thesis, University of Pennsylvania, 2002.

[6] A. Kind, R. Plekta and B. Stiller : *"The potential of just-in-time compilation in active networks based on network processors"*, in Proc. of OPENARCH'02, pp. 79-90

[7] T. Egawa, K. Hino, Y. Hasegawa: "Fast and Secure Packet Processing Environment for Per-Packet QoS Customization", in Proc. of IWAN'01, Philadelphia, USA, October 2001, LNCS 2207, pp. 34-48.

[8] H. Otsuki and T. Egawa, "A Retransmission Control Algorithm for Low-Latency UDP Stream on StreamCode-Base Active Networks", in Proc. of IWAN'03, Kyoto, LNCS 2982, pp. 92-102

[9] K. Calvert, J. Griffioen, and Su Wen : *"Lightweight network support for scalable end-to-end services"*, in Proc. of ACM SIGCOMM, 2002, pp. 265-278.

[10] M. Conrad and H-J Hof : "A Generic, Self-organizing, and Distributed Bootstrap Service for Peer-to-Peer Networks", in Proc. of IWSOS'07, The Lake District, UK, September 2007, pp. 59-72.

[11] S. Martin and G. Leduc : *"Ephemeral State Assisted Discovery of Peer-to-peer Networks"*, in Proc. of 1st IEEE Workshop on Autonomic Communications and Network Management, Munich, May 2007, pp. 9-16.

[12] S. Martin and G. Leduc : *"An Active Platform as Middleware for Services and Communities Discovery"*, in Proc. of ICCS 2005, LNCS 3516 (part 3) pp. 237-245.

[13] P. Gill, M. Arlitt et al. : "The Flattening Internet Topology: Natural Evolution, Unsightly Barnacles or Contrived Collapse?", in Proc. of PAM08, Cleveland, USA, April 2008, LNCS 4979, pp. 1-10.

[14] J. Lu and J. Wang : *"Performance Modeling and Analysis of Web Switches"*, in Proc. of 31th International Computer Measurement Group Conference, Dec. 4-9, 2005, Orlando, Florida, USA, pp. 665-672.

[15] S. Wen and J. Griffioen and K. Calvert : *"Building Multicast Services from Unicast Forwarding and Ephemeral State"*, in Proc. of IEEE OPENARCH'01Anchorage, Alaska, USA, Apr. 2001, pp. 37-48.

[16] I. Wakeman, A. Jeffrey, R. Graves, T. Owen : "Designing a Programming Language for Active Networks", unpublished work presented at Hipparch workshop 1998. *citeseer.ist.psu.edu/wakeman98designing.html*

[17] S. Martin and G. Leduc : *"Interpreted Active Packets for Ephemeral State Processing Routers"*, in Proc. of IWAN'05, Sophia Antipolis, LNCS 4388, pp. 156-167.

[18] D. Wetherall : *"Active network vision and reality: lessons from a capsule-based system"*, Operating Systems Review, vol.33, ACM, Dec. 1999. pp. 64-79.

[19] S. Bhattacharjee, K. Calvert E. Zegura : "*On Active Networking and Congestion*", Technical Report GIT-CC-96/02, Georgia Institute of Technology. ftp://ftp.cc.gatech.edu/pub/coc/tech_reports/1996/GIT-CC-96-02.ps.Z

[20] S. Martin : *"WASP : Lightweight Programmable Ephemeral State on Routers to Support End-to-End Applications"*, Doctoral Thesis, University of Liège, 2007, ISSN 075-9333. *ftp://ftp.run.montefiore.ulg.ac.be/pub/RUN-BK07-02.pdf*

[21] D. Meng, E. Eduri, M. Castelino : *"IXP2400 Intel Network Processor IPv4 Forwarding Benchmark Full Disclosure Report for Gigabit Ethernet, revision 1.0"*, The Network Processing Forum, March 2003.