**UNIVERSITÉ DE LIÈGE**
**Faculté des Sciences Appliquées**
**Institut d'Électricité Montéfiore**
**RUN - Research Unit in Networking**

# WASP

## Lightweight Programmable Ephemeral State on Routers to Support End-to-End Applications

**Sylvain Martin**
Licencié en Informatique
Aspirant du FNRS

Submitted for the doctoral
Degree in Computer Science

July 2007

# Abstract

We present WASP (World-friendly Active packets for ephemeral State Processing), a novel active networks architecture that enables ephemeral storage of information on routers in order to ease distributed application synchronisation and co-operation.

We aimed at a design compatible with modern routers hardware and with network operators' goals. Our solution has to scale with the number of interfaces of the device and to support throughput of several Gbps. Throughout this thesis we searched for the best trade-off between features (platform flexibility) and guarantees (platform safety), with as little performance sacrifice as possible.

We picked the Ephemeral State Processing (ESP) router, developed by K. Calvert's team at University of Kentucky, as a starting point and extended it with our own virtual processor (VPU) to offer higher flexibility to the network programmer. The VPU is a minimalist bytecode interpreter that manipulates the content of the "ephemeral state store" of the router according to a microprogram present in packets. It ultimately allows the microprogram to drop or forward the packet on any router, acting as remotely programmable filters around unmodified IP routing cores.

We developed two implementations of WASP: a "reference" module for the Linux kernel, and, based on that prototype experience, a WASP filter application for the IXP2400 network processor that proves feasibility of our platform at higher speed. We extensively tested those two implementations against their ESP counterpart in order to estimate the overhead of our approach. High speed tests on the IXP were also performed to ensure WASP's robustness, and were actually rich in lessons for future development on programmable network devices.

The nature of WASP makes it a platform of choice to detect properties of the network along a given path. Thanks to per-flow variables (even if ephemeral) and its ability to sustain custom processing at wire-speed, we can for instance implement lightweight measurement of QoS parameters or enforce application-specific congestion control. We have however opted – in the context of this thesis – for a focus on another use of the platform: using the ephemeral state to advertise and detect members of distributed applications (e.g. grid computing or peer-to-peer systems) in a purely decentralised way. To evaluate the benefits of this approach, we propose a model of a peer-to-peer community where peers try and join former neighbours, and we show through simulations how efficiency and quality of user experience evolve with the presence of more WASP routers in the network.

# Résumé

Nous proposons *WASP* (*World-friendly Active packets for ephemeral State Processing*), une nouvelle architecture de réseaux actifs qui permet de stocker dans les routeurs pendant un court laps de temps des informations provenant des paquets acheminés par le routeur, de manière à simplifier la synchronisation et la coordination des applications distribuées.

L'enjeu principal dans la conception de la plate-forme WASP est d'obtenir un système programmable qui soit compatible avec l'architecture des routeurs modernes et avec les objectifs des opérateurs du réseau. La solution proposée doit entre autres rester efficace quel que soit le nombre de cartes E/S dont dispose le routeur et supporter des débits de plusieurs Gbps. Nous avons donc cherché le meilleur compromis entre les fonctionnalités (pour une meilleure souplesse du modèle de programmation) et les garanties offertes (pour la sécurité du réseau), tout en évitant de sacrifier les performances.

Nous avons pris comme point de départ le routeur *Ephemeral State Processing* (ESP) développé par K. Calvert et ses collaborateurs de l'Université du Kentucky, que nous étendons avec un processeur virtuel (le VPU) pour améliorer la souplesse de sa programmation. Le VPU est en réalité un interpréteur de bytecode minimaliste qui peut manipuler le contenu d'une banque d'état éphémère (*ephemeral state store* ou *ESS*) présente sur la carte d'E/S en suivant un microprogramme attaché au paquet traité. Outre les interactions possibles à travers l'ESS, chaque paquet WASP a ainsi la possibilité de décider s'il doit ou non continuer son chemin dans le réseau. WASP agit donc comme une série de *filtres programmables* autour d'un cœur de routeur IP classique.

En plus de l'implémentation de référence (sous la forme d'un module pour le noyau Linux), nous avons réalisé une version de WASP pour le processeur IXP2400 et nous comparons les performances obtenues avec celles du filtre ESP qui nous a fourni le concept et le code source de la banque d'état éphémère. En particulier, nous avons procédé à une série de tests à hauts débits pour ajuster la robustesse de notre solution et vérifier sa capacité à traiter les requêtes en soutenant le débit des liens Gigabit Ethernet.

Par sa nature-même, WASP est une plate-forme parfaitement adaptée à la détection des propriétés du réseau relatives à une connexion donnée. Grâce aux variables (éphémères) que chaque flux peut maintenir et à sa vitesse de traitement, il est par exemple possible de mesurer certains critères de qualité de service ou de mettre en œuvre des mécanismes de contrôle de congestion propres à une application donnée. Nous avons cependant préféré nous concentrer dans cette thèse sur un autre usage de notre plate-forme: annoncer et détecter via l'ESS les membres d'une application distribuée (telle qu'un système pair-à-pair ou une plate-forme GRID), et ce de façon totalement décentralisée. Afin d'évaluer les bénéfices d'une telle approche, nous proposons un modèle de communauté pair-à-pair où les membres tentent de rejoindre la communauté en recontactant d'anciens voisins et nous montrons, à travers des simulations, l'évolution de l'efficacité de la communauté et de la qualité perçue par les utilisateurs en fonction du nombre de routeurs WASP présents dans le système.

# Contents


**1 Introduction** **1**

1.1 Active Networking . . . 1
1.2 Ephemeral State Processing . . . 2
1.3 Active Packets for Ephemeral Store . . . 3
1.4 More Building Blocks . . . 4
1.5 Project Goal . . . 4
1.6 Structure of This Work . . . 5
1.7 Previous Publications . . . 6

**2 Active Networks** **7**

2.1 What Are Active Networks . . . 7
2.1.1 Packets Carrying Programs . . . 7
2.1.2 Programmable Switches . . . 8
2.1.3 Deploying Active Networks . . . 9
2.1.4 Offloading Active Services . . . 10
2.2 Major Active Platforms . . . 10
2.2.1 ANTS . . . 10
2.2.2 Protean . . . 12
2.2.3 PLANet / SwitchWare . . . 13
2.3 Sample applications for Active Networks . . . 14
2.3.1 Active Caches . . . 14
2.3.2 Multimedia Flow Transcoder . . . 15
2.3.3 MergeCast and video conferencing . . . 16
2.3.4 Active Monitoring and Management . . . 17
2.4 Open Problems and Future Use . . . 18

**3 Network Processors** **21**

3.1 Routers design . . . 21
3.1.1 Traditional Routers Design . . . 21
3.1.2 Intelligent Line Cards and Network Processors . . . 22
3.1.3 Maximum Headroom, Please ... . . . 23
3.1.4 PowerNP and IXP2xxx . . . 23
3.1.5 Related Work . . . 24
3.2 Overview of IXP2400 Network Processor . . . 25

3.2.1 Processing Elements . . . . . . . . . . . . . . . . . . . . . . . . 26
3.2.2 Storage Elements . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.2.3 Developing on the Radisys ENP2611 card . . . . . . . . . . . . . 28

**4 The WASP Platform 33**
4.1 Model of a WASP Router . . . . . . . . . . . . . . . . . . . . . . . . 34
4.1.1 WASP Packets . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.1.2 The WASP Node . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.1.3 World-Friendly Platform . . . . . . . . . . . . . . . . . . . . . . 36
4.2 From ESP Operations to Wasp VPU . . . . . . . . . . . . . . . . . . . 37
4.2.1 A Virtual Processing Unit for Ephemeral State . . . . . . . . . . 39
4.2.2 Packet Variables . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.2.3 Environment Variables . . . . . . . . . . . . . . . . . . . . . . . 45
4.3 The Ephemeral State Store . . . . . . . . . . . . . . . . . . . . . . . 46
4.3.1 Ephemeral State Store Implementation . . . . . . . . . . . . . . 47
4.3.2 Managing the State Store . . . . . . . . . . . . . . . . . . . . . . 48
4.3.3 Finer Access Control . . . . . . . . . . . . . . . . . . . . . . . . 50
4.4 Reference Implementation on x86 . . . . . . . . . . . . . . . . . . . . 51
4.4.1 Validating the VPU's behaviour . . . . . . . . . . . . . . . . . . 52
4.4.2 Experimenting WASP with Linux . . . . . . . . . . . . . . . . . 53
4.4.3 More Efficient Access to ESS in WASP . . . . . . . . . . . . . . 57
4.4.4 Too Cheap, Really ? . . . . . . . . . . . . . . . . . . . . . . . . 60
4.4.5 Node and Interfaces Statistics . . . . . . . . . . . . . . . . . . . 62

**5 Experimenting WASP on IXP2400 67**
5.1 Development on IXP . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
5.1.1 Overall Implementation . . . . . . . . . . . . . . . . . . . . . . 68
5.1.2 Parallel Programming on the Microengines . . . . . . . . . . . . 69
5.2 The WASP microblock . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.2.1 Structure Placement . . . . . . . . . . . . . . . . . . . . . . . . 70
5.2.2 Redesigning the Fetch/Decode . . . . . . . . . . . . . . . . . . . 72
5.3 WASP processing delay . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.3.1 Profiling the WASP microblock . . . . . . . . . . . . . . . . . . 73
5.3.2 There (on the IXP) and Back . . . . . . . . . . . . . . . . . . . . 74
5.3.3 Embedding Measurements on Microengines . . . . . . . . . . . . 76
5.3.4 Larger Entries and `map` Opcode . . . . . . . . . . . . . . . . . . 79
5.4 High Availability at Higher Rates . . . . . . . . . . . . . . . . . . . . 81
5.4.1 Behaviour of the ESP microblock . . . . . . . . . . . . . . . . . 81
5.4.2 Behaviour of the WASP Microblock . . . . . . . . . . . . . . . . 85
5.5 Throughput Tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.5.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.5.2 Results with Count/Compare Instructions . . . . . . . . . . . . . 89
5.5.3 Results with Collect Instruction . . . . . . . . . . . . . . . . . . 91
5.6 Compiling WASP programs on the IXP . . . . . . . . . . . . . . . . . 92

5.6.1 Environment for Run-time Compiled WASP Programs . . . . . . 93
5.6.2 Just-In-Time Compiling of WASP programs . . . . . . . . . . . . 94
5.6.3 Towards Self-Optimizing WASP Component . . . . . . . . . . . 96
5.7 Uninterrupted Processing: Lessons Learned . . . . . . . . . . . . . . . . . 98
5.8 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100

**6 WASP as Discovery Middleware 103**
6.1 The case for Discovery Middleware . . . . . . . . . . . . . . . . . . . . 103
6.2 Discovery: Flavours and Existing Solutions . . . . . . . . . . . . . . . . 105
6.2.1 Local Service Discovery . . . . . . . . . . . . . . . . . . . . . . 105
6.2.2 Global Service Discovery . . . . . . . . . . . . . . . . . . . . . 105
6.2.3 Proxy Services in a Transit Network . . . . . . . . . . . . . . . . 106
6.2.4 Joining a Peer-to-Peer Community . . . . . . . . . . . . . . . . . 107
6.3 MagNet: Service discovery with WASP . . . . . . . . . . . . . . . . . . 108
6.3.1 Flooding Locally . . . . . . . . . . . . . . . . . . . . . . . . . . 110
6.3.2 Persistent Data in Ephemeral Store . . . . . . . . . . . . . . . . . 110
6.4 History File Processing . . . . . . . . . . . . . . . . . . . . . . . . . . 111
6.4.1 The Community Model . . . . . . . . . . . . . . . . . . . . . . . 111
6.4.2 Bootstrap Quality Indicators . . . . . . . . . . . . . . . . . . . . 112
6.4.3 Behaviour on a “Regular” Network . . . . . . . . . . . . . . . . 113
6.5 Active Domains boosting P2P . . . . . . . . . . . . . . . . . . . . . . . 115
6.5.1 Registering Membership in the State Store . . . . . . . . . . . . 116
6.5.2 Keeping the Community Running . . . . . . . . . . . . . . . . . 117
6.5.3 Getting the Community Running . . . . . . . . . . . . . . . . . . 118
6.5.4 Other Affecting Parameters . . . . . . . . . . . . . . . . . . . . 119
6.5.5 Dynamic Addressing vs. Active Domains . . . . . . . . . . . . . 119
6.5.6 Avoid the Need for an Initial List . . . . . . . . . . . . . . . . . 120
6.6 Enforcing Registration Fairness . . . . . . . . . . . . . . . . . . . . . . 122
6.6.1 Hash-Requesting Packets . . . . . . . . . . . . . . . . . . . . . . 122
6.6.2 Accessing Election Result . . . . . . . . . . . . . . . . . . . . . 124
6.6.3 Practical Implementation of Code Hashing . . . . . . . . . . . . 124
6.7 Conclusion and Future Work . . . . . . . . . . . . . . . . . . . . . . . . 126

**7 WASP and Beyond 127**
7.1 Rerouting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
7.1.1 Issues with Rerouting . . . . . . . . . . . . . . . . . . . . . . . . 128
7.1.2 Network-Friendly Rerouting . . . . . . . . . . . . . . . . . . . . 129
7.1.3 Validating Source Addresses . . . . . . . . . . . . . . . . . . . . 131
7.2 Multicast to Small Group . . . . . . . . . . . . . . . . . . . . . . . . . 133
7.2.1 Small Group Multicast . . . . . . . . . . . . . . . . . . . . . . . 134
7.2.2 Application-level Multicast . . . . . . . . . . . . . . . . . . . . . 134
7.2.3 Multicasting with ESP and Lightweight Modules . . . . . . . . . 135
7.2.4 Building Small-Group Multicast with WASP . . . . . . . . . . . 135
7.2.5 Pending Problems with WASP-SGM . . . . . . . . . . . . . . . . 138

7.2.6 Interconnecting Multicast Islands . . . . . . . . . . . . . . . . . . 140
7.3 Deployment scenarios . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
7.3.1 WASP-aware line card . . . . . . . . . . . . . . . . . . . . . . . 141
7.3.2 WASP filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
7.3.3 WASP in a non-intrusive test bed system . . . . . . . . . . . . . 142
7.3.4 Isolating WASP traffic on the router . . . . . . . . . . . . . . . . 143
7.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145

**8 Conclusion 147**
8.1 Towards a WASP Socket . . . . . . . . . . . . . . . . . . . . . . . . . 147
8.2 Rethinking the State Store? . . . . . . . . . . . . . . . . . . . . . . . . 148
8.3 More on the “Best Effort” We Provide . . . . . . . . . . . . . . . . . . 148
8.4 The Role of WASP in Autonomic Networks . . . . . . . . . . . . . . . 149
8.5 Towards User-Friendly Rerouting in WASP . . . . . . . . . . . . . . . 149
8.6 Benefits of WASP for Research Network Operators . . . . . . . . . . . 150

**A WASP Opcode Reference 153**
A.1 Control Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
A.2 STACK Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
A.3 ALU Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
A.3.1 Miscellaneous ALU Operations . . . . . . . . . . . . . . . . . . 154
A.4 Branch Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
A.5 Memory Operations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
A.5.1 Memory Movement Operations . . . . . . . . . . . . . . . . . . 155
A.5.2 Index Register Manipulations . . . . . . . . . . . . . . . . . . . 156

**B WASP and ESP Packets Format 157**
B.1 Count Packet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
B.2 Compare Packet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
B.3 Collect Packet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
B.4 Rchild Packet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
B.5 Rcollect Packet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159

**C Patches brought to ESP 161**
C.1 Erratum #69 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
C.2 Wrong CRC polynom . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
C.3 ESS chains update . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
C.4 No Queue Manager . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
C.5 Overlapping Ring-Buffer . . . . . . . . . . . . . . . . . . . . . . . . . 162
C.6 Leaking Classifier . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
C.7 Trashing CRC on Header Update . . . . . . . . . . . . . . . . . . . . . 162

**D Code Samples 165**
D.1 Microstore Reprogramming Benchmark . . . . . . . . . . . . . . . . . 165

# Acknowledgements

There are several people that have all brought their small stone to this work and I want to thank them all, even if I may miss some of them here.

Of course, I'd like to thank Pr. Guy Leduc for his invaluable support, and Pr. Ken Calvert for his interest in my work. Special thanks also fly to Jiangbo Li for his help on understanding the ESP package, Lukas Ruf, Jean-Patrick Gelas and Lennert Buytenhek for their expertise and advice on the ENP2611 board, Cyril Briquet and Cyril Soldani for their discussions on Peer-to-Peer and Grid systems, and Hugues Smeets, a long-time friend whose wisdom was precious to "keep things small and simple" when designing WASP virtual processor.

Enfin, j'adresse bien-sûr un tout grand merci à mon épouse Violaine et à ma famille pour leur patience et leur attention tout au long de ce marathon.

*There's no place like 127.0.0.1*

# Chapter 1

# Introduction

Not less than 10 years ago, the idea that the whole planet would be interconnected by a network capable of supporting instant communications, conferencing and gaming for millions of individuals, which could offer access to distributed data storage capable of answering more questions than professional encyclopedias and that encyclopedias, scientific publications and daily news themselves would move to that new media and be maintained collectively by communities of individuals was still science fiction.

Not less than 5 years ago, the idea that this planetary network known as (the) Internet would be accessible *anywhere* by devices you could carry along for the whole day was science fiction as well. Nowadays, "infospheres" detailed in scifi novels only differ from today's Internet by the exotic "neural interfaces" scifi authors can think of...

Yet, managing and maintaining that network remains a daily burden for thousands of operators and administrators all over the world, requiring constant manual intervention on equipment for upgrading, repairing and maintenance. Many scientists as well as field technicians agree to say that IP as we know it today has shown its limits in several ways, yet proposals for a next generation of planet-wide network layer haven't replaced it so far – for many reasons, few of them being technical.

## 1.1 Active Networking

During the last decade, the *active networking* research field has investigated and evaluated ways for "bringing life to the network", evolving from the static management nightmares we know into a dynamically modifiable architecture where protocol stacks could be updated to face end-users and operators demand – the same way as "plugin" technologies have allowed extensibility of applications on desktop systems.

With active networking deployed, we could benefit of sufficient programmability to make QoS, multicast and whatever else the future of the Internet brings a simple matter of injecting code into the network. Many different ways to achieve those goals have been proposed, and the most relevant to this work have been summarised in section 2.2. Some proposals replace IP itself while others simply "extend" the existing network layer. Others again are orthogonal to data forwarding and rather act as a new generation of control

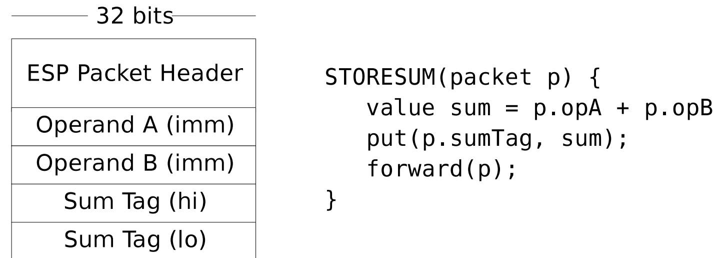



**Figure 1.1**: *Packet format for the* `sumstore` *operation and pseudocode to be executed on the router when such a packet is received.*

protocols. In all cases, supporting active networking with legacy hardware is usually not possible as protocol implementations are often hardwired in equipment. However, recently introduced *network processors* [HiFn04, IntelPRM, IntelHRM] have changed the way routers are designed and it is now only a matter of hardware and human resources investment to deploy a programmable solution in a small sub-network. As those processors evolve, higher bit-rates and higher loads are achieved with the same programmability, potentially making the technology available closer and closer to the core of the network. Bringing an open programmable platform to the Internet is no longer a technical impasse, which may give a second breath to active networking. Chapter 3 will review two of those processors and their potential applications in the field of active networks.

## 1.2 Ephemeral State Processing

*Ephemeral State Processing* (ESP) [Calvert02] is one of those approaches to make the network more "informative and capable" developed by Dr. Kenneth Calvert and Dr. James Griffoen at University of Kentucky. Its goal is to provide an open interface to deposit, retrieve and process small pieces of data in routing elements while keeping IP's inherent properties such as robustness, anonymity, generality and processing cheapness. ESP does not propose a *replacement* for Internet Protocol, but rather comes as an extension to the existing network layer that should provide the bare minimum flexibility required to help user applications build more capable solutions such as reliable multicasting [Calvert01].

ESP defines a small set of generic operations that manipulate the state stored on the routers. Each *ESP packet* processed by an ESP router invokes one of these instructions over a specific set of data and may lead to modification of the ESP packet's content, modification of the router's stored data and a decision on whether the packet should be forwarded or discarded. To better figure out how this works, we will reuse the `SUMSTORE` packet example from N.R. Imam's thesis [Imam03]. We assume that ESP routers implement pseudocode defined on Fig. 1.1. The header of ESP packets contains a numerical operation code that identifies the computation to be applied. Once the router identifies the ESP packet as such, it retrieves the numerical opcode and branches towards the processing routine that implements that specific operation.

The "sumstore packet" also contains the operands to be applied for that specific computation, such as the immediate values involved in the computation and the specific entry in the router's store (identified by the "sum tag") to be manipulated. Values are extracted

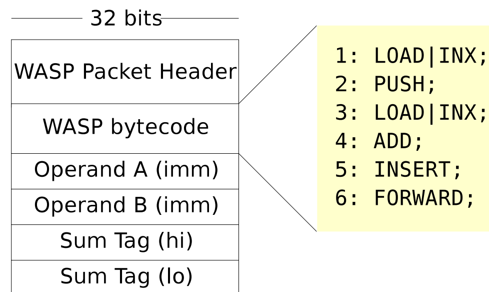



**Figure 1.2**: *Structure of a WASP packet for the `sumstore` operation with detailed bytecode implementing the operation.*

from the packet, addition is computed and written into the router's store under the "sum-tag" key, as detailed by the pseudocode given on Fig. 1.1. The tag is a 64-bit integer that acts as a variable name on that router. For a short period after that packet has been forwarded (and that's why the store is said to be *ephemeral*), any packet that reuses the same tag can retrieve the value computed by the `SUMSTORE` packet.

## 1.3 Active Packets for Ephemeral Store

ESP operations can be used in more complex applications such as reliable multicast: receivers that detect a missing frame send a `NACK` packet that sets state in routers towards the source. That state can be used by other `NACK` packets to avoid feedback implosion at the source and by the retransmitted frame to focus retransmission towards receivers that are actually missing the frame. An implementation based on the generic `COMPARE`, `COUNT` and `COLLECT` operations is given in [Calvert02]. However, as the authors propose more applications, it appears that new – and more complex – operations are required. The more complex operations become, the less obvious it is for the network application designer to pick the correct one. Moreover, such application-dedicated operations contradict with the generality principle that is essential to wide deployment of any solution.

In WASP (World-friendly Active packets for ephemeral State Processing) we attempt to overcome such situations by replacing the pre-defined generic operations of ESP by "microcode" carried by packets themselves, much like early works in Active Networking suggested that *capsules* would carry their own forwarding code instead of using IP's generic function, with the notable difference that an ESP operation is orders of magnitude simpler than IP forwarding. A WASP router is thus in many ways similar to an ESP router, with its packet classifier, its ephemeral stores and its IP routing logic, but it replaces the set of pre-defined (and pre-compiled) operations with an interpreter for the program expressed by the microcode.

Fig. 1.2 shows the code to be attached to a WASP packet to implement the "sumstore" instruction. Each of these "micro-instructions" is one byte long in the packet's program. The *Virtual Processing Unit* interpreting these packets is detailed in section 4.2. For now, we can assume that WASP works like a stack-based CPU. Operands A and B are placed on the stack (lines 1-3) and then added together (4). The "INX" modifier advances the "data pointer", accessing packet's operands in a way that might remind you Turing machines,

and when "insert" is invoked, the data pointer is over "Sum Tag". The value on top of the stack is thus written under the key "Sum Tag" in the ephemeral store (5), after which the packet is forwarded.

## 1.4 More Building Blocks

In addition to that interpreter, WASP also extends the environment provided by ESP. It is for instance possible to retrieve basic statistics about a given interface or generic information about the router itself (such as its network address, QoS and multicasting abilities or a local time counter). These additional building blocks enable interesting network monitoring functions which can benefit from the ephemeral store, but that weren't possible with ESP alone. With the sole addition of a local time stamp, for instance, an end-user application can program packets to monitor the *jitter*[1] they experience within a network. The potential benefit of such measurements to validate inter-domain service layer agreement (SLA) has been presented in [Boschi05].

The local time $t_0$ observed when packet $P_0$ crosses the router is written in the ephemeral store and next packet $P_1$ can compare this with observed local time $t_1$ and build average, maximum and minimum values over a few packets (how few should be decided according to the "ephemerity" of the store) before a special packet collect those values in each router.

Other statistics such as depth of output buffers, transmission errors, etc. or access to the network packet header will of course allow more applications. Those applications mainly focus on *small tasks* that could be helpful for monitoring or detecting properties of specific network areas such as path taken by a given flow or a tree towards a given destination from multiple sources. In parallel, WASP may also be a key feature to ease the deployment of fully distributed systems such as peer-to-peer overlays since its presence *within* the network could naturally replace things that currently require centralized servers such as discovering existing peers of a community.

## 1.5 Project Goal

In this work, we have defined and implemented a reference version of the WASP VPU that has been integrated in the ESP implementation for Linux/netfilter architecture. Whether interpretation of WASP "bytecode" programs on a router could be made "too cheap to meter" as ESP is not obvious, and one of the challenges we explored with that reference implementation is what techniques can make the interpreter competitive over native code that implements ESP operations in an ESP router before an implementation based on IXP2400 network processor could be envisioned.

ESP had a very conservative approach about packet sources and destination addresses, in the sense that all a packet could do to affect its forwarding is to drop itself. In WASP, we investigate to what extent we can offer more flexibility to the network programmer. In

[1] variation of inter-packet reception time

many cases it can be interesting to allow packets to return prematurely to their source or to allow them to change their destination under strict control of the router.

A cornerstone of this project was the implementation of the WASP interpreter on a IXP2xxx network processor. Along with this document, we produced a proof-of-concept implementation of the interpreter on a "filter" box in order to study its performance compared to a native operation as in ESP, and whether it is realistic to hope wire speed processing of WASP packets.

Finally, we intended to explore the potential applications of our new framework. From the start, it was felt that most interesting WASP applications would be in a "middleware" role where it can offer support and network knowledge that distributed applications typically lack. This is probably the aspect of this thesis where most of the future work should be done, building real test cases with modified applications and IXP or x86 variant of the WASP filter box.

WASP initially stood for "Weightless Active packet for ephemeral State Processing". While working on the project it has become clearer and clearer that "W" could not be "Weightless" (as low as we made it, there *is* an overhead against ESP) but should rather be "World-friendly". An architecture like WASP would have no chance to escape research labs nowadays if it were not Inherently Safe and Resource-Aware for both the Routers, Network Operators and End-User point of view. We hope the reader will agree with the fact that "WASP" sounds better that "ISRARNOEUASP" and forgive the resulting buzzword.

## 1.6 Structure of This Work

In chapter 2, we will review active networking research and cover the major active platforms that have inspired the development of the WASP project. Chapter 3 will then give the reader an overview of what network processors look like and how they can be used to build programmable routers. We will detail the IXP2400 processor that is being used in this work as well as the development environment around the *ENP-2611* Radisys card, a commercial product embedding IXP2400 core in a Gigabit-router-on-PCI-board.

Chapter 4 presents our WASP architecture, its internal design and some applications it can support in addition to ESP operations. It will also present the WASP prototype for the Linux kernel and results of performance benchmarks. The IXP2400 implementation is then discussed and compared to ESP in chapter 5.

WASP can also have a great impact as a *middleware* for larger distributed systems. In chapter 6, we illustrate this role through two examples: service discovery and boosting joining procedures in a peer-to-peer community.

In chapter 7, we explore the benefits and drawback of allowing WASP to alter the destination of packets, a feature that could potentially boost the flexibility of the platform, but that turned out to be unpractical when trying to implement it using WASP only.

Our conclusions are finally presented in chapter 8.

The reader can find detailed programming information about the VPU in appendix A. For reader's comfort, we included the packet formats and pseudocode of ESP operations in the appendix B (original documentation in from [Calvert05w]), facing the correspond-

ing WASP microbytes emulating them with the VPU. Full sources of our reference implementation can be found online [Martin06w]. The detail of patches we brought to the esp-ixp2400 package is given in appendix C.

## 1.7 Previous Publications

We presented WASP router and the aspects relative to rerouting at IFIP International Working Conference on Active Networks (IWAN'05) in [Martin05b]. A previous paper presented at "Active and Programmable Grid Architecture and Components" workshop of ICCS 2005 ([Martin05]) addressed the potential application of WASP as a service discovery framework. The aspects of using WASP as a peer-to-peer discovery middleware have been presented at the 1st IEEE Workshop on Autonomic Communications and Network Management (ACNM'07) in [Martin07].

Besides those papers that are directly related to this work, we presented RADAR in [Martin03] and [Martin02], a topology-discovery framework specially designed to meet the lack of automatic setup of overlays for ANTS heterogeneous networks. Some of the aspects of RADAR can be useful when using WASP for service discovery, and in other aspects, WASP could be used as a substitute for RADAR, providing a way to discover "active forwarders" in a network without requiring the complete active platform to be deployed in all equipment.

A derived version of our "state of the art" chapter and "service discovery" section have been integrated in the deliverables 1.1 [Jelger06] and 2.1 [Fdida06] of the European IST "ANA" Project (Autonomic Network Architecture), respectively.

*Captain's log, stellar date 139.165.223.2*
*It's now been weeks since we're stuck in Autonomous System 7007, looking for a class C domain we could land on ...*

# Chapter 2

# Active Networks

## Abstract

*This chapter gives the reader an overview of past research in Active Networks, and more specifically, focusses on the applications that motivated the Active Networks approach – most of which have then turned into "use cases" for many researchers.*

*We also study the concepts and mechanisms in use in three "major" platforms against which WASP will be compared later in this work.*

## 2.1 What Are Active Networks

In the traditional model of Internet, router software contains the whole logic to forward packets to the proper interface solely based on the *destination address* carried by packets and the content of the *routing table*. All the additional complexity (congestion control, error recovery) is handled by end systems. Nowadays this simple "store and forward" model is extended over and over to provide more security (firewalling and filtering rules), better guarantees (differentiated services through queueing and scheduling) and improved performance (explicit congestion notification, local packet caching and retransmission).

Breaking with that model, the DARPA "Active Networks" project proposed that end-systems (and thus end-users) could replace that plethora of specialised protocols with *programs* injected into the network and processed by the routers, which would turn in some *execution environment* combined with an abstract operating system [DARPA99] that would coordinate the evaluation of those programs.

### 2.1.1 Packets Carrying Programs

Amongst the proposed models, the *capsule* is a paradigm where each packet contains the complete program stating each step to be performed in order to forward the packet. A typical capsule will read some packet status, lookup for information left in the active router by other capsules and then issue a "forwarding table" lookup to the NodeOS to know on which interface it will go. When pushed to the extreme, the capsule could even

choose its position in the interface's queues or completely handle its own forwarding table. Since it's fairly difficult to put such complex programs within data packets, the ANTS project [Wetherall98] proposed that code should actually be downloaded separately and *cached* on the router. Each ANTS capsule only carries a *reference* of the code that should be used (computed from a MD5 hash of that code) for the purpose of locating the appropriate code in cache or to download it from the previous node.

On the other side, in the CANEs project [Merugu99, Sanders01], programmability is only offered at pre-defined *slots* that are present in a *generic* packet processing function. In essence, this is very close to the programming model of *netfilter* in Linux kernel.

The various active platforms proposed usually differ by the technique they use for transporting code, by the control they offer on the underlying node and the mechanisms they enforce to ensure proper operation of the node.

Running user-issued programs within the network has however raised a number of issues. While many approaches have been envisioned to make those programs safe (through interpretation or sandboxing, for instance), or give proof of their safety (known as proof-carrying code), it remains unlikely that we could afford such checks for every data flow the router should handle, and the deeper we go into the network, the more unlikely it becomes.

### 2.1.2 Programmable Switches

For the network operators point of view, giving user control on how the packet should be handled is perceived as a threat rather than a progress, since planning and provisioning bandwidth becomes almost impossible. However, the idea of having quickly deployable packet-processing functions on demand has received significant interest. The notion of *active services* encompasses those projects where extensible functionalities are available to the user by means of operator-provided *plugins*. The selection of those plugins, the subset of nodes where they could be installed and the downloading and installation procedure remains under the strict (automated) control of the network operator.

The way users specify which active services should be applied to their own packets may vary from platform to platform, but the general trend is to opt for a web service where the users set up *overlay topologies* interconnecting sites that should benefit of the same service. The main drawback of this approach is that it's often hard to extend it to the interdomain, since it's unlikely that all the operators on an Internet path will agree on which plugins to provide, the very same way they don't agree on a global multicast or QoS infrastructure.

It is therefore not surprising that research being done in "programmable" networks aims at different goals than "active" networks proper, for instance more focusing on techniques that allow quick deployment and autonomic reconfiguration of distributed Spam/DDoS/Worms detection and prevention mechanisms for the operator rather than offering more flexibility to the end-users.

### 2.1.3 Deploying Active Networks

How could we test a new network paradigm that changes even the way forwarding is achieved ? The idea of having "native" packet format for active packets, replacing even the IP layer, and being exchanged only between active routers has been abandoned quite early, as well as the idea of using something else than IP for locating machines that were part of the active network.

Yet, it was necessary, for testing purpose, to have paths with active routers *within* the core network, so the A-Bone [Berson02] project has been set up and offered NodeOS-compliant platforms for packets carrying the Active Network Encapsulation Protocol (ANEP) header.

Apart from test bed environments, however, it's unrealistic to expect every router to be active (and to support the execution environment a given active packet requires). Core routers typically handle millions of flow (c.f. [Sivakumar00]) and therefore need to be as stateless as possible. Moreover, technologies like MPLS are common within the network core, meaning that the packet might actually not even *see* core routers.

In several proposed architectures, only the *edges* of a domain are active and interconnected with legacy hardware. The main two proposed approaches to handle heterogeneity of active/legacy nodes within the network are *active option* and *overlays*. In the overlay approach, active nodes in the network are identified and connected to each other by means of *tunnels*, making the active packet appear as a legacy packet until the next active node receives it, and making the tunnel appear just as a point-to-point wire for the active nodes.

On the other side, the active option paradigm [Wethereall96] suggests that active routers will be able to identify active packets and do what's required and that legacy router will handle them exactly the way they handle legacy packets. In this framework, it is up to the active protocol designer to ensure that her solution keeps working (or degrades gracefully) when the density of active nodes in the system decreases.

Both approaches have benefits and drawbacks. While overlays maintain the illusion of a fully-active network, they involve substantial building and maintenance overhead. Several works explore how this can be done in specific applications (such as YOID [Francis00] in the case of multicast transmission or Chord and Pastry [Stoica01, Rowstron01] for distributed hash tables), but dealing with unplanned overlay requirements across different network operators remains unsolved so far and solutions sketched such as X-Bone or OPUS [Touch00, Braynard02] require a quite heavy infrastructure for operating successfully.

Another challenge for overlays is to maintain the illusion that the tunnel has foreseeable characteristics while the underlying network might very well change the route tunnelled packets take, suffer from congestions due to "perpendicular" traffic, etc. A regular point-to-point wire will not reorder packets it carries; a tunnel could very well do, and it is not obvious to predict when it will do[1].

Several parameters distinguish overlay networks from 'real' networks, like the fact that link costs may change at any time (due to a change in the underlying topology), or the fact that there's usually no broadcast facility to discover peer routers.

---

[1] This is discussed in section III (Protean Network Management) of [Sivakumar00]

With the "active option" approach, no such infrastructure is required, but we might very well find no active router on a given path while a small deviation of the standard path might have found one. Moreover, it's up to the application designer to ensure its solution degrades nicely when fewer nodes in the system are active.

[Sivakumar00] highlights 4 questions to be answered by an active network architecture:

1. Where should active routers be placed in the networks?
2. If not all routers are active, how can we abstract the state of 'non-active' portions of the path?
3. How do active routers discover and communicate with other active routers?
4. How do users discover third-party services?

### 2.1.4 Offloading Active Services

To some extent, many of the applications suggested by the active network community do not need to take place *on the routers*. They mainly need to be applied *within the network*. If an overlay topology is built, nothing actually requires that the machines that process active packets *are* the physical devices that forward the packets.

In many "high-performance active node" research [Calvert99], active packets are identified by the ingress line card and dispatched through the switch fabric to a "processing card", an additional hardware component that acts as a line card as far as the switch fabric is concerned, but which actually features general-purpose processor capable of doing the active processing required and then re-inject the packet in the switch fabric once its actual destination (or queue level) has been decided.

The Tamanoir Active Node [Gelas02] extends this concept by suggesting the use of a processors cluster site connected to the border routers so that even heavy tasks (such as video flow transcoding) can be achieved at high rates.

However, the authors of [Sivakumar99] claim this approach cannot effectively support many of the services that motivated the design of active networks in first place. It would be for instance more difficult for an "offloaded" active service to know the current state of the routers it monitors (e.g. what is the *instantaneous loss rate* of a wireless link, and should we provide more forwarding error code to increase the chance that the packet transmits successfully).

## 2.2 Major Active Platforms

### 2.2.1 ANTS

ANTS [Wetherall01] is a capsule-oriented platform, where code used for packet processing has access to a limited set of primitives:

- environment access functions such as retrieving node address, channel properties, localtime.

- store and retrieve data from the protocol-specific *soft-store*
- route the capsule towards another node or deliver it to a local application.

Capsules code is organised into *protocols* (e.g. set of capsules that share code and have access to the same *soft-store* on a router). Both capsule and execution environment is written in JAVA, which gives the portability, mobile code support and a good substrate for the required safety. ANTS comes with its own code download mechanism, based on MD5 hashing of the required bytecode (carried by the capsules), so that the *previous active node* can be requested to transmit the code group needed for interpretation of a given capsule while maintaining the guarantee that the code that will operate on the capsule *is* the code expected to be used[2].

An open-source distribution of ANTS running on top of the Java Active NodeOS ([Tullman01]) is available at university of Utah since 2001, which has led to a collection of ANTS-derivative works, some improving the performance while maintaining the programming model, others providing alternate code download mechanisms to keep services deployment under the control of the network operator. While ANTS itself supports partially active networks, nothing is provided to automate the process of setting up neighbour active routers. In the specifications, ANTS node could also host various *services* that capsules could connect to using the `findExtension` primitive. Such services could have featured a database lookup, or any other 'specialised' function, but it has - to our best knowledge - never been used or implemented.

Beyond the fact that its unclear a heavy environment such as JAVA can be supported in a core router[3], the design of ANTS raises other issues.

First, each protocol has the option of storing data in the *soft store* (a key-based object storage). These data can be explicitly removed by the protocol or they can be reclaimed by the node after a pre-defined 'idle' timeout. Every time a specific item in the soft store is reused, the 'idle' timeout is restarted, which means a thin flow of capsules is sufficient to lock some memory on the node for arbitrarily long periods of time. To keep things running, ANTS will use per-context memory allocators, making sure than one protocol cannot consume the whole memory of the node. However, defining the appropriate limit while maximizing the number of running protocols is not a trivial task. This however implies that several 'contexts' may have to synchronise themselves and cooperate to handle a capsule, for instance when code for that capsule need to be downloaded or when the capsule is delivered to an application, which results in additional context switching and inter-context communication overhead.

In addition to ease memory management, per-protocol store is a primary requirement to ensure it is not possible to write a 'scanning' protocol that crawls the network, detecting what other protocols are running and modifying their state for malicious purposes: protocols only compete for resources but they do not interfere with each other state or capsules.

---

[2] as long as we trust MD5 as a one-way hash function, and if we can trust the router operator to run a properly-implemented ANTS router

[3] [Wetherall99] reports 4,000,000 packets/sec (minimal packet size of 70 bytes at OC-48 wire speed) in the CISCO 12000 series, cf. [Tolly99]. On the other hand, the "Click Modular Router" achieves 'only' 70,000 packets per seconds on PC hardware according to R. Morris et al. (SOSP'99)

By a sophisticated combination of JAVA sandboxing features, appropriate design patterns for exposing information about the node without letting capsule code alter them, and thanks to the inherent type-safety and bounds-checking of the language, ANTS manages however to keep the node as a safe place for everyone – what alternate solutions based on native code cannot usually enforce unless the code has been compiled locally with a 'secure' compiler adding safety checks.

Yet, malicious code *may* be written in the ANTS toolkit. Because the JAVA language provides no mechanism to guarantee code completion in bounded time constraint, even a small capsule can consume CPU resource indefinitely. In ANTS/Janos platform, this is achieved by *watchdog timers* that will ensure long-running forwarding routines are terminated. Once again, it is not obvious to define those watchdog timers so that they catch *only* misbehaving code. A more concerning issue is that, according to Hawblitzel et al. [Hawblitzel98], it is unsafe to terminate runaway threads in the JVM[4].

A steady flow of capsules where each capsule executes a small action could very well be consuming the same amount of CPU resources as a sparse flow where each capsule asks for long operations and yet the later one would be terminated by the watchdog and the code would likely to be tagged as misbehaving, preventing subsequent capsules with the same code to execute on the node.

From [Wetherall99], there's an intriguing idea that TTL-based bounds may not be enough (even in the 'PLAN' approach where TTL of child packets is subtracted from TTL of parent packet), because they do not prevent a given packet to consume all its resource *in a specific location*. In other words, an initial resource bound required to reach a fair amount of receivers through multicast-like service may still allow an attacker to send a capsule that quickly reaches its target and then loops or ping-pongs between a few systems in the target's domain[5].

### 2.2.2 Protean

The PROgrammable TEchnology for Active Networks project [Sivakumar00] follows the 'extensible services' approach. A common packet classifier decides which *user network context* (UNC) should be applied to each packet. Those contexts contain a collection of *(event, handler)* binding, customizing the processing that the packet will receive. Events include system events such as "incoming packet", "packet ready for transmission" or "packet dropped", but also triggering of periodic timers or custom events that other event handlers fire off. The handlers are kept in dynamically loadable kernel modules, identified via *unique service profiles* and optionally retrieved from other nodes using the dedicated SPINE infrastructure.

The protean switch is equipped with a compiler for a safe subset of C language (subC, see [Sivakumar99b]) that will produce native code safe for kernel-level operations out of

[4] quoted from [Moore02b]

[5] the authors of [Wetherall99] suggests that (1) code that can be proved "safe" can get executed (e.g. forwards toward fixed destination and don't create clones), (2) "unsafe" code is reviewed and validated by a trusted authority using digital signature that can be checked by active node to know whether they can be running the stuff without headaches and (3) that remaining code gets "best effort" servicing

source text downloaded from a nearby cache or directly from the source – a technique that is quite common in the "programmable switch" approach[6]. From a security perspective however, this might be easier to fool than ANTS's hashing scheme (e.g. nothing guarantees that the source retrieved from a nearby cache will actually do what the end-user expects).

Unlike what happens in *ANTS*, it does not only define the node architecture, but also network services that are required to support operations of the nodes. Among other things, the PROTEAN framework suggests that only a portion of the nodes – the edge of the domain – might be programmable by the user, for scalability purposes. In addition, PROTEAN comes with its own hierarchical lookup service to retrieve the storage location of a code module based on compound names such as "edu.uiuc.crhc.timely.siva.md5cksum". Finally, PROTEAN offers a *link abstraction* to compute and provide estimated characteristics of tunnels between programmable nodes in terms of loss rate, delay and bandwidth, helping the services programmer build services that will work even when not all nodes are active.

### 2.2.3 PLANet / SwitchWare

Compared to most other projects that distribute the active code either via *out-of-band* mechanism (e.g. active packets contain an identifier used to retrieve the plugin either from a cache or from a code server [Bossardt02]) or via *in-band* mechanism (e.g. the code is present in the packet), PLANet uses an interesting mix of the two. The active packets contain scripts expressed in a safe language – PLAN [Hicks98]: A Packet Language for Active Networks – that will make use of out-of-band installed *active extensions* that are under the control of the network operator. The PLAN scripts therefore acts as a "glue" for operator-offered services composition.

Safety and security in *PLAN* is achieved through the language design rather than from virtual machine properties. PLAN is a functional language whose expressibility is limited but that guarantees termination of programs. Active extension can balance that limitation by offering more complex services.

One of the main drawbacks of PLAN is that the program representation can be quite long and that converting that representation into something that can be handled by the interpreter (e.g. parsing the code, unmarshalling packet-carried data) can be inefficient. SNAP (Safe Networking with Active Packets [Moore99]) is another packet language defined in the context of the *SwitchWare* project of University of Pennsylvania designed to fix that problem. While PLAN uses a high-level functional language that has to be converted between network representation and executable representation at each node, SNAP bytecode is executable *in situ* in the packet buffer, combining a compact representation with high speed interpretation.

For a programmer's point of view, SNAP splits the packet between "stack" and "heap", where the heap contains for instance the data arrays for the active extensions. The authors of SNAP have proven that each instruction either grows the stack by one item, or shrinks the stack. Moreover, SNAP can only take *forward* jumps, which means that evaluation

[6]the OKE Corral (IWAN'2002) uses similar approach

time is now a *linear* function of the code size (in PLAN, the execution time could be bounded by a function of the code size as well, but it could be *exponential*). Finally, when a new packet is sent, its initial *resource counter* value is subtracted.

As a result, if we consider a set of packets $N = P_1, \ldots, P_n$ in the network, the sum of resource counters for each packet is a *termination function* for the computation expressed by those packets, and thus bounds the overall amount of processing required on that packets. The following properties are demonstrated on SNAP packets of length $|p|$ (in bytes):

1. On any node, processing a packet $p$ only takes $O(|p|)$.
2. On any node, processing a packet $p$ only requires $O(|p|)$ memory.
3. The overall network bandwidth consumed by a packet $p$ is at most $O(n|p|)$ where $n$ is the resource bound of the packet at its creation.

An exhaustive list of SNAP bytecode is given in Jonathan T. Moore's dissertation "Practical Active Packets" [Moore02]. Among the interesting derivative works, a compiler to translate PLAN into SNAP is presented in [Hicks01] and a just-in-time compiler of SNAP bytecode on the PowerNP network processor is presented in [Kind02].

## 2.3 Sample applications for Active Networks

Several proposed applications of active networks proposed as "show cases" have become so popular in the active networks community that they're now often reused as use cases for people designing new active platforms. This section presents some of those applications, highlighting the possible benefit of active networks in those cases.

An interesting investigation found in [Sivakumar00] is to classify the services according to the requirements they have:

**routing** the service allows the modification of the path packets take, either to provide better quality of service, multipath routing for resilience, or mobility support.

**differentiation** the service provides differential packet processing by selecting scheduling, dropping priority, etc. to be applied.

**data manipulation** services like transcoding, compression, decompression, encryption, decryption operate directly on the application data flow, modifying even the payload (and thus potentially the number of packets and their size) of the flow.

**data forwarding** services may alter the end-to-end communications by caching, snooping and retransmitting data, but they do not "produce" new data themselves.

### 2.3.1 Active Caches

In several client-server applications, including the Web, user-perceived performance benefits of the presence of *caches* within the network, installed on intermediate systems (usually *proxies*). Since only a small fraction of available content (i.e. the *popular* content) accounts for most of the traffic, keeping a copy of popular items in a cache located

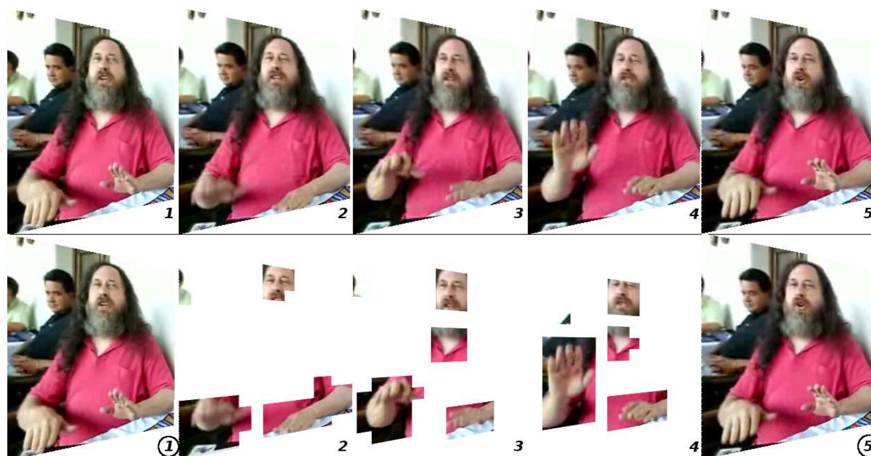



**Figure 2.1**: *A uncompressed video flow (top) and the corresponding keyframes (1,5) and deltaframes(2-4) flow*

nearby clients usually save both server bandwidth and request latency. [Arlitt95] estimates that only 0.3 to 2.1 percent of server-offered content is requested, and that it is frequent that only 10% of the content accounts for up to 95% of all requests. Several studies [Lopez95, Bowman95, Gwertzman95] have exhibited the potential advantage of hierarchical distributed caches where a local cache miss can be resolved at sibling or parent caches. The Internet Caching Protocol [Wessels97] is explicitly designed for such cases.

Beyond Web content caching [Lefèvre03], the RESAM Laboratory of ENS-Lyon combines *Internet Backplane Protocol* ([Plank01]) with their Tamanoir architecture to provide a large caching facility with a world-available API, used for instance in the implementation of multicast *repair servers* or as a repository for large e-mail attachments ([Bassi02]). Using active networking, it is also possible to get rid of the management hassle inherent to hierarchical caches. Rather than having a few proxy caches with large storage capabilities, [Bhattach.98] suggests that all the nodes should be involved and provide room for a few objects. Those caching routers use a self-organising policy to know what router should cache what object. The "lookaround" policy suggests to reserve a small portion of the storage capacity to keep *pointers* to objects stored in neighbouring nodes, a scheme that we find back in peer-to-peer distributed storage research such as the OceanStore project [Kubiatowicz00].

## 2.3.2 Multimedia Flow Transcoder

### MPEG Encoding

A video MPEG flow consists of *frames*, each encoding the image to be viewed at a given time to render the movie. Some of these frames (I-frame) are independent of any other frame and are encoded the same way as a still JPEG picture. Such "Intra-frames", also called "keyframes" usually provide highly redundant information over time, and thus mainly serve as reference to "delta" frames such as P-frames (predictive) or B-frames (bidirectional). P-frames encode only the blocks that differ between previous I-frame and

the frame to be rendered, making compression more efficient e.g. when items are animated over a still background, while B-frames may reference to both previous and next I-frames.

The price to pay for that improved compression is that a P- or B-frame cannot be rendered properly without its corresponding I-frame(s).

#### Transcoding

A problem that arises when one tries to send a multimedia flow to a collection of receivers is that all the receivers might not have the same capabilities[Tschudin98]: some may lack the CPU power to decode a more complex encoding, others may not have the required bandwidth to support the whole flow. Others, again, might be hardcoded to support some encoding and know nothing about the newer technologies the sender is streaming.

The idea of active video/audio transcoders [Amir98] was to detect those situations and to install transparent repeaters in the network that would help the source by providing streams with lower qualities (and lower bandwidth) or transcode the whole stream to a newer encoding (such as MPEG to H.323 conversion and back). While this is a good example of CPU-intensive feature that could happen in the core network, it is quite arguable (and has been often criticised) whether it's a good thing to have it *within* the network rather than at the source:

1. unless the stream originates from a regular user with a low-bandwidth line, offering both streams at the source is a quite valid option,
2. recently introduced MPEG2000 encoding (and all previous wavelets encoding prototypes) allow to scale down the quality by just dropping "extra frames", making decoding-and-recoding obsolete.

In many scenarios, active *transcoding* could thus be efficiently replaced by active *dropping*, making sure that, if not enough resource is available to carry the whole stream, then at least the most interesting parts of the stream are indeed received. Early works with MPEG [Bhattachrjee96] have shown that dropping entire group of pictures or dropping B-frames when I-frames's fate is uncertain can lead to significant improvement on the received signal. With layered video streaming in general, and MPEG2000 specifically, it gets even easier to keep useful information with a reduced bitrate since the encoding is organised in such a way that it's sufficient to drop packets that belong to one layer to get smooth degradation of stream quality.

Yet, in some specific applications [Sacks05], video transcoding remains an interesting option, especially when it appears that channel properties have evolved and that more forward-error-code is now required to have proper transmission.

### 2.3.3 MergeCast and video conferencing

Besides unicast and multicast, active networks can potentially support more paradigms for exchanging packets. *Anycast* (sending a packet towards a group, guaranteeing that at least one member of the group will get the information) is one of them. Another approach

that has received substantial attention from the active networks research community is the *opposite* function of multicast, where a large amount of senders want to report information towards a single receiver. This may happen in conferencing environments (where everyone should receive an audio stream that is obtained by combining the individual audio flows of each member), but it is also required as soon as feedback is wanted for a multicast transmission.

The advantage of multicast is that it hides the amount of receivers from the sender, allowing to send a single packet to an arbitrarily large community without modifying end-system code to make it scale. As soon as one wishes to know whether data have been received properly by the community, it is no longer possible to hide the amount of receivers from the emitter as he will be exposed to the individual “acknowledge” message of every receiver, a problem know as “Ack implosion” in the literature. In essence, the receiver do not need to know *which* end-system didn’t received the message, not even *how many* of them didn’t receive it properly (though this information can be used to set up state in the network for the retransmission, as shown in [Calvert01]): we only need to know that a retransmission is needed.

The idea of a concast/mergecast protocol would be that all the individual information from the senders can somehow be *combined* into a single message that the receiver will use. Another common example would be the gathering of temperature monitoring information from a large amount of sensors: each sensor transmits its actual temperature and the receiver wants to retrieve things like the average, minimum and maximum values among all the sensors. The *merge function* in Concast ([Calvert01b]) is defined by 4 methods through which both the multicast acknowledge aggregation, conferencing stream aggregation, sensor averaging can be expressed:

**getTag** maps each packet to class of equivalence identifier

**merge** that actually combines the values contained in the packet with those already aggregated under the router-stored state for that class of equivalence.

**done** which is a predicate telling whether the whole state has been computed or if more packets are still needed

**buildMsg** which packs combined state into a new packet when the aggregation of values is done.

The Application of ESP in video conference support based on a concast-like services are detailed in [Bond02]. Similar issues, and other deriving from the same generic application, are further developed in [Yamamoto03].

### 2.3.4 Active Monitoring and Management

Network management remains a field where active networks are highly appealing, and recent activities around *autonomic networks* are somehow a by-product of active network research. Collecting information from a large pool of machines, identifying recurring events or coordinating actions is indeed still impractical with standardised tools such as SNMP. Active management entities could receive complete programs that would watch a specific combination of events (where traditional management systems only allow to

install triggers for a single event in best cases), and exchange information with peer agents in other monitored systems or take immediate response as chosen by the management station.

Another advantage of active management is that it allows the system to react even when the management station is offline. A important number of recent active applications proposals are targeted at Distributed Denial of Service or worm propagation fight-back. The ability of running downloaded code (from secure sources) that tune packet filters according to a new attack signature or to exchange and collect information from thousands of sites and correlate them to identify the attack pattern stems for programmable equipments very similar to those proposed by active networks, even if here the end-user is not invited to make use of active code.

Some other platforms such as [Schwartz98] or [Moore02b] also attempt to propose active network solution for the "management agents" paradigm, coupling a safe language with an interface to node's Management Information Base (MIB). Mobile Agents platforms are indeed more interesting if one can be sure that a misbehaving agent (due to a programming error, for instance), will not be able to remain indefinitely in the network once unleashed. Resource-bound design of active network can help here design agents that can only live for a few dozen of hops and will then have to return to the *network operations centre* where the information they gathered is analysed and another replacement agent might be released. With that paradigm, however, we lose a bit of flexibility for safety, since it means agents cannot "settle in" at a specific node to keep monitoring it.

## 2.4 Open Problems and Future Use

While all active platforms aim at running code on routers for the purpose of improving application performance, it is interesting to note that not all applications of active network require the same level of participation from the network. Aggregating multimedia flows, for instance, requires each packet to be processed but it doesn't necessarily require that they're processed on each router. It doesn't even imply that they're processed on routers *at all*. In many such applications, what's interesting is not to run code *on the router*, but rather to run code *within the network* at some crossing point for some flows. In the Tamanoir project [Gelas02], for instance, one can to take advantage of those cases and perform the required processing on a cluster co-located with the router and later re-inject packets into the network.

Technologies for running code safely on remote locations (some of them showing close similarities with solutions proposed by the active network community) are now gaining maturity and are widely used in e.g. grid computing ([Allan01]). However, without support from the network, it is still pretty hard to know *where* the service should be deployed. Remote code execution frameworks still often assume that the code initiator knows that already, or just let the scheduler decide based on computing resource availability.

A similar issue is encountered in peer-to-peer networking where it's usually necessary to know the location of a peer in advance to be able to join the community.

### Safety in Active Networks

It was fairly clear from the start of active network research that the environment offering execution of active code would have to be *safe* – that is, it would prevent outages caused by malicious or incorrect code [Moore02]. Avoiding router crash, or interference with other traffic immediately comes to mind, as well as avoiding active packets to modify critical resources like the general IP table.

Another desired property is that the active network framework shouldn't make denial-of-services easier to build than they already are. If left unconstraint, an active network technology has the potential to create as many clones of a packet as the user want (possibly replicating them at a large amount of remote nodes) and make them all go to the same victim destination. With a low cost for the attacker, the victim will then be overloaded with junk traffic, preventing it from receiving regular requests. We invite the reader to refer to [Bossardt05] for details.

To guarantee safety, active platforms usually use sandbox interpreters or type-checking compilers.

### High-Performance Active Networks

Our overview of active networking research wouldn't be complete without mentioning projects aiming at high-performance processing of packets with a dedicated forwarding code, like PAN (Practical Active Networking, [Nygren99]), ANN [Decasper99] or CLARA [Ott00]. Using C rather than interpreted languages or clusters of PCs as routers, these projects are interesting building blocks for a network operator that wishes to setup a low-cost packet processing infrastructure and keep the control on what's being executed.

### Long Live Active Networks ?

While interest for active networking *per se* has strongly decreased over the last years, we can find active and programmable network inspiration in various recent disciplines. This is e.g., the case in wireless sensor networks, where *retasking* the nodes (e.g. to adapt their behaviour to new monitoring objectives or optimize their protocol even after deployment) involve code dissemination with properties similar to those active networks have proposed [Levis03].

We can also see inspiration from active networking in the emerging autonomic networks research area [Schmid06]. Many of the aspects promoted by autonomic networks such as automated code distribution or self-configuration have already received significant research in the context of active network platforms or applications.

Moreover, the promise of additional flexibility and intelligence offered by active network research made it a playground of choice to develop prototypes of what could be autonomic networks in the future. Many of the applications proposed by active network

researchers were instances of self-optimising services, or more recently, ways to achieve self-healing of the network.

As detailed in [Xie05], programmable networks are almost mandatory to build a network resilient to flash crowd and DDoS attacks, as new patterns of attack may appear at any time. Be it for detection modules that depend on application-layer protocol, for appropriate push-back protocols (to throttle the offending traffic) or for network reorganisation (to keep cross-traffic running), attempting to pre-program a resilience solution sounds like a futile exercise. With the recent development of programmable network hardware such as network processors or the Field-Programmable Port Extender (FPX), one can now design pattern-detection engines that enforce malware removal in the access network rather than on the end-systems [Lockwood03], and with proper collaborations of local security monitors, we might even have a chance to contain appearing worms before they cause excessive damage [Hwang05].

*A robot must communicate via a series [of] beeps and bloops, as long as such action does not conflict with the First or Second Law.*
*– Mike McCain, "The Nerd Test".*

# Chapter 3

# Network Processors


## Abstract

*This chapter will introduce the technology of network processors and detail how those components can be used in router design. We will then concentrate on the specific hardware used in this thesis (the Intel IXP2400 network processor and the Radisys ENP2611 board), and describe our development environment.*


## 3.1 Routers design

### 3.1.1 Traditional Routers Design

One cannot design a good active router without first understanding how *packet switches* are built and how they've evolved to cope with multi-gigabit interfaces we have in network cores nowaday.

The first generation of packet switches (see Fig. 3.1) was built with mainstream computer having several *line cards* (e.g. network interface I/O cards) connected to the I/O bus. The interface cards typically handle LLC and MAC protocols, store received packets in *queues* (either in main memory or located on the line card) and notify the core processor of the availability of a new packet – usually using *interrupts* mechanism. The main processor will retrieve the packet from the queue, decide of the appropriate output card according to the routing table and add the packet to the corresponding queue.

While this is inefficient to many aspects, such routers can be build from cheap components, operated with free software and still offer decent performance for individuals, small communities and corporates. The Linux router on a Intel (or clone) is the best proof first-generation routers aren't dead.

However, to handle higher speed with the same technology, the first-generation model isn't sufficient. Both the CPU, the IO bus and main memory quickly become bottleneck points that throttle the router performance. By putting more "intelligence" into the line cards, it is possible to let them decide themselves the ouput port of each packet, offloading the main CPU, and providing a naturally scalable and parallel model. A more complex

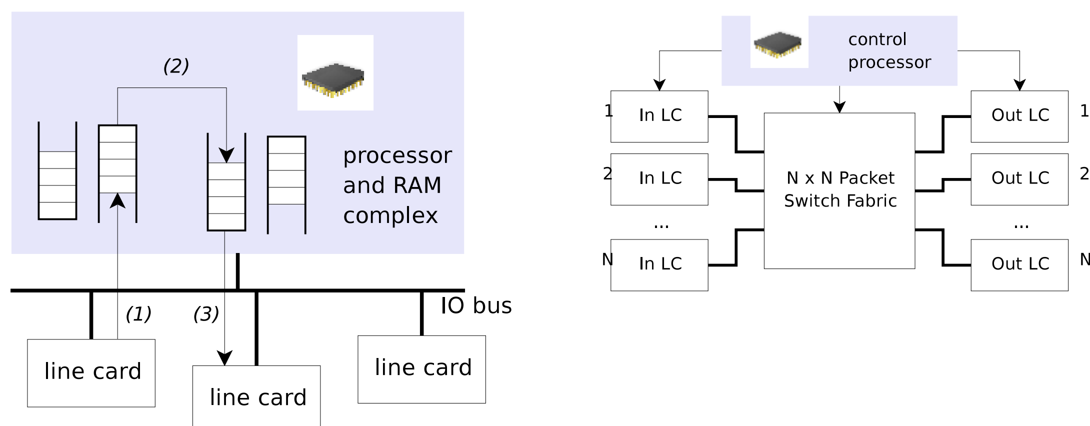



**Figure 3.1**: *(left) First Generation Router. Main CPU may be involved in fetching packet from the linecard (1), moving the packet into the output card's queue (2) and delivering packet to the output linecard (3). (right) 3rd generation packet switch.*

bus is however required to allow line cards to directly exchange packets without the help of the central CPU, which is now only responsible of *control* protocols.

The bus of those second-generation routers can however become a severe bottleneck at high speeds, since it can only handle *one* packet exchange between two line cards at a time. To allow 1Gb speed on every of 10 line card with any traffic pattern, for instance, the bus should be able to sustain 10Gb. Third-generation packet switches thus replace the shared bus by a *switch fabric*, a N-to-N connecting element that can allow direct interconnection of any pair of line cards simultaneously. There are, of course, restrictions and it is usually not possible for the switch fabric to handle a set of requests where two input cards request the same output card. In such situation, at least one of the packet will have to be blocked for another turn.

### 3.1.2 Intelligent Line Cards and Network Processors

Initially, a line card has little job to do. It is mainly a hardware trie search engine that will lookup for information associated with a given IP address, plus performing sanity checks on the packet such as CRC checking, TTL decrementation and the like. As new functionnality are added to the network, however, the line card also has to identify packets that belong to the same connection (e.g. for firewalling) or to a given quality of service class. More complex functions such as enforcing dropping preferences in the output buffers or scheduling packets from different queues also become frequent.

It is common to design a router as having a *fast path* that handles all the 'regular' packets and delegates the remaining packets to a more complex software process (i.e. the *slow path*). Fast path is usually implemented using Application Specific Integrated Circuits and therefore offers virtually no room for extensibility.

Using technologies like ASIC or FPGA, it becomes difficult to face the increasing demand for such new functionalities. Moreover, once a given set of functionnalities has been implemented on a chip, it is not trivial to modify them. *Network processors* present

an alternative design where dedicated processing units can handle packets even on the datapath.

Along with those dedicated packet processing elements (e.g. *microengine* on the Intel IXP family or *picoprocessors* in the IBM PowerNP family), the network processor chip combines a set of dedicated coprocessors for hashing, trie lookup, packet copies, etc. The whole chip is operated under the control of an embedded generic-purpose processor that implements the slowpath functions, forwarding tables maintenance, chip initialization and guarantees synchronization with network processors on other linecards.

### 3.1.3 Maximum Headroom, Please ...

As detailed in [Campbel02], it is critical for a router to be able to process even minimal-size packets *at line rate* – that is, even if the router only receive minimal size packets, it should still be able to fully utilize the output links. If a router fails to meet this requirement and, for instance, can only sustain 50Mbps when all packets have the minimal size (assuming 100Mbps fast Ethernet links), then an attacker can easily deny routing to other traffic with a traffic volume that the router should normally handle without problems.

It is important not to misinterprete this rule. If we consider $T_{min}$ as the transmission time for a minimal packet, we could be utilizing the full output capacity even if the router took more than $T_{min}$ to forward the packet. The reason is that the forwarding logic could be using *pipelining* and separate the forwarding process into independent *stages* that can happen in parallel. If all those stages can complete in less than $T_{min}$ then we can still sustain a full flow of minimum-size packets. If, in addition, a given stage $i$ receives $k$ execution units – that is, we have replicated hardware and a dispatching technique that delivers packets from previous stage $i-1$ to unit $(i,1)\dots(i,k)$ then we can theorically allow a processing time $kT_{min}$ for step $i$.

### 3.1.4 PowerNP and IXP2xxx

Among network processors two main designs have drawn more attention in active networking research [Sterbenz02], due to the genericity and the performance they are capable of: the IXP processors family from intel [IntelPRM, Johnson02] and the IBM PowerNP [Allen03], now known as the HiFn 5NP4G processor [HiFn04]. Both hardware share similarities in their basic design:

- the presence of a “controlling processor” with slower speed but capable of everything a generic processor can do,
- multiple sub-processors (pico-processors or microengine) that can manipulate every packet
- multiple levels of memory, with variable size and latencies, and mechanisms in sub-processors to hide memory latencies.
- co-processors for checksums, trie lookup, encryption/decryption, either on-die with the core controller and sub-processors or as externally available dedicated units.
- communication interface with either “media chips” (like gigabit ethernet, OC-3 through OC-48) or with a switch fabric.

The main way by which those systems differ is the intended mapping of code blocks on processing elements. The PowerNP architecture promotes the "run-to-completion" model where each packet is handled by a single hardware thread, from reception from a media card to enqueueing into switch fabric interface. All the pico-processors of the PowerNP share the same code memory which is sensibly larger than the individual *code store* of IXP's microengines. In the IXP processor, by contrast, the relatively small microengine's code store advocates for a pipelined model where each microengine is responsible of a given functionality (receiving, checking sum, looking up IP table, enqueueing) and has specific hardware resources to allow threads among a single "stage" to cooperate more efficiently. With introduction of IXP2xxx family, this even goes further as each microengine is natively chained with two other microengines through specific "neighbour registers".

Taking results of the previous section into consideration, it becomes obvious why network processors have such large amount of fairly simple processing elements. The IXP1200, for instance had 24 execution contexts distributed among 6 microengines. The IBM PowerNP NP4GS3 has 16 dual-threaded picoprocessors and the next generation of IXP products (IXP2400) has 8 pipelined microengines with 8 hardware contexts each. The goal is simple to understand: provide more *headroom* for interesting packet processing – that is, more instructions cycles available beyond the bare IPv4 forwarding, while still meeting the "full output link utilization" constraint.

Later products like IXP2800 features up to 16 multithreaded microengines, achieving 10Gbps ethernet support with a single chip!

### 3.1.5 Related Work

In [Spalink01], the authors evaluate various forwarding functions on the IXP1200 processor and show what performance we can expect from the router. The problem of dynamically creating optimized pipelines has been addressed in [Campbel02] by load-time modification of code constants to avoid extra memory lookups used by common dynamic binding features. Authors of [Kind02] detail the two 'programming models' for network processors: *run to completion* or *pipeline* and spot that hardware design of a network processor can make it unsuitable to some model. In the case of run-to-completion model, they implement a just-in-time compiler for SNAP language.

One of the active-network related research activities around network processor focus at building datapath dynamically [Ruf05], and managing the heterogeneity of resources (e.g. multiple level of memory, restricted amount of code store, usually split over a potentially high number of locations. An interesting alternative depicted in [Baron05] consists of using IXP network processor as TCP *offloader* for high-end servers. Some other applications are even more "exotic", using the microengines to accelerate database operators such as scans and joins [Gold05] or DNA processing [Bos04].

It would of course be over-selective to mention only IXP and PowerNP while the wikipedia lists about 10 vendors of "network processors". The term "network processor" actually hides a much wider variety in hardware designs, dedicated units and programmability, such as CISCO's NPE-G1, Xelerated X11, C-Port's C5 DCP and more [Kohler04].

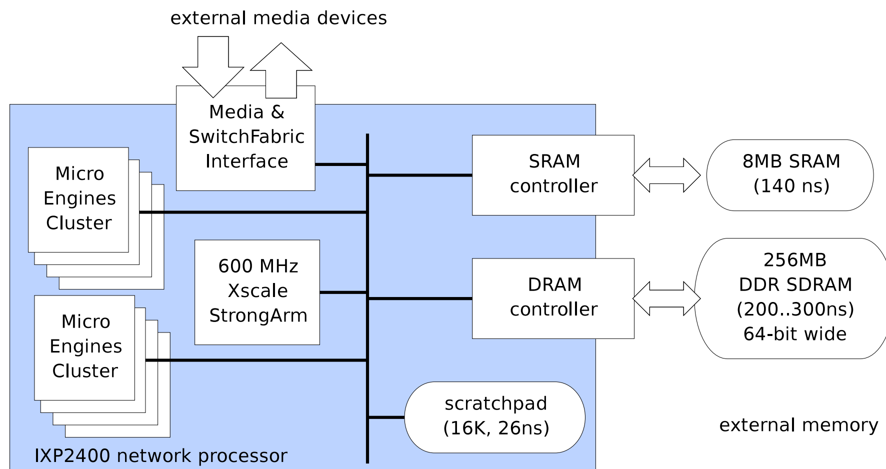



**Figure 3.2**: *IXP2400 network processor structure diagram, annotated with timings reported by [Lu05]*

Some, such as the C-Port's Digital Communication Processor, are even more monstruously powerful than the Intel and IBM products, featuring 16 "channel processors" each coupled to a programmable Gigabit Ethernet / Packet Over Sonet (POS) physical channel and a 50Gbps internal bus to interconnect all internal elements. Others such as EZchip and Lucent processors have well-defined *task-optimized processors* (TOP) organized into distinct pipeline stages. The Xelerated X11 even pushes this approach to the extreme with 24 elements of a "programmable pipeline" (PISC blocks), each allowed to execute a few instructions per packet before the packet is handed over to the next PISC.

For active networking purposes, these devices are however less pratical to deal with than IXP features. It's not rare with the Xelerated processor that packets needs to be sent over a "loopback" wire after being flagged for additional processing, for instance. Such considerations give a newer look at works such as NetVM [Baldi05] that attemps to abstract the particularities of each hardware (including generic programmable devices such as FPGAs equipped with *softcore* blocks) into a single model for the programmer.

## 3.2 Overview of IXP2400 Network Processor

The Intel IXP2400 network processor is a programmable chip capable of processing packets at rates approaching 4Gbps. The most popular products featuring an IXP2400 are the Intel IXDP2400 development board and the Radisys ENP-2611 card. The processor itself doesn't contain logic to receive and transmit packets over the Gigabit Ethernet or OC-xx physical medium, but it rather communicates with external IO controllers through a generic *media and switch fabric interface* (MSF interface). Depending on the board design, the IXP can be used as a "standalone" component (in the case of Radisys card) or as a media card for a switch-fabric based system (in the case of IXDP card).

The IXP processor also features components that facilitate its control from the PCI bus. While in most cases it is just a convenient way to reset the card or inspect its state, it can theorically be used to transfer packets (at degraded rates) between several IXP-based

cards connected in a single PC. However, if one wants to build a larger router (cards mentioned above range from 2 to 4 gigabit ports) with IXP processors, a switch fabric system and IXP2850-based blades should be preferred.

### 3.2.1 Processing Elements

At the lowest level, packets received by an IXP equipment will be handled by the *microengines*. These are 8 independent processors running at 600MHz[1] with a dedicated RISC-inspired instruction set, each having up to 8 hardware contexts (threads) and a control memory of 4K instructions. Each microengine features 256 general-purpose registers (GPR) and a local memory of 640 words that can directly feed the ALU, plus 256 "transfer registers" and 128 "next neighbor" registers that can be used to communicate with SRAM, SDRAM and chained microengines.

Each microengine features a *context arbiter* that will perform round-robin selection among the activable threads everytime the running thread releases the processing hardware (for instance, waiting for a memory operation to complete). Microengines are chained by hardware, one to another to help build processing pipelines where writes into the "next neighbor" register of one microengine are immediately available in the "next neighbor" register of the next microengine for reading, avoiding the need for external RAM to exchange context information about the packet being processed. If memory latency can be avoided, each *microword* of the microengines is processed in one cycle.

Over those microengines stands the XScale core, an ARM5TE-compatible processor clocked at 600MHz which has access to the same SRAM and SDRAM resources as the microengines as well as to some of the microengines resources (such as their control store). The XScale is capable of hosting an embedded system such as QNX, VxWorks or Linux that will provide a programming environment for the microengines, implementing control protocols such as ICMP or OSPF, filling forwarding tables used by the microengines, starting and stopping microengines and filling their code store with appropriate programs. If needed by the network application, the XScale core can also be a good place to handle "exception" packets such as fragmented packets, ARP requests and whatever might be processed on a "slow path" due to code size restrictions in the microengines. Unlike the microengines, the XScale also has code and instruction caches that hides SDRAM latencies and a Memory Mapping Unit required for Linux operations.

### 3.2.2 Storage Elements

Much like processing, storage on the IXP platform is split between different units, each having their own technological properties and being best suited to some aspects of packets handling. The DRAM[2] is the largest one, ranging from 256 to 2GB of memory that use the same technology as main memory of modern PCs, and the same drawback concerning access latency. DRAM is appropriate for storing the StrongArm's operating system and

[1] according to table 157 of [IntelHRM]

[2] IXP2xxx models actually use 100MHz DDR SDRAM. We will simply label it "DRAM" in this work, for readability.

| Storage Type | access (ns) | bus (ns) | IXP2400 cycles |
|---|---|---|---|
| External DRAM | 226 (+12) | 59 (+4) | 180 |
| External SRAM | 81 (+5) | 51 (+4) | 84 |
| On-chip scratch | 21 (+3) | 37 (+3) | 38 |
| Local Store | 11 (+4) | 0 (+4) | 11 |

**Table 3.1**: *IXP2800 memory model used in [Labrecque06]; access external DRAM takes 12+4+226+59=301 ns, 285 of which are "pipelined"*

programs and typically used for storing packets that are being forwarded and other bulk data.

Basing on informations provided in programmer's reference manual [IntelPRM], we may assume that DRAM access may take around 150 microengine cycles to complete. The actual latency of DRAM may depend on the memory actually installed and of the memory access pattern (see [Kozierok01w]), still results presented in [Labrecque06] and in [Lu05] confirms that order of magnitude, despite the fact they only consider IXP1200 and IXP2800[3]. The memory-related parameters of that thesis are mentioned in table 3.1, and illustrate the potential benefit of *burst* transfers where the "pipelined" portion will be incurred only once for several accesses.

The SRAM area typically spans a few megabytes, but it can be accessed twice faster (with a latency around 85 cycles). Another important factor is that while DRAM only allows data stores and loads, SRAM is capable of atomic operations (addition, bit setting and clearing or "test-and-set") and allows individual words to be locked thanks to a Content-Addressable Memory (CAM) companion unit. The SRAM will typically contain structures that needs to be manipulated by several threads in parallel.

When access speed really becomes a critical factor, it is also possible to use the on-chip *scratchpad* SRAM. The scratchpad only provides a few (16) kilobytes of memory, but can be accessed in about 16 processor cycles and has the same kind of atomic operations as the off-chip SRAM. The scratchpad is typically used to store datapath variables, counters, etc. that need global access but cannot suffer the performance penalty of an external memory access.

The other storage facilities (local memory and registers) are duplicated in every microengine but cannot be accessed from outside. The *local memory* provides 640 words of storage with a 3 instructions latency, but with mechanisms such as access pipelining and post-incrementation that can help manipulate that memory without delay cycles. The register banks are large enough to accomodate not only procedure-local variables but also application-global variables. While registers are usually bound to a specific hardware thread, the microengine assembler provides a facility to allocate and use "global" registers that will be visible by all the threads of a single microengine.

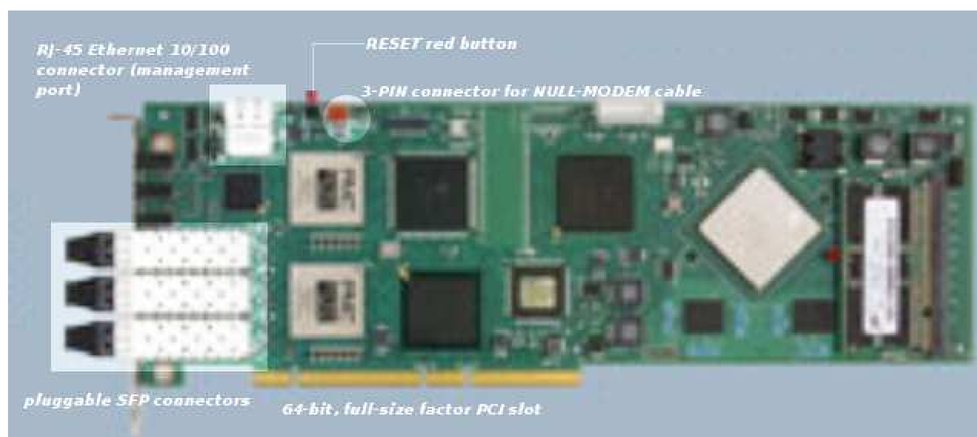



**Figure 3.3**: *The Radisys ENP2611 card, with 3 optical ports*

### 3.2.3 Developing on the Radisys ENP2611 card

The Radisys ENP2611 card is a PCI card hosting a complete system with a IXP2400, dedicated DRAM and SRAM memory and up to three gigabit ethernet ports interfaced with the MSF through a FPGA bridging chip. Despite the ENP-2611 card has 3 gigabit ethernet connectors, these are the last things we'll be using for development. Instead, the management port – a traditionnal RJ-45 connector for 100BaseT Ethernet will carry most of the traffic between the XScale embedded operating system (in our case, a patched version of Linux 2.6.15) and the host machine.

A "null modem" cable (shipped with the card), connects the 3-pin UART slot to the serial port of the host PC. With a program like minicom installed – or any equivalent program you're happy with – we can connect to the most elementary development interface that the card features and control the booting sequence (the card features a pre-installed version of RedBoot manager on flash) or run diagnostic tests. Once the system is booted, the serial interface is also the only console we have until a *telnet* can be run over the management port.

It should be noted that what the ENP2611 card really miss is a user-friendly printed manual with an all-in-one development kit on CD. Most of this section is the result of several weeks of crawling through kernel sources, downloading of restricted Software Development Kits at Intel and MontaVista corporations and discussions with people on the ixp2xxx mailing list. A large amount of our findings have been made public step by step on the ENP FAQ, now promoted as the accompanying wiki for the IXP2xxx support project on SourceForge[Martin06]. Without the work performed by Lennert Buytenhek to integrate ENP2611 support into linux 2.6.16 kernel, the story would have pretty much ended here.

[3] estimations presented here assume IXP2400 is just a IXP2800 clocked at 600MHz instead of 1GHz

### Loading Something on the Card

Loading a linux kernel on the ENP is no different from loading a Unix kernel on a diskless X workstation, except that we're now in the 21th century ...

- DHCP configuration: the host (or any other machine on the lab LAN) should be running a DHCP daemon for giving the card its own IP address. With proper configuration, RedBoot should report the management port's IP address and the server it has used just after memory scrubbing. We can notice that it reports MAC address as well (which can be helpful in DHCP configuration). We can check this step was successful by just "pinging" the card.
- TFTP configuration: this is the protocol used when loading linux kernels on the card. Here too, it's more convenient to have it on the host. It's as simple as to put any file in /tftpboot and issue the `load -v -r -b 0x02000000 -m tftp -h xxx.yyy.zzz.uuu /tftpboot/any-file` command to check if it can work properly.
- NFS configuration: only the kernel is transferred to the bootloader. Once it will be launched, the Linux kernel will need its root filesystem with `/dev`, `/proc` and the appropriate files in `/bin` and `/sbin`. MontaVista comes with an almost complete base filesystem, but an appropriate debian installation will work as well.

### Drivers for the ENP2611 card

Together with the core support of the board, the Linux kernel provides a network driver (`enp2611_mod.ko`, developed by Lennert) for the gigabit Ethernet ports. Only two of the available eight microengines are required for this very simple functionality. Microengine 0 will be in charge of receiving packets from GigE chips through the Media Switch interface, and microengine 1 will be transmitting packet of the kernel to the Ethernet chips. For both, packets are stored in DRAM (that is, where Linux kernel stands too) and the hardware-supported ring buffer in *scratchpad* memory connects the microengines with the linux kernel.

However, Lennert's driver is just that: a driver. While it has microcode loading facilities for its own purposes, it will not help load our microcode into the microengines. To use the full power of intel's IXA SDK, we should instead use the device drivers provided by Intel. These are made of two parts: a user-side library and a kernel-side driver, glued together by the /dev/medrv0 device. By porting the `halMev2.ko` kernel module to the 2.6 kernel architecture, we now have the ability to use the microcode loader for UOF files (the native binary format used by intel tools) from user programs.

In addition to medrv0 device, application written for the IXA SDK typically use `/dev/spi3br` and `/dev/pm338xx` devices to program the gigabit ethernet hardware of the ENP card. Examples provided by the IXA SDK as well as demonstration provided with the ENP SDK are oriented around the idea of a system application that takes the control of all your board's resources (microengine, GiGE controllers, SPI3 bridge) for a given purpose. The code for this application will e.g. issue proper IOCTL calls to enable reception/transmission by the gigabit ethernet chips, or prepare SRAM/SDRAM values

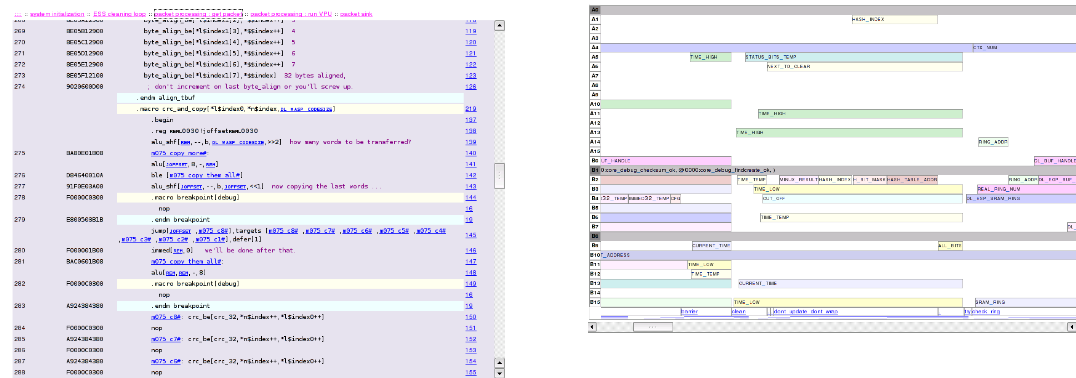


**Figure 3.4**: *output of the* `beautify.pl` *script: annotated microstore content (left) and register allocation map (right)*

according to the expectations of the code running on microengines. The “system application” takes care of loading the UOF file, starting/stopping the microengines and it can as well report/configure various parameters of the running application.

Like in the case of HalMev2 driver, ENP drivers are specific to linux 2.4, but hopefully, they were way smaller than the microengines driver and easier to port to 2.6 kernel.

### Improving Our Tools

Writing code for the microengines is one thing, having the code running on the IXP is another. We need a *control application* running on the XScale core to transfer the binary `uof` file in the control store of the different microengines, using the UCLO (microcode loader) and the HalMEv2 libraries provided in the IXA SDK.

Based on L. Buytenhek tutorial tools and the control application of the IPv4 forwarder (part of the ENP2611 development kit) and of the ESP filter, we grew our own “swiss army knife” for WASP. More than a mere program loader, our control application for WASP is capable of reporting and analysing most of the dynamic aspects of WASP and ESP microblocks behaviours, allowing us to inspect virtually any storage item on the microengines, either on demand or repeatedly in a pre-programmed statistic-gathering loop.

While the XScale core can natively only access external memory resources such as scratchpad, SRAM or DRAM, we used the HAL library to suspend the microengine we want to inspect, replace a portion of its microcode store by a sampling program that reads the requested data item that will be then read through a status register.

Based on our experience in operating system kernel development, we also extended the control application with generic debugging features such as registers inspection, microengine state dump and breakpoint management. This “debugging shell” in the control application was preferably used in conjunction of a post-processed HTML view of the `.list` files produced by the microassembler, which mixes the actual generated machine code and the (pre-processed) source code.

These human-readable views of the microengine code store are generated by a home-brew perl script (`beautify.pl`, see Fig. 3.4) that process `.list` files and produces HTML files of indented, annotated pre-processed code as the microengine see it. Its most interesting uses were:

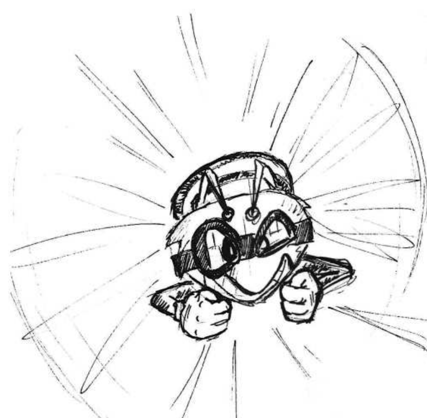
1. get a clear view of how code has been pre-processed, and what code (sometimes out of many hardware-specific alternatives) has actually be generated even before giving it a run.
2. the ability to quickly locate all instructions of a certain type (e.g. all references to a control/status registers, or all the manipulations of scratch rings), even when nested deep into macros libraries, and immediately see enclosing macros and source lines involved
3. when given a breakpoint address by the debugger, quickly retrieve the corresponding code in the `.list` file, see the corresponding source file(s) and obtain registers references by hovering the source. One can then ask the debugger for each register's content.

In order to get a global view of registers allocation and lifetime, and getting ourselves rid of "too many GPR" assembler errors, we extended the use of *beautify.pl* to produce a *map* of the register usage, allowing the developer to get an immediate overview of which portions of its code use more registers, which register needs to be reallocated to different physical registers, etc.

At several points, it allowed us to pinpoint registers that were kept for abnormally long code regions compared to the locations they were supposed to be useful.

# Chapter 4

# The WASP Platform



*WASP is faster than ANTS*

## Abstract

*This chapter describes the components of our architecture: WASP packets, the ephemeral store and the virtual processor, illustrating their role through simple "use cases" of active networks.*

*We will motivate and detail the restrictions we imposed on the architecture to keep it network and router-friendly. We also give the results of performance comparison against the ESP active router for the x86 "reference" implementation.*

Before going into the details of the WASP router design, it can be useful to detail what kind of "applications" WASP targets. First of all, WASP is not a replacement for IP, and is not concerned with *routing* proper. It is rather a pre or post-processing stage that can be placed between the network and the router core. Second, WASP is designed to handle *small* tasks. Both code and data portions of a packet have small size compared to most active networking architectures, and chances are that all implementations will also restrict the number of ephemeral store accesses a single packet can request.

To many respects, WASP adheres to the "philosophy" of Ephemeral State Processing (ESP) and of IP itself by being *anonymous*, available to everyone and *too cheap to measure*. Processing a WASP packet should have the same order of magnitude as processing an IP packet[1] so that a router can still operate at full rate regardless of how many percent of its traffic is actually WASP packets. In addition, a WASP router doesn't care of the identity of end-users when it processes packets: anyone is allowed to manipulate any entry in the state store[2] – much like an IP router doesn't care about who has sent or will receive a packet when it forwards it. Those two properties (anonymity and lightweight) imply that there's no need for the router to perform authentication of any kind before it processes a WASP packet and that there's no need for individual accounting.

A first use case for WASP is the identification of properties on a specific path: as packets traverse the network, they can gather router IDs, measure how long they remain on a given node, how long has elapsed since the last packet of the same flow has been

[1]This is clearly not the case for ANTS capsules, for instance

[2]with a small exception for "super packets", as we shall see later in section 4.3.3

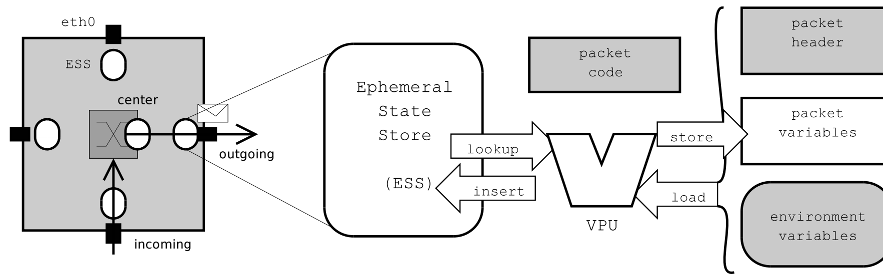



**Figure 4.1**: *A WASP router and WASP execution environment. Gray items mean the VPU has read-only access to the resource*

processed by the router, etc. Yet, it will be up to the end-system application to process and react to such extra feedback. Another field of application is the coordination of end-system that contribute to a group without requiring a global knowledge of that group. ESP already takes advantage of this for multicast retransmission, but WASP could extend it to peer-to-peer systems, flash crowd controlling, etc. A small set of such sample applications are presented in the next chapter.

Last, but not least, WASP does not place any assumptions on the number of active nodes in the network to operate properly, and it meets the triple objective of *world-friendliness*:

**User-Friendly:** WASP should offer significant programmability while allowing the applications running on end-systems to know to what extent they can trust what they get from the active network.

**Network-Friendly:** WASP should not become a nuisance to networks (and operators). It should require no configuration from the operator and the network load it produces should be predictable.

**Router-Friendly:** Active packets should not be able to harm a router nor degrade its performance.

## 4.1 Model of a WASP Router

The overall model of the WASP router is derived from the Ephemeral State Processing [Calvert02]. The core of the router is the *forwarding logic*, which implements the basic IP service consisting of looking up the IP table, switching the packet to the appropriate output queue(s) and updating the IP header.

Between this core and the network interfaces are the Ephemeral State Stores (ESS) where WASP processing is applied, as shown on Fig. 4.1. In addition to ephemeral stores associated with network interfaces, there is a single "center" location which can be used by more "control-oriented" packets. An active packet processed by the router typically crosses three logical locations:

1. the *incoming ESS*, bound to the network interface that received the packet,
2. the *center ESS*, which all packets will traverse, and

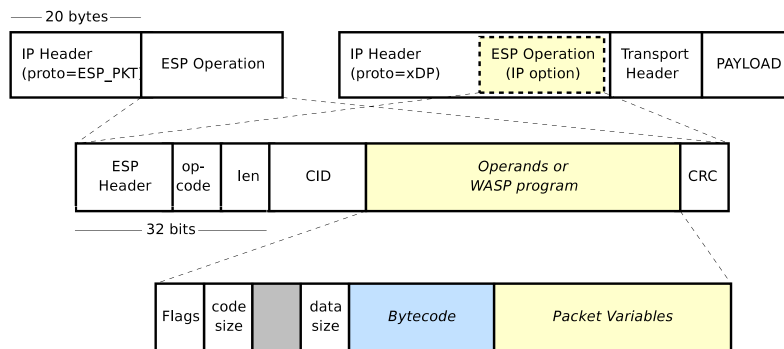



**Figure 4.2**: *WASP packet format, both standalone (left) and piggybacked (right)*

3. the *outgoing ESS*, bound to the network interface that will transmit the packet.

Each packet can request execution on any combination of these three location but it can in no way alter the state in other stores. When forwarded from the “southern” interface to the “eastern” interface, a packet may not, for instance, drop or inspect state in “northern” or “western” stores.

A special field of the ESP/WASP header[3] instructs the router on which locations the packet requests evaluation. This flexibility allows reduction of processing overhead to the minimum while enabling interactions between packets coming *from* a host and packets *returning to* that host (by e.g. making forward packets using the `ESP_OUTPUT` and backward packets using the `ESP_INPUT`) as well as enabling interactions between packets from multiple sources that go to the same destination.

Keeping most of the processing at the *interface card* level is a key element for design scalability: on a router with higher throughput, the processing load can be distributed among the available processors. As a result, the *center* location should be avoided every time possible and the protocol designer should remain aware that WASP processing at the router center might involve the delegation of the packet to a generic-purpose processor and that the router implementers might enforce rate-limitation of active packets on that location.

The specificity of WASP is that each Ephemeral State Store is coupled with a *Virtual Processing Unit* (VPU) that handles interactions between state store and packet storage according to a microcode carried by the packet itself.

### 4.1.1 WASP Packets

WASP uses *in-band* code transmission: each packet contains its own code and the data on which it can operate. The evaluation of packet code (up to 256 bytecoded micro-instructions called *microbytes*) terminates when a *packet control microbyte* is encountered, which tells the router what to do with the packet. In addition to “forward” and “drop” semantics, WASP allows the packet to be sent back to the source at any router,

[3]the `loc` field, which has one bit for `ESP_INPUT`, `ESP_CENTER` and `ESP_OUTPUT` respectively

which can be useful when a quick feedback of a discovered state is required (e.g. filtering more packets in the router ahead of a congestion point).

During interpretation, the data part of the WASP packet is available as a 128 byte region of random-access memory and only the IP header is readable. Other parts (WASP bytecode, other IP options, transport payload) are unavailable to WASP code.

As depicted on Fig. 4.2 WASP packets may exist in two flavours, both being inherited from the ESP framework. The most simple is the *standalone wasp*, where the WASP packet is simply encapsulated in a IP header (just like an ICMP packet is encapsulated in IP). This will be the preferred case when WASP is used to implement a new protocol, or when WASP packets act as autonomous monitors of the network's state.

On the other side, WASP packets can also be *piggybacked* with other protocols, in which case they are stored in an "option header" of IP, which may be useful when WASP is used to regulate the flow of an existing transport protocol.

### 4.1.2 The WASP Node

Each WASP node has a certain number of *Ephemeral State Stores* that will associate 64-bit keys (also called a *tag*) with small, fixed-size data. Each ESS is associated with a *Virtual Processing Unit* (VPU) that processes the WASP packets. Since all exchanges between packets occur in the ESS, there is no need to store VPU state between the evaluation of two packets. This greatly simplifies the synchronisation problems, even on a multi-processor system, since it means we can bind VPU data to one real CPU rather than to an ESS.

Before a VPU starts evaluating a packet, it retrieves the node and interface *environment variables* and exports them as banks of read-only memory to WASP code. These variables will typically include the node IP address, netmask, local time, etc. plus statistics about the current interface (recent packet transmission statistics, queues status, etc.), which can be useful for applications sensible to network conditions.

Considering the restricted resources of network processors, we tried to keep the design of WASP's virtual processor as simple as possible, which makes it look more like an embedded microcontroller than a modern microprocessor, from an architectural point of view, but the resulting interpreter for the base instruction set is only 4KB long – smaller than the operation handlers of the native ESP filter.

### 4.1.3 World-Friendly Platform

#### WASP is User-Friendly

The additional flexibility provided by the presence of the VPU allows a larger scope of problems to be addressed and it offers a more natural programming language than ESP's high-level instructions. Yet, a WASP router does not alter packet flows of users that decide not to use WASP and do not allow interaction between separate WASP flows unless end-systems explicitly use the same tags. This goal is very close to the *flexibility* goal mentioned by [Moore01] and the *usability* goal in [Bond02]: offer a easy-to-use interface to the active network that abstracts the inherent complexity of network programming.

To achieve that abstract objective, anonymity, automatic management or best-effort approaches are the keys to "*Keep It Small and Scalable*"[4].

### WASP is Router-Friendly

Even if WASP is based on active packets, it is much more restricted than general-purpose *capsules*, and user's bytecode cannot waste router's resources. For instance, WASP bytecode language prohibits backward jumps and all instructions have predictable execution time, which makes packet processing time trivial to control (as shown in SNAP [Moore02]), and typically linear with the packet's size. Router's memory is of course taken into account as well, mainly thanks to the ephemeral state store, which automatically reclaims entries a fixed period after their creation.

If such restrictions are impractical for general-purpose services, they perfectly fit the lightweight control tasks that WASP will have to perform. While alternative solutions exist, such as associating a counter to any backward jump in the code, we believe the benefit we could get in WASP is not worth the additional management required.

### WASP is Network-Friendly

Even if we take care of router resources, an ill-intentioned active packet could easily create an avalanche of clones to overload its destination. Among 'first generation' active platforms, PLAN [Hicks98] was the only project that addressed this issue by making sure all children's resource counters receive a portion of the parent's counter. Unfortunately, picking a "good" initial resource bound remains a complex issue.

In the case of the *WASP* platform, packets do not have the ability to create child packets unless they are targeted at a multicast address, but they can *drop* themselves or *return* to their source. If we focus on applications like service discovery or server load balancing, we have no real need for more: we can store the new destination in the active packet, *return* the packet towards its source and let the source issue a new connection attempt to the real destination.

## 4.2 From ESP Operations to Wasp VPU

In the original Ephemeral State Processing router, each packet carries opcode and operands for a *single* operation. This operation, however, is usually rather complex and may involve several ESS references (for reading or writing), access to packet-stored "immediate" values and even perform arithmetic or logic operations. If we transpose these packet-triggered operations on the ESS to the design of microprocessors, the ESP operations looks like instructions of a Complex Instruction Set Computer (CISC) machine. As an example, the two operations `count` and `collect` involved on Fig. 4.3 could be described as follow:

[4]http://en.wikipedia.org/wiki/KISS_Principle

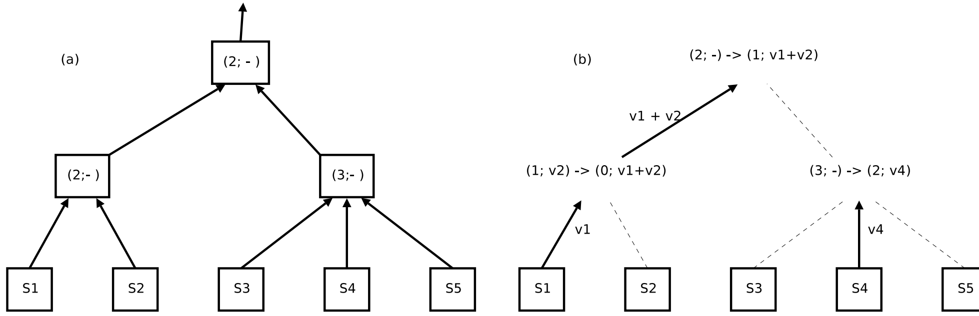



**Figure 4.3**: *(a) First round labels each node with its number of children with* `COUNT` *packets. (b)* `COLLECT` *packets then compute the local aggregated value and forward it upwards when no more values are expected on the local node*

**`COUNT`**$(value_tag, threshold_imm)$**:** retrieve $val$ from the ESS using $value_tag$ and set $val$ to zero if undefined. Increment $val$ and store it back in the ESS using $value_tag$. Packet is forwarded only if $val$ is below or equal to the $threshold_imm$ parameter carried by the packet.

**`COLLECT`**$(value_tag, counter_tag, value_imm, operator_imm)$**:** retrieve $val$ from the ESS using $value_tag$, then apply merging operator on $val$ and $value_imm$ and store the result back in the ESS under $value_tag$. If $value_tag$ was undefined, the value carried by the packet $value_imm$ is stored without applying the operator. Available operators include addition, minimum and maximum. After this, $counter_tag$ is used to retrieve $ctr$ from ESS is decremented and stored back. If the decremented value of $ctr$ is zero, the packet is forwarded, otherwise, it is dropped.

These are only two "simple" operations out of the 5 defined in [Calvert05w]. Those two operations are for instance combined in a two-rounds process, illustrated on 4.3 that could for instance return the maximum value detected by a large set of sensors by having each sensor sending `COUNT`$(T, 1)$ first and then `COLLECT`$(V, T, x, MAX)$ afterwards. Note that, as detailed in [Calvert02], this protocol is not protected against packet losses, but it is protected against machines that are unresponding for a whole round (these will not send "count" so they do not prevent "collect" packets to go forward). Two other operations `RCHILD` and `RCOLLECT` are provided when achieving reliable collection of data is required, requiring 8 and 13 32-bit words of argument each.

`COUNT`$(T, 1)$ is an interesting special case as, if generated from a collection of machines to the same destination address, it will label each router with the number of "children" it has in a distribution tree. Each packet will travel the tree towards the root (its destination) as long as no other packet with the same tag has passed through there and stops on the first router that is a "branching point" with packets from other machines. Note that children in this context will be machines *or* routers: in order to know the number of end-systems, what is required is `COLLECT`$(T, V, 1, ADD)$.

To achieve that flexibility, the `COLLECT` operation relies on the `operator_imm` field that tells how to combine two values, and branches to one of the pre-defined handlers for that operator. A similar behaviour is observed in the `COMPARE` instruction which has a field selecting among *equal?*, *less-than?*, *less-or-equal?*, *greater-than?* and *greater-or-*

*equal?* predicates. The programming model offered by those instructions remains 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 of how to use `COUNT`, `COMPARE` and `COLLECT` to achieve that goal. In many other situation, we just experience the frustration that the operations defined are too specialised to be of any use.

While ESP itself mainly focuses on multicast, reliable multicast and implementation of *concast*-like services [Calvert01b], the *ephemeral state store* itself could be applied in many other situations such as setting up and maintaining peer-to-peer overlays, terminal mobility support, or application-managed packet dropping. The restricted instruction set is of course not the only limiting factor and thus our proposed architecture also extends the ESP router with a *rerouting* facility (see Sec. 7.1.2) or with basic node and interfaces status reporting (see Sec. 4.4.5).

### 4.2.1 A Virtual Processing Unit for Ephemeral State

Most ESP operations are non-trivial to describe, and their (sometimes lengthy) pseudo-code often involves temporary variables, primitive operations on the ephemeral store, packet-stored variables and immediates as well as control structures (e.g. *if-then-else* blocks). WASP started as a challenge of designing a small virtual machine that could express those operations with something looking more like *microcode* for a virtual processor. The ephemeral state and the packet-stored variables would then appear like memory units in that virtual machine just like we have RAM or content-addressable memory in a real machine. Going further with that concept, decisions of packet forwarding or discarding (and, as we introduced them later, *return* or *reroute*) would be control opcodes just like a regular computer has opcodes for suspending operations, return from procedure calls, or invoke the operating system.

The most complex of ESP operations features not less than 7 conditional branches but it still didn't require a loop-like flow control mechanism. Drawing inspiration from *SNAP* ([Moore02]), which also offers programmability through a microcode-like instruction set, we decided to enforce the absence of loop as a key design element of WASP's virtual processor. That certainly reduces the set of applications that can be implemented with WASP, but it also means that we keep execution time linear with the size of the packet size, which is a serious advantage to achieve "too cheap to measure" WASP packets.

The fact that the ALU and registers of the WASP VPU are 64-bit wide is of course motivated by the fact operands of ESP packets, tags and values in the ESS are themselves 64-bit wide. We expect that the additional cost for implementing the VPU on a 32-bit machine should be affordable compared to the cost required to decode and execute one extra bytecode. Another nice property of a 64-bit ALU is that it allows us to manipulate keys for the ESS easily and maybe combine packet-carried values with ESS values to create new keys on the fly.

### Instruction Size

We chose to stick to a single instruction size of one byte as much as possible. In the context of WASP VPU, compactness of the encoding is certainly a desired property as the code will be attached to data packets. In a few exceptions, we allow instructions to have one extra word of immediate operands. While instruction sets such as x86 may achieve compact programs, it comes at the cost of a much more complex decoding logic[5] that we cannot afford if we want to meet code storage and execution speed constraints.

Much like a RISC machine, the WASP VPU only offers a limited set of opcodes and it will often require that an operation such as “add 4 to memory location X” is broken down into several steps such as loading registers with immediate value and memory content, perform the addition (which operates on registers only) and write back to memory. Yet, RISC machines have longer (usually 4 bytes) instruction words and plenty of general-purpose registers, which –again– we cannot afford.

### Implicit or Explicit Operands

Another discriminating factor in CPU architecture is how operands are “named” in the instructions. RISC systems typically have only *explicit* operands, that is in e.g. an addition, there are fields for identifying the two source registers as well as the destination registers. In other architecture, some operands are *implicit*, e.g., because one of the source operands is automatically reused as destination.

Most modern processors can access any register at any time with any opcode. While this offers a great flexibility for the programmer, it is impractical to interpret, even on an architecture that has more registers than the interpreted one.

If we consider a virtual instruction such as “add R3, R7, R2”, the interpreter would be required to get the current value of the 3rd and 7th registers in the register bank and store the resulting sum in the 2nd register. We couldn’t map the virtual register bank on some hardware registers of the real CPU as it would imply the CPU has to operate not on a specific register (such as GPR3), but rather *on a register which identity is computed at run time*, based on the fields of the virtual instruction. No instruction set we’re aware of supports such run-time indexing of the register bank and thus the VPU would have to emulate its register bank through memory.

As a result, most of the interpreted architectures, including SNAP and WASP[6], stick to a *stack*-based design where the actual register to be involved in ALU operation is known when the interpreter is compiled and where intermediate values are explicitly saved and restored from a LIFO structure.

### Accumulator and Stack

Breaking with the pure stack-based model, we decided to give the VPU an *accumulator* register that will be the sole location manipulated by most of the opcodes. Much like the

[5] integer addition has 14 different encodings, not mentioning the optional prefixes

[6] This is e.g., the case of JAVA and PostScript languages as well.

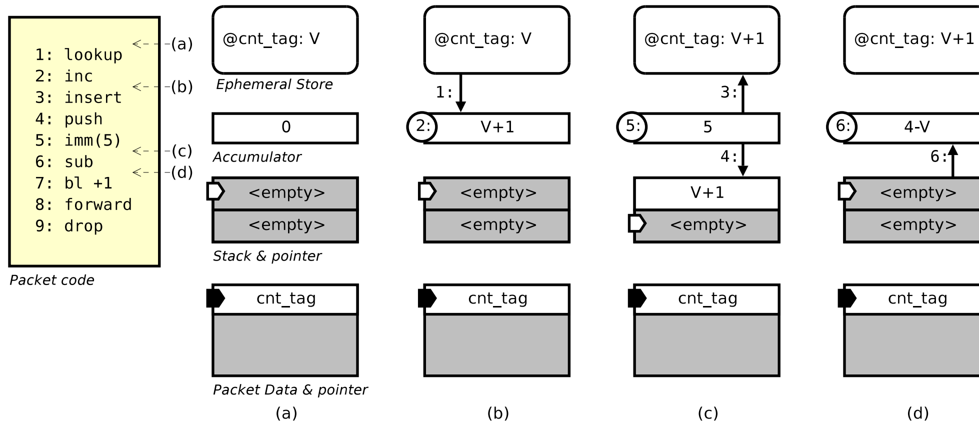



**Figure 4.4**: *VPU interpretation of COUNT WASP packet*

design around an accumulator simplified multiplexing/demultiplexing logic of ancient 8-bit microprocessors, the VPU interpreter code for almost all the opcodes will be simplified if we know the accumulator is the only place where results go, as we can then map it directly to one of the real processor's registers. Single-operand instructions such as *inc*, *dec* and *not* will for instance operate directly on the accumulator. When a second operand is required for arithmetic and logic operations, the top of stack is used. The stack is manipulated through *push* and *pop* opcodes and ALU operation may remove one item at the top. In a pure *stack-based* machine, *add* would instead use the two operands from the stack, remove them and push the result instead.

In some sense the accumulator can be seen as an extension of the stack for ALU operations. However, other operations (such as *lookup*) are implicitly destructive and will replace the old value of the accumulator with the new one. Requiring the code to explicitly save the previous value through a *push* when needed is one of those compromise that can keep the interpretor small enough. Unlike a stack, the accumulator is never "full" or "empty", and code for altering or checking the stack pointer can thus be omitted in many cases.

### A Simple Example: Emulating COUNT Instruction

Fig. 4.4 illustrates how a small bytecode program can implement the ESP "COUNT" operation and how the ESS, accumulator, stack and packet data are used together to achieve this goal.

(a) Packet emulation always begins in a known, clean state with data pointer at the start of data packet, accumulator cleared, etc.

(b) `lookup` uses the packet-carried key(`cnt_tag`) to access the ESS and store the value in $acc$. Our semantic is to leave the accumulator unmodified when the key is missing, and since the VPU guarantees $acc$ to be initially zero, we can skip the "set to zero if undefined" step.

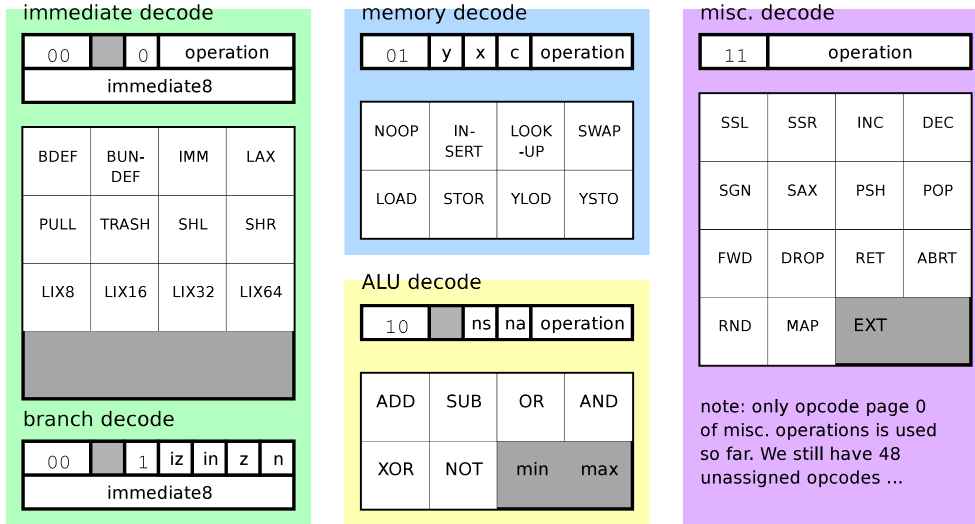



**Figure 4.5**: *The instruction set of WASP processor, grouped by "decoding family" that share common sub-operations*

(c) after incrementation, the counter value is written back in the ESS with `insert`. Note that the data pointer hasn't moved so far, so we're still reusing `cnt_tag` as key. The incremented value $V + 1$ is also explicitly saved on the stack with `push` opcode.

(d) as detailed at the start of section 4.2, COUNT packet is only forwarded as long as the updated value $V$ is below a packet-stored threshold. In this packet's case, the threshold was the immediate value $5$. The last four instructions perform the comparison and either `forward` or `drop` the packet accordingly.

### Hierarchically-Organised Decoder

While designing the instruction set for WASP, it appeared that most of the instructions could be grouped into a "family" that would share a significant part of the interpretation code. For instance, all instructions in "immediate decode" need to extract one extra byte from the instructions stream and all "ALU" operations need to retrieve an operand from the stack and update ALU flags. We decided to extend these properties and to provide "per-family" instructions modifiers that will e.g. tell whether the result of an ALU operation should or shouldn't be written to the accumulator[7], but also whether the data pointer in the packet should be advanced (the `|INX` modifier that will appear several times in examples). This is one of the design decisions that has lead to an interpreter smaller than the implementation of ESP operations on the x86 target.

[7]comparison operations can usually be achieved by subtraction or bitwise logical functions where only the flags are updated and the numerical result is discarded

### 4.2.2 Packet Variables

The `COUNT` packet is one of the rare cases where there's no need for storing information in the packet as result of evaluation, and the threshold was small enough to fit an immediate constant inlined in the code. In many other situations, we need true variables to be present in the packet:

- larger operands are impractical for "imm(xx)" opcode and will rather be loaded from the packet's "data" section;
- we might want to retrieve the current value of `cnt_tag` on a router, store it in the packet and inspect it on the end-system;
- we might even want to build a list of values on the packet, gathering one item on each traversed router.

In the case of WASP, values stored in the data section of WASP "programs" are used as banks of RAM memory by the virtual processing unit. On the other side, ANTS capsules can reference any number of sub-objects of any type. These are de-serialised when the packet is received by the router an re-serialised when packet is forwarded[8]. Previous studies with ANTS [Wetherall99] have however highlighted that serialization steps can be a significant share of active packet processing (up to 42% and 32% of the smallest ANTS capsule, respectively) and motivated active network designers to find solutions where data can be manipulated *in place*.

WASP data section also has a fixed size. When the source emits a WASP packet, it decides how many bytes will be present in the data portion and give them their initial value. If we want a "traceroute" packet collecting the IP address of traversed routers, and expect up to 16 routers to be met, we need to provision 64 bytes of data storage at the source and to write packet code that will write values one after the other (e.g. using an additional packet variable as an index in that array). This contrasts with the SNAP language where packets carry a stack of data that can grow and shrink as the packet gathers or consumes values on routers, at the expense of a more complex reassembling before the packet can be forwarded. The fact that an ESP (and WASP) instruction may be attached to a regular TCP or UDP packet by the source advocates for a *fixed-size* data section rather than something like SNAP's stacks.

#### Accessing Packet Variables

The VPU offers `load` and `store` opcodes to exchange data between packet variables and the accumulator. The actual variable to be read or written to is pointed by the *index register* $X$. While RISC computers typically have a memory address encoded with the "load" instructions, the VPU requires that the address is first loaded in the index register through one of the `LIXxx` opcode and only then used with `load`, `store` or any of the ESS access instruction (where it indicates the key to be used).

The analysis of operations available on the ESP router has revealed frequent cases where a variable is read, modified and written back before any other memory reference is

[8]This is sometimes also known as *marshalling* and *unmarshalling* of the data

issued, meaning that we can save one of the addresses if all the memory operations use the index register to indicate the address they're operating on. It is also frequent – due to the small program size and the absence of loops – that variables can be organised in the packet so that code access them in sequence rather than in random order. This has motivated the presence of the `INX` (for INcrement indeX register) modifier that allows any of the memory-referencing instruction to post-increment the data pointer.

To many aspects, this makes data access from the VPU look much like accessing the data tape of a *Turing Machine*, with the option of jumping to a specific position when needed. When data access are carefully designed and take advantage of `INX` to avoid extra explicit addresses in the program allowed us to halve the length of bytecode implementing `COLLECT` instruction, with a substantially improved interpretation speed.

**Variable Types (absence of ...)**

The virtual processor only knows two types of data: integer values and keys, and in most cases, it doesn't even distinguish the two. As soon as a data item is loaded in the ALU, the VPU assumes it is an integer and allows all ALU operators on it. Similarily, any data item[9] in the data section can be used as a key regardless of *how* it was put in the data section.

In contrast, the SNAP interpreter also supports floats, strings (though there's no operator to modify them) and opaque data types and the language itself is *type-safe*. Unlike WASP, SNAP is mainly designed as a glue between core services, some of which may require string arguments (such as a path in the Management Information Base of SNMP) and others may return parameters for other services as strings as well. It is clear that interfacing service components in a way that allow component $X$ to trust parameters received from component $Y$ even if they have been exposed to a script requires stronger type checking (and the presence of core-service definable types) than implementing WASP programs.

The ability to manipulate keys through the VPU is a new functionality brought by WASP; ESP packets are less flexible in that regard since the semantics of a given data in the packet is defined by the operation's pseudo-code and hardcoded in the router. Since an end-user is allowed to generate ESP packets requesting any keys anyway, we don't believe restricting modification of those keys by the packet program improve the safety of the platform. This may even be a key feature to enable more sophisticated programs that are sensible to some context information to pick the key they're going to use or decide of a random key on the router itself.

The only data type really manipulated by the VPU is thus 64-bit, unsigned integers. However, our experience with ESP has shown that applications designers rarely need 64-bit for their items. To allow storage of thresholds, IP addressses etc. without wasting space in the ESS or in packets, we allow the `load` and `store` microbytes to operate on smaller data items (bytes, 16-bit and 32-bit words). The application designed, through one of the `LIXxx` microbytes, indicates the transfer size for subsequents `loads`/`stores` and the VPU automatically expands and truncates values as they are moved between the

[9] properly aligned on a 64-bit boundary

ALU and storage elements. As a side effect, INX advances by *one packet data unit* as defined in the "size" part of the index register: `LIX8(p) ; INX` will position the data pointer one byte after `p` while `LIX32(p) ; INX` will place it 4 byte after `p`. The result is that, once the size and the base of an array of homogeneous items has been defined by `LIX`, we can scan the array by just repeating `LOAD | INX` microbyte.

### 4.2.3 Environment Variables

With ephemeral store and the VPU alone, our expressiveness remains quite restricted. As explained in the introduction, most of the monitoring applications require additional information such as comparing the timestamps of two packets, evaluate the current load of the output link, etc. Other interesting information may be found in the IP header itself, such as the actual source and destination, or the TTL of the packet. It is frequent to offer a programming interface to access such information in active networking. In SNAP, for instance, it is provided by a collection of context-polling opcodes, but none is defined with ESP. There was actually no need to define such an interface in ESP as the processing was implemented by native code, but in WASP, they become primitive features just like ephemeral store access.

Rather than dedicating opcodes à la SNAP, we chose to offer node, interface and packet information through banks of read-only memory that WASP programs can access just like they access packet variables. Memory that can be addressed by `LOAD` and `STORE` microbytes is split in banks of 32 bytes, and each bank indicates whether it can be written or not.

#### Planned Per-Node Information

The *node information bank* contains environment variable that are global to the whole router including the *features bitfield*, which indicates whether the node supports MPLS, IPv6, DiffServ, etc. This also includes for instance the node's *timestamp*, with enough precision (e.g. $1\mu s$) to allow packet-related functions to be described, but sufficiently large (e.g. 32-bit) so that we can at least detect period changes. In addition, a second field giving the *node uptime* in seconds can be used when larger timescales are needed.

This bank also provides the *node identifier* and its *domain identifier*, two 64-bit numbers that help figure out the topology of the network. Every VPUs of a single router will of course provide the same *node identifier* and similarily we expect all the routers of a given autonomous system to have the same domain identifier. While the "domain" as seen by wasp doesn't necessarily match the autonomous system number, it requires that any node between two nodes $n_1$ and $n_2$ that are member of domain $D$ is also a member of domain $D$. Moreover, some of the functionalities reported by the *features* field are inherently homogeneous over the whole domain. If a node $n$ advertises e.g. DiffServ quality-of-service support, it implies the whole domain it belongs to is DiffServ-compatible.

Only core functionalities are described in *features* field, so that end-system can quickly have a overview of what the network supports. Other bits might define whether the current node is an ingress, egress, core or end system and whether it is in a stub, transit or core

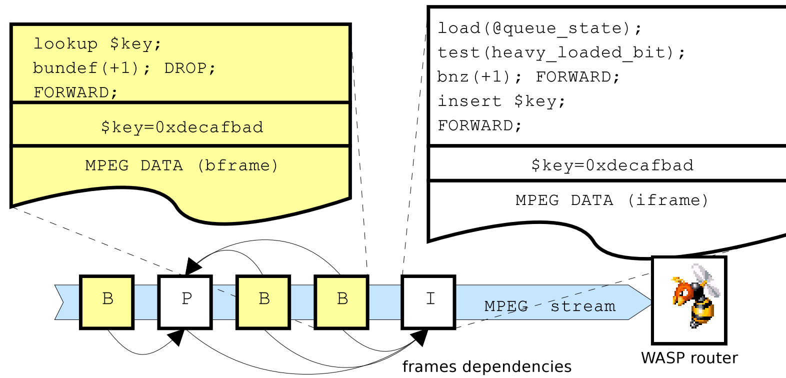



**Figure 4.6**: *WASP code attached to MPEG I and B frames to implement smart dropping*

domain. More detailed information could be provided at the discretion of the operator, for instance by means of protected tags computed from a hash of a standard capability name.

## 4.3 The Ephemeral State Store

### A Simple Example

Most active protocols will need information to be stored temporarily on intermediate nodes, so that it can be later retrieved by other active packets. Following the example of MPEG flow processing (see Sec. 2.3.2), we could drop intermediate *B frames* of a video stream if the *I frame* they refer to has been dropped by the node or if it is likely to be dropped, for instance due to a congested output link. This requires *I frame* to leave information on the router status for further frames. The state defined by the *I frame* should be distinct from states of other traffics (e.g. other applications, other end-systems) and is only useful as long as some depending *B frames* are present in the network. It is important for network availability and performance that this local storage remains easy to manage and can automatically discard information that is no longer pertinent.

Figure 4.6 illustrates an implementation of that selective frame filter with WASP programs. The code for *I frame* checks a bit in the interface environments variables to evaluate the load on the output interface (e.g., set by a RED queue manager). If the queue is too heavily loaded, a new entry is created (`insert`) in the router's store.

All *B frames* that depend on that *I frame* will carry the same key that acts as a unique identifier of the application flow and group of pictures (e.g. `0xdecafbad`), and use it to retrieve (`lookup`) the state and drop themselves if they are instructed to do so.

### Soft-Store vs. Ephemeral Store

ANTS [Wetherall98] and many other platforms use *soft-state*-based memory management to release memory that has not been used by packets for a given amount of time. Each item of a soft-store has an associated expiration timer that is reset everytime the item is accessed. Soft-state is common practice in network protocols to ensure devices do not

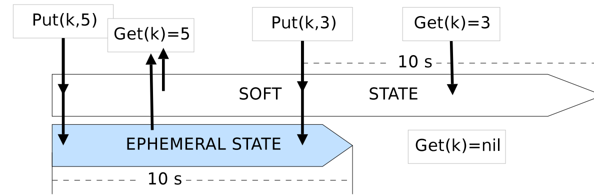



**Figure 4.7**: *Comparing the soft store against the ephemeral store with $\tau$ being 10 seconds*

keep obsolete information: as long as the entities refresh their entries at least every $\tau$ seconds, the information is kept. If an entity fails to refresh the information for more than $\tau$ seconds, the information is lost. Unfortunately, the soft-store may become pretty hard to manage, especially when it comes to tell whether there will be sufficient memory to accept a new flow.

It has been shown in [Calvert02] that memory will be much easier to manage in the *ephemeral store* approach, that is if the store only keeps data for a constant period (10 seconds), *regardless of how frequent the data is referenced during that period* (see Fig. 4.7). If we also ensure that all the data slots in the store have the same size, collecting free-for-reuse slots becomes simple enough to execute without disturbing packet forwarding tasks on the router, and checking if the router will have sufficient resources to process an additional flow simply requires that the router checks how many different slots are used by the flow. ESP terminology uses *ESS entries* or when it refers to the $(key, value)$ pairs and the 64-bit word used to identify the value is called *tag* or *key*.

Note that no access control is required for tags. It is simply assumed that each source picks up a random 64-bit word and uses it whenever it needs a key. How members of a distributed application agree on the same key for a given session, how this key is exchanged to avoid eavesdropping is left to the application itself. With randomly chosen tags, [Calvert02] shows that the probability that e.g. a user (among 16 millions) picking tag $X$ for the time period $[t \dots t + \tau]$ has a probability of $10^{-12}$ to experience a collision, meaning that the probability of at least one pair of users (still among 16 million) pick the same tag remains of $10^{-5}$. Even under those circumstances, a collision will only be experienced if the two flows share at least one WASP router on their path.

## 4.3.1 Ephemeral State Store Implementation

The Ephemeral State Store is implemented as a two-table structure. The largest one (typically held in DRAM) is the *entry table*, which stores the (tag, value) pairs. Each time a new entry must be created, the entry immediately after the last one is allocated, until the `next_available` pointer reaches the `last_cleared` , which corresponds to a full table. Every time ticks, a cleaning procedure sweeps the table and advances the `last-_cleared` pointer so that all the remaining valid entries have a creation time less than $\tau$ seconds in the past. Since the entries are stored in chronological order, cleaning operation is fairly simple.

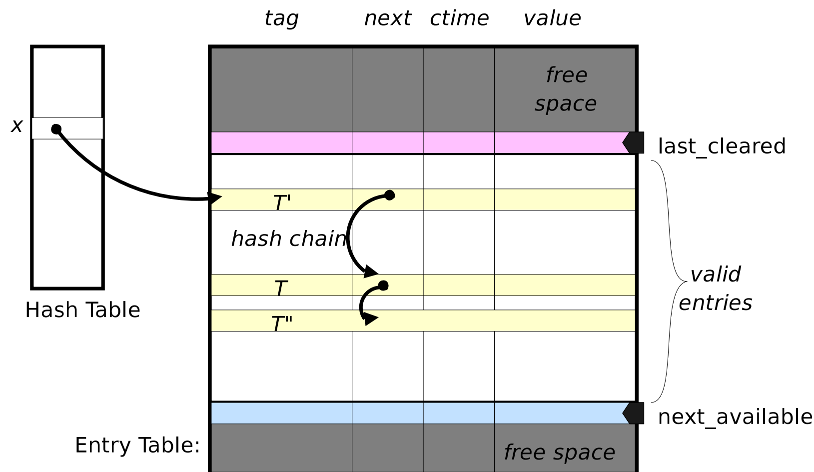



**Figure 4.8**: *Layout of the Ephemeral Store. Valid entries are between* `last_cleared` *and* `next_available`

In addition to the entry table, the state store also features a *hash table* that contains $2^k$ chain pointers[10] and that will be used to retrieve a specific tag in the entry table. When a given tag $T$ is searched, we first compute its hash over $k$ bits $x = h(T)$ and lookup $hash[x]$. If that entry is empty, the tag $T$ is not present in the store, otherwise, $hash[x]$ indicates the address of the oldest entry which tag matched the hash. As shown on Fig. 4.8, when several tags in the store matched the same hash, they are chained together thanks to the “next” field of the entry. The ESP router will then walk the chain until the matching tag is found.

In the native implementation of the ESP, operations are divided into three phases. In the first phase, all the lookups in the ephemeral store are performed, and addresses of the entries in DRAM (either existing or created) are kept in local registers[11]. The second step is the operation proper, which only manipulates the local registers. If the operation completes successfully, the third phase will write back the results using the addresses kept in phase one. The third phase thus requires no additional walking through the table.

### 4.3.2 Managing the State Store

One of the advantages of the ephemeral state is that, unlike in the case of soft-state, the system designer can compute in advance how much memory will be enough to handle the worst traffic scenario (e.g. full load on the interfaces with the smallest packets, all requesting new entries in the ESS). In [Calvert03], the authors illustrate that principle with the implementation of ESP on the IXP1200 network processor. With a flow of $10^5$ packets per second, and if at most 2 entries can be created in the state store by each packet (corresponding to 1280 bit-second of storage), $2 \cdot 10^6$ entries of ephemeral state will never overflow – which asks for 46MB of ephemeral state on the device.

[10] In [Imam03], the hash table was $2^{18}$ slots large

[11] or other low-latency memory resources such as the scratchpad or local memory in the case of IXP2400 implementation

However, according to [NPForum03], a dual IXP2400 system (such as the IXDP2400 development board) may face up to $12 \cdot 10^6$ packets per second when all packets have minimal size (64 bytes). With a packet size of 128, 256, and 512 bytes, the forwarding rate drops to around $6.5$, $3.8$ and 2 Mpps, respectively. In the worst case, it means we now need 2746 MB of ephemeral state if we want to guarantee that overflow cannot occur. For WASP, where ESS entries have been extended to 32 bytes, those 2746 MB only allow the referencing of one entry per packet, which will probably reduce the flexibility of WASP programs significantly. On the other hand, well-built WASP packets will typically *reuse* entries created by other packets, which suggests a smaller state store could statistically handle a mix of legacy and wasp traffic with an acceptable overflow probability. In that case, however, the router wouldn't be protected against denial of ESS service anymore.

We take the option of *not* addressing this issue in WASP design but rather to leave it open to the platform implementation. There are, however, various elements of ESP/WASP design that could help build solutions:

- since a network operator uses easily identifiable *private keys*, a WASP router could reserve a portion of its physical memory to support private keys only, offering a guarantee that network management applications based on WASP remain available even in the event of a denial-of-storage attack.
- the 'computation ID' (cID) of ESP packets clearly identifies which packets *need* to be processed on the same store. For both performance and fairness consideration, the implementation is free to handle WASP packets on any of those ESS provided that packets carrying the same cID are handled on the same ESS.
- a domain operator does not have to worry about fairness among end-users in general, but only about fairness between *its own clients*. In other words, all the traffic from one ingress point could be aggregated when it comes to tell whether the request for a new slot can be serviced.

We suggest that the WASP header receives carry the *ingress identifier* (iID) , an opaque 16-bit value that any WASP router within a domain could use to quickly identify the quota information that should apply to that packet. When a new packet arrives at an ingress point, the router will stick the identifier of the receiving interface in the WASP header.

Note that despite packets from the same source with different *computation ID* could be handled on different ESS (and thus not sharing fate concerning ESS availability), packets with the same cID *must* be handled on the same ESS, regardless of their source/destination addresses: failing to do so would prevent some ESS-based application to work properly. As a result, even if many stores are available, it wouldn't be correct to simply combine the iID and the cID to balance requests among them and provide fairness in that way.

Depending on the domain size, there might be too many or too few iIDs to identify all the ingress interfaces. We could imagine to combine the *previous domain*'s iID with the unique interface ID of the ingress router to get a better iID (that is, more accurately identifying the traffic source) when few interfaces are used. Note however that, unlike what's done in [Yang05], the new iID still fits 16 bits and we do not build a 'list of iIDs' that could be used as a path identifier (though the WASP code would of course be free to do so).

In the opposite case where the domain has more than $2^{16}$ interfaces, further aggregation will be necessary (that is, the iID will not uniquely identify one interface but rather a subset of the interfaces). This will act exactly as if there were smaller 'aggregator' domains submitting traffic to the 'core' domain instead of one, large domain. Similarly, iIDs could be hashed on the router to retrieve one of the $K$ available quota slots – which would only guarantee coarse fairness – or index a table of quotas defined by a service-level agreement with the client connected to the corresponding ingress interface. The operator is even free to re-allocate iIDs generated by the ingress routers and mix the two techniques to offer fine-grain fairness to "premium" clients and coarse-grain fairness to the masses.

### 4.3.3 Finer Access Control

As soon as WASP is used to locate services, packets need to use a *well-known* key to access information other participants might have left in routers. Such a well-known key can be for instance produced by hashing a service name, which makes them easier to guess for an external attacker than random keys of section 4.3. Therefore WASP introduces *protected tags* that can only be modified by *super packets*.

If the domain operator ensures that no super packets can come from outside, the end user can be sure that the information bound to the tag has been set up by the domain operator. The node determines whether a tag is protected or not by checking its key against a specific pattern[12], and will allow writes to such tags only to packets that are marked 'super' in their WASP header. Of course, this only works if the network manager filters out super packet coming from the outside.

#### Hash-Requesting Packets and Private Tags

When participants and attackers can come from the same domain, protected tags are no longer helpful. For such cases, WASP offers *private* tags, which work like protocol-private data in ANTS. Unlike other tags, the application programmer has no direct control on the key that will be used for private tags. Instead, the WASP node will hash the code contained in the packet and use the result as the *private key* for that packet, which is kept secret by the router. To make sure that regular packets do not attempt to use brute-force scan, private tags have an identifiable prefix and any attempt to use keys with that prefix explicitly will abort packet execution.

If the hash method is carefully chosen (e.g. a one-way hash like SHA-1), it means that packets will have access to the *same* private space *only* if they have the same code, which means we are sure they play the same game with same rules. Under those circumstances, an attacker can only hope to break the protocol by sending more (or less) packets than expected by the protocol – which a properly designed protocol should handle anyway. Note that the implementer of a WASP node is free to use any hash method that best suits its hardware as the resulting hashes are used only on the computing node. The only rule

[12] highest 8 bits are all 1 in current implementation

is that packets with the same hashed part operate on the same private tag and that packets with different hashed parts operate on different private tags.

#### World-Readable, Protocol-Writable Tags

While private tags guarantee that a collection of participants will modify the state in the router following a common set of rules (i.e. the protocol), their cost may not be acceptable for packets that just need to follow the decision without altering the state (e.g. a multimedia stream). Each packet would also have to carry the *whole* protocol so that it receives the same hash value, regardless of what part of the code is useful for itself.

As a result, the `expose` opcode allows a hash-requesting packet to have its private state accessible read-only as a protected tag. The result is a new ESS tag that contains a *link* to the private tag, which is transparently resolved by the VPU when a packet tries to read it. Writing to an exposed tag via a link is of course not allowed.

Note that the presence of a link only tells that it exposes private data, but not *what protocol* exposes them. It will thus be up to the protocol designer to ensure that the key used for exposing the data cannot be guessed by an attacker before the link is created. A simple way to achieve this is to generate the key from a random number on the router and inform participants of its value *after* data has been exposed.

## 4.4 Reference Implementation on x86

Before starting the implementation of WASP on a network processor, it was important to validate the concept and estimate the potential performance on a well-known architecture. The availability of the ESP component as a Linux module (for version 2.4) led to the natural choice of x86/Linux for the reference environment. Later, this module has been ported to kernel 2.6 for both x86 and x86-64. An experimental port for the XScale architecture (the controlling processor of the IXP boards) is under completion at the time of writing.

The Ephemeral State Processing component for Linux is mainly a *netfilter* module [Welte07] that will intercept and process both stand-alone and piggybacked ESP packets. The *netfilter* API has been added to Linux kernel 2.4 to ease the construction of modular firewalls. Several *hooks* are added to the default packet processing routine so that custom checks can tell whether packets can go on (`NF_ACCEPT`), should be dropped `NF_DROP`, etc. Netfilter defines 5 logical "locations" where hook code can be applied, out of which three are especially interesting as they perfectly match the *locations* defined in ESP.

**NF_IP_PRE_ROUTING:** this hook is applied after IP packet has been received on a network card and checked for validity, but prior routing decision is made. The "indev" argument is pointing towards the receiving device;

**NF_IP_FORWARD:** this hook is applied right after the routing decision is made. Note that packet destined for or originating from the local machine are *not* processed here.

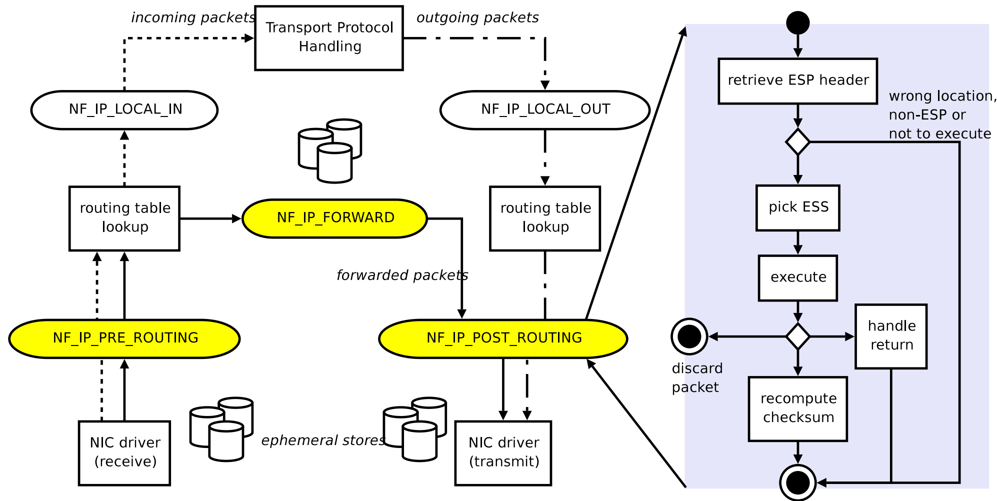



**Figure 4.9**: *Hooks in the Netfilter Architecture used by WASP router*

**NF_IP_POST_ROUTING** this hook is applied before a packet is delivered to a target network card for emission. The "outdev" argument will point to the device that will have to send the packet.

In the context of this work, we modified the processing function of the ESP component (depicted on Fig. 4.9) so that it can handle a new "operation" which happens to be the interpretation of WASP bytecode. This policy guarantees a fully backward-compatible packet format and processing semantic with ESP. In addition to the *netfilter* hook proper, the WASP/ESP module also sets up a periodic timer that will be in charge of collecting the expired entries from the stores and updates the *interface statistics* available through WASP. Most of the node and interfaces environment variables remain however statically evaluated at module initialization and reloading the module will be necessary after one changes the IP address of a network card or connects a 100Mbps card to a 10Mbps peer.

### 4.4.1 Validating the VPU's behaviour

Once the code for the virtual processor was written, and before starting the integration in a Linux module, we ran a collection of tests to ensure that the VPU was operating as expected, including checks of the arithmetic operations, insertions/lookups in the state store, branches, etc. in a user mode environment. These tests consist of a collection of benchmark "packets" that are prepared in data structures and submitted to a VPU instance. The resulting packets are then checked against assertions to ensure the proper results were obtained.

This testing environment also proved very useful to design the WASP programs used to emulate ESP operations. In a normal packet processing environment, the core `v_init` and `v_execute` functions are called one after another and the resulting state of the persistent `VPU` structure is ignored: only the content of the packet itself and the return code telling whether packet is forwarded, dropped, etc. are meaningful. In the case

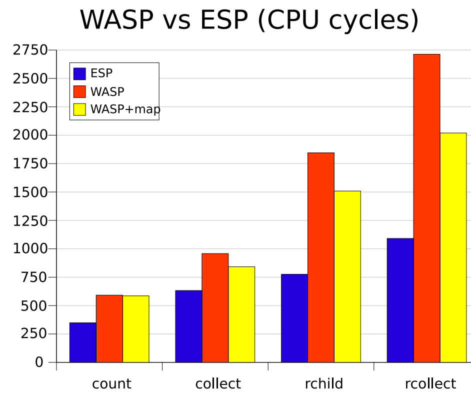



**Figure 4.10**: *Comparing operation processing time, when executing native code (ESP), interpreting WASP code based only on* `insert/lookup` *(WASP) or using the additional* `map` *opcode (WASP+MAP).*

of the functions provided in `vpulib` such as `vpu_trace`, we manipulate the packet's bytecode to insert "breakpoints" after every instructions and force `v_execute` to operate step by step on the packet. We can then read the state of the VPU to provide a meaningful trace of the packet's execution.

A second use of the "VPU library" in user mode is to profile the execution of VPU and ESP operations. The *time stamp counter* of the Pentium processor is used to compute how many CPU cycles have been spent during the processing of `v_execute` and `v_init` in order to compare their overhead against "native" implementation of ESP operations. Since code execution can be subject to several unpredictable events such as cache misses, mispredicted branches or even simply interruption by kernel code, the actual measurement is repeated 1000 times[13], and the average timing is reported. Special care needs to be taken to ensure that the virtual node remains in the same state between the tests, so that the same (longest) code sequence is evaluated at each iteration. In several cases, these profiling output were precious to choose one approach over the other in the implementation of the VPU, such as deciding of the most interesting size for internal structures or organization of virtual registers.

Fig. 4.10 shows the resulting measures obtained with this technique on a Pentium III machine running at 1GHz. Those results are pretty encouraging as we managed to have interpreted code taking no more than 250% of native code, but there's clearly room for improvement. We will see in Sec. 4.4.3 how we can modify the access to ESS entries (through the `map` opcode) to speed up WASP interpreter.

### 4.4.2 Experimenting WASP with Linux

First of all, we have checked that WASP packets were handled as expected. The WASP Linux module is installed on *bumblebee*, an AMD 300MHz debian machine, directly connected to our workstation (asmodan). A collection of small tools running on asmodan

[13]This is an empiric-defined value. Our tests have shown that values obtained with more iterations were not more precise. The average also ignores iterations interrupted by the kernel.

**Figure 4.11**: *Running WASP "count(5)" packets experiment. We can observe 5 packets that made it through the WASP filters (1), then 5 packets that were dropped by the receiver (2). Despite Ethereal's ignorance of the protocol (3), we can identify ESP header (4) and WASP bytecode (5) in the "raw data" section*

will then send WASP or ESP packets to bumblebee, using the *raw socket* API to forge the appropriate transport-layer protocol unit. In addition to the debugging messages logged by the module, we can use network inspection tools such as *ethereal* [ethereal] to validate our scenario. Of course, in this setup, is is mandatory that packets are processed on the *input* location.

Fig. 4.11 illustrates one of the most elementary test we can run: a collection of 10 COUNT packets with a threshold of 5 are sent to bumblebee, all using the same tag. The first five of them will find a counter value below 5 in the ESS and are thus allowed to pass the netfilter, while the remaining five ones should be dropped by the filter. After passing through the netfilter, the WASP packets are delivered to bumblebee's network stack, and as the Linux kernel doesn't know about ESP/WASP transport protocol, an ICMP error message (destination unreachable) is sent back to the source. These message are especially useful for debugging as they include the offending packet in their payload[14], which allow us to check the VPU's output.

For more sophisticated test cases, such as `collect` operation where we need to send `count` packets first, the ESP statistics provided through the `/proc` filesystem can give a nice overview of what's going on. E.g. after sending 10 `count(5)` and 10 `collect(3)` packets, we can see that the netfilter hook has been invoked 20 times, and that 6 packets were accepted (presumably the 5 first `counts` and the last `collect`) and that 14 packets were dropped (last 5 `counts` plus the nine first `collects` which had to "wait" for the final result to be available). Once again, this can be confirmed by ethereal and we can check in the final ICMP error packet that the sum value is indeed correct. Such quick checks can become handy when test cases include thousands of packets rather than a few.

[14]The IP header plus 64 first bytes of payload, according to RFC 777

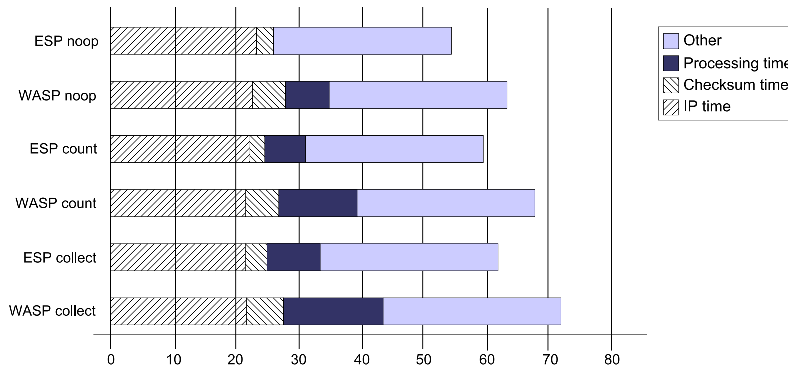



**Figure 4.12**: *Forwarding latencies on mylady, in microseconds, showing how different stages of packet processing contribute to the latency*

We further confirmed the experiment by repeating it over a topology including Linux routers and Cisco equipments to ensure "regular" routers were capable of forwarding WASP packets even if they were not equipped with WASP, using both x86, x86-64 and XScale systems as a host.

Robustness tests included the submission of `count` WASP packets to a host during a period of several hours. This test has revealed a flaw in the ephemeral store implementation from University of Kentucky [Calvert03w], due to a missing check in the ESS cleaning process. When `ess_clean` function reached the end of an ESS table, it wasn't correctly wrapped back to the start of the table, leading to a kernel crash. This flaw and a few others were fixed in [Martin06w] and reported to Calvert's team.

### WASP on a Linux Router

The timings presented on Fig. 4.10 only take into account the time required to execute the WASP packet's bytecode or the ESP operation's routine, only one of the steps among all the operations involved in forwarding a packet on a Linux router. To evaluate the actual overhead, we compared the forwarding latency – i.e. the time spent between the reception of the packet on *eth0* and the transmission of the forwarded packet on *eth1* – on *mylady*, a 300MHz Pentium II router featuring Linux kernel 2.4.18. To avoid interference with other *netfilter* hooks, the *iptables* firewall was deactivated during the tests, and we tried to minimize cross-traffic on the routers to minimize queueing delays. Under these conditions, *mylady* took on average $44.6\mu s$ to forward an ICMP "echo request" packet and $99.8\mu s$ to reply to a "ping".

Packets were captured using *ethereal* tool and stored in *libpcap*-compatible traces, which were then processed by a custom Perl script to extract arrival and departure time of each packet, computing the average latency for each protocol (ICMP, WASP, ESP, UDP, TCP, etc). As Fig. 4.12 shows, there's only a difference of 13% between the total forwarding time of `ESP:count` ($59.42\mu s$) and `WASP:count` ($67.65\mu s$) packets. A similar interpretation overhead can be observed for `collect` operation (16%, as WASP took on average $71.8\mu s$ against $61.77\mu s$ for ESP).

To better understand where the difference comes from, we measured the latency of the same packet flows when the WASP/ESP module is not loaded, which gives us the

*IP forwarding time* of each packet. "IP forwarding" include the time required to retrieve the packet from the interface card, lookup the routing table, enqueueing and transmitting the packet. Since all our packets have similar size and go to the same destination, the IP time is roughly constant for each packet. What does more depend on the packet size, however, is the time required by our module to compute the *CRC checksum* over the ESP operation (or WASP program). While its contribution is sensibly smaller, it may require between 600 and 800 cycles on a Pentium-based machine, and it needs to be computed twice per packet (first to check the received packet is correct, and then to reflect the new payload), which contributes to 7% of the forwarding time in the case of `WASP:count`, for instance.

What remains corresponds to the time spent in `linuxesp_hook` function and can be further split between *processing* the WASP code proper and other supporting code, such as locating the ESP header, identifying the proper state store, checking the location, etc. In addition, some of the supporting code must be called three times (on input, output and center hooks) even if only one location needs to execute the code. To evaluate the contribution of the ESP processing routines that we profiled in section 4.4.1, we issued `noop` packets that have the same size as `count` packets but the simplest forwarding code. To implement `ESP:noop`, we simply used the `ESP:compare` packet, that has the same size as `ESP:count`, but we replaced the processing function with a function doing no ESS access and just indicating that the packet should be forwarded. In the case of WASP, the `noop` packet starts with a `FWD` microbyte and is padded with `NOP` microbytes to have the same size as `WASP:count`. The difference between `WASP:noop` and `ESP:noop` latencies is the overhead required to initialize the VPU, and will be present for all other WASP packets. Since time for "other support code" measured with `ESP:noop` is independent of the packet constant, we can now estimate the *processing time* contribution of each kind of packet.

**Portability**

For reader's information, here follows the list of things that need adjustments in WASP Linux module to allow portability across x86, x86-64 and XScale architectures.

- CRC32 code needs adaptation to work on 64-bits machine (mainly adjusting some variables type).
- some of the kernel functions involved in packets reflecting (`ip_route_output` are obsolete in kernel 2.6 and had to be replaced using `ip_route_output_key`).
- encoding of ESP header is no longer correct on big-endian machines. We replaced the C bitfields with explicit operations on individual bits of the "flags" field.
- GCC back-ends for ARM and Intel x86 family do not have the same structure padding and alignment rules. This is especially an issue with the ESP header which is only 6 bytes long to ensure word-alignment of ESP operands. On the StrongARM target, this structure was silently expanded to 8 bytes by the compiler, screwing up packet format.

Listing 4.1: *implementation of ESP "count" operation in Linux*

```
static int esp_count(int ess_no,operand_t* operands)
{
  value_t     cur_val;
  ess_item_t*   item=ess_find_create(ess_no, COUNT_VAL_TAG, NULL);
  if (!item) return -ESP_ERR_ESS_FAILURE;
  cur_val=ess_read(item);
  cur_val=value_increase(cur_val,1);
  ess_write(item,cur_val);
  if(value_comp(cur_val, COUNT_THRESH_HOLD)<= 0) {
    return ESP_ACCEPT;
  } else {
    return ESP_DROP;
  }
}
```

### 4.4.3 More Efficient Access to ESS in WASP

Among all microbytes, interactions with the ephemeral state store are the most important to tune, as they are the most costly operations the VPU will have to handle. In the case of ESP implementation (on both x86 and IXP network processors), for instance, a lookup does not only return the *value* of the tag, but also a pointer to the entry itself that is later used to write back the new result after the operation is completed (see line 11 on listing 4.1). Beside the cost of looking up the hash table (and hashing the tag), this can save us from walking the chain in the ephemeral state in the case of collisions in the hash table.

In the case of WASP VPU, however, only the values are viewed by the programmer, and a lookup-and-update cycle can only be identified by the fact that the same key is used for the *lookup* and the next *update*. Things are further complicated by the fact that the data pointer could have been moved or the key could have been modified in the packet's storage area by `store` microbytes issued between `lookup` and `insert`. In other words, we need to store the "resolved pointers" to ESS entries in a cache transparent to the bytecode programmer. We tested two cache policies, using the benchmarking framework described in section 4.4.1:

**no caching** every lookup or insert is independent of any previous ESS operation.

**small cache** the cache has a single entry that keeps the last resolved pointer.

**full cache** the cache has one entry for every possible key location. Since there's a maximum of 128 bytes for packet data and that keys must be aligned on 64-bit boundaries, that makes a maximum of 16 tags to keep track of. When the $k$th key in the packet is used to lookup an entry, the corresponding $cache[k]$ entry is used and the $k$th bit of cache controlling variable is set.

Table 4.1 show the measured timings for each caching policy and compare them with the "native" implementation of the equivalent ESP operation. Note that the *small caching*

policy behaves better than *full caching* here. This can be explained by the fact that ESP operations (as described in [Calvert05w]) don't look up for a given variable more than once and that updates can be done before another lookup is issued. The "small caching" policy is extremely efficient in those circumstances as the cost for setting up and maintaining a more complex policy such as *full caching* is thus greater than the benefit one can expect from the cache hit ratio.

Yet, generally speaking, it could be interesting to have a more flexible caching policy than "small cache". An implementation of WASP on IXP microengines, for instance, could take advantage of the content-addressed memory facility to provide a cache of a few keys without requiring operations such as `store` to be modified to enforce "full cache" consistency.

### Mapping Larger Entries

The idea of ESP was to provide a small amount of data per storage entry – namely a single 64-bit word. However, when we consider the applications of ESP (and WASP), the state we process rarely remains atomic, but instead consists of tuples. The `RCHILD` and `RCOLLECT` operations used for the robust aggregation service, for instance, may require 3 or 4 logical variables in the ESS to store all its state. In WASP, this is even more concerning, since performance depends on the access pattern of keys in the ESS. We could make the VPU more efficient and easier to program if we'd allow larger memory entries in the ephemeral state.

We therefore decided to size up ESS entries to 32 bytes[15], which is enough to hold all state required by the most complex ESP operation using a single key. Yet, 32 bytes remains small enough so that protocols that required only one 64-bit value do not waste too much memory. If we refer back to Fig. 4.8 that depicts the internal structure of the ephemeral store, we can observe that a single entry requires 24 bytes of storage for the tag, the 64-bit value and the control fields. In other words, merging two entries of the "regular" ESS is enough to provide the requested 32 bytes of storage and, as soon as a majority protocols require at least two values per node, we're actually improving memory efficiency. While the reference implementation simply sizes up *all* entries, we could rewrite the ESS access and cleaning procedures so that two 24-bytes entries are effectively merged when a "bank" is requested.

For the network programmer, these enlarged entries are available through an additional bank of memory, using a new indexing register ($Y$) and a new pair of opcodes for

[15] which happens to be the size of a "memory bank" in the VPU: the granularity at which we can define read/write capabilities and access non-contiguous physical memory

| | no caching | full | small | mapping | native |
|---|---|---|---|---|---|
| count | 721 | 637 | 592 | 586 | 349 |
| collect | 1245 | 1082 | 958 | 842 | 633 |
| rchild | 2058 | 1830 | 1845 | 1509 | 775 |
| rcollect | 2980 | 2438 | 2394 | 2020 | 1091 |

**Table 4.1**: *Comparing caching policies. Timings in CPU cycles on 1GHz Pentium III*

**Figure 4.13**: *A summary of new memory-related opcodes.* `insert` *and* `lookup` *for direct access to a 64-bit value from the ESS,* `load`*,* `store`*,* `yload` *and* `ystore` *for exchanging data between the accumulator and memory banks,* `lix`*,* `map` *and* `swyx` *defining what is available through the* $X$ *and* $Y$ *banks*

loading and storing values (namely `yload` and `ystore`). We can thus read individual bytes, or aligned 16, 32 and 64-bit words out of a mapped entry, which gives additional flexibility to the programmer over the fixed-size of the regular entries. Having two separate banks for ESS mappings (defined through the new `map` opcode) and packet memory compensate the lack of a more efficient way to implement truly random access pattern to memory resources of the VPU.

With the `swyx` opcode – which can be used to swap banks $X$ and $Y$, we can even benefit of that new flexibility to perform copies from one portion of the packet to another or between "environment variables" and packet data. The behaviour of those new opcodes is summarized on 4.13.

As already shown on Tab. 4.1 and Fig. 4.10, rewriting WASP packets with proper use of the `map` opcode has allowed us to reduce execution time by about 30%. We can even expect higher improvements on IXP architecture thanks to burst transfers with DRAM. The "native" code we compare to, however, still uses 64-bits per entry and would equally benefit of larger entries even if we preferred to keep the same reference through all the tests.

Using the Pentium-based implementation, we compared the execution time of a sequence of `lookup` against a `map` followed by several `yload`. Tests show an improvement of 18% (two variables per bank) to 35% (four variables per bank) in the processing time as soon as several variables need to be updated, and we expect improvement to be even more important on saturated ESS storage (e.g. when collisions occur inside of the ESS hash table).

Besides performance improvement, larger entries also play a significant role in the support of operations atomicity in WASP. For protocols correctness, it is important we can assume that state stored on routers always remains in a coherent state. Either the program behaves as expected and a new (coherent) state is written into the store, or the store should be left untouched. In ESP, this is simply achieved by splitting the implementation in three phases (lookup, compute, write back) that are typical of transactions-based computing.

By *mapping* ESS entries that contain full protocol state, we can offer the same correctness without placing additional constraints on how packet code should be written. Modified state is only written back to the ephemeral store when validated by another `map` opcode or one of the packet control opcode `forward`, `return` or `drop`. If an error is detected (either by the VPU or the packet's program itself through `abort`), the modified state is discarded and nothing is written back in the store.

Another potential interest in larger entries resides in the support of *bloom filters* [Bloom70] operations with the VPU. In peer-to-peer literature, Bloom filters [Zhao03] are commonly used to offer a compact representation of a set associated with information about where a more detailed set can be found. In its current implementation, data aggregation using `RCOLLECT` requires that each data source indicates whether it has left state in the store or not by creating an entry whose tag is the node's identifier. While Calvert et al. suggested that bloom filters could be used instead to avoid O(N) state required on the node, implementing bloom filters over 64-bit is less convincing than with 256-bit entries. In the future, this might lead to additional microbytes that would copy, set or test bits over a whole memory bank in a single VPU instruction.

### 4.4.4 Too Cheap, Really ?

Our goal is to make WASP "too cheap to measure" compared to IP and ESP. To evaluate whether we reached that goal, we ran a serie of throughput measurements comparing how fast the x86 reference implementation is capable of handling a flow mixing *wasp* packets (a variant of *count* instruction, 92 bytes on wire) and *bulk* packets (an IP header with 1024 bytes of payload). We crafted a packet-generator with raw sockets API running on *bumblebee* and sending packets through *asmodan* towards a local source. Both machines are connected using PCI RTL-8139 network cards that operate in full-duplex 100Mbps mode[16].

Our packet generator runs a loop sending 100,000 packets – mixing bursts of $k$ wasp packets and $(100 - k)$ bulk packets – with the `sendto` system call. If `sendto` fails to enqueue the packet (e.g., because we already filled up kernel queues faster than what the 100Mbps card can handle), we introduce a small delay and try again until the packet is successfully enqueued. We then monitored a collection of flows with varying values of $k$ (and thus, varying average packet size) using both *ethereal* and *iptraf* and measured the generated load.

The first observation we can make is that the throughput of flows *bumblebee* can produce depends on the actual average packet size of that flow: while large packets can easily fill 97% of the wire, that value drops as soon as we approach 50% of small packets and only 24Mbps are used when all packets are 92 bytes large. Another observation is that, without the intervention of a rate controller at the emitter, *asmodan* is barely capable of receiving 40% of the packets in the best case (largest packets). When the average packet size decreases, the drop rate even increases and nearly 99% of the packets are dropped when minimal size is used. While we haven't investigated the actual reason of

[16] as reported by ethtool

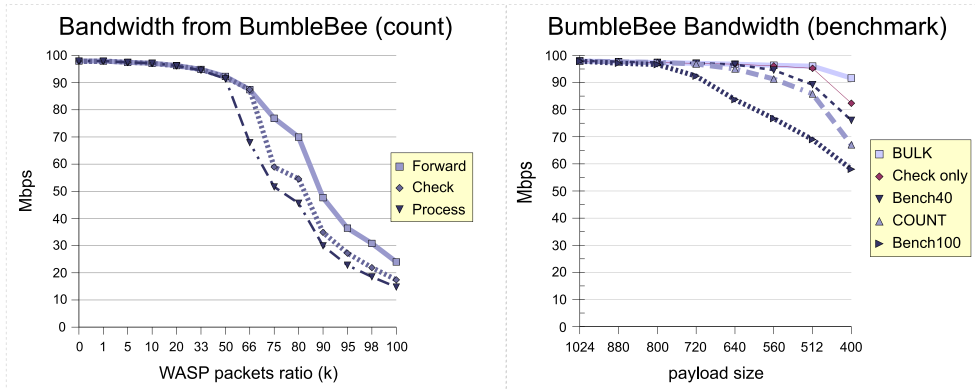



**Figure 4.14**: *Maximum throughput achievable by 300MHz host (bumblebee) with (left) a mix of 92 bytes WASP and 1058 bytes bulk packets and (right) flows made of bulk, count and benchmark packets with homogeneous (but varying) packet size.*

these drops, chances are that the receiving card is actually unable to follow 100Mbps when all packets on the wire are to be caught.

A possible way to throttle down the emitter is to run *iptraf* on bumblebee while emitting. This has the effect of virtually halving the submitted throughput (55Mbps whith largest packets only), which the receiving card can now follow. We varied the average packet size in this scenario and it appears that the loss rate can be maintained around 1%. While we haven't investigated this behaviour either, it sounds reasonable to assume that, when *iptraf* is running, packets we submit are also reported by the card, requiring two transfers over the PCI bus.

### Host Performance

In order to measure the speed at which *bumblebee* can submit packets, we're running the packet generator without throttling in three different scenarios:

**forward:** no WASP module is loaded, neither on bumblebee, nor on asmodan. This gives us a comparison point about the performance of the legacy linux network stack.

**esp checking:** WASP module is installed on bumblebee, but packets generated request execution on *input* ESS and thus WASP hook terminates early for all packets. In the case of "bulk" packets, this happens immediately after the classification tests while WASP packets will also inspect the "location" bits. Note that we reorganized `linuxesp_hook` so that CRCs aren't checked until we have the confirmation that we are on the proper location.

**wasp processing:** this time, WASP packets request execution on *output* ESS and are processed before being enqueued on *bumblebee*. For those packets, two CRCs are computed (over the 52 bytes of WASP payload) and the "count" instruction is processed.

We can observe on Fig. 4.14(left) that, as long as there's at least 50% of "large" packets, all configurations behave roughly the same. Another interesting fact is that after

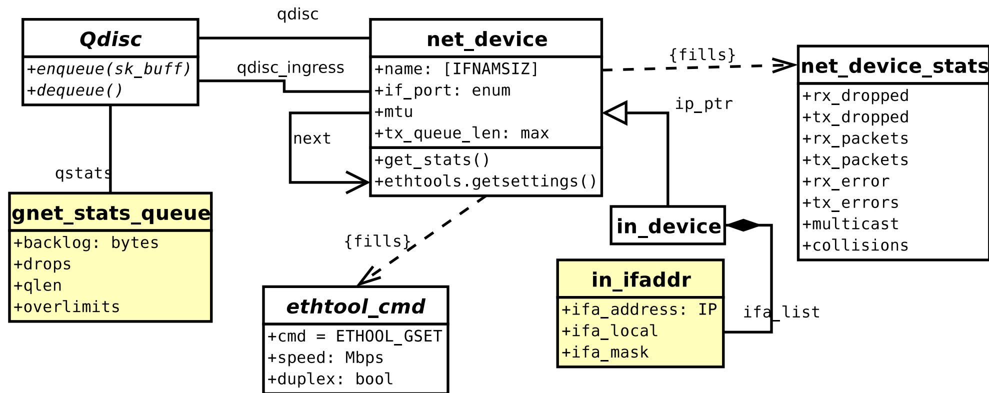



**Figure 4.15**: *A simplified look at the structures, function calls and fields involved in node and interface statistics reported by WASP*

the "threshold" of 50%, the ratio between measured throughput (with WASP module installed) and reference throughput (without WASP) remains fairly constant. The reader should also note that while "normal" count packets are dropped if the threshold is met, the tweaked packets we generate here are *always* forwarded.

### A More Comprehensive Benchmark

In most WASP applications where bandwidth is a concern, we do not use *standalone* WASP packets, but rather programs attached on regular traffic (e.g. for enforcing preferential dropping on a video flow, measuring jitter). To simulate that behaviour, we artificially "inflated" WASP packets to make them 1058 bytes on wire as well. In that case, the performance degradation is solely due to our netfilter hook processing, and not to some interface-card side effect.

If we ensure that CRC manipulation only covers the WASP program (not the entire packet), then our "count" packets are indeed undistinguishable from "bulk" packets for payload sizes above 640 bytes. We thus ran further measurements with a *benchmark* packet that performs 8 ESS accesses, periodically change the key and the *computation ID* used by the generator (to fill the ESS) and copy a bank of memory into the packet. As Fig. 4.14 shows, that "benchmark" packet can still be generated at 60Mbps when all packets in the flow require WASP processing – clearly expensive enough to be measured, but still a honorable performance for the modest hardware running the packet generator. A flow made only of `WASP:count` packets behave slightly worse than a flow containing 40% of "benchmark" packets and 60% of "bulk" packets.

With packet sizes lower than 100 bytes, even the code checking whether the packet should be executed on the VPU or not consumes too much CPU cycles to keep the interface fully busy and we therefore didn't include those figures here.

## 4.4.5 Node and Interfaces Statistics

Gathering and presenting statistics over node operations through the *environment variables* memory banks in WASP is probably one of the trickiest part of implementing WASP

on Linux. Information is scattered in multiple structures (see Fig. 4.15), which all have their own purpose. The `net_device` structure for instance, which is our root to access most information, is over 200 lines long and, besides locks and queue entries of all types, mainly contain pointers towards protocol-specific data, queueing disciplines instances, etc. Unfortunately, an important number of these structures have been revised between versions 2.4 and 2.6 of the kernel and at the time of writing, we're still lacking a comprehensive documentation source covering the Linux network stack (i.e. an equivalent of [Wehrle03] or [Rio04] for 2.6).

When our module initializes, the `vpu_prepare_node` function has to walk the `net_device` list, checking each interface's name and looking for a suitable entry in the list of `in_ifaddr` associated with the IP-specific sub-structure `in_device`, so that we can fill address and netmask field for each interface statistic bank. We also want to offer a "primary node address" in the node state bank (i.e. as the node ID) but Linux doesn't provide such abstraction. Instead, each network device may have its own address. We thus scan all the network interfaces to retrieve one that has a suitable address (e.g. we might prefer a routable address over a private 192.168.xx.xx) and pick one of the remaining addresses as the node's default address. In addition, we prepare the "static" part of interface-related state.

In order to support QoS-related applications, we want to give per-interface statistics such as packet drop ratio, or current and maximum bandwidth. Some of these information can be obtained from the `net_device_stats` structure and are periodically sampled by `vpu_update_stats()` function. Only the queue's length is updated "live" for each packet processed by the WASP filter. It should be noted that our proof-of-concept implementation of WASP on Linux doesn't fully support QoS parameters yet and e.g., always advertises the full capacity to be available.

Due to the absence of division in the VPU, WASP programs might prefer prepared statistics such as the ratio between current traffic load and interface capacity rather than the nominal traffic load itself. The interface capacity, however, is surprisingly difficult to obtain and doesn't appear anywhere in the device-related structures. Analysis of management tools such as [ethtool] revealed that special IOCTL calls were made to the device to have the device driver fill an `ethtool_cmd` which reports interface speed (defined during layer-2 negotiation) and full/half duplex status. For devices that do not support *ethtool* operations, the `if_port` may provide the information, but it is clear that some work should still be done in this area to have proper capacity reporting of non-Ethernet devices.

### Concurrent Access

The *VPU* structure will contain all state required for execution of WASP bytecode. In order to ensure proper execution of a WASP packet, we need to make sure that only one thread at a time will attempt to use a specific VPU structure. While the usual solution to this problem is to associate a *spinlock* with the structure, we can avoid this here and offer higher throughput to multiprocessor systems by having *as many VPU structures as hardware CPUs*. Using the `smp_processor_id()` macro available in Linux, we can

pick the right VPU every time we have to process a packet. Since the hooks are processed with interrupts disabled[17], there's no risk for a given CPU to start handling a new WASP packet before the one we're interpreting gets completed.

It should further be noted that no restriction is put on *which location* these VPUs are bound to. A given VPU structure might once be associated with *ethX* interface and then be associated with *ethY* a few microseconds later. Moreover, several running instances could be associated with the same *ethX* location : the ephemeral state stores have been designed to allow such concurrent accesses with minimal performance penalties. Not only the ephemeral stores are dynamically rebound to the VPUs, but also the node configuration and interface statistic banks. As the VPU cannot modify these statistics, there's no need for synchronization procedures here either. At worst, the thread that performs ESS cleanup and statistics update could modify the banks while a packet takes a snapshot of the whole bank. Like IP, WASP is a "best effort" service and it will be up to the application designer to take care of such updates if they are relevant for the service being developed.

### Endianness Strikes Back

The support of statistics banks in WASP raises another issue about endianness of local data storage. Data in WASP packets are always stored in *network byte order* (big-endian) and, for performance reasons, the VPU registers as well as entries in the ephemeral store are in *host byte order*. This is usually not a concern because `load` and `store` instructions (that manipulate packet data) do network-to-host or host-to-network reordering as they move data. However, if a program writes data byte-per-byte in a ESS entry and another program reads them back using 64-bit access, the actual data pattern it will get will depend on the host's processor. While the problem currently remains open, there are two approaches we could envision:

- Enforce network byte ordering everywhere but inside VPU's registers. This requires simply to modify `yload` and `ystore` to make them look more like `load` and `store`. This is transparent for the IXP network processor, but will be expensive for x86 processors which have to do the conversion.
- Enforce network byte ordering for "statistics banks" and "configuration bank" so that it is possible to use 64-bit data movement to retrieve them quickly and keep ESS in host-ordering. The resulting programming model for the VPU would however have undefined behaviour when e.g. one writes two 32-bit quantities in an ESS entry and read back one 64-bit word.

## Conclusions

We have provided a reference implementation of WASP router as a Linux *netfilter* module for x86, x86-64 and XScale architectures. We also compared the performance obtained

[17] Our experiments with WASP have shown that hook code has full access to packet structure and that it may be executed in non-interruptible context.

by WASP versus those achieved by the native implementation of ESP operations. While the execution time of WASP may be twice the time for native ESP handlers, we illustrated that the actual impact on the *forwarding latency* on a Linux router is similar for the two (between the time required to forward a similar packet and the time required to reply to a *ping* control packet).

Regarding performance on end-systems, we have shown that, as long as packets remain large enough (half of the *Maximal Transfer Unit* of 1500 bytes in our experiments), the bandwidth achieved by a user-process forwarding WASP packets is identical to the one achieved with regular IP packets of the same type (about 97% of the 100Mbps link). We also have observed that, for IP packets as well as WASP packets, the end-system have increasing difficulties to achieve full wire speed as the size of packets decreases.

Comparatively, ANTS [Wetherall99] announced a maximum of 16Mbps for packet size approaching the MTU, and with a latency between 500 and 700 $\mu s$ (against around 70 $\mu s$ for WASP). Such comparison should of course be moderated by the fact that those performance depend strongly on the actual hardware used, and by the fact ANTS provides a flexibility we cannot compete with.

The whole design of WASP virtual processor has been organised around the idea that processing WASP programs should be possible on a network processor – namely the IXP 2400. At the end of this work, we have all reasons to believe that the implementation of WASP should indeed be possible on Intel IXP2400 and that the performance gap between WASP and IP (when measurable) has good chances to be even smaller on that platform. In his thesis, N. Imam concluded that an IXP1200-based system could switch ESP packets at line speed with around 48MB of ephemeral storage. Whether we could do the same on IXP2400 (which has gigabit Ethernet rather than 100Base-T) with WASP or not couldn't be predicted accurately at this time, and chances are that we'll have to enforce restrictions on the packet-per-second ratio if we want to achieve full bandwidth.

*A Klingon Warrior uses only machine code,*
*keyed in on the front panel switches*
*in raw binary.*
– The Klingon programmer, Steve Baker.

# Chapter 5

# Experimenting WASP on IXP2400


## Abstract

*This chapter details the structure and the working of a WASP implementation on the IXP2400 network processor. We then compare the performance of WASP and ESP in three aspects: forwarding latency under low stress conditions, forwarding latency when saturating the state store and throughput under varying state store conditions. We finally give guidelines for a future self-optimizing WASP framework that could produce pre-compiled operations for the most frequent programs.*


## 5.1 Development on IXP

Over 20000 lines of assembly code written by 2 corporates and 2 universities, with heavy use of macros and branches optimised for speed that will be split on 8 multi-threaded processors. That short description could sound scary to many programmers, yet this is what we have to deal with in the case of WASP on IXP.

We are well aware of the existence of the `micro-c` language introduced by Intel with the IXP2xxx development kits, but still we decided to keep working at the lower level of micro-assembly. This choice is motivated first by the immediate availability of receiver, transmitter and classifier microblocks for the ESP filter written in micro-assembly, as well as macros for ephemeral store management that we will reuse in our work. Rewriting (and debugging) those blocks in `micro-c` would have required substantial work at a time we weren't very comfortable with the SDK and the IXP hardware.

The second motivation comes from the nature of the microblock we plan to develop. While writing the reference implementation of WASP interpreter in C, several parts needed to be repeatedly disassembled and studied with care to locate performance bottlenecks and whether dedicating a register to this or that variable, whether making this or that function static for in-lining would benefit interpretation speed. Such "hide-and-seek" games with the compiler for performance profiling are only possible when one has

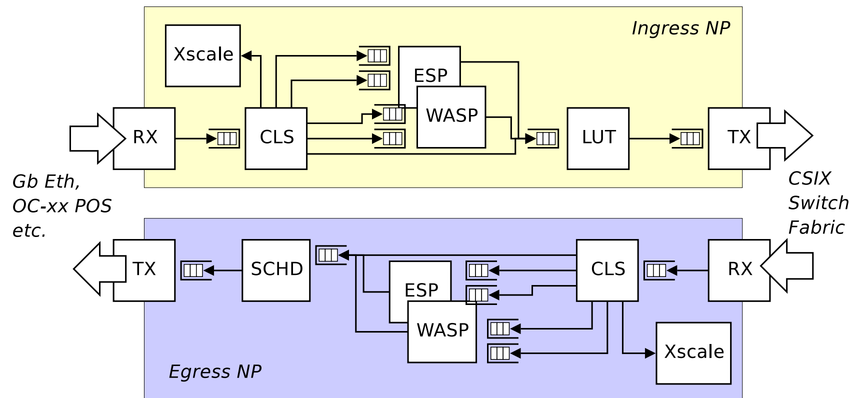



**Figure 5.1**: *Microengines and buffers in a (hypothetical) WASP NP-blade for a switch-fabric equipped router, showing individual MEs for packet reception (RX), CLasSification, Look Up ip forwarding Table, SCHeDuling and transmission (TX)*

a strong knowledge of the compiler's behaviour and the architecture instruction set, both of which we were lacking for the IXP.

By nature, the WASP interpreter does not involve complex algorithms or data structures, and we already had gone through "youth error" with the reference implementation. We thus opted for micro-assembly WASP with good confidence that we wouldn't face the nightmares of spaghetti code or assembly source refactoring.

Most of the information we could give about our experience on IXP would likely be perceived as "implementation details" by most of our audience and we have tried hard to keep this chapter readable for someone that has no preliminary knowledge of network processors (other than what we explained in chapter 3). Yet, our experience taught us that in the field of network processors, every bit of published information may save days of measurement and debugging to other developers. The reader should therefore not be surprised to find this chapter rich in footnotes.

### 5.1.1 Overall Implementation

An application on the IXP is typically split in several components (microblocks and software components on the XScale) that will be running on the different processing elements. Pipe-line processing is typically assisted by hardware *scratch rings*[1] programmed to relay packet handles (plus optionally few metadata) between microengines.

The packet content is transmitted directly from I/O buffers into DRAM (by the RX microblock) and only meaningful parts will be fetched on demand (e.g. by the classifying microblock).

Fig. 5.1 shows how WASP and ESP microblocks could be inserted in a typical line card application using a double-IXP2xxx board connecting multiple gigabit Ethernet ports

[1] Alternatively to the use of scratch rings, we can exchange information using the "next-neighbour" (NN) registers – a ME-to-ME FIFO queue facility. While NN registers offer lower latency (e.g. they do not require access to external memory), they restrict the flexibility as the "walk path" from one ME to the next cannot be re-wired. Using NN registers is thus interesting mainly when there is a single preferred path between processing elements.

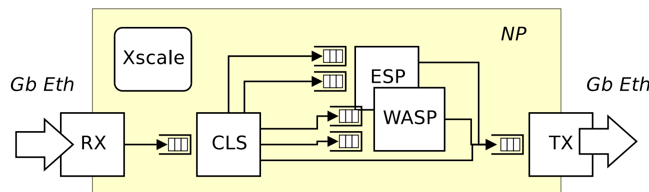



**Figure 5.2**: *Internal structure of our WASP/ESP packet filter. We actually allocated four MEs for WASP/ESP processing and have one "spare" ME.*

on one side and a CSIX-compatible switch fabric on the other side. As it is usual for such setup, the XScale processor is typically involved only with "exception" packets such as ARP request/replies or control protocols (e.g. packets directly addressed to the router itself) – i.e. what we're used to call the "slow path" in router design.

For our proof-of-concept implementation, we placed our WASP microblock in a much simpler (and cheaper) test bed depicted on Fig. 5.2. In our case, there is virtually no slow path processing as we limited ourself to layer 2 forwarding. The XScale's role is here limited to microcode loading, monitoring and debugging. These three tasks are fulfilled by a Linux application that controls the microengines *via* the Hardware Abstraction Library provided by Intel. In addition to basic counters reporting (already available with the ESP package), we extended that control application with an "online" debugging mode allowing inspection of virtually all ME resources (general-purpose registers, local memory, transfer buffers, etc.) and application-specific registers monitoring, as well as real-time ephemeral store dumping.

## 5.1.2 Parallel Programming on the Microengines

Multi-threading on the microengine differs from typical multi-threading on a time-shared (preempted) system. Here, context switches only occur when the running context explicitly releases the processor with e.g. a `ctx_arb` microword. The hardware context arbiter will then evaluate the other threads in a round-robin fashion and check their events status registers to identify a thread that will receive the processor.

Since we have a massively parallel architecture for processing WASP packets, it is important to ensure that computations will still be carried out properly. We must for instance ensure that two `count` packets or one `count` and the subsequent `collect` packet update the counter entry in a coherent fashion.

Still, it wouldn't be practical to lock individual ESS entries due to the absence of atomic operations with the DRAM controller. It could also be extremely inefficient as a WASP interpreter thread cannot stop interpreting a packet (that e.g. should wait for an entry to be updated) and continue interpretation with another packet.

Instead, we impose (as in [Imam03]) that packets that operate on the same entry use the same *Computation ID* (*CID*). The computation ID acts as a "flow identifier" for the classifier, which will allow for parallelism by multiplying the number of queues towards the processing ME and which will guarantee ordering of packets carrying the same *CID* by placing them in the same queue.

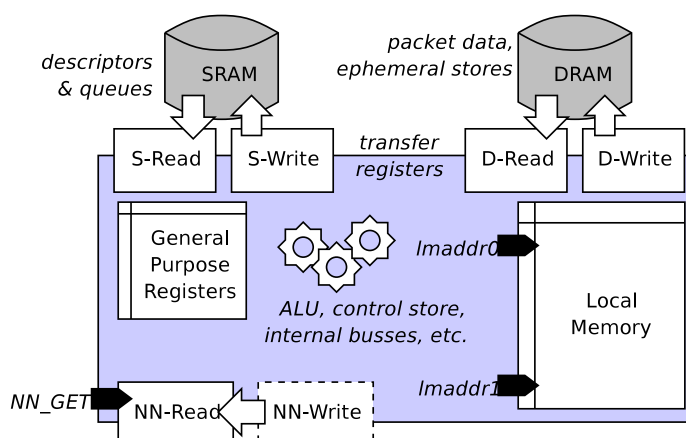



**Figure 5.3**: *The microengine resources as seen by a thread in the WASP microblock*

Each thread in the WASP microblock can pick packets from any queue, but the queue will then remain "locked" to other threads until the packet is completely processed and ESS entries are updated. This means that we can now safely process `count` and `collect` packets operating on the same entry even in the absence of per-entry lock, but this implies an upper bound on the throughput we can achieve for a single computation.

## 5.2 The WASP microblock

Each thread in the WASP microblock performs the following three tasks every time a packet is received on the scratch ring:

1. fetch WASP code and data from DRAM and check their integrity,
2. interpret the bytecode until a terminating opcode is encountered,
3. write back modified parts (packet banks as well as ESS entries) to the DRAM, and compute the new CRC checksum if needed.

### 5.2.1 Structure Placement

An important point in the VPU design is how we map the logical VPU structures (e.g. stack and bytecode) onto the various storage facilities of the microengine. Considering that accessing external memory is orders of magnitude longer than reading a local register, it is clear that we have to restrict ourselves to the microengine-local resources for storing VPU state and required packets parts.

Even inside the microengine, we have an impressive number of storage facilities that all have their properties that make them more or less suited to different content and access patterns. The SRAM and DRAM transfer registers, for instance, use separate banks for reads and writes. It is therefore not possible to use them as temporary storage since we will be unable to read back what we have written there. At best, they might be used to

| | size | indexed | inc/dec | offset | read back |
|---|---|---|---|---|---|
| GPR | 128B | no | no | no | yes |
| SRAM xfer | 64B | shared | ++/– | no | no |
| DRAM xfer | 64B | shared | ++/– | no | no |
| NN regs. | 64B | per-me | ++ | yes | yes |
| local mem | 320B | per-ctx | ++/– | yes | yes |

**Table 5.1**: properties of the memory element on the IXP2400, along with the size in bytes per context

keep interface configuration or statistics. Hopefully enough, much details of the actual microengine structure can be ignored and the programmer can work with a simpler model depicted on Fig. 5.3.

For most structures in the VPU, we will prefer storage with indexed access and index auto-increment after a memory move would be a plus. This will indeed be a plus for data banks (for implementing `INX` microbyte) as well as for the stack or sequential code progression. Table 5.1 summarises the properties of the different memory areas within the microengine. We consider here that the microengine's "next neighbour"(NN) registers have been set up so that the ME accesses its *own* NN rather than another ME's. It appears that local memory will be one of the most interesting locations, among other things because it has two independent index registers that are automatically saved on context switch.

Comparatively, we have only one `NN_INDEX` register for the whole microengine, and one `T_INDEX` register that is shared for SRAM transfer and DRAM transfer registers for all contexts of the microengines. In other words, if we want to preserve those index content, we will be required to identify all instructions that could cause a context switch while they are alive and save the index value in a GPR.

**Bytecode**

We decided to use the NN registers to store up to 64 bytes of code. The additional latency for reading back values written in NN registers is not really an issue since code is only written once (in packet fetch phase), and the only place where we need extra attention will be ESS access instruction (which may release the hardware during DRAM transfers).

Comparatively, it would be more complicated to place code in, e.g., SRAM transfer registers, since reads and writes are handled by different physical registers. It is clear that writing packet bytecode to SRAM and getting it back in the transfer registers would be awfully inefficient. The DRAM transfer registers could sound more appropriate, as they do contain the code at least during packet fetch phase. Unfortunately, we don't have enough room in the 64 bytes per context to store a useful load of bytecode *and* still keep enough room to retrieve ESS entries during packet execution.

**VPU state**

The program counter, virtual processor flags and accumulator, as well as other variables that constitute the VPU state do not need indexed access at all. They can thus safely be held in general-purpose registers. The notable exception is the management of *memory banks* X and Y. We have so far the need for 4 registers of meta-data per bank (including 2 registers for caching the current 'line' of 64 bits within the bank), and implementation of load/store operations is simplified if we can abstract whether we operate on X or Y bank. A proper design of local memory allows us to implement this elegantly: a simple increase of the index will then "shift" to the alternate 'Y' bank and decreasing the index restore the use of the "main" bank.

The second local memory index is dedicated to the VPU stack. Still, we may temporarily need it for other purposes (like advancing to the next 'line' in a memory bank) in some instructions and we will have to save/restore it manually in these instructions. We believe that it is a necessary trade-off to take advantage of the extra space available in the local memory to keep full (WASP) packet data on the microengine during interpretation.

### 5.2.2 Redesigning the Fetch/Decode

After external memory accesses, conditional branches are one of the most costly operation on the IXP microengines. While almost every instruction will complete in one cycle, a branch requires discarding the instructions that are already in the pipeline. As in many RISC-inspired architectures, the IXP micro-assembler allows one to indicate the CPU whether (and how many of) the instructions laying after a branch opcode should be executed *even when the branch is taken*. Such instructions are said *deferred* and are one of the keys for microcode optimisation.

Deferring instructions is convenient when you have many independent things to do in a code block and that only a few things among them depend on a simple condition (e.g. a bit set or cleared), but according to our experience, they are almost impossible to fill when it comes to implement a *decision tree*, especially when the next test to perform depends on the result of the current test. Bad news is that the instruction decoding in our C reference implementation is very close to a decision tree, especially since it first decodes instruction families and then opcode within a family.

In several cases, it will be preferable to use a *jump table* rather than a decision tree. The microengine instruction set indeed allows one to jump at a computed location[2]. There are mainly three ways we can use that `jump` microinstruction:

1. A pure "trampoline" table, where each slot has only a single (non-conditional) branch to a block of code that resides anywhere in the program. This will however lead to poor performance as we will experience a double jump latency.
2. A pure "switch" code block where each slot takes $n$ instructions and can either continue to the next slot or branch to some "all done" label. While this may be preferable for performance, it requires a perfect sizing of each slot to work properly, which makes it very inflexible to changes.

[2] in the form base+offset, where the offset is typically computed first in a register

3. A hybrid switch/trampoline where you give enough room to perform initialisation steps in the defer slots of the impending branch to a larger chunk of code (if needed). It can be especially interesting when many cases merge into a single code block after case-dependent initialisation is performed.

Reducing the amount of cycles spent in fetch/decode part is crucial to have an interpretor that can compete with ESP native operations, but as usual, it led to many trade-offs, be it on code readability or ease of debugging (the encoding of PC location, for instance, would sound totally awkward to an external developer).

More annoyingly, it questioned some of our initial opcode encoding schemes, requiring a change in instructions mapping or in modifiers semantics[3]. We have good hope that these changes were needed to have an instruction set that is easy to implement on *any* hardware platform, but it leaves a bitter taste anyway.

## 5.3 WASP processing delay

### 5.3.1 Profiling the WASP microblock

The IXP microengine features a high-precision timestamp counter that is incremented every 16 clock cycles (with microengine clocked at 600MHz[4]), which gives us a time granularity of 26.66 $ns$.

With that timestamp counter, we have measured the processing time of both WASP and ESP microblocks from the reception of the packet descriptor from the classifier to its dispatching on the transmission scratch ring. We expect that the time we measure that way is a good estimate of the overhead that WASP or ESP packets experience compared to a regular packet of identical size.

The processing time of the last packet is stored in a GPR on the microengine and periodically retrieved by the code on the XScale. There is still work to do to automate aggregation of those measurements, but as a first estimation, we have an average processing time of 2.93 $\mu s$ for a `WASP:count` packet with a deviation of 0.204 $\mu s$ among 42 sampled timings. We should keep in mind here that a DRAM access implies a non-deterministic delay estimated between 200 and 300ns, which explains why the relative error is so high. Comparatively, 58 `ESP:count` packets took on average 2.14 $\mu s$ with a deviation of 0.085 $\mu s$.

We can thus estimate that the time required to *process* a `WASP:count` packet is 137% of `ESP:count`, a fairly encouraging result for the IXP implementation, since under the same circumstances[5], the x86 WASP implementation took nearly twice as much as its ESP counterpart.

---

[3]e.g. the `CLR` bit in memory opcodes that tells whether the accumulator should be cleared before the lookup is done is now active when cleared rather than when set.

[4]again, according to hardware manuals [IntelHRM], table 157. Empirically confirmed by the 10 $sec$ lifetime of entries in the ESS, considering the way timestamp counter is converted to check delays.

[5]at the moment, we don't use any information cached from the previous ESS lookup when we update the entry

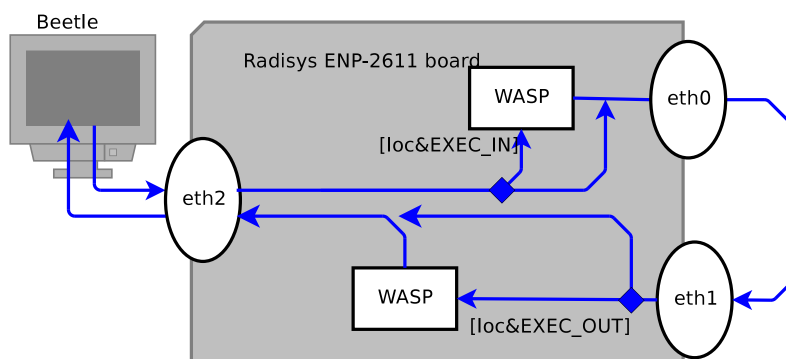



**Figure 5.4**: *Testbed setup for "There and Back" latency measurement. The control station on the left runs both (custom) traffic generation and Ethereal as a monitoring tool.*

We should keep in mind that the measurements here take actually more than just interpretation into account: packet fetch, CRC etc. are part of the WASP/ESP microblocks, while they weren't taken into account in the x86 comparison (on Fig. 4.10, section 4.4.1).

### 5.3.2 There (on the IXP) and Back

We set up a test bed to measure packet latency through the WASP/ESP filter, trying to avoid the need for precise clock synchronisation between the involved machines. We have indeed frequently observed a time offset of 10 $\mu s$ when synchronising our Linux PCs with a local NTP server. We expect to measure latencies near to 50 $\mu s$ according to our preliminary tests, and the NTP offset could introduce a significant bias in our results.

We thus opted for a setup where traffic is injected and collected by the same machine, a 1GHz Pentium III (Beetle) running Ubuntu Linux 2.6.12 and featuring a 1Gbps PCI network card[6]. This machine is connected to port 2 of the ENP-2611 board. We set up a static forwarding rule that delivers packets from port $i$ towards port $(i+1)\, modulo\, Nports$, and wired ports 0 and 1 together.

As a result, a packet emitted by Beetle will be received on port 2, forwarded on the "loop" wire, received again on port 1 and sent back to Beetle, experiencing 3 queueing and serializing over Ethernet wires and two IXP processing (see Fig. 5.4). Measurements are performed through the Ethereal application, using a script deriving from our tests on x86 Linux router. The wiring is such that the WASP box only processes packets with `EXEC_IN` bit set on port 2 (from Beetle) and packets with `EXEC_OUT` bit set on port 1 (before sending them back to Beetle). The main expected advantage from this unusual setup is that we do not have to rely on strong clock synchronisation. Considering we are measuring times near 50 $\mu s$ and that it is is not rare to see a time offset of over 10 $\mu s$ when checking synchronisation with a (local) NTP server every second, we believed it was preferable not to introduce that additional bias to our measurements.

For WASP packets, we can further control whether we want the packet processed once or twice by using the *execution location* flag. We could thus theoretically estimate

[6]Broadcom Corporation NetXtreme BCM5701

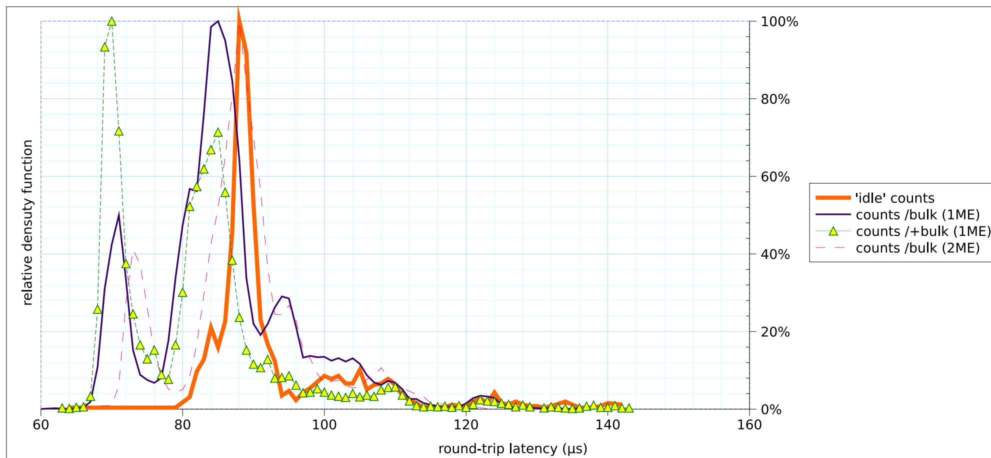



**Figure 5.5**: *Distribution of measured delays from Beetle for* `WASP:count` *packets, with 0 ('idle'), 2.7 ('/bulk') and 5.4 ('/+bulk') Mbps of background traffic. The distribution shown is relative to each experiment's most represented delay.*

the amount of time consumed on the WASP microblock through the difference between the delays of a packet that has both `EXEC_IN` and `EXEC_OUT`, and of a packet having only one of those bits set.

Typically, not all `COUNT` packets will experience the same delay. The first packet after ESS cleaning, for instance, will have to create a new entry while other packets simply update an existing entry. In order to see to what extent these different code paths lead to different delays, we extracted the distribution function of various scenarios, as reported on Fig. 5.5.

### Cache miss and other side-effects

Going further with our measurements, it appeared however that the actual average delay we measured was very sensitive to the amount of traffic crossing the router. For instance, a regular "ICMP echo" packet will take around 76 $\mu s$ to cross the IXP and come back when there is no background traffic, while as soon as we start a stream of "bulk" packets (2.7 Mbps using frames of 1066 bytes on wire), the average latency of ICMP packets drops to 64 $\mu s$.

We observed a similar effect with `WASP:GET` packets and Fig. 5.5 shows how it affects measurements of `COUNT` packets. The thick "idle" line, plotting latency distribution when we send one `COUNT` packet per second with no background traffic, has a single strong spike matching with the observed average latency (93.48 $\mu s$). This spike covers 76% of the actual packets and is centred at 88.2 $\mu s$. When we add the stream of bulk packets ("/bulk" line), the major part of the traffic is now distributed on two spikes; a small one centred on 73.74 $\mu s$ that covers 16% of the traffic, and the larger one centred at 87.61 $\mu s$ that covers 63% of the traffic.

Now, as we double the amount of bulk traffic (e.g. comparing the "/bulk" line with the "/+bulk" one), the distribution of packet latency between the two spikes changes. We now have 38% of the traffic processed in 70 $\mu s$ and only 49% of the traffic taking on average

84 $\mu s$. Nothing in our code on the IXP could explain such a behaviour, so we suspected something strange was happening with cache performance on the PC platform itself.

We thus repeated the experiment, replacing our Beetle with a dual-core 3GHz Xeon running SuSE Linux 2.6.11 in 64-bit mode and featuring an on-board Intel 82541GI/PI Gigabit Ethernet controller. It definitely confirmed that we shouldn't rely on a PC to measure network latency at 1Gbps. Indeed, on the Xeon, no spike appear in the graph and it looks like packets now experience a delay that is uniformly drawn between 30 and 150 $\mu s$. Under those circumstances, we have no hope to accurately measure anything that happens on the microengines, so we opted for modifying receive/transmit microblocks of the IXP application to make them measure directly the packet latency.

While we haven't investigated further what caused the curious phenomenon detailed in this section, we believe that the behaviour observed with Beetle could be explained by the way network drivers are typically implemented in Linux: when a packet is first received, the network card issues an interrupt that will eventually make the CPU execute driver code. On most drivers, however, the driver will not just pick one packet, but it will greedily consume packets on the card until buffers are empty (or a given time quota has been reached). The kernel firewall behave similarly, running in a "tasklet" (also known as a "bottom half") that acts as a dedicated thread inside the kernel. Code and data locality is thus higher when we process one packet within a burst than when a single packet is processed by network driver, then by firewall, etc.

The Intel 82541GI network card additionally implements interrupts moderation[7] to avoid excessive CPU time consumption by frequent switches between user and kernel mode, at the cost of more variable in-card delays. Obtaining accurate measurements with such hardware might require us to disable interrupt delaying in the card, and might also have to hack the kernel to ensure timestamps for packet reception/transmission are performed as close as possible from the actual I/O operations.

### 5.3.3 Embedding Measurements on Microengines

#### Methodology

Our implementation of the WASP filter reuses the generic "receive" (RX) and "transmit" (TX) microblocks provided by Intel as part of the IXA SDK. Transfers with the Media Switch Frame (MSF) is typically something that requires perfect knowledge of the IXP hardware and extremely precise synchronisation between threads to sustain the wire speed[8] – two objectives that are rather incompatible with code readability, despite the extensive commenting of the code package. We have thus long kept modification of those blocks as a last resort until it became obvious that PC hardware couldn't fulfil the task.

Packets received by the MAC cards are delivered in chunks of 128 bytes called "mpackets" to the microengines. Threads on the RX microblock pick those mpackets as they arrive and attempt to rebuild the packet they come from using state associated with each

[7] see application notes [IntelAP450, IntelAP453] for details on interrupt moderation and small packet traffic performance optimisation

[8] e.g. some operations such as obtaining the buffer for the next packet are performed ahead of time while we increment counters for the current packet and thus replicated with every 'receive case'

media port in the local memory. When the packet is completed, its buffer handle is transmitted to the classifier block.

We modified the RX block to capture the local time counter immediately after the MSF wakes the thread[9]. In case we identify the reception of a "Start of Packet" (SOP) mpacket, that timestamp will be stored along with reassembly state in the local memory. In all other cases, the captured timestamp is discarded. When an "End-of-Packet" (EOP) mpacket is detected, the timestamp associated with the corresponding SOP is added to the packet descriptor in SRAM that will be available for later processing blocks.

The timestamp is then kept untouched by classifier and WASP/ESP microblocks. It will be retrieved with other buffer metadata by the TX microblock. When the last mpacket has been transferred back from DRAM to the buffers of the transmitting interface and we have acknowledged the MSF[10], a second local counter capture takes place. The difference between this timestamp and the one we retrieved with metadata gives us the *software* processing time of the packet on the platform: we might miss a constant and payload-independent additional response delay from the MAC units and the MSF[11].

To avoid extra interference, microengines simply keep the last measured delay in a register that we periodically sample from the XScale. This technique gives us more precise delay measurement, but we lose the ability of isolating traffic classes as we did with Ethereal in previous tests.

Still, We have a different counter for packets that are exactly one *mpacket* (128 bytes and less) and packets that are 2 *mpackets* or more. We can thus isolate some ESP/WASP operations by artificially "inflating" some packets types (e.g., having `count` packets sized up to 400 bytes to keep them apart of the `get` or `collect` packets that we plan to measure).

**Results**

In the conditions described above, our WASP filter forwards `WASP:count` packets with an average latency of 6.342 $\mu s$. Dumb packets of identical size have an average latency of 2.451 $\mu s$ and `WASP:count` packets that are not processed (i.e. that have all their execution location bits cleared) take on average $2.908\,\mu s$. The same experience repeated with `ESP:count` packets showed an average latency of $5.470\,\mu s$.

In other words, WASP processing is 47% of the complete latency for a `count` packet, and the interpreted version takes 116% of the native version time. The results for `collect`, on the other side, are less impressive: 136% of the native version. With $8.581\,\mu s$ for WASP against $6.295\,\mu s$ for ESP, we clearly see here the impact of a longer WASP program.

[9]threads that are willing to process incoming mpackets add their identification, together with the address of a transfer register where the status word will be written, to the "freelist" maintained by the MSF and are automatically activated when the mpacket is ready in the transfer buffers

[10]that is, at the start of phase 3, where debugging counters are incremented, etc. and before the buffer is returned to the free pool

[11]by manual inspection of the microengine state, we have estimated this additional delay between 3 and 6 $\mu s$, but such measurements are hard to reproduce and can only be obtained with sufficiently large packets with the current code.

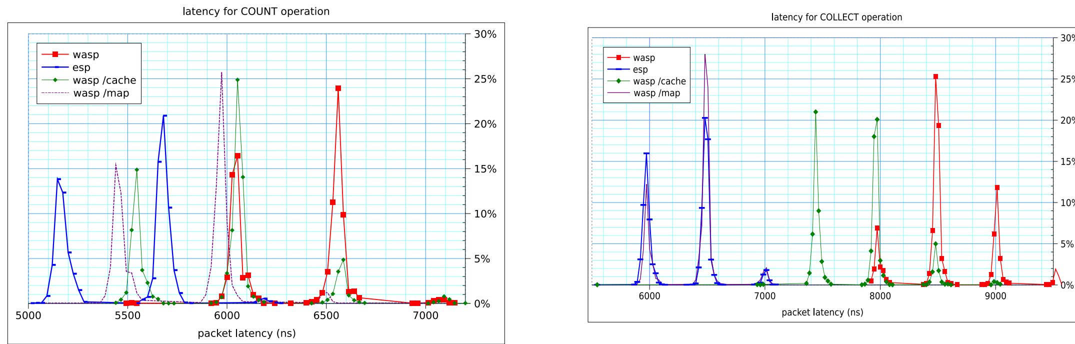



**Figure 5.6**: *Distribution of packet forwarding latency at low throughput for* `count` *and* `collect` *instructions*

We then implemented caching of ESS pointers between `lookup` and subsequent `insert` on the same key the same way we did in the x86. We can then observe an improvement of $0.37\,\mu s$ for `count` (one SRAM and one DRAM access saved when writing back the counter), and $0.78\,\mu s$ for `collect` (two SRAM+DRAM accesses saved). In the IXP implementation, caching the last ESS pointer comes virtually for free[12]), but the effect remains modest (10 and 13% of the interpretation time respectively) compared to the 23% improvement observed with x86 processors (see Sec. 4.1).

Comparatively, the IPv4 forwarder demonstration application[NPForum03], takes on average 6.92 $\mu s$ to forward a 64-byte packet under 25% throughput (see Fig. 5.7). This demonstration application from Intel involves looking up the destination's IP address and resolve the appropriate MAC address for the next hop, while our application is a simple filter that doesn't alter any of L2 or L3 headers.

As shown on Fig. 5.6, we observe again that the latency distribution is mostly split in two (or sometimes three) spikes. We can notice too that 95% of the samples are located no further than $52\,ns$ from the spikes centres. We can likely consider that this corresponds to variance in latency of DRAM accesses[13].

A most intriguing fact is that the spacing between the spikes' top is on average $516\,ns$ with very small deviation [14] independently of the actual packet's program and regardless of whether ESP or WASP is considered. Our first hypothesis was that some collisions in the ESS hash table occurred for some of the lookups, leading to extra DRAM access, but probing the distribution of chain lengths during the test revealed that all chains were of one single element. We also observed that the time between packet reception and packet drop (for dropped `WASP:count`) does not exhibit such a "slotted" distribution, but rather has a continuous, Gaussian-like shape (avg. $4.35\,\mu s$, stdev $1.41\,\mu s$).

On the other side, the delay of $516\,ns$ on average almost exactly correspond to the time required to transmit a minimal-sized packet (64 bytes) on the 1Gbps medium. While

[12]basically, they can remain in transfer registers if we ensure that other operation do not trash the retrieved ESS entry

[13]remind that while [Johnson03] reports a latency of 120 cycles (200ns) for a DRAM access, other sources[Lu05] report from 200 to 300ns, which would confirm the $52\,ns$ variance observed here

[14]Actually, we can only observe either 507 or $533\,ns$ due to the $26.6\,ns$ granularity of our measurements

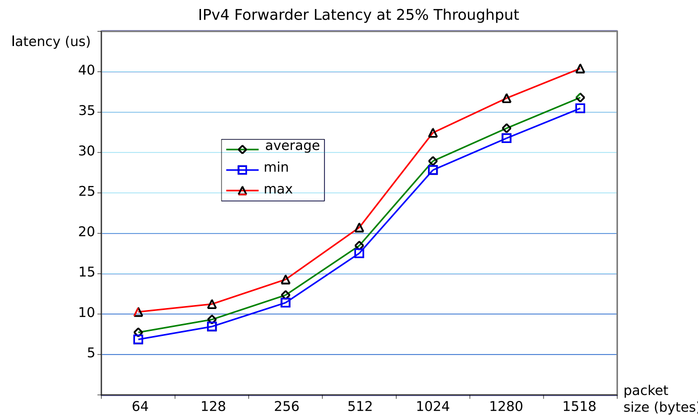



**Figure 5.7**: *IPv4 forwarding latency, according [NPForum03]*

we haven't investigated this any further[15] at the moment of writing, it sounds reasonable to consider that the spacing between spikes is introduced by the transmit component depending on whether it is found idle or busy when our transmit request is submitted.

### 5.3.4 Larger Entries and `map` Opcode

The timings for `collect` in the previous session are based on bytecode that exclusively uses `lookup` and `insert` microbytes. We have already shown in the case of x86 implementation that it is more interesting for performance purpose to replace these multiple accesses by a single `map` opcode [Martin05b].

On the IXP, saving space in the ephemeral store is of increased importance if we want to support 1Gbps, so we modified the lookup/insert code as well as store cleanup routine to accommodate *both* 24-byte (single 64-bit value) and 48-byte (mapped bank) entries. Since each ESS entry is aligned on a 64-bit boundary, we can use the 3 lowest bits of "chain" pointers to store extra information about the entry itself, and one of them has been used to tell large entries apart from small entries.

Supporting `map` opcode also required a couple of modifications in `store` implementation so that we identify correctly the following cases when packet interpretation ends:

- the packet data is dirty (e.g. we used `store` on a packet data bank): we need to write those data back in DRAM and compute a new CRC.
- a mapped ESS entry has been updated during interpretation and is still waiting to be written back in DRAM.
- a mapped ESS entry has been created during interpretation and should be written back.

[15] further investigations would e.g. require us to guess the internal state of the transmit logic and e.g. save measured latency to `delay_ready` or `delay_busy` accordingly. However, according to [Johnson03] (p. 123), there isn't anything like a "ready to transmit" bit in the IXP2xxx MSF, and therefore no way for us to further confirm our hypothesis.

We should remind that in the case of x86 implementation, using map (compared to a small cache implementation) has virtually no impact on count instruction, but improves latency of collect by 18%. Still, interpretation time takes respectively 166% and 133% of their native ESP counterpart.

Figure 5.6 shows that there is only a small improvement (merely 1%) in the case of count. Comparing packet latency measurement against the cost of forwarding a WASP:count packet that does not require execution, we can estimate the average interpretation time of count /map as $2.866\,\mu s$ while count /cache takes $3.062\,\mu s$, giving a 6% improvement[16]. In both case, these measurements only take into account count packets that have to create the ESS entry.

The most interesting result is however the fact that WASP:collect /map achieved a forwarding latency almost equal (101%) of native ESP counterpart. Indeed, an ESP:-collect packet needs to lookup and update two keys, requiring 4 DRAM access at best[17], and 3 SRAM accesses in the case where the second entry needs to be created. Comparatively, both the packet counter and the computation accumulator are held in a single entry when using map, and collect suffers no entry creation penalty.

Finally, only 1 SRAM and 2 DRAM access are now needed, which could save between $680$ and $880\,ns$ in memory accesses, and the resulting bytecode is simpler, which finally make us save $1416\,ns$ against WASP:collect /cache. While they achieve similar latency at low load, it should be mentioned that we expect processing of an ESP packet to be essentially *memory-intensive* while WASP packets are *cpu-intensive*. It will thus be important to repeat those tests at higher loads to ensure we are not saturating microengines with computation to the point it is too busy to fully benefit of memory waits by other threads.

### Atomicity and Aborted Mappings

Another advantage of using map is that we can more easily implement atomic operations (see section 4.4.3). In case of interpretation error (stack overflow, etc.) or if the abort opcode is executed, the packet interpretation is halted *without writing back the mapped bank in the ESS*. What remains in the state store is thus the (consistent) state found before starting packet execution.

However, we need to take extra care when aborting a packet *that just created a new entry* via the map opcode. When a packet creates a new entry, we give the interpreter a zeroed bank and the “defined” bit in its status is cleared. The new entry is then chained but its state will only be defined for the first time when packet execution completes (or when another map will be issued by the same interpreter). It may be important for some protocol to distinguish a zeroed entry that was already present from a new entry creation.

- Once an entry has been allocated, it is generally not possible to “free” it before it is reclaimed by the store cleaner. As a result, entry allocation and chain linking *must* be kept together if we want to avoid wasting entries.

[16] *interpreting* WASP:count /map thus takes 111% of *processing* ESP:count, while *latency* of WASP:count /map takes 106% of ESP:count.

[17] assuming that there is no chain to walk

- Modifying the chain separately from looking for the entry (e.g. at packet forwarding) will require to check the DRAM (or SRAM) again to ensure we still know the end of the chain.

The approach we envision is to let map create the new entry anyway, but to tag it as "invalid" (using the extra status bits mentioned above) until it has been validated by a write-back. This way, the chain for $hash(k)$ is immediately valid and we can use a single-shot write-back in DRAM, and still have WASP programs detect appropriately the absence of a valid state if the creator has been aborted. In that case, one "dangling" entry for $k$ remains allocated in the store, and another packet that invokes $k$ can still use it (and define it), but the entry's lifetime will be defined by the first packet. This approach will be completed by a map | die modifier which instructs the map opcode that packet execution should be immediately aborted if the entry does not exist. This eliminates the need for constructs such as "map, then branch to abort if not defined" in bytecode that would have inevitably led to dangling entries.

## 5.4 High Availability at Higher Rates

As we have seen in section 4.3, it is possible, due to the nature of the Ephemeral State Store, to engineer a ESP component so that we can guarantee proper execution of any program simply by enforcing a limit on *how many keys* a packet can create. We have also seen, unfortunately, that 681 MB of DRAM is required per Gigabit Ethernet port – almost three times the amount of memory available on our ENP-2611 card.

In this section, we will study how the ESP microblock and our WASP microblock behave when approaching these levels of performance. Note that the nature of the traffic we send prevents us from using an existing traffic generator such as D-ITG[Avallone04] or thrulay[Shalunov05w], for instance.

### 5.4.1 Behaviour of the ESP microblock

#### Test Setup

For these tests, the ENP-2611 card has been configured as having a single *port pair* between the two machines *Wasp* and *Beetle*. Any packet received on one port is forwarded on the other, and vice versa, as one could expect from a regular switch.

The ESP and WASP microengines are both given 32 queues to receive packets from the classifier, of 2048 entries each[18]. We have 4 ephemeral store of 8MB DRAM each, with a hash table of 64KB.

The test traffic consists of pairing count and collect packets (e.g. every collect packet reuses a key that has been installed by a previous count packet, with sufficient

[18]these multiple queues allow parallel processing of WASP programs while enforcing ordering preservation between packets that belong to a same computation

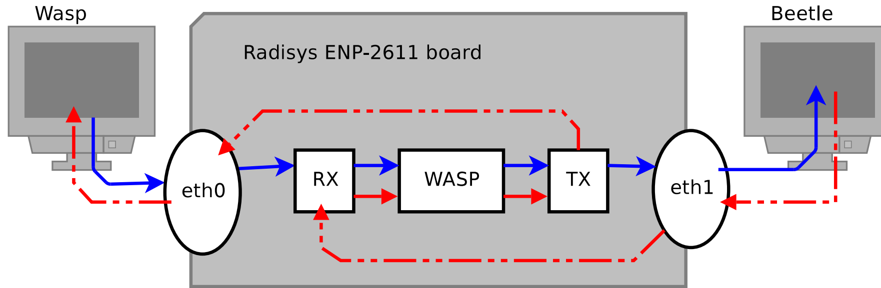



**Figure 5.8**: *The setup used for the state-store stressing testbed. Note that only a few tests will involve traffic on the Beetle-to-Wasp path.*

spacing[19] between corresponding `count` and `collect`. We used the x86 internal timestamp counter[IntelRdtsc, Bindels] to enforce a minimum delay between two subsequent `send` system calls.

The traffic generation tool we use here can adjust the number of keys used by the test traffic in order to accommodate a given size of ephemeral store. In this scenario, we would like to avoid exaggerated variation in chain length while avoiding the need for a complex function generating random keys[20]. The technique we used was to reuse some bits of the CRC that protects the ESP PDU $i$ as the lowest bits of the key for PDU $i + 1$. Every $N$ packets, we simply restart with our first key, where

$$N = \max_i \{x = 2^i : 48x < |ESS|\}$$

### Forwarding Latency with Short Chains

The maximum capacity of the generator on *Wasp* (dual-core Xeon, 3Ghz) was 907 Mbps with all packets having the maximum size and 170 kpps (98 Mbps) when using only minimum-size packets. This is still quite below the theoretical maximum throughput of a single port-pair Gigabit Ethernet Forwarder where each interface can receive 1.488 Mpps [21], but it already gave us the opportunity of stressing the state store.

We first placed ourselves in an optimistic scenario (on average 7 entries in a ESS chain) where the generated traffic will only require from 1 to 6.4 MB of state store and using the full processing capabilities of the chip (16 threads doing ESP packets processing). The latencies and chain length distribution in the resulting scenarios are depicted on Fig. 5.9.

Note that the packets whose size are mentioned on Fig. 5.9 are the `collect` packets. All the `count` packets were 128 bytes large in this test.

In all cases, the percentage of packets effectively forwarded by the ENP card was at least 98.5%. We have tuned the “threshold” of `count` packets so that none should be dropped as a result of normal processing, but if a `collect` packet is processed with a

[19] `sendespmorecc.c` will send 1024 `count` packets before the first `collect` is issued

[20] The ESP implementation uses hardware-accelerated `hash_64` and keeps appropriate lowest bits depending on the table size

[21] if we take inter-packet gaps (12 bytes) and preamble (8 bytes) into account and assume minimum-sized packets (64 bytes)

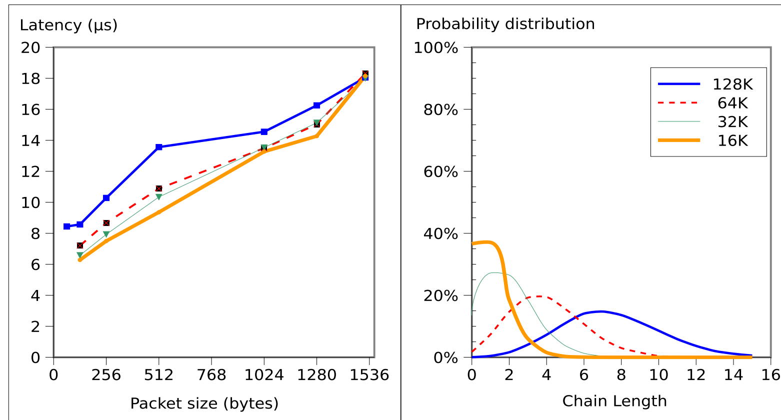



**Figure 5.9**: *(left) `ESP:collect` packet latency with varying packet sizes and (right) ESS chain length distribution for various state store usage ranging from 16384 to 131072 unique keys*

`value` entry that is older than the `count`[22], we might experience a drop anyway. We estimate that this occurs for 79% of the packets actively dropped by the ENP card – which still represents less than 0.8% of the received traffic. The remaining dropped packets are due either to lack of space in the ESS (16%) or a missing `count` tag when a `collect` packet is processed (4%).

The most likely source of packets loss in our tested is thus the network card on the receiving PC (1GHz Pentium III) which may not be able to process all those small packets (or the Linux kernel lacking buffers to process them all).

### Saturating the State Store

In the following tests used a packet flow mixing `collect` packets sized up to 438 bytes (including Ethernet headers) and `count` packets of 78 bytes, which corresponds to 253 kpps (17% of the maximum throughput per interface).

We then selected the number of different keys generated to saturate the ephemeral state store (in its current state, it can handle 34950 new keys per second) in order to have most of the chains longer than 10 entries. The test has been repeated by limiting further and further the number of different *computation IDs* in use, thereby limiting the number of queues used to relay ESP packets between the classifier and the processing microblocks, but also the level of parallelism in the processing microblock.

With two threads (on the same microengine) and 256K entries ($2q256K$ traffic pattern[23]), we start seeing forwarding latencies approaching several milliseconds, and the number of packets dropped by the ENP card dramatically increases.

What actually occurred is that, given the restricted size of the *hash table* of our ephemeral store (16K entries for a 8MB store), the average chain length grew up to 16 ESS entries for the same hash, a threshold at which the processing time of an individual `find_create` was exceeding the inter-packet time for our test traffic (at 211 kpps, we receive a new packet every 4 $\mu s$).

[22] meaning that `count` tag has already been collected, but `value` hasn't been so far

[23] which correspond to the setting -storesize 8000000 and a key mask of `0x1ffff`

We automated the process described in section 5.3.1, giving a more precise gathering of the number of cycles spent *in the ESP microblock*. With the $2q256K$ pattern, we have between a mean processing time of 6.75 $\mu s$ with a deviation approaching 3 $\mu s$, against 2.14 $\mu s$ on average per packet when the system is near idle state.

With 4 threads, the average time spent on a packet in the ESP microblock is 8.40 $\mu s$ (with a deviation of 4 $\mu s$), as the DRAM bus is now facing twice the amount of requests. However, the rate at which our 4 threads consume packets from the ring buffer is sufficient compared to the offered traffic.

Reducing the number of entries used in the state store to 64K also solves the problem. Due to the restricted size of the ephemeral store, however, we are not capable of reproducing the same experiment with 4 running contexts (the store then saturates and we cannot increase the chains length further). By submitting traffic at both ends of the ENP-2611, we can still increase the load to a total of 393 kpps (2.54 $\mu s$ inter-packet arrival time) and put the ESP microblock to a similar state where queues stay almost full for a few seconds before getting empty again.

### Experimental Results vs. Queueing Theory

With an average processing time of 6.6 $\mu s$ ($2q256K$), two "servers" (the individual worker threads) and inter-packet arrival time of 4 $\mu s$, the system should in theory be able to drain buffers to a normal state. However, we observed that queues are heavily loaded most of the time.

We should first note that what we measured isn't exactly the inter-packet departure time of the queues. the processing time doesn't take into account the time needed by a thread to acquire a packet descriptor on the SRAM queue. Since all the threads of a microengine poll the 16 queues in a round-robin fashion, we might be missing a delay of several SRAM accesses (150 $ns$ each) before one of the thread probes one of the two queues in use.

Moreover, the traffic pattern isn't as regular as we could hope. Because we first send a burst of 1024 `count` packets (which are smaller), we have a temporary burst rate approaching 274 kpps (3.64 $\mu s$ inter-packet time), and exactly enough buffers to accommodate that burst when only 2 queues are in use.

We also don't have precise measurements of the load the ephemeral store cleaning might add on the DRAM and SRAM controllers. Cleaning indeed happens "between" packet servicing, and simply means the system will temporarily serve queues with 7 threads rather than 8 – which should be unnoticeable when only 2 or 4 queues are loaded. Cleaning, however, will issue continuous requests to the DRAM controller to check each entry's expiration time and SRAM requests to update the chain pointers in the hash table, which might temporarily increase the processing time and fill up buffers.

Finally, a cleaning burst is likely to be followed by a collection of entries creation, which is also significantly more expensive than entry lookup, which will delay buffer drain.

Further significant modifications of our statistic grabbing tool would be required to investigate those kinds of behaviours, collecting queue state, traffic patterns, servicing time in a single experiment.

### 5.4.2 Behaviour of the WASP Microblock

Now that we have a clearer understanding of the dynamics of ESP and the state store, we can compare latency of a flow of ESP count/collect packets against an equivalent WASP flow. We picked generator parameters so that the chain length distribution in the state store and the traffic rate in terms of packet per second is the same in both scenarios.

It should be mentioned that in this type of scenario, it makes little sense (if any) to study the latency of `count` and `collect` packets separately. Indeed, both will be received in the same queue and may access ESS entries that are in same chains. The effect of e.g. a longer average processing time for `collect` packets will increase the average queueing time (and therefore the latency) of `count` packets.

To keep things comparable, an ESP flow assuming a store of size $S$ will typically compete with a WASP flow assuming $2S$[24]. The implementation of `WASP:collect` we're considering here is indeed using `MAP` and needs only one entry per "computation", therefore requiring us to double the number of individual computations if we want to keep the same chain length distribution.

#### Store Cleaning Effects

As in the case of ESP processing, we first studied the behaviour of WASP for critical chain length. Theoretically, at a certain length, the processing time will be such that microengines cannot sustain the offered traffic. Buffers between the classifier and the processing microengines will then fill up permanently and excessive traffic will be dropped.

In this section, we mainly focused on a test bed with 2 worker threads (thus only two distinct CIDs) on the same microengine and a virtual store size of 128K entries ($2q128K$) for ESP and WASP($w2q128K$) (giving an average chain length of 8 entries and a maximum chain length over 15 entries, see Fig. 5.9).

We indeed observed such behaviours when offering a combined load (from *Wasp* and *Beetle*) of 440 kpps with latencies then approaching $3\,ms$, but as soon as the load gets over 250 kpps, we can see transient states with high buffers usage that drain at a speed depending on the offered load. The periodicity of those buffer-filling states is close enough to the ESS entry lifetime to suggest that we are observing the result of a state store cleaning. Unfortunately, the mechanism we use for probing buffer usage is too intrusive to allow more than one probe per second – a rate at which providing quantitative result, but since we observe less than 2 seconds with empty buffers between two bursts at 434 kpps, we can reasonably suggest that the "knee" load is somewhere between 434 and 440 kpps.

It should be noted that our traffic unfortunately starts using virtually all the keys of the store at the same time, meaning that store cleaning happens in "bursts" during which

[24]e.g. $2q128K$ is achieved through `-storesize 4000000` while $w2q128K$ requires `-storesize 8000000`

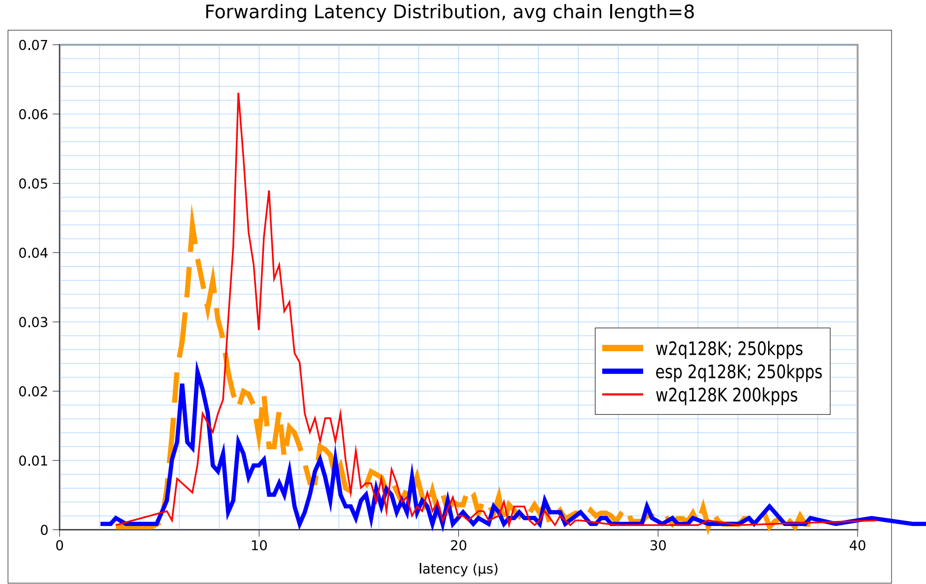



**Figure 5.10**: *Evolution of the latency distribution as offered load approaches saturation*

the processing time in the WASP microblock can be up to twice the average value. Be it because of an increased load on the DRAM controller, the need for creating new entries or because chains are found locked by worker threads, the result is that the now-longer WASP processing struggles the consumption of WASP packet from the classifier's buffers that will fill up. The time needed to drain buffers will then depend on the difference between the offered load and the current throughput of the WASP workers, which has been measured on average at 400 kpps in the setup, but showing variations from 380 to 440 kpps.

Measuring the packet latency through the system with load over 250 kpps gives virtually no clue on the behaviour of the processing microblock: as illustrated by Fig. 5.10, the latency is then mostly dominated by the queueing delay and almost fits a exponential distribution. Moreover, the smallest deviation in the generated load may translate into an effect on the packet latency that is close to the observed difference between WASP and ESP. We have e.g. measured an average latency of $14.08\,\mu s$ for WASP packets at 266 kpps against $15.36\,\mu s$ for ESP.

We thus tried to study the average processing time $\overline{T_P}$ on WASP and ESP microblocks, but the best we can say is that it fluctuates between $3.8$ and $4.4\,\mu s$ without showing any clear relationship between the load and $\overline{T_P}$, nor any clear way to tell which of ESP or WASP performs better. At 418 kpps, for instance, $\overline{T_P}$ for WASP is $3.85\,\mu s$ and $4.12\,\mu s$ for ESP, but WASP $T_P$ increases up to $4.4\,\mu s$ as the load is increased to 470 kpps while we observed the minimum value of $\overline{T_P} = 3.88\,\mu s$ at 465 kpps for ESP.

### Latency at Non-Critical Loads

Since we know the buffering effect becomes sensible mainly at loads of 250 kpps and over, we ran a set of experiments to collect packet latency between 220 and 240 kpps.

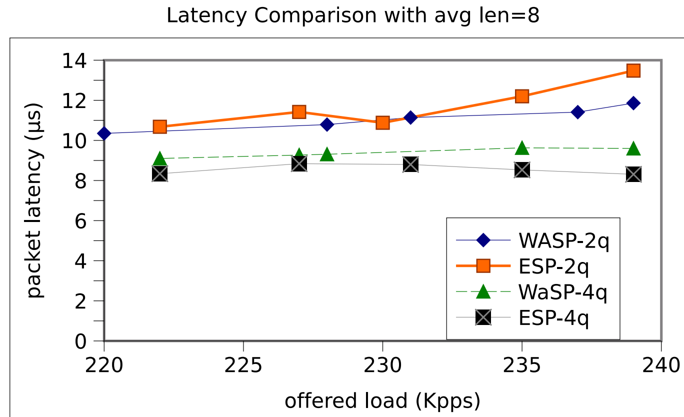



**Figure 5.11**: *Average latency*

This range has been chosen to compensate the difficulty to precisely obtain the same load with the two different generators. All the observed latency values[25] were below $40\,\mu s$ this time, which confirms store cleaning has a much moderated effect on our measurements. Still using the same traffic patterns $2q128K$ and $w2q128K$, we can see on Fig. 5.11 a small advantage for WASP and a smooth increase of the latency with the offered load.

We then repeated the experiment with 4 worker threads, illustrating how ESP better benefits of additional parallelism. This time indeed, WASP performance is slightly lower than ESP, but more intriguing, ESP latency tends to *decrease* as the load is increased. For the comparison, we measured latency at 200 kpps with 4 workers, giving very close latencies of $8.4$ and $8.66\,\mu s$ for ESP and WASP, respectively.

# 5.5 Throughput Tests

## 5.5.1 Methodology

As observed in section 5.4.1, none of the machines in our lab are powerful enough to stress a single Gigabit Ethernet wire in terms of packet-per-second.

We could have used a cluster of emitters and receivers at lower rates, using switches to aggregate traffic between clusters and the ENP-2611. Yet, our traffic-generation tools aren't designed to cooperate with other generators, potentially leading to duplicate entries creation or other sorts of ESS oddities. Moreover, not all 1000baseT switches might be able to handle the highest rate of 1488 kpps of the Gigabit Ethernet.

Since we are dealing with a programmable device, we might have opted for a purely software approach, rewiring the *scratch rings* that feed the classifier and sink of the WASP and ESP microblocks. An external tool on the XScale core would then pre-fill packet descriptors with a given traffic pattern, and an additional microblock would simply bridge sources and sink, gathering throughput measurements. While this might be the preferable approach in many cases, it is also unfortunately the most code-demanding one.

[25] we are collecting one sample every 20 ms, a granularity imposed by the Linux scheduling interval

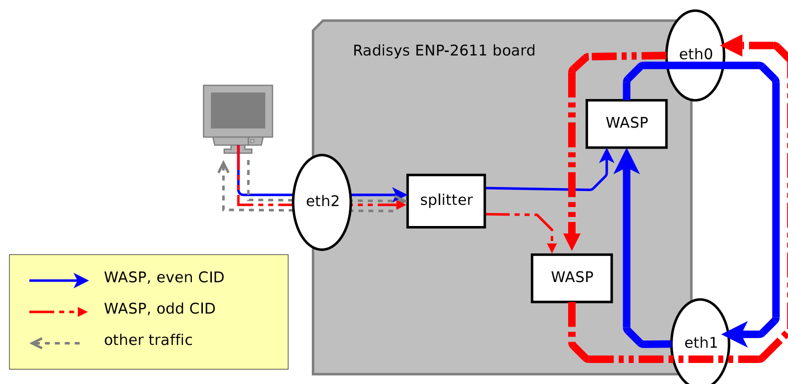



**Figure 5.12**: *Packet flow in our testbed, highlighting the full-duplex use of the loop-back link.*

The approach we finally keep draws inspiration from the purely software method presented above, but uses wires instead of lines of code. Once again, we will connect port 2 of the ENP-2611 board to a single test machine (*Wasp*) and wire the two remaining ports together, as depicted on Fig. 5.12. However, the static forwarding rule of the classifier has been rewritten to enforce the following policy:

1. regular packets coming on port 2 are simply returned to port 2.
2. WASP/ESP packets that are flagged as having errors, or that cannot be processed because of their "location" flags are delivered on port 2 regardless of their input port.
3. WASP/ESP packets coming on port 2 are delivered either on port 0 or 1 depending on a bit of the computation ID.
4. packets received on port 0 are delivered on 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 will keep packets until they drop themselves. We can "load" this loop by sending incremental traffic from the test machine and watch the impact on the system. Note that one WASP microengine actually processes packets from both "odd" and "even" CIDs, but leaves half of each to the other ME, unlike what the picture might suggest.

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. While we lack a mechanism to purge the loop manually (other than reloading the software), rule #2 ensures that we do not keep corrupted packets in the measurement.

Temporarily disabling rule #1, we can measure a maximum throughput of 2972 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.

| # queues | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| WASP (1ME) | 315 | 528 | 608 | 653 | 692 | 746 | 756 | 764 | 788 |
| ESP (1ME) | 390 | 672 | 950 | 1096 | 1183 | 1258 | 1380 | 1430 | 1546 |
| WASP (2ME) | 315 | 632 | 836 | 1026 | 1106 | 1190 | 1242 | 1293 | 1552 |
| ESP (2ME) | 390 | 779 | 1047 | 1298 | 1483 | 1603 | 1680 | 1744 | 1744 |

**Table 5.2**: throughput (kpps) of WASP and ESP microblocks, with single entry per hash chain and varying number of active queues and hardware contexts

## 5.5.2 Results with Count/Compare Instructions

We injected an increasing amount of `WASP:count` and `ESP:compare` packets in the loops, using the "transmitted packet" counter of the TX microblock to estimate WASP and ESP throughput. Note that the byte code of WASP packets as well as the implementation of the ESP operation have been modified to allow unlimited thresholds in both cases.

We also observed that packets that simply start with a `FWD` microbyte can be processed at a maximum throughput of 2966 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 `WASP:count`) will grow to 100 bytes on wire[26] and will limit the throughput to 2046 kpps – 98.22% of the theoretical maximum for packets of that size.

### Performance with 1 Entry per Chain

Like in the state-store stressing test bed, we have repeated the experiment using fewer microengines by reducing the number of *CID*s present in the test traffic. Table 5.2 shows the throughput obtained by adding more processing power and more parallelism. The best performance of ESP and WASP on a single microengine correspond to 58 and 38% of the maximal throughput, respectively.

Note that using 8 different queues may not be enough to reach the maximum throughput of the WASP or ESP microblock. Even with no hardware thread starving on the microengine, the presence of 8 permanently empty queues introduces a significant delay for acquiring a new packet to process. We have thus observed throughputs ranging from 606 to 788 kpps with 8 hardware threads doing WASP processing when varying the number of queues from 8 to 16 (which is the total number of queues one ME checks).

With a lower impact, the worst spacing between two used queues may also affect the performance. Using queues 0 to 7 and leaving queues 8 to 15 empty, for instance, leads to a throughput of 755 kpps while we can achieve 764 kpps if we enforce that no more than 2 empty queues appear between 2 used queues.

We then reproduced the experiment with one to 8 threads balanced on two different microengines to estimate how increased CPU power helps the performance. A "4 threads" setup thus means that we will have 2 active queues served on each microengine.

Comparing rows 1 and 3 in Table 5.2 also confirms that the WASP interpreter is CPU-bound. Indeed, while balancing the load on two different microengines leads to throughput increased by 20% with ESP, WASP sees its throughput improved by 57 (4 queues) to

[26]including Ethernet header and checksum, or 120 bytes with inter-packet gap and preamble)

| chain len (1ME) | 2 | 4 | 6 | 8 | 10 |
|---|---|---|---|---|---|
| ESP (compare) | 1502 | 1403 | 1296 | 1207 | 1124 |
| WASP (lookup) | 946 | 931 | 902 | 868 | 826 |
| WASP (map) | 774 | 757 | 730 | 699 | 680 |

| (2ME) | 2 | 6 | 10 |
|---|---|---|---|
| ESP | 1733 | 1470 | 1314 |
| W(lk) | 1858 | 1686 | 1476 |
| W(map) | 1526 | 1396 | 1225 |

**Table 5.3**: Throughput (kpps) with 16 queues, depending on the average chain length, with one (left) or two (right) microengines.

70% (8 queues), and the maximum throughput when all 16 queues are used has almost doubled.

### Increasing Chain Length

So far, the throughput we can achieve with one WASP microengine is only 50% of ESP performance. The results are a bit more in favour of WASP when using two MEs (88%). However, these figures are obtained with an extremely low load on the ESS (on average one entry per chain), which may not be a fair test. Indeed, the longer the chains, the less WASP interpretation will have a negative impact on microengines' CPU availability.

We extracted chains of colliding keys observed in the store during the previous experiments (Sec. 5.4.1) 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 $k$. In all cases, we used 16 independent computations, avoiding extra delays due to empty queues, and allowing for the maximum CPU parallelism on microengines.

With a single ME for processing, we can note that the gap between ESP and WASP performance is reduced as the average chain length increases (from 50 to 60%). 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).

We have further tried to reduce the cost of the `WASP:count` program, using earlier versions. Since the microengine is fully used, shortening the WASP program by a single byte immediately translates to lower inter-packet departure time directly corresponding to the number of cycles we saved. Removing the `YNC` opcode in `WASP:count`, for instance, still keeps the `count` semantic, but using the first slot in the mapped bank rather than the second one (as `WASP:collect` requires). In that case, we can see that the throughput is boosted from e.g. 730 to 800 kpps in the intermediate case of chains with an average length of 3.5 entries. Similar improvements are observed for all chain lengths we have tested.

Another interesting result is achieved when using the old version of `WASP:count` that uses `lookup` and `insert` microbytes rather than `map`. In this case, not only the code is shorter, but it also requires less bandwidth on the DRAM bus.

An intriguing fact is that, when we fully load the two microengines, the `WASP:-count` 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

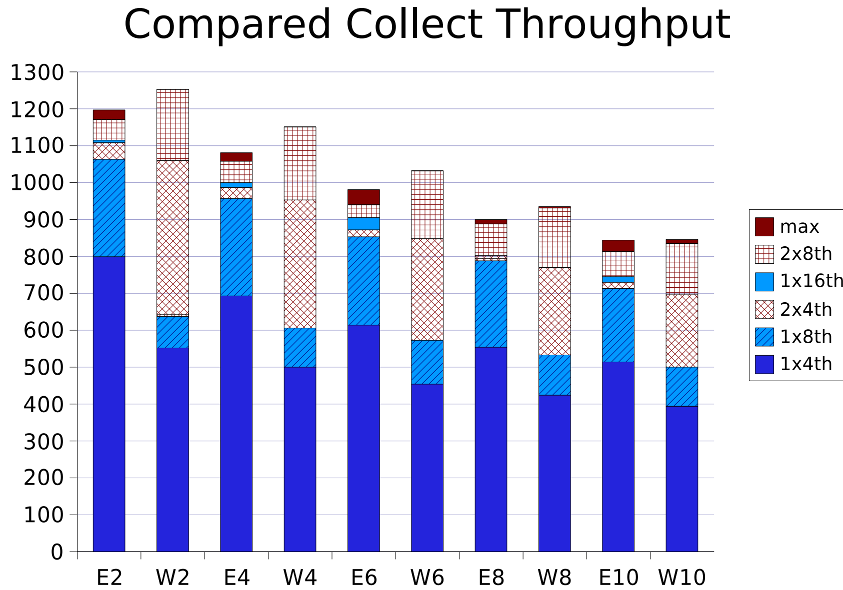



**Figure 5.13**: *Measured throughput for the* `collect` *operation with chain length varying from 2 to 10 entries, for both ESP and WASP (e.g. E6 is ESP walking a 6-entries chain), varying the amount of threads and microengine used for processing (e.g. 2x8 is 2 microengines, each processing 8 queues).*

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`.

Note that we can obtain much more precise throughput measures by deactivating store cleaning. The resulting values we obtain that way confirmed the result presented here, but they require less time for the average to "stabilise" and less feedback from the experimenter to ensure the proper value is reached.

### 5.5.3 Results with Collect Instruction

Another benefit of disabling store cleaning is that we can now keep a long chain without necessarily keeping the packets that created that chain in the loop. For instance, a collection of "count" packets with a threshold of 0 can build a chain of length $k$ before we send another set of packets with an infinite threshold but that only lookup for the $k$th key in the chain. Using this technique, it becomes possible to measure throughput of more complex operations with sufficient precision.

We modified the code of `collect` to ensure it will no longer drop packets and extracted chains of colliding keys from state store dumps of previous state store saturation tests. A vertical bar in Fig. 5.13 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 (e.g., from 1 ME x 4 threads to

1 ME x 8 threads) and how WASP rather benefits of an additional microengine, even with the same number of threads (e.g., 1 ME x 8 threads to 2 ME x 4 threads).

We have seen in section 5.3.4 that we could come with a similar processing time for the `collect` operation, and as expected, on a single microengine, WASP remains way behind in terms of throughput (see e.g., the 1x8 series). With two microengines doing WASP processing, however, we can now (slightly) outperform ESP when using the full power of two microengines. It should be noted, too, that in this case, both ESP and WASP have to commit the packets' variables into DRAM. The performance improvement can thus be fully attributed to the reduction of memory accesses during the ESS entries lookup.

We should note however, that in the worst case (10 memory accesses per ESS lookup), WASP reaches a throughput (829 kpps) that is only 98% of ESP. This is a rather surprising fact given that the performance gap was expected to increase as the number of memory access per lookup increased. We need to keep in mind, however, that despite WASP will require 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.

Without extensive additional tests and tools to measure the DRAM bandwidth on the IXP, we cannot directly assert that lower performance of the $W10$ test is due to a memory bottleneck. It is clear, however, that reading a full bank (48 bytes, including meta-data) while walking the chain in the state store was an optimistic decision that lead to sub-optimal behaviour once the chain gets long enough. We thus replaced the chain-walking code of the `MAP` opcode so that it only fetches 24 bytes of memory during the chain walk and issue an extra access to get the remaining 24 bytes once the correct entry is found (the "max" series for $W8$ and $W10$ on Fig. 5.13). This indeed slightly improved the throughput (from 829 to 840 kpps for $W10$ and from 924 to 928 kpps for $W28$), but actually degrades the performance for chains of 6 entries and below.

It should be mentioned that with the first 24 bytes of the entry alone, we can already set the VPU in a state capable of processing the following instructions. The remaining 24 bytes of data would only be required on the next `YNC` instruction, which allows us to resume execution of the VPU while we have a memory access pending. This is typically the kind of programming technique that the IXP microengines are designed to do efficiently, and our first estimations suggest that we could achieve up to 877 kpps. It would require, however, a significant revision of our code since we need to detect the case where data are still pending, and update the partially mapped bank transparently.

## 5.6 Compiling WASP programs on the IXP

So far, we have only discussed the possibility of *interpreting* WASP programs. The choice for an interpreter-based approach was motivated by the desire to support as many applications as possible. Ideally, an end-system using a new variant of a control operation could simply issue the code and the WASP routers wouldn't even notice that it is something new.

It is clear, however, that interpretation *has* a cost, and that replacing the interpretation by some pre-compiled native code could potentially save resources on the network processor. We estimated that interpreting one extra NOOP consumes 36 cycles on the microengine and may take 0.05 $\mu s$ more for each packet. When time on the CPU of the microengine is the performance bottleneck, even such a small increment in processing time may lead to 160,000 to 320,000 packets per second that we cannot process anymore when approaching the maximal throughput (2046 and 2966Kpps, resp.).

Through this section, we discuss two potential approaches for translating WASP bytecode into microcode for the IXP microengines: just-in-time compiling (section 5.6.2) and statistical optimiser (section 5.6.3). They both perform the same kind of code translation, but differ by how many programs may benefit of a native version of a program and how often the set of native programs available is updated. Implementing such optimisations for WASP is beyond the scope of this work, but we tried to provide landmarks and to identify technical difficulties for future works.

### 5.6.1 Environment for Run-time Compiled WASP Programs

The ephemeral store library for IXP2400, as detailed in [Calvert03] and [Imam03], has been designed to minimize the cost of access synchronization[27]. An important side effect, however, is that we cannot use the same ESS from two different microengines.

As a result, any run-time compiled instruction will have to co-exist with the WASP interpreter on a single micro-engine. This isn't much of a problem for the available size in the *istore*, as the interpreter currently consumes only 1753 out of the 4096 available uwords where the ESP microblock took up to 3182 uwords[28].

#### Reusing Interpreter Code

Out of the existing code, the "packet fetch" and "packet sink" parts of the WASP microblock could probably be shared by the interpreter and pre-compiled programs. Most of the WASP microbytes have a sufficiently short implementation to be written directly in a "compiled code chunk". ESS-related microbytes – as well as a few memory-related instructions such as LIX – are significantly more complicated, and we could prefer to see them available as "independent" code chunks that would be *called* by the compiled code.

Considering how those "heavy" microbytes are implemented in the current interpreter, and especially, the tight binding between the microbyte-specific code and the microbyte fetch/decode, we're very likely to need two instances of e.g. the LOOKUP microbyte, one that is designed to be invoked as part of the interpreter, and another that is designed to be called by a pre-compiled code chunk.

[27] e.g. chains in the hash table are locked via ME-local content-addressable memory, input queues are shared using global GPR, etc.

[28] partially thanks to our new code for fetch and sink phases

### 5.6.2 Just-In-Time Compiling of WASP programs

Just-in-Time compilation is a dynamic code generation technique that translates bytecode for a virtual machine into machine code at runtime, prior to executing it natively. The experience from general-purpose bytecode languages such as Java and LISP have illustrated the benefits that just-in-time (JIT) compiling can bring to bytecode evaluation. As soon as code reuse is sufficient, it gets more efficient to compile the bytecode into machine code for the host platform and then execute that machine code. The actual performance benefit will actually depend on several factors such as how many times the code is reused, whether it contains loops or not, how complex the JIT compiler is, etc.

In [Kind02], the authors show that it is possible to perform JIT compilation of a *SNAP*-derived language on the PowerNP network processor. The authors depart from definition of the SNAP[Moore02] bytecode specification by allowing (restricted) loops in their packets. They then compare the processing time of the interpreter (initialization, sand-box checks, instruction interpretation and cleanup) versus JIT-compiled execution (compilation, execution).

They study three kinds of executions:

**LCLE:** Long Code, Long Execution, e.g. a *traceroute* implementation as executed by a router, that has no loops and executes most of the instructions present,

**LCSE:** Long Code, Short Execution, e.g. a *traceroute* implementation as executed by an end-system, which would terminate after only a few instructions,

**SCLE:** Short Code, Long Execution, e.g. the *congested hop counter*, which iterates on the available queues of each router and gather statistics about their congestion status.

Only in the third case, we can see a significant improvement due to the JIT compiling. For LCLE-packets, there is a slight penalty from the compilation phase, and LCSE may suffer from a even higher penalty since *all* the code is compiled even when only a couple of instructions will actually be processed. Note that there is a similar problem with the sand box checks in the interpretation version, but with a smaller impact. With the exception of SCLE programs (which have no equivalent in WASP, since loops are forbidden), the main benefit of JIT compiler for packet-carried code will depend on the ability to cache and reuse results of compilation by subsequent packets.

#### Dynamic Code on the IXP

Compared to the PowerNP architecture, however, the Intel IXP network processor appears poorly suited to just-in-time compiling. First the microengines themselves have a very small instruction store (4K word on IXP2400 against 128K words for the PowerNP), meaning that the StrongARM control processor will be the sole component that could perform the compilation. Passing a packet to the StrongARM core for code compilation will unfortunately involve additional delays, making the potential benefit of JIT much more hypothetical than on a PowerNP.

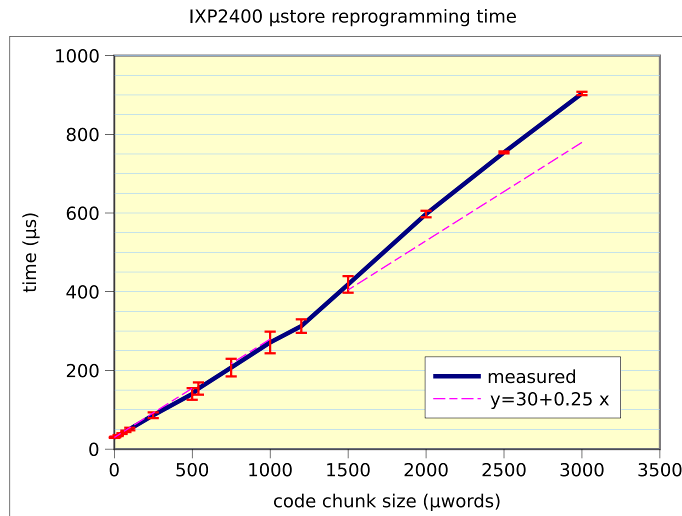



**Figure 5.14**: *reprogramming time of the IXP2400 network processor, measured by interruption of a busy loop on the reprogrammed microengine*

Moreover, the instruction store (*istore*) of a microengine cannot be modified while the microengine is active. The StrongARM therefore has to deactivate temporarily the microengine whose *istore* is to be reprogrammed, write the new content and then reactivate the *istore*.

According to [Spalink01], it will take 800 cycles (4 $\mu s$) to rewrite 10 instructions of the *istore* of an IXP1200. Projecting those results to the IXP2400, a full rewrite of the *istore* could take 1.638 ms[29] under the conservative assumption that the memory technology used for the *istore* hasn't evolved much between the IXP1200 and IXP2400.

Comparatively, Lee and Coulson [Lee06] report a delay of 60 ms for halting/updating/restarting a microengine. While the authors acknowledge the fact that reprogramming a whole microengine is actually rare when reconfiguring a network application, they did not provide a detailed report of the actual time required by each part of the process, nor the potential cost of a small update.

With proper adaptation of our controller/debugger for the WASP microblocks, we have gathered measurements of delay experienced by the "regular" code running on a microengine when another part of the same microengine is reprogrammed, as reported on Fig. 5.14. Our code issues 3 calls to the halMEv2 driver – `halMe_Stop`, `halMe_PutUwords` and `halMe_Start`, with varying amount of microwords submitted through the `PutUwords` call. The `PutUwords` call itself will take care of saving microengine state, load the proper registers and feed the microstore with new bytes.

On the microengine side, we have a single thread running a loop that continuously reads the timestamp register and detects any iteration beyond a given threshold (64 cycles, while our loop is 8 instructions plus a hardware yield). All other threads on the microengine are suspended to avoid interferences. The last abnormally iteration's length is collected in a specific GPR where the user can read the measurement. Meaningful code samples for this experiment are documented in Appendix D.3.

[29] $409,6 \cdot 4\,\mu s$

We have observed an average delay of $30.3\,\mu s$ when reprogramming less than 10 instructions. At this scale, the delay is mostly dominated by the two kernel/user transitions, which will highly depend on the operating system installed on the XScale core.

The additional delay per 10 instructions reprogrammed vary between 2.2 and $3.6\,\mu s$, with the higher values observed with chunk sizes between 1200 and 2500 instructions (9600 to 20000 bytes). These are also the ranges where the higher deviation is observed in the measurements, and it sounds reasonable to say that for those sizes, the cost of reading instructions on the XScale produces an additional overhead, since we are approaching the size of the data cache (32KB).

These results suggests a full-reprogrammation cost close to 1.6 ms for one microengine. We should however keep in mind that this doesn't take into account the delay before the microengine can be halted, which depends on the frequency of voluntary CPU release in the microengine code.

#### Code Caching and Memory Storage

The time required for compiling and installing JIT-generated code is not the only argument for avoiding JIT in the case of WASP. We have to keep in mind that the active router may have to handle flows from thousands of users, all having the potential of submitting new code in their packets. In these conditions, storage for JIT-generated code becomes a scarce resource, and the active node becomes vulnerable to several denial of (active) service such as:

- "regular" active service not finding translated code in the code store because of excessive amount of service running at the same time,
- delivery of "regular" packets delayed because of repeated submission of (useless) code to the JIT translator.

To avoid such misbehaviours, it would be necessary to authenticate the individuals that submit code requests to enforce quota so that each flow has a fair chance of having its own code compiled and available. Such requirement would be incompatible with our "anonymous" service objective.

### 5.6.3 Towards Self-Optimizing WASP Component

While there are chances that a 'just-in-time' approach wouldn't worth the price for WASP programs, this doesn't mean that we cannot take advantage of pre-compiled code anyway. An interesting alternative consists in letting the "control plane" decide which programs are requested frequently enough to be worth optimizing by a native code chunk. In this section, we will study how we could evolve the existing WASP interpreter towards a more "self-evolving" group of components.

#### Components Organization

**profiler:** Before starting compiling anything, we need to gather statistics about the frequency of each program processed by the WASP interpreter. Since a CRC check-

sum of the packet code has already been generated by the interpreter when receiving the packet, all we need to do is to increment the $k$-th coarse-level counter every time a packet with $n$ lowest bits of $crc(P)$ matching $k$ is forwarded.

**sampler:** When the self-optimizer running on the XScale control processor has identified a set of programs that represent an important share of the traffic, it can request more accurate sampling for a few values of $k$. This sampling involves delivering a complete copy of the packet's bytecode to the control processor over e.g. a scratch ring.

**analyzer:** On the control processor, the sampled packets are further classified using a more precise hashing mechanism. Immediate values embedded in the packet's bytecode may also be replaced by wildcards, etc. The statistical analysis will then identify programs that are frequent, but also those who are more computationally intensive and for whom a compiled version would mean a higher performance improvement.

**compiler:** Still hosted on the control processor, this component will produce the microcode to be added, and the associated matching rules for the classifier.

**match-checker:** Extending the classifier, this component will be required to identify packets for which we have compiled code available in the ESP and WASP microblocks.

The match-checker is probably the component that will require the most careful design and implementation. It is unclear at the moment whether it should rather be implemented in the classifier or in the WASP/ESP microblock. The IXP hardware suggests one of the two following implementations:

1. The whole packet's bytecode is hashed and the content-addressable memory (CAM) hardware will try and map the hash to a code identifier. The total number of pre-compiled code chunk we can support here will be limited by the size of the hardware CAM (16 entries per ME in IXP).
2. The packet's bytecode is compared word-by-word to a trie structure that contains code chunk identifiers (that maps to pre-compiled code chunks). The trie is preferably stored in local memory and has limited degree per node.

Note that the trie-based structure could offer *partial* bytecode compilation: even if it is not possible to provide pre-compiled alternative for all the possible WASP programs, many of them could be boosted by a code chunk that would perform the few first instructions (e.g. `map`, etc) that are common to many packets. By having "intermediate" code chunk identifiers in non-leaf nodes of the tree, we can compensate the match-checking cost by a shorter interpretation time.

Note too that beyond the obvious generation of code replacing *processing* of packets, we could also generate code that optimizes the fetching/checking phase for a given class of packets, for instance fetching less bytes from DRAM or skipping byte-alignment phase for packets that are already aligned.

### Seamless Microengines Reprogrammation

As detailed in [Lee06], programming a microengine may require a substantial amount of time. Buffering up to 58MB of packets while a component restructuration occurs due to a policy change may be affordable, but it is certainly not desirable when we are simply *optimizing* something that is already present.

Hopefully enough, when packets are passed from one microengine to another they do not care about their peer microengine, but rather about the *ringbuffer* that is used to relay packets between the two. In other terms, we could perfectly halt a WASP microblock running on ME#16 and start another one on ME#17 and keep this unnoticed by the classifier microblock *as long as the ring on which packets are delivered is kept unchanged*. Moreover, the atomicity of ring-handling operations is guaranteed by the hardware and it is perfectly safe to have threads in different MEs trying to get data from the same ring in parallel.

Provided that we have a *spare microengine*, and with proper synchronization from the control processor, we could then program the spare ME with the new code and then swap the ME running the old code and the spare ME seamlessly for the rest of the application. Special care will be needed when we are "deactivating" some compiled code, especially if the match-checker component is running on the classifier. In that case, we indeed need to make sure that no packets requesting the leaving code chunk remain in the queues leading to the reprogrammed microengine *before* the actual ME swapping takes place.

## 5.7 Uninterrupted Processing: Lessons Learned

Our number one design goal for the WASP filter box is that high load or misbehaviour of the WASP service may not lead to packet losses or other sensible performance degradation of the *forwarding* of non-active packets.

Our experience with programming and debugging the WASP filter on IXP taught us how an apparently insignificant implementation choice could corrupt the behaviour of the whole box. This section reports some of those cases that we believe to be worth mentioning and for which we can suggest good-practice rules for network processor programmers.

### The Classifier Gets Going...

Most of the behaviour of WASP/ESP microblocks can be safely ignored by the rest of the network appliance since they are running on separate micro-engines and therefore do not consume CPU resource of the other processing elements such as classifier, etc. We should take more care, however, when modifying the classifier. Indeed, in that case CPU cycles on the microengine will be shared by all kind of traffic, and we should thus ensure that the most complex cases do not require more than the "budget" corresponding to the maximum packet rate.

In a single port-pair forwarder, the system will then receive a new packet to process every 336 $ns$, which would mean a budget of 201 cycles for each packet if we had only one

thread in the classifier. In a more typical setup (e.g. 4 interfaces, 8 threads), a new packet could appear every 168 $ns$[30] and we are allowed 806 cycles (1344 $ns$) in the classifier.

Our measurement reports on average from 434 to 475 $ns$ per ESP packet on the classifier (using different traffic patterns, ESS loads, etc.), not counting the latency for enqueuing the descriptor and fetching the next one (which overlaps and would take some additional 80 $ns$). We can thus expect headroom of 492 cycles per ESP packet with the current classifier structure, which could easily accommodate for one more DRAM access (from 120 to 180 cycles).

We need to ensure that threads in the classifier do not get blocked while trying to enqueue packets to the TX microblock[31]. We observed such behaviours during development of the WASP microblock, for instance as the result of a "breakpoint" instruction. The classifier and the RX microblocks were indeed simply performing a busy loop while checking that a given queue is capable of holding one more packet or not, leaving the actual task of dropping excessive packets to the MAC chips.

This would of course be unacceptable on a production-grade appliance, and quickly becomes annoying on a prototype. Indeed, most of micro-engine reprogrammation or status query features require that the current context releases the execution on the chip. Even a soft-reset is impossible when the micro-engine is caught in such a busy loop. The most elementary fix we implemented consists in releasing CPU[32] before looping, provided that the code does not need some exclusive access when the enqueuing attempt occurs.

In its current state, the classifier in WASP prototype will detect if a queue remains full for an abnormally long time and enters a debugging mode if it happens. That 'debugging mode' could easily be replaced with code that drops or forwards packets.

#### Keep The Buffers Available

Even with dedicated CPU resource and well-behaving classifier, there is unfortunately a way our WASP/ESP microblock could deny forwarding for non-active traffic: the lack of buffers.

The WASP/ESP filter – much like any network application – uses a list of free buffers where the RX microblock will pick a handle every time a new packet starts and where other components (TX, ESP, WASP, classifier) will return buffers once a packet has been completely processed. We unfortunately experienced several cases where all the buffers we provisioned for our application ended up in the 32 ring-buffers leading to the ESP microblocks (a grand total of 16K entries, while we have 12K buffers available), leaving the RX block starving for buffers regardless of whether it receives active or regular packets.

It should be also mentioned that it is surprisingly easy for a network processor programmer to introduce a "buffer leak" in his program, by simply omitting to enqueue a buffer in the free list, or (more subtle) by having the same buffer enqueued in more than one queue as the result of microblock processing (which may later result in loops and item loss in the free list). Such errors are extremely hard to pinpoint and an autonomic

[30] assuming worst case of 4x1488Kpps

[31] as well as between RX and classifier microblocks

[32] through `ctx_arb[voluntary]` uword

network device that makes use of 3rd-party code blocks should likely provide a "garbage collector" component that would identify buffers that are "in use" for too long and return them to the free list.

Moreover, ring buffers leading to "auxiliary" component like the ESP/WASP filter should be better calibrated than the naïve approach currently used in WASP. A good practice rule should be to ensure that

$$\sum_i Q_i + P < T - (FP + FQ)$$

Where $Q_i$ is the size of an auxiliary queue, $P$ the number of auxiliary threads (which can hold a buffer each), $T$ the total amount of available buffers, $FP$ the number of processing threads on the *fast path* and $FQ$ the number of queues on the fast path.

### Keep The Chains Short

The length of chains in DRAM is probably one of the major issues to solve if we want to allow wire speed processing of ESP/WASP packets. In their introductory paper [Calvert02], Calvert et al. assumed that not only *pointers* of the hash table, but also *tags* and *chain pointers* could be stored in SRAM, while DRAM would take care of *values*. Even with the maximum of 64MB SRAM, we could only hold 5.33M entries that way while a single Gigabit interface will require 14.4M entries[33]. Chances are that this motivated the alternate design used in IXP-based implementations.

Moreover, the initial assumption was that the ratio ($h$) between the number of entries in the store ($m$) and the number of pointers in the hash table could be kept between 0.5 and 2. Comparatively, our prototype setup uses a ratio $h = 0.04$, and could reasonably reconfigured to offer $h = 0.37$ (512KB of hash and 8MB of state per store). Again, this speaks in favour of our "larger state per key" approach.

Simply enforcing that no more than $m/\tau$ entries per second can be created on a single store is thus not enough to guarantee good behaviour, even if we can offer fairness to the various traffic sources. Even with a small ratio of $m/\tau$ entries created and sufficient knowledge of the hash function used by the router, an attacker could easily craft a stream of requests that builds a chain sufficiently long so that further packets will experience arbitrarily long processing time, denying execution for other ESP/WASP traffic and severely reducing the amount of buffers available for other traffic.

Instead, we should measure the average service time for ESP packets for various chain lengths, and enforce a chain length limit according to the estimated rate of ESP packets and the available number of hardware threads.

## 5.8 Conclusions

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

[33] assuming each packet is allowed to create an entry that remains present for 10 seconds

operations as seen in ESP. We confirmed the advantage of larger entries for the ephemeral state store on this platform.

It should be noted, though, that the VPU processing makes more intensive use of the microengines ALU and that 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*, while ESP will experience only small throughput improvement whether the same number of flows are processed by 1 or 2 microengines. We therefore believe that it would be preferrable to integrate both code (ESP and WASP), maybe as well as runtime-compiled optimizations of frequent WASP programs, on a single microengine, which would better balance the ALU usage.

We also observed that the increase of the average chain length and the absence of a mechanism limiting the longest chain in the state store may have an important impact on the forwarding latency of WASP and ESP packets. For applications doing network performance measurement or that try to enforce a given quality of service, it might even be preferrable *not* to execute the WASP program in that case and just use default forwarding.

We envision that this could be solved by modifying the internal structure of the state store, replacing the "flat" hash table by something capable of splitting a chain into a trie-like structure, or simply by denying the creation of a new entry when the chain length exceeds a given threshold. We unfortunately lack time for doing a proper test of these proposals and they will be kept for further research.

Altogether, we are still far from a ready-to-deploy software package for a multi-linecard router. We believe however that the time invested in this implementation and benchmarking is rich in lessons for further development on network processors and that the IXP series is a good tool to prototype solutions that should work with multi-Gigabit interfaces. The dual-IXP2xxx development board is however preferable over our ENP2611 board for someone willing to simulate a linecard on a switch-fabric based router.

# Chapter 6

# WASP as Discovery Middleware

*"Do I* look *like a capsule?"*


## Abstract

*In this chapter, we present how WASP can be used as building block of distributed applications, and more specifically, how it can help scattered members of such applications to discover each other through the ephemeral store, using examples motivated by Grid and Peer-to-Peer computing frameworks.*

*Through simulations, we provide a quantitative estimation of the benefit we can expect on the peer bootstrapping process with varying amount of WASP routers in the network and how it depends on user's average behaviour or ISP use of dynamic addressing.*


## 6.1 The case for Discovery Middleware

Through the literature review of chapter 2, we have seen that active networks can offer a number of interesting improvements to resource usage or quality of experience for the end-user. Real-time auctions delivery, cooperating hierarchical web caches and video stream adaptation were the most popular use cases, and they could be joined nowadays by new applications inspired from Grid computing. Collecting available computations sites, routing computations requests towards the closest (or less loaded) point of presence or even performing distribution/aggregation of computation requests hierarchically could be seen as so many additional examples where in-network function can help a distributed application on end-systems. For network operators, the incentive for integration of such services is the same as e.g. the support of HTTP proxy caches: reducing the traffic towards the global Internet through local processing and storage.

Nowadays, structured peer-to-peer [Keon05] variants, and Distributed Hash Tables (DHT) in particular [Stoica01, Rowstron01, Maymounkov02], seem to have unlimited applications and are seen in proposals for document preservation [Kubiatowicz00], multicast streaming [Castro03] and even new network infrastructure [Stoica02] or routing [Caesar06, Caesar06b]. In many cases, a peer-to-peer network could offer a similar

service to the end-systems. Some technical issues haven't been completely solved yet, mostly in the field of locality awareness (also called "network congruence"), which are of higher importance for multimedia and real-time gaming, but projects such as OASIS [Freedman06] are a first step in the good direction.

Yet, the unability to *trust* operations performed by members of the network limits the application field of peer-to-peer solutions, and there may still be an interest for network operators to support "value-added services" themselves. Technically speaking, however, most of the services mentioned above are poorly suited to the hardware of routers and other network devices. Offloading their processing task to a "service providing farm" raises another problem as flows and service processing elements no longer see each other – something that active applications are typically not ready to handle.

The presence of WASP-capable routers in the network allows an elegant solution to that kind of problems. The state store can indeed be used to exchange location of "service providers"[1] with end-user flows, and the lightweight nature of WASP allows us to inspect packets at wire speed on the border routers. We further discuss this approach in section 6.3.

We have observed that a similar use of the state store could be useful for peer-to-peer overlays. One major problem to be solved in these systems, before we can depend on them for network architecture, is how to bootstrap them. To the question "*How do you find a contact node in the overlay to join?*"[Castro02], the answer is too often "*leave that to the end-user*". We studied the solutions proposed for P2P bootstrapping in existing implementations, aiming for a fully decentralised and self-configuring mechanism. As reported in section 6.2.4, we mostly found a collection of hacks working around the lack of a proper support from the Internet architecture: ultimately, either the user will have to provide a contact node, or the software vendor will have to provision a server to process every bootstrap request.

In other contexts, that problem of communication *bootstrapping* is typically solved by a designated router on the local network advertising e.g. the local entry point to a distributed database (such as DNS server advertised through DHCP). But peer-to-peer networks are not part of the architecture and there is apparently no incentive for network operator to advertise the closest machine member of every possible P2P network – which may not even be in their own network. Here too, the key/value pairs storage offered by WASP may offer an interesting alternative to the existing solutions: rather than relying on a well-known end-system to gather membership, we could have the information collected and available *in the network itself*.

In section 6.4, we propose a model of a commonly used bootstrapping mechanism (history lists) in which nodes try and join the community by contacting nodes that were their neighbours in the previous session. We also propose metrics to evaluate the performance of that mechanism under varying network conditions. We then show how the presence of "active" routers featuring an ephemeral store may improve the performance of that method in section 6.5.

[1] *i.e.* "nodes offering the packet filter/split/merge service", not Internet Service Providers

## 6.2 Discovery: Flavours and Existing Solutions

### 6.2.1 Local Service Discovery

The most obvious services the user can think of is usually the one closest to the physical world: printing, e-mail servers, local file storage, etc. One of the goals of IPv6 in this regards was to bring "plug and play" to the network, and with the increasing success of wireless and portable devices, it has almost turned into a requirement. Considering inter-dependency between service discovery and naming in local networks, the Zeroconf IETF working group [IETF06] has studied the problem and proposed DNS-Service Discovery (DNS-SD) [Cheshire05]

Even in IPv4, much of the bare IP configuration can now be delegated to DHCP [Droms97] servers, which reply to broadcast requests on the local network. In the case where no DHCP entity is present, which could be the case in an ad-hoc network, link-local IP addresses can be used to allow interconnection of devices even in the absence of any infrastructure. The "Bonjour" technology available in MacOS X [Apple05] is an example of how those link-local addresses, together with mDNS-SD [Cheshire05b] (the 'service discovery' variant of DNS) can be used to achieve interconnection in such 'unpredictable' environments. Devices with more specific needs typically come with a vendor-specific feature advertisement and lookup (e.g. the Cisco Discovery Protocol), which is limited to one or a few LANs.

When the 'locality' goes beyond a simple LAN, but remains topologically close to the client, the *expanding ring search* [Boggs82, Boggs83] is one of the most common techniques used. The key idea is to make use of multicast support to send the service location request towards a multicast address that all potential service providers will be listening to. Since we want to avoid to flood the whole domain with requests, the search is performed ring after ring, using the *time to live* field of IP packets to increase the search distance allowed at each step.

### 6.2.2 Global Service Discovery

On the opposite side of the spectrum, global services discovery will locate an item *wherever it stands*. Such systems usually expect that the client has located a nearby node that is member of a distributed database. Domain Name Service is of course one of the oldest example of such distributed lookup facility, where responsibilities of each node depends upon its level in the hierarchy. Similar hierarchical databasesare proposed in active networks literature, such as Spine, the network infrastructure protocol used with the *protean* active network. It is based on a tree where each node represents an autonomous domain, labelled by a name and a collection of leaf nodes. Each spine node lists known active routers, each "user network context" (e.g. user-installed packet processing functions).

When the nature of information can no longer be made hierarchical (or cannot be arbitrarily split), hierarchical systems such as DNS become less obvious to use and one might prefer the *distributed hash table* (DHT) approach [Stoica01, Rowstron01].

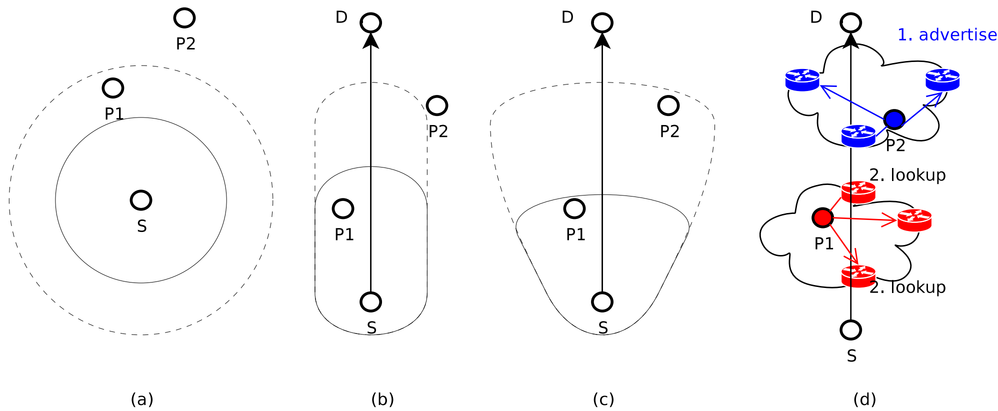



**Figure 6.1**: *Comparing expanding ring search (a) with two variants of oriented multicast search (b and c), and the domain-wide advertise/lookup mechanism (d).*

### 6.2.3 Proxy Services in a Transit Network

There is a number of situations, however, where the service neither fits a local search, nor a global. Application-level caching and multimedia filtering are examples of such services that could be located in ISPs or even transit networks and for which ring search sounds unappropriated. Many applications of active networking to GRID computing platforms [Lefèvre02] also falls in this category.

Unlike the case of local or global discovery, the end-system $S$ (see Figure 6.1) that looks for a proxy service is planning to use that service when communicating with a destination end-system $D$ whose address has been learnt by an out-of-band method. The distance should be measured based on the shortest path $S-D$, rather than between the endpoints themselves, in order to limit the path stretch when making a detour through the proxy service.

Among the existing solutions, one can mention oriented multicast [Magoni02], which suggests a new forwarding mechanism allowing a query message initially flowing from $S$ to $D$ to be duplicated and forwarded in directions orthogonal to the $SD$ path. By orthogonal, the authors mean that it will be flooded on every interface that does not lead either to $S$ or $D$, therefore limiting redundancy. Moreover, such duplicates receive a new TTL value (the range) that controls how far from the shortest path the request may go (cases (b) and (c) on Fig. 6.1). Unfortunately, the oriented multicast protocol (OMP) does not allow one to control the amount of replies for a given query, except if used in an expanding oriented search approach. Moreover, a malicious source could even bomb the destination using a very widely available service and a too broad range.

Alternatively, one could set up an infrastructure that dynamically creates overlay networks interconnecting end-systems and intermediate proxy through tunnels that achieve the desired topology. This assumes that every potential proxy has joined a distributed database that can be searched for proxies nearby a given path. Unfortunately, the existing proposals following this approach (e.g. OPUS [Braynard02], or the X-Bone [Touch00]) do not come with a truly scalable and completely decentralised method for identifying available proxies in a very large-scale network. Moreover, maintaining the infrastructure (that is, capturing the global network topology), monitoring the available resources and

expressing applications needs in a generic fashion remains a resource-intensive activity, even when hierarchically distributed like in OPUS.

In section 6.3, we will show how, with sensibly less support from all sides, WASP manages to offer enough information to *end-systems* and *service providers* through domain-wide advertisements (see 6.1d) so that they can take strategic decisions themselves.

### 6.2.4 Joining a Peer-to-Peer Community

In virtually all peer-to-peer applications, the operation of *joining the network* is separated into two steps:

1. find a *contact node* (or bootstrapping peer);
2. use that contact node to locate your neighbours in the network.

Depending on the desired network properties, the process of locating neighbours will of course vary, involving e.g. searching nodes with an identifier close to yours, detecting peers in your physical vicinity, etc. The incoming peer therefore needs a way to probe its current neighbourhood and ask peers for their neighbours list, in order to compare them and improve its own set.

The role of the *contact node* in this process is to provide the incoming peer an initial set of peers so that it can start this incremental neighbourhood selection. Most (if not all) architecture papers consider the location of that contact node as an implementation issue and assume the joining algorithm already has an "entry point" in the network.

There are different techniques implemented to locate that contact node, but none of them are fully satisfactory:

**user knows:** the application expects the user to provide the location of another running node to join, and provide a way for starting a 'stand alone' node. This is for instance the case in Chord [Stoica01].

**static node:** the application is bundled with a few addresses (or DNS names) of machines that will handle the bootstrap process. These machines (known as *pong servers* in the gnutella network [Limewire01w]) will register every peer and reply with a list of peers that are believed to be still alive. The obvious drawback is that as soon as the pong server is put offline, the application simply stops working.

**address-encoded:** the user gives the application data that contain the address of the pong server to be used. This is for instance the case of the *BitTorrent* protocol[Cohen04w] where a `.torrent` file contains both metadata of what you will download, and the location of the *tracker* that will be your "pong server". However, the `.torrent` file needs to be transmitted through some off-line means (e.g. using mails, newsgroups or a website) and there is no collaboration between peers downloading different contents.

**pool service** Jelasity et al. [Jelasity06] suggest that a pool of potential peers should be maintained by a separate distributed service. While this allows quick spawning of P2P topologies from a set of stable nodes (interesting in the case of Grid computing), they still rely on off-line mechanisms to include the node into the pool in first place.

**history file:** rather than relying on some external pong server, each peer could store the list of neighbours it identified in the last session and try to reconnect to them (this is the case e.g. in FreeNet[Clarke00]). Depending on how long your system has spent offline, the size of that history and how dynamic the network is in general[Rhea04], this technique might offer fair performance or turn into a total nightmare. The actual success of the application will strongly depend on the presence of *super peers* that are running 24/7 at fixed IP address and on a mechanism to identify super peers in that history.

An intriguing alternative[Castro02] suggests the use of a DHT (the Universal Ring) to associate identifiers of *service DHTs* with a list of contact nodes for that specific DHT. The authors advocate that the Universal Ring, being more widely supported that application-specific overlay, could be more easily located and could become part of the network architecture itself.

The techniques suggested to locate nodes of the Universal Ring are however not more convincing than what we found in other literature. Moreover, the proper operation of the Universal Ring requires that each member of the ring and each service obtain a digital certificate for their private key, which is – in our humble opinion – only practical in very restricted deployment scenarios.

## 6.3 MagNet: Service discovery with WASP

Rather than using a typical "all-in-one" active network approach, the custom service could be offered as follows:

1. identify anchor points in the network (*i.e.* machines able to host the service),
2. detect at which anchor point(s) service deployment is strategically most useful,
3. route packets requiring the service towards deployed *service provider(s)*[2],
4. apply merge/split/filter service on packets received by the service provider(s).

We can thus decouple the service processing from the service discovery discovery, and use WASP to locate the most interesting service provider(s) independently of what the service will actually do. Once service-providing node(s) have been located, the end-system can adjust application behaviour so that the relevant flow goes through the discovered provider(s).

When a new applicative flow is initiated, small active packets are used to probe the network on the route to be taken. Each time such a probe crosses a WASP node, it will lookup the node store to see if it can find *advertisements* of the expected service, consisting of the provider address and cost for reaching that provider from the local node. The same kind of active packets can also be used by the service providers to install advertisements in routers of the local domain. Figure 6.2 illustrates that two-phase process: servers $A$ and $B$ first flood the domain with WASP packets advertising their presence, avoiding to re-install an advertisement in a router that already contains a better one (e.g. advertising

[2] *i.e.* "nodes offering the packet filter/split/merge service", not Internet Service Providers

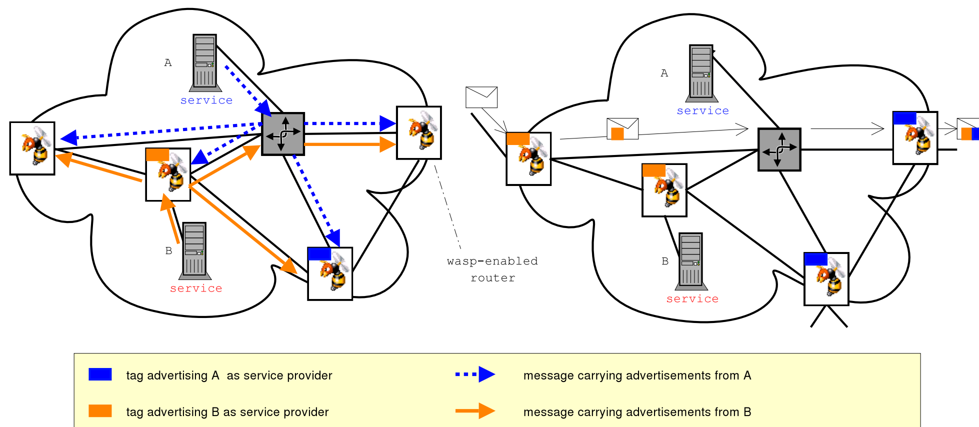



**Figure 6.2**: *Advertising (left-side) and looking for (right-side) service*

a closer or less loaded service provider). A source $S$ can then use another WASP packet to record those advertisements as a list of provider $P$ and branch point $X$ information: $(P_{addr}, X_{addr}, cost(S, X), cost(X, P))$.

This approach – codenamed "MagNet"[3] – allows the network operator to remain in control of the additional load generated by service lookup in his domain by deciding the refresh rate for advertisements. However, since advertisement is limited to the domain that hosts the service provider, the end-systems still need to have an initial destination and will only find providers from domains that lead to that destination. In other words, unlike what a global *anycast* [Partridge93] service could offer, you cannot "get the closest news aggregator" with MagNet, but you can "get the closest news aggregator between me and slashdot".

To work properly, MagNet discovery requires that end-systems and operators agree on a *well-known tag* under which advertisements for a given service will be stored and on the data layout of advertisements. An ESS entry could easily accommodate for a few provider addresses and their associated costs (hop-count distance, server load or even a mix of the two).

Thanks to the additional programmability offered by WASP, applications that use MagNet to locate service providers are free to apply the search behaviour that best suits their need. When looking for a HTTP cache, for instance, one might prefer to *return* towards the source at the first match. Another application might prefer to filter out poor advertisements, or keep searching (keeping track of the best advertisement) along the whole path and only return if a provider with a sufficiently attractive cost is found. A grid application that has to retrieve and merge several results from scattered computing sites might instead prefer to retrieve all the potential provider and later inspect those results on the end-system to see where the deployment of merging function would optimise the transfers.

---

[3]With our mechanism enabled, providers can "attract" relevant packets in their neighbourhood, but they don't affect "regular" (non-magnetic) packets – hence the codename.

The service application can equally benefit of WASP programmability to express policies that will decide whether an advertisement should replace another one in the router, how much advertisements are stored, and so on. The advertisement packets may also report the location of other providers to their emitter, so that the available resources of a domain can coordinate their efforts by themselves.

### 6.3.1 Flooding Locally

In order to advertise the service, providers have to locate WASP routers in the local domain and send them WASP packets that will install advertisement tags. We benefit here from the fact that WASP processing is *optional* so no overlay of WASP-enabled routers need to be pre-established. In our previous work [Martin03], we showed how knowing the routing table of the local domain suffices to discover all the active routers of that domain. The topology internally built out of the link-state database of the underlying routing protocol is annotated with the capabilities of routers so that only WASP-capable routers are explicitly refreshed.

As an alternative, we could use the 'opaque LSA option' of the *link state advertisement* messages. Data in those opaque LSA are flooded to the whole network but they are ignored by the routing algorithm of OSPF. This approach is used in [Keller03] to exchange CPU, memory and load information of active nodes to allow resource allocation within a given domain. In both approaches, the routing daemon needs to be modified to allow external programs to retrieve information gathered in the database, and the 'opaque LSA option' also requires an API to define the local option to be transmitted by OSPF.

Note that there is an implicit trust relationship between the users of Internet and operators of transit domains. If we want that relationship to extend to services provided by the operator, we need to make sure that advertisements indeed come from the operator. An attacker that would manage to put his own address in an advertisement for e.g. HTTP proxy service could gain a privileged position to eavesdrop traffic from other users. This can be prevented by using only *protected* tags to advertise services. Protected tags (as described in Sec. 4.3.3) can only be written and modified by so-called *super packets*, and we expect the network operator to clear the "super bit" of WASP packets of all packets it receives from other domains.

### 6.3.2 Persistent Data in Ephemeral Store

A particularity of *ephemeral* storage is that the advertising tag will be deleted after a fixed period $\tau$, regardless of any refresh we could try to perform. Therefore, there may be a small delay between the moment where a WASP router decides to remove an advertisement tag and the moment where an advertisement refresh comes. Even if the server manages to learn precisely the tag's lifetime $\tau$ it cannot completely avoid the risk that client packets may not see any advertisement. If this risk cannot be afforded, it is still possible for a service to use two separate keys $k_1$ and $k_2$ that will be refreshed with a period $\tau + \epsilon$ but such as advertisements of $k_1$ and $k_2$ are separated by a delay of e.g. $\tau/2$.

A client that doesn't find the "primary tag" (referenced by $k_1$) can then check the "backup tag" (referenced by $k_2$) to see whether the service is really missing.

Depending on the service constraints and on the end-application policy, several variants to this scheme can be implemented, such as:

- using the router's clock to decide whether $k_1$ or $k_2$ should be preferred when inserting/looking up information, in order to improve our chances to get information in one try.
- always update both keys once one of them has been inserted so that the application can get the most recent information regardless of whether it uses $k_1$ or $k_2$.
- use $k_2 = f(k_1)$ with a key-modification function $f(k)$ that can be computed by the VPU (e.g. toggling a given bit pattern) to avoid the need for carrying two keys in the packet.

## 6.4 History File Processing

Among the mechanisms presented in section 6.2.4 to join a peer-to-peer community, bootstrapping based on history files is the only one that is truly decentralised. This section presents a model of this process along with performance indicators that we will use through the rest of this paper.

The reader should stay aware of the major inherent drawback of history processing: it requires an initial list. For machines that have been running at least one session, joining the network again is only a matter of patience, but for those system on which we just installed the P2P software, we need some out-of-band mechanisms to obtain a history. We will first assume that we have "imported" that initial list (e.g. shared by e-mail, from a friend inviting the user to join, or retrieved from a web search). We will see later in section 6.5.6 how we could bypass this requirement.

### 6.4.1 The Community Model

The community is made of a collection of *peers* that are connected to the Internet via a specific *stub* AS. Each peer belongs to a given *class* that defines its average online and offline time (e.g. "stud" nodes that are online for 2 out of 22 hours, "home" nodes that are online for 2 out of 14 hours and "desk" nodes that are online for 8 hours out of 24). We assumed here that, for a world-wide deployed system, we could reasonably consider that nightly shutdowns have no globally observable effects.

Each peer's behaviour is defined by the state transition diagram of Fig. 6.3. When a peer enters or leaves the "offline" state, it will pick a random duration $d$ before it leaves or returns to that state. As soon as the machine is powered up, it will contact peers in its history list ("connecting" state) observing a random pause $dy$ before each attempt[4]. Since we want to avoid a partitioning of the whole peer-to-peer network, we only consider that

[4] $dy$ follows an exponential distribution around an average delay of 5 minutes, which we selected to approximate the timeout of a TCP connection establishment

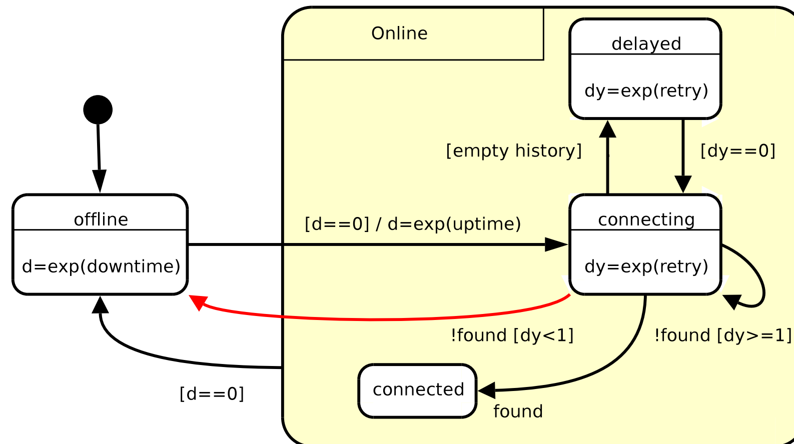



**Figure 6.3**: The UML state transition model of the peers

a neighbour has been found when we manage to contact a node that is in “connected” state. When a peer has scanned its whole list without managing to contact anyone, it will enter the “delayed” state where it remains inactive for a short time interval before it tries scanning its list again.

Note that the system is greedy in the sense that a peer that has already found a neighbour will continue processing the rest of the list and the neighbour list of its neighbours. At shutdown, the peer will only keep a fixed amount of addresses in its history list (typically set to 10 in our experiments) and it will prefer long-established neighbours over other addresses.

Note too that the value of $d$ is the *intended* session length. When the system is in the “connecting” state, we added a random return to “offline” modelling the behaviour of a frustrated user who connected his machine mainly with the idea of using the peer-to-peer system and just powers it off because the service is too long to set up.

In the following text, we will use the term “online” to refer to peers that are in one of “connecting”, “connected” or “delayed” state.

### 6.4.2 Bootstrap Quality Indicators

In a typical simulation of the community we described, we start with a predefined amount $N_0$ of online peers. After a progression phase, the system will oscillate around the “equilibrium” amount of online peers $on_{th}$ which can be derived from the online and offline time. For instance, when only *desk* (A) and *home* (B) classes are involved, we will have

$$\frac{on_{th}}{N} = \alpha\frac{u_A}{u_A + d_A} + (1-\alpha)\frac{u_B}{u_B + d_B} \tag{6.1}$$

where $N$ is the total amount of peers and $\alpha$ is the ratio of *desk* nodes among them ($u_A$ and $d_A$ being respectively the average time spent online and offline in minutes).

In order to compare the quality of different bootstrap mechanisms, we measure the following indicators:

**failed attempts:** this is the ratio between the *unsuccessful* attempts and the total amount of connection attempts, that is, those who contact a machine that is down, that is not connected to the community yet, or an identifier that is no longer (or not yet) associated with a machine that can join the community.

**frustration ratio:** is the ratio between the number of sessions aborted before their scheduled "ontime" expires (e.g. due to a bored user) and the total amount of sessions in the simulation.

**bootstrap efficiency:** measures the percentage of the time spent online during which the node actually has access to the community.

$$eff = \frac{\sum T_{connected}}{\sum T_{online}}$$

For most indicators, the progression phase exhibits different values than the oscillation (equilibrium) phase. There are thus two kinds of scenario we might study:

**bootstrapping:** We initiate the community with a small amount of connected nodes which are all aware of one another (e.g. through coordinated configuration) and then study whether (and how fast) the whole community can reach the "equilibrium" phase. This allows us to simulate to what extent history-based peer bootstrapping is viable as the sole mechanism for a peer-to-peer community.

**survival:** We initiate the community with a number of connected nodes that is close to $on_{th}$ and study how long this "equilibrium" can be maintained. This can be used to simulate the history-based peer bootstrapping as a fallback mechanism when another system (e.g. a pong server) suddenly becomes unavailable. Due to the lack of a global rendez-vous point for the community, the network can remain "online" only as long as at least two members remain *connected*. If we enter a state where the last *connected* member goes offline, any connection attempt will fail and the community will not recover.

### 6.4.3 Behaviour on a "Regular" Network

Intuitively, the probability of a successful contact will depend on the amount of connected peers, which itself depends on the amount of online peers. In turn, a high success probability will increase the number of connected peers and therefore reduce the probability of anticipated poweroff.

We ran a sequence of experiments with a 1000-peer community, varying the amount of "stud", "home" and "desk" machines to explore values of $on_{th}$ ranging from 90 to 165 online nodes on average per simulation. The number of connected peers at the start of each simulation has been set to match $on_{th}$ for that specific setup, allowing us to have more accurate results on a relatively short (2000 minutes) time span. We can see on Fig. 6.4, however, that the actual number of online peers may be up to 17% below the expected $on_{th}$, which can be explained by the fact that the formula for $on_{th}$ doesn't take into account anticipated poweroffs.

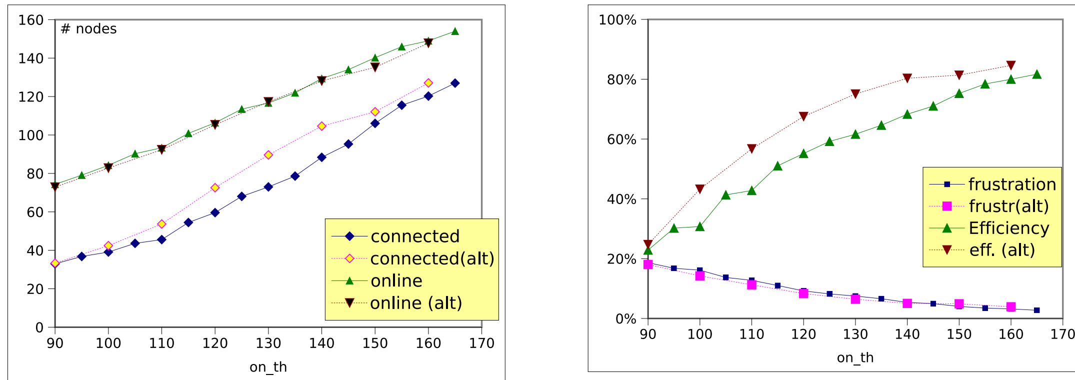



**Figure 6.4**: *(left) Average number of online and connected peers for different values of the theoretical $on_{th}$ parameter. (right) Efficiency and frustration ratio varying with the $on_{th}$*

| | $u_A, d_A$ | $u_B, d_B$ | $\alpha$ | $on_{th}$ | $len$ | $eff$ |
|---|---|---|---|---|---|---|
| H+S | 120,720 | 120,1200 | 0.387 | 110.1 | 94 | 0.43 |
| D+S(alt) | 480,960 | 120,1200 | 0.082 | 109.9 | 112 | 0.57 |
| H+D | 120,720 | 480,960 | 0.958 | 150.0 | 109 | 0.75 |
| S+D(alt) | 120,1200 | 480,960 | 0.750 | 150.7 | 151.4 | 0.81 |

**Table 6.1**: comparing efficiency in two simulations with identical $on_{th}$, but different mean session length ($len$); $\alpha$ is the portion of nodes of type $A$ in Eqn. 6.1. Up and down times given in minutes.

As the theoretical number of online peers increases, we can see that the frustration ratio decrease from 19% to 3%, which is accompanied by a more accurate approximation of the actual average of online peers by $on_{th}$. A few additional experiments with a home:desk ratio of 5:5, 3:7 and 1:9 (leading to $on_{th}$ values of 237, 275 and 313 respectively) confirmed the progression we observed. With an average of 313 online machines, the efficiency reaches 97% and the frustration ratio tends towards 0, which makes the relative error on $on_{th}$ of only 2%.

We repeated the experiment using only machines from “stud” and “desk” class, and report the result in the “(alt)” data series of Fig. 6.4. While the average number of online peers for these simulations is virtually identical to the figures obtained with “stud+home” and “desk+home” mixing of the previous simulations, the alternate simulations exhibit a higher efficiency. We then investigated the mean session length (see table 6.1), which revealed as expected a longer mean session in the alternate simulations. This mean that, for identical average community size, it is preferable to have fewer “better peers” if they have a longer average session length.

One should note that in both simulations, efficiency and frustration can be directly expressed as a linear function of the probability of a successful connection attempt. This confirms our a priori feeling that we should try to improve the probability of a successful “hello” if we want to improve the overall system performance.

## 6.5 Active Domains boosting P2P

A simple way to increase the chances for a hello message to be successful is to make it more capable of detecting running peers on its way. With regular IP processing, a "hello" packet will test only one machine and will be successful only if that machine is connected[5]. If instead the "hello" packet could be distributed to all the machines running in a given domain (e.g. all the clients of one ISP), our chances of having a positive answer will become:

$$P(success) = 1 - (1 - P(conn))^N$$

**P(success)** probability of a successful reply if the packet is targeted to a machine in domain $D$.

**P(conn)** probability for a machine of $D$ of being connected. We assume in this formula that the domain $D$ is homogeneous and that all its clients have an equal probability of being member of the community.

**N** is the number of clients in domain $D$ that have already been members of the community.

Such a technique, however, would lead to excessive unsolicited traffic. We can still achieve similar improvement with substantially less overhead if edge routers of the ISP domain are capable of storing information about which of its clients are member of the community. Such an ISP is then called an *active domain*, and the community members connected through it are said to be *active peers* if their software is capable of periodically sending active packets that store their address on the border router.

In the following simulations, we model an *active domain* as a domain on which it is possible to issue a *probe* that will return the address of a random peer in the community that is client of that domain. Even a successful probe that returns an address does not immediately cause a transition to the *connected* state. Rather, the address is pushed at the head of the 'pending list', and will be processed as the next peer to probe.

There are two situations where a peer can benefit from the active routers support: either it could have the address of an "active peer" in his history list, or it could be itself a client of an active domain and do a 'local probe' for connected peers. The results presented in the following sections assume that both probing mechanisms (i.e. local and remote) are in use. Our early simulations with remote probing only showed that adding local probes does improve efficiency, but only by a few percents.

Note that the above formula only holds for domains where $P(conn)$ of clients is independent. This won't be the case, for instance, if $D$ is a geographically concentrated area and if all machines in $D$ observe similar nightly shutdowns – such as in a corporate network [Bolosky00]. It should be considered as a theoretical upper bound on $P(success)$ rather than a way to predict it. Note too, that it is the probability of a successful connection attempt *given that the packet is sent to an active domain*. We should thus moderate it with the probability that an entry in the history list is a client of an active domain.

[5] in that case, we simply have $P(success) = P(conn)$

### 6.5.1 Registering Membership in the State Store

Compared to most of the active network frameworks proposed in the last years, the support we require from the network in this chapter is extremely modest. The most elementary requirement is the presence of a publicly available *state store* on the active router where packets could resolve a community name (or its hash, the *community key* $K$) into an address (or a short list of addresses) of community members.

Using this framework, the peer discovery protocol could be implemented with two (simple) WASP programs:

**`peer_adv(K, addr)`** this packet is sent periodically by community members to a random peer. When processed on an outgoing interface, it will try to add its source address $addr$ to the list maintained under key $K$ in the ESS. If the list is full, it will return to its source where it could trigger a self-regulation mechanism (see below).

**`peer_probe(K)`** this is the probe packet sent together with connection attempts to the addresses mentioned in a history list. When processed on an interface, it will look for a list of peer in the ESS and copy the addresses into packet's "scratch" area. It will return to its source if any address has been found and goes on its way otherwise.

A peer that bootstraps will first issue a *local probe* that looks for a WASP router with membership information in its local domain (e.g. using a random target address and a small TTL). Then, together with the connection establishment attempts for each address in the history list, the peer issues a *remote probe* that will come back if the community key is found in a router on the path to the probed address. The same `peer_probe()` program executed on the *outgoing* interface of the source's domain or on the *incoming* interface of the destination's domain can implement local or remote probe respectively.

On each WASP router, we only need one entry (32 bytes) per community, independently of the number of peers that are members of this community. Each entry should be refreshed at least once per "ephemeral period" $\tau$ (typically 10 seconds), and preferably by drawing a random delay uniformly between $\tau/2$ and $\tau$ to approach the random peer selection mechanism mentioned in the model.

While a single hardware context on a modest network processor is reported to handle about 200,000 ESS requests per second [Calvert03], thus potentially supporting up to 2 million members in a single domain like a charm, it is clear that it would be preferable to use the feedback information provided by returned `peer_adv` to estimate the amount of local peers and adapt advertisement period accordingly.

Similarly, if we have each online peer issuing one probe every 5 minutes on average, a single hardware context could handle aggregated requests for around 63 millions peers, in the unlikely event that they all probed an address behind our router. Still, even a simple network processor such as the IXP1200 used in [Calvert03] could use up to 16 hardware contexts for WASP packets processing, and we could easily offload the egress router by adding WASP processing on access routers too.

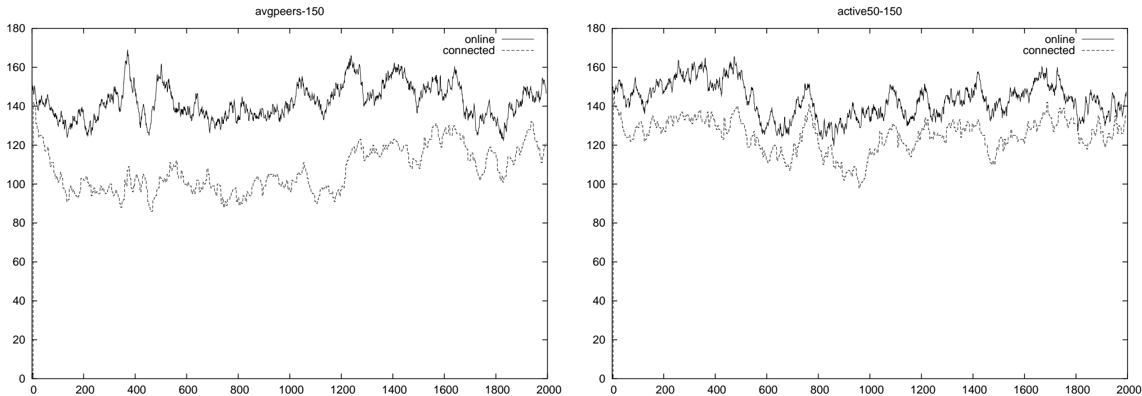



**Figure 6.5**: *plotting the amount of online and connected peers over time in a regular (left) and an active (right) simulation*

### 6.5.2 Keeping the Community Running

In these simulations, we will investigate the benefit of active networks during a "survival" scenario. 1000 nodes are first randomly assigned to the different domains which are then randomly "activated" to reach 100 active peers[6]. All the following simulations use a population of 950 "home" machines and 50 "desk" machines and are started with $on_{th} = 150$ connected peers. Each simulation set contains 20 independent runs of 2000 minutes.

**regular** This is our "reference" simulations set, with no active peers. As detailed in the previous section, this leads to an average of 106 connected peers, a system efficiency of 75% and a frustration ratio of 4%.

**act:med** This set has on average 103 active peers and an average domain size of 20 peers. It leads to an average of 128 connected nodes and improved efficiency of 87%. We can also notice the low frustration ratio of 0.99%

**act:small** In this set, we have on average 100 active peers and an average domain size of 10 peers, which leads to 125 connected nodes and system efficiency of 86%. Smaller domains thus clearly offer more modest performance boost over the "regular" network, but still, this remains clearly a boost over the reference set.

**act:huge** Here we have only one active domain whose size is on average 113 peers, resulting in an average of 130 connected nodes and system efficiency of 89%. It also has the lowest frustration ratio of 0.66%.

All the active simulations thus outperform the reference simulations, be it by the size of the community they manage to maintain, the number of anticipatively terminated sessions (e.g. frustrated users), or the amount of time required to get connected to the system. These performance gains still hold with a population of 2000 peers as long as other parameters are also scaled accordingly (e.g. 200 active peers and preserving the average domain size).

[6]As a side effect of this policy, we will experience a higher variance in simulations featuring only 10 domains

| | $sust$ | $postavg$ | $t_{80}$ | $t_{100}$ | $t_{sust}$ |
|---|---|---|---|---|---|
| regular | 106 | 108 | 607 | 815 | 943 |
| act:small | 125 | 129 | 263 | 348 | 676 |
| act:med | 128 | 131 | 196 | 265 | 562 |
| act:huge | 130 | 131 | 150 | 226 | 584 |

**Table 6.2**: number of connected peers at equilibrium ($sust$), after equilibrium level has been reached ($postavg$), and time (in minutes) required to reach 80, 100 and $sust$ connected peers.

It should also be mentioned that, for the user starting his machine, the improvement of "only" 14% in efficiency in our simulations implies an application that is ready for use twice faster.

### 6.5.3 Getting the Community Running

Now that we know the average number of connected peers that the system is able to maintain in various settings of active domains, we can compare how fast the system reaches its "cruise level". The following simulations have been started with relatively few ($N_0 = 30$) connected machines (that could e.g. be the set of systems that has been up during the week-end), and let the system evolve to restore its equilibrium.

Table 6.2 summarises the result we obtain for those simulations. For each setting, it shows the average amount of connected peers the system can sustain (as measured in section 6.5.2) and the average time needed to reach that level for the first time ($t_{sust}$). The value of $t_{sust}$ identifies a "knee" point on the curve where the system has reached its equilibrium and now oscillates around the average value. We also measured the actual average value *past* the knee point (that is, for $t$ in $t_{sust} \dots 2000$), as reported in the *postavg* column[7]. So not only the active routers can help having more nodes connected on Mondays morning, but it can also cut to 60% the time required to rebuild the community ($t_{sust}$) in the regular network. If we are rather interested in how fast each setting can reach an arbitrary value, we can see in columns $t_{80}$ and $t_{100}$ that even the less optimistic scenario (act:small, with an average of 10 members per active domain) is more than twice as fast as the regular network, and that with larger active domains, we can even be 4 times faster.

Another benefit brought by active networking here is the size of the initial set required to actually bring the community connected. We reproduced the experiment with ($N_0 =$) 25, 20, 15, 10 and 5 machines initially connected to see how many of the 20 simulations could still "take off" and grow to the expected equilibrium value. Indeed, if new peers cannot find those "initial members" quickly, the initial members themselves might disconnect and we will end with an "aborted" community where no one can connect anymore.

With a regular system, things starts getting wrong with $N_0 = 20$, where 10% of the simulations aborted, and further degrades so that with $N_0 = 12$, we have less than 50% chance of seeing the community taking off. On the other side, a system with 100 active peers in a configuration similar to *act:med* could still take off in all simulations with

[7] We can observe here that postavg is systematically slightly above the average of section 6.5.2, which could be due to the shorter runs not being able to compensate for the oscillations amplitude

| act. $\backslash P_{dyn}$ | 0% | 10% | 20% | 30% | 50% | 70% |
|---|---|---|---|---|---|---|
| 0% | 72.4 | 67.9 | 64.7 | 58.3 | 33.3 | 20.3 |
| 5% | 83.4 | 81.6 | 79.8 | 78.3 | 68.5 | 51.0 |
| 10% | 86.3 | 85.6 | 85.1 | 84.7 | 80.8 | 74.7 |
| 20% | 90.2 | 90.1 | 90.0 | 89.3 | 86.2 | 85.7 |

**Table 6.3**: Impact of dynamic addressing on system efficiency for various ratio of active domain support.

$N_0 = 12$ and gives 90% and 60% of chances of a successful take off with $N_0 = 10$ and $N_0 = 5$ respectively.

### 6.5.4 Other Affecting Parameters

The important random variable through these simulations is the probability of finding an online machine in a given time period $T$. The different scenarios we investigate in this section all alter the 'default' probability.

It is clear, too, that this probability depends on how many different addresses we can test during period $T$. The delay between two connection attempts, for instance, directly influences the percentage of time spent online, regardless of whether or not we have active nodes.

Note that, in most implementations, the peer will scan several addresses in parallel. There is however, a maximum amount of attempts that we could do on a given platform, meaning that e.g. we could have an average number of scanned nodes on a $\tau$ time slice that is $k$ times higher than what we observe in our simulations. Simulation results can still be applied to such a system if we assume that the individual probability for a node to be online is actually $k$ times lower in the real system than in the simulation (e.g. a given node doesn't connect 2 hours every day, but rather 2 hours every $k$ days).

### 6.5.5 Dynamic Addressing vs. Active Domains

So far, we have assumed through all our tests that a peer leaving the system and then joining it again will always reuse the same address. However, an increasingly high number of ISPs only offer *dynamic* addressing to their clients: every time a machine connects to the Internet, it will receive one of the addresses from the ISP pool, that it will keep during its whole session, but chances are very slim that the same address is allocated twice in a row to the same machine, especially hours after the last session.

In the following simulations, peers have probability $P_{dyn}$ of being client of a *dynamic domain*. Those domains will assign a new address to their clients every time they connect.

If we assume that the community members (both online and offline) represent a fraction $k$ of the number of addresses available in the domain's pool, there is now a probability $(1 - k).P_{dyn}$ that an address we find in our history list no longer corresponds to a peer, but that it rather has been reallocated to a machine that doesn't run the P2P software.

The first row in table 6.3 shows the efficiency of the P2P community with $P_{dyn}$ varying from 0.1 to 0.7 and $k$ being fixed to 0.1 (i.e. the size of each dynamic domain's pool is 10

times its number of peers[8]) without any active router. As we can see, the system efficiency quickly degrades when we add more dynamic domains. We also observed a significant degradation of the frustration ratio and the average community size.

On the other side, adding *active* networks results in an improvement of the bootstrap efficiency and other studied parameters. Moreover, it is not important to have a high number of active domains to obtain a significant effect: 5 percent of domains being active is enough to gain 11% of bootstrap efficiency, but we need half the domains to be active if we want to gain another 11% of efficiency.

It is interesting to note that a dynamic domain that is capable of storing information for active packets will behave here like a static, active domain. Indeed, the fact that new addresses are allocated every time a node connects is compensated by the fact we can obtain the address of a community member (if any) using any address previously seen in that domain.

Moreover, as depicted in Table 6.3, the presence of a few active domains in the system can compensate the degradation resulting from the presence of dynamic domains. Even with only 10% of the domains supporting the active packets, we can almost annihilate that degradation and keep the same efficiency regardless of the amount of dynamic domains in the network.

There is a new phenomenon that appears with $P_{dyn} > 0.4$. Some of the nodes might end up with only unassigned addresses in their history list[9], meaning that they have virtually no chances of connecting to the peer-to-peer network. From $P_{dyn} = 0.7$, the phenomenon can no longer be neglected since it will affect on average 2% of the members – a value that will quickly grow over 25% of the members when $P_{dyn} = 0.9$.

Note that even in extreme conditions such as $P_{dyn} = 0.9$, where a regular network couldn't maintain the community alive for more than a couple of hours, the presence of 10% of active peers allows the system to survive for arbitrarily long time, although with a degraded efficiency (around 66%) and a significant number of nodes that might end up with a useless history list (18%, against 26% without active nodes).

Indeed, the community model only takes into account the age of a neighbour when it picks the addresses it will archive in its history list for the next run. While an active address has more chance to be kept from one run to the other, the address from an active disconnected peer will not be preferred over a dynamic, connected peer, although the latter will likely be useless in the next run.

### 6.5.6 Avoid the Need for an Initial List

So far, we have illustrated that exchanging member addresses at active routers could help the peer-to-peer community to recover from unusually low activity and that its efficiency can be boosted through the improved probability of a successful connection attempt.

[8]While there are typically almost more clients than addresses in an ISP that does dynamic addressing, it would be utopian to assume all those clients already run our P2P software

[9]the simulator considers an address unassigned when it belongs to the pool of a dynamic domain, but not currently assigned to any member of this domain

Still, there is a major drawback of history-based peer-to-peer systems we haven't addressed yet: the need for an initial history list. An instance of the P2P software freshly installed on a machine has no other system to connect to. Most existing systems will overcome this through off-line process to obtain this list, such as publishing it on a forum or manual transmission through e-mails. In this section, we will review a collection of techniques enabled by the presence of WASP in the network that could avoid that need.

First, when the new machine's ISP runs WASP routers, we might build our initial list by looking for other peers in our own domain. Picking a random destination address and sending a probe should be sufficient here to hit the border router on which other peers of the same domain have advertised their address. Relying only on this automatic setup might work well when the software is actually "pushed" by the ISP (such as a GRID platform promoted by the managers of the domain), but it will generally be frustrating for lambda users downloading some P2P software regardless of whether their ISP supports WASP or not and whether there is already a sufficient user base in their vicinity.

We can envision another successful deployment scenario if the software vendor can afford setting up a WASP router at the entry point of his own network domain. In that case, the software can be designed so that, as a last option, it probes the vendor's domain for some initial contact node. One might valuably argue, though, that this is no different from running a pong server from an architectural point of view.

The good thing WASP routers offer here is that all we need to find is *active domains* that contain members, not members themselves. If the vendor cannot upgrade to WASP routers but WASP is sufficiently spread on the network, he could simply run a scanning software that would probe all ISPs and include a list of active domains as the "initial peers list" for the shipped software.

If we want to build a peer-to-peer system based solely on history lists, as opposed to systems where that list is just the most scalable and preferable mechanism before we fall back to a pong server, it is clear that we need an additional way of rebuilding the list when it is empty or useless. Once again, thanks to the presence of WASP router, the process of scanning the network for other peers is greatly simplified by the fact that we simply have to send a packet to any address (even if not currently assigned to a running machine) of an active domain where another peer is running to get a match. Knowing that, we can opt for different techniques that will gather addresses to test:

**netstat:** The P2P application might periodically run a netstat-like tool to learn the current connections the hosting system has with other machines, then send probes to those locations to see whether peers can be found.

**web browsing:** Rather than waiting for the user to establish connections, we might import the history of previous connections from e.g. the user's browser. We can then build a list of destination addresses out of the URLs and check if we can see any active router.

**address book:** Another potential source of peer addresses would be the address book of the user's mailing application. Out of "John.Smith@BigISP.com", we can extract the IP block associated with "BigISP.com" and send active probes to see whether there are online peers in that location.

The drawback with the "web browsing" approach is that there are unfortunately little chance that popular web servers are co-located with potential peers. There is a way an active "server" site could be helpful if it is popular enough. Indeed, we could technically use a website like slashdot or google as a rendez-vous for machines looking for a peer to join (which might visit the site too). It would however require the community to proactively scan for active routers frequently visited by users and to maintain ephemeral state present by periodically refreshing on routers that would otherwise never carry traffic for our P2P activities.

Comparatively, the "address book" scheme is more likely to point us to domains that host *users* rather than services. With a simple component filtering out most frequent webmail providers, we can concentrate our search efforts on parts of the network that could be running compatible P2P software.

## 6.6 Enforcing Registration Fairness

In applications like peer-to-peer community discovery as we presented above, it is of increased importance that the actual *protocol* is respected by the different parties. Amongst other things, we want to guarantee that:

1. $X$ cannot prevent other peers from appearing in the list if they registered themselves
2. $X$ cannot insert "fake" IP addresses in the list[10]
3. on sufficiently large timescale, all machines have similar chances of appearing in the list of machines that will be returned in probe replies.

Clearly, if we leave WASP unrestricted, there is no chance we can enforce such kind of rules. Indeed, because of the nature of community discovery application, the store's key needs to be *well-known* (e.g. a hash of some community name) and we cannot opt for some kind of "protected key" this time. Note however, that the problem isn't specific to WASP and that even if we implemented the "probe" and "register" algorithms as new ESP instructions rather than using the bytecode interpreter, we still couldn't prevent an attacker from using e.g. `COUNT` or `COLLECT` operations to alter the list we're gathering.

What we need is to ensure that *only the correct protocol* code can be applied to the corresponding key. With pre-compiled operations, this could be achieved for instance by using a part of the key to encode restrictions on the set of operations allowed.

### 6.6.1 Hash-Requesting Packets

We can implement a similar protection in WASP by hashing the bytecode into a $n$-bit key and use a specific bit pattern (e.g. a $k$-bit prefix, with $n + k = 64$) that is automatically added to the code hash to form the "private key". We then modify the implementation of `LOOKUP`, `INSERT` and `MAP` microbytes to guarantee they will abort packets that try to use a key using the $k$-bit prefix reserved for private keys. This will fullfill requirements

[10] we assume here that we can at least rely on the ISP that owns the WASP router for ensuring that source addresses aren't forged.

Listing 6.1: pseudo-code for fair `peer_adv` election process on a WASP private entry

```
1   if (MAP(private) == CREATED) {
      private.goal = random();
      private.list[0] = ip.src;
      done;
    } else {
6     addr = ip.src;
      d = addr XOR private.goal;
#repeat i=0..k
      if (private.list[i] XOR private.goal > d) {
        tmp=private.list[i];
11      private.list[i]=addr;
        addr=tmp;
        d = addr XOR private.goal;
      }
#endrepeat
16    done;
    }
```

(1) and (2) in our list (provided that the information added in the list comes from the IP header rather than from some packet variables), but it doesn't enforce fairness by itself.

Given a set of machines $P_1...P_n$, and a WASP-based protocol that allows only $k$ addresses to be stored in a list, a simple way to offer each machine a fair chance of appearing in the list would be to impose nodes to wait for a random delay ranging from $0.5\,\tau$ to $1.5\,\tau$ ($\tau$ being the lifetime of ephemeral entries – typically 10 seconds) between two registration messages, therefore randomizing the first $k$ nodes that will be allowed to appear in the list at each entry expiration.

Unfortunately, we have no way to prevent an attacker from issuing more traffic than expected, and $k$ cooperating attackers could then deny listing for regular peers by simply using an inter-registration interval sufficiently small compared to $\tau$. This is an important concern, since they will by this mean become the preferred contact nodes for an important share of the community. In most P2P applications, peers joining the network via such compromised contact node will have no way to tell whether they're actually using the real network or a attacker-controlled fac-simile of that network.

To avoid those risks in the community discovery protocol (and in any other election-style protocol based on WASP), we suggest that the membership registration bytecode first detects whether the `MAP` opcode has created a new entry or not and generates a *random goal* for the current round. Whenever a new address is to be added in the list, we then compare the distance between the new address and the goal (e.g. using a XOR instruction) with distances of addresses that are already in the list. The new address is then only allowed to replace another one if it has smaller distance.

Note that we can easily support independent elections for many communities without altering election code by simply hashing a *community key* in addition to the bytecode.

### 6.6.2 Accessing Election Result

The `peer_adv`/`peer_probe` mechanism involved in the case of peer discovery differs from regular election protocol in the sense that the election result needs to be *public*. Not only the participants (in the current domain), but every peer, needs to know who has been elected.

A simple approach would simply be to extend the protocol presented in listing 6.1 so that each packet either participates in the election or grabs the result depending on some packet variable. The drawback of this approach is that it imposes that all the peers interested in the election result can only use the result in a way that has been foreseen by the protocol designer. For higher flexibility, we would prefer a mechanism that decouples *how the election is performed* from *what it is used for*, and that somehow (partially) *exposes* the state manipulated by the protocol.

It should be noted, too, that enforcing a fair election process would be useless if we just publish the election result in a well-known regular tag that any malicious packet can modify. When considering one-to-one or one-to-many communication patterns, we can work around that risk by generating a random tag as part of the protocol and delivering it to the other participant(s). In our context, however, we have no way to communicate all peers that could potentially lookup on a router $R$ for election result without also communicating the information to attackers.

The most promising alternative consists in embedding the election code in a special portion of the `peer_probe` packet, so that it isn't interpreted when the packet is received, but remains available for a special `MAP_ALT` opcode. The corresponding semantic is "please retrieve content that has been generated according to the included protocol". The information is then of course mapped in *read-only* mode, and not all the state may be available. The example presented in listing 6.1, for instance, requires that `private.goal` remains undisclosed. We envision that many other protocols could have similar requirements, and we propose that the *hash-request* option of WASP packets include additional flags indicating how the entry can be exposed to public packets. Potential options would of course include "do not expose" or "fully expose", but also "only expose first longword" and "expose all but first longword".

### 6.6.3 Practical Implementation of Code Hashing

#### One-Way Hash at Wire Speed

Performing one-way hash for an important number of packets on a routing equipment may quickly become a potential bottleneck in the implementation. While some network processors are equipped with security co-processors[IntelPB] that provide hardware implementation of MD5 and SHA-1[11], most will have to work with a software implementation.

In [Yue06], a pioneering paper in the domain, the authors have studied the performance of several cryptographic algorithms, including unkeyed SHA-1 and MD5 on the

[11] in addition to ciphering standards like 3DES or AES

IXP microengines. Good news is that hashing algorithms require no external memory lookups beyond fetching the plaintext and storing the digest. Moreover, unlike keyed message hash (e.g. MD5-HMAC), they can live without any sort of internal look-up table or per-flow state.

Software implementation of MD5 on the IXP2400, for instance, consumes about 600 instructions, and about 60% of the execution time is spent in regular ALU instructions (plus about 40% spent in fetching data from DRAM). Comparatively, the SHA-1 hashing is twice larger, spends a significant amount of time (10%) processing "load immediate" instructions (for initialization purpose).

MD5 hashing also outperforms SHA-1 by a factor 2 in terms of throughput. [Yue06] reports 1.2, 2.4 and 4.8 Gbps throughput with 1,2 and 4 microengines respectively. Note that since the MD5 algorithm is totally cpu-bound, the performance is completely independent on the number of threads doing MD5 computations.

Since the WASP interpreter itself is also cpu-intensive, it would probably be preferrable to off-load MD5 to a thread in a ME that rather performs memory-intensive operations, such as a forwarding table lookup component or the ESP processing microblock.

### A CRC-Based Alternative

In many contexts, the computation-intensive MD5 might be a strong blocker for the activation of private state support on a WASP router. On the other side, virtually all network processors offer a hardware implementation of CRC algorithms that could be used to produce a hash of the bytecode. Since the result of hashing is never disclosed to other equipments, each router indeed has the opportunity of picking the hashing method that best suits its owner's preferences.

Given a protocol $P$ with $H_P = crc(P)$, it is however trivially easy to fix attacking code $Q$ into $Q'$ such that $crc(Q') = H_P$[mdgray03w]. The idea is thus to hash *a mix* of the bytecode and of a secret bitstream so that the attacker ignores the result $H_P$ to be met. Note that simply prefixing or XORing the bytecode with some secret value offers poor protection, while interleaving bytes of plain bytecode with secret bytes seems to offer a decent protection.

The weakness of this mechansim is that the *same* secret is used for hashing *all* protocols. As a result, the attacker might use a simple protocol $A$ that installs a well-known piece of data in the state store, and then generates variants of another protocol $B$ that simply checks whether state has been created or not, and compare the content of the state with the well-known signature if anything is present. If the signature is found, that means that the attacker has found $B_i$ such that $hash(A) = hash(B_i)$ (with probability depending on the "quality" of the signature), which could substantially ease the discovery of the secret bitstream used to scramble the bytecode. If the attacker has full access to a 1Gbps interface, the $2^{32}$ packets testing all the possible CRC values could be sent in about 3000 seconds, which suggests the router's secret should be changed frequently enough to keep protocols' state truly private, but still kept for long enough to minimize the event of a protocol state "disappearing" due to a change of the secret bitstream.

## 6.7 Conclusion and Future Work

It is our belief that the currently existing peer-to-peer applications lack a scalable, robust and user-friendly way to let peers join the network, be it for the first or the $n$th time. We have however no doubt that peer-to-peer technology has now given enough proofs of its potential and that we are likely to see more and more applications and architectures involving P2P networks in the future.

Through this chapter, we have shown how the presence of programmable ephemeral state in the network could allow a significant improvement of P2P system performance, and that it might even allow us to distribute peer-to-peer software without having to dedicate online resource to its support.

In section 6.2.4, we mentioned the two-phased nature of joining a peer-to-peer system. The second phase (neighbours selection) is oversimplified in our simulations, and while we have good hope that our proposal could equally apply to e.g. a Chord ring, we still need additional simulations with an enhanced model taking into account ring maintenance algorithms to validate our belief.

While the CRC-based approach presented in section 6.6.3 sounds promising, we do not have, at the time of writing, more precise information about the robustness of this method, and one should certainly not implement it for purposes other than testing.

Finally, the approaches for initial list avoidance presented in section 6.5.6 clearly need more research before getting convincing. We have however good faith that they can be useful building blocks of a heuristic technique which – given sufficient WASP support from the participating domains – could strongly reduce the number of cases where installing the software requires extra user intervention.

# Chapter 7

# WASP and Beyond

*ANTS can handle harder tasks than WASP*


## Abstract

*It is tempting to propose more disruptive services relying on the presence of a generic key-value store on the router's fast path, such as* traffic rerouting *([Stoica02, Schmid04]). The potential applications of this "extended" WASP would for instance include the support of fast terminal mobility and load balancing for server farms. In this chapter, we rather focus on improving the feasibility of Internet-wide multicast, as most of the applications based on* `collect` *operations suggest that we're responding to a multicast solicitation.*

*Allowing WASP programs to change packet destination, however, is a radical decision compared to the conservative restriction we presented in chapter 4. We will investigate through this chapter what kind of compromise we could find in order to keep that rerouting extension "world-friendly" before studying of what help it could be to multicast distribution.*

*The use case of multicast through WASP also raised questions regarding the actual implementation and deployment of WASP in a production network. Whether it comes on a programmable line card, as a filter box or as a router "sidekick" box has implication on what we can practically do.*


## 7.1 Rerouting

One of the most fundamental principles of IP forwarding is the fact that packet destination is decided *once* when the packet is emitted by the source. While the actual path it takes can still be unknown (e.g. routers might reconfigure while the packet is on its way), the packet should eventually reach the machine corresponding to what the source has decided to be the destination and only that machine. There are, however, several cases where this is not the most preferable behaviour.

It is common practice in Web hosting, for instance, to run a cluster of servers behind a front-end switch or proxy responsible for load balancing. In that situation, changing

the destination of the packets at the switch may be sufficient to balance the load among servers without requiring clients to be exposed to the individual machines of the cluster[1].

Another strong example is the case of mobile terminals [Gopal03], where the same end-system $M$ receives successive *foreign addresses* reflecting its actual location, in addition to the *home address* that uniquely identifies it. Even with Mobile IP – a protocol that allows such foreign addresses to be dynamically allocated and bound to the home address, there are practical limits to the speed at which new foreign addresses can be assigned.

Drawing inspiration from "Internet Indirection Infrastructure" [Stoica02] and "Network Pointers" [Tschudin03] proposals, we wanted to integrate a generic way of *changing packets' destination* while they're on their way – what we call *rerouting* the packets – into WASP, with the hope that it would greatly extend the expressivity of the platform.

Finally, rerouting may offer help to support partial multicast on the Internet. Actually, an operator that offers multicast to his clients virtually offers no more than multicasting a flow from one of his client to his other clients. Without the support of a worldwide multicast-capable backbone, chances that we can receive e.g. CNN by multicast in Japan remains thin. With WASP and rerouting, clients of a multicast-capable (and WASP-capable) domain may cooperate and setup unicast-to-multicast rerouting at domain border so that they appear as a single client to CNN.

### 7.1.1 Issues with Rerouting

Because this is a major difference with the traditional IP model, special care needs to be taken to ensure such destination changes do not put the network, the router or the end-user applications at risk.

#### Multiple Lookups

Forwarding tablelookup is a complex operation, involving several access to high-latency memory and usually the use of ASICs or dedicated coprocessors. If we allow WASP packets to lookup that table more than once, there's a potential risk that the lookup engine is not sufficiently available to process other packets, leading to packets queueing at the interface, even if there are chances that high-end network processors could handle two IP lookups per packet. Problems mainly arise when rerouting is requested *at the output interface* (see Sec. 4.1), and especially when it appears that the new destination should be reached through *another* interface. Should we allow the packet to leave the router with that new address or should we instead give the packet back to the forwarding core of the router? None of these alternatives are satisfactory as they both allow a WASP packet to concentrate resource consumption on a single spot of the network, for instance by repeatedly requesting rerouting to another output interface than the current one (and

[1] Note that in more subtle scenarios the proxy running on the front-end may require to inspect application payload to pick the most appropriate router [Zhao05], in which case there's more than the target address to modify

therefore making a single packet stay in the forwarding core of the router until its TTL reaches zero).

**Looping Packets**

Letting the packet leave router $R$ on any interface with any destination address is not a more desirable alternative. In most cases, the network will recover this at the next hop $S$ by forwarding the packet on the shortest path from $S$ to the new destination, just like $S$ would have processed any packet to that destination, with the exception that here, the interface where the packet goes might be the one it comes from.

More malicious packets, however, could then "turn back" to their initial destination as $S$ has sent them back to $R$, forcing $R$ to send the packet to $S$ again, and making the packet "ping-pong"ing between two routers, increasing artificially the load on the link between them. Even if we ensure that packets never do such "turn back", larger loops can be built, concentrating resource consumption in a slightly larger area.

**Where Do My Packets End Up?**

The main concern for the end-user will be to ensure that packets still reach the expected destination, even when re-routing applies. When the sequence of destination addresses to the final target is given explicitly in the packet's variables, the offered service has the same security semantics as loose source routing in IP. When that sequence is retrieved from ESS, however, we need to ensure that no one is trying to abuse the end-systems to gain an intermediate position on a specific data flow. We are confident that protected/private tags should however help the end-user build reliable rerouting-based applications.

If we're to implement rerouting, it is clear from the points expressed above that we need to enforce restrictions on *where* those rerouting operations may take place and what subset of destinations is allowed at each "rerouting-enabled" location. As a primary restriction, rerouting is an *inter-domain* functionality and as soon as the "input" interface of an ingress router has processed a WASP packet, no other rerouting of that packet is allowed until the packet reaches the output interface of the egress router in the local domain. That way, rerouting becomes almost transparent to the operator: packets "enter" the domain with the destination that will be used along the domain until the last router has processed them.

### 7.1.2 Network-Friendly Rerouting

It should be reminded that there is a strong hierarchy between transit domain operators in the Internet, mainly due to economical customer/provider relationships. Only a few "Tier-1" operators provide world-wide connectivity and most ISPs cannot afford the services of a Tier-1. Instead, they establish contracts with their direct customers (namely "Tier-2") or with the customers of these laters ("Tier-3").

With the exception of Tier-1, an autonomous system on the Internet can thus be seen as an operator that pays his own providers to get connectivity and sells back that connectivity to its own clients. To cut down operation costs, operators at the same level in the hierarchy

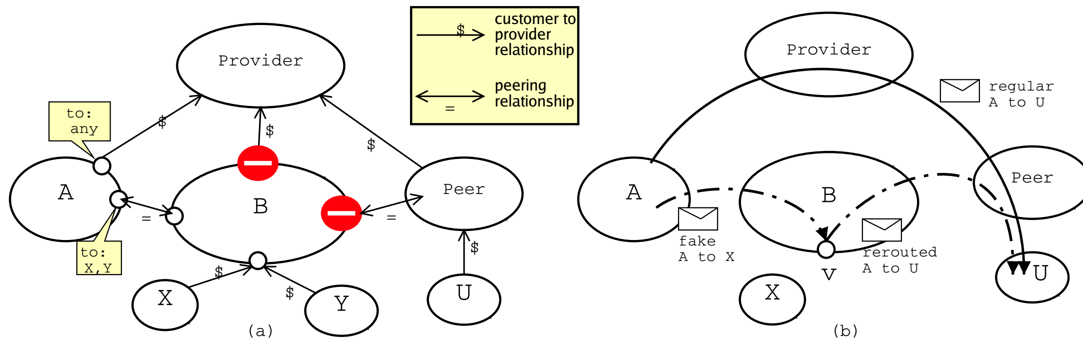



**Figure 7.1**: *Typical inter-domain policies for domain A (a), broken by blind rerouting (b)*

will typically try to establish *peering* contracts to exchange traffic between their respective customers without paying their provider to do so.

### Invitations-Based Rerouting

From the operator's point of view, the main difficulty in rerouting comes from the fact that, generally speaking, we do not want to allow any packet to be received from any link. Business agreements with other peer domains, for instance, may only allow a domain $A$ to use link to domain $B$ to reach $B$'s clients, but not to reach other peer domains or providers of $B$ (see Fig. 7.1a). Nowadays, most of these agreements are enforced by filtering routes advertised by BGP [Rekhter04] rather than by filtering packets, but blindly enabling rerouting of WASP packets could lead to situations where a packet leaves $A$ with a destination address falling in one of $B$'s clients and then reroute itself to another *peer* domain on $B$'s ingress router, thus cheating the business model (see Fig. 7.1b).

WASP overcome those problems by means of *invitations* left in the ESS by former packets. When a WASP packet executes on an interface VPU, it can create a new tag carrying its source address by means of the `invite` opcode. The binary pattern of the key used with *invite* tells the VPU that the value can be safely used as a redirection target. Depending on how the interface is configured, the `reroute` opcode will accept either any target value (e.g. for customers-ingress links) or will be restricted to tags and invitations (e.g. for any other link). This way, "ping-ponging" between domains is no longer possible if all egress interfaces restrict rerouting to invited destinations. An invitation to address $Y$ present on the interface means that the peer router for that interface has recently sent a packet coming from $Y$, and thus it should be able to route another packet towards $Y$ properly.

We can also prevent cheating on the business model if non-client (guest) packets can leave invitations only on *incoming* VPU of ingress routers. More precisely, if we make sure that WASP packets from peers and providers are tagged as guest when they reach the *center* (see Fig. 4.1) of their ingress router and that *invite* opcode is not allowed for guest packets, then a packet $p$ received on a non-client ingress interface that is targeted to a client domain can only be delivered to a client domain. By contradiction, suppose

$V$ is the first VPU where rerouting changes packet $p$'s destination towards a non-client destination $U$. This is only possible if an invitation to $U$ is present in $V$, however:

- if $V$ is on a core router, it cannot have the invitation since guest packets can only leave invitations on their ingress VPU (unless source addresses are spoofed by one of $B$'s clients).
- if $V$ is on a border router, it implies $V$ is bound to an outgoing interface, say *itf*, towards domain $U$. Note that $p$ can only reach that hypothetical interface if *itf* connects to both clients and non-clients domains (likely to be a configuration error).

In Fig. 7.1, note that $B$ is not protected if $A$ allows blind rerouting on its egress interface to $B$ (we could easily disable blind rerouting on output interface, anyway). If both $A$ and $B$ configure rerouting properly, malicious clients from $A$ cannot lead $A$ to a situation where it unwillingly misroutes packets through $B$.

Note that even if rerouting does not, by itself, allow source spoofing (which is the root of most DDoS attacks [Bossardt05]), it might disturb tools based on header-hashing for traceback of packets used to react to those attacks. Allowing inter-operations between rerouting and traceback might include storing previous destination in a field of the WASP packet or keeping traces of applied rerouting on routers and will be an interesting challenge for future work.

**Invitations and "Out"vitations**

At a given ESS location that allows rerouting, we will have two types of packets: packets that *leave* the domain through that interface and packets that *enter* the domain through that interface. Both are usually allowed to drop an invitation at the interface. However, we do not want the domain ingress to "reflect" packets by allowing an incoming packet to follow an invitation left by another incoming packet. Invitations will be carrying a status bit indicating whether they were left by entering or leaving packets and when a packet will try and reroute using an invitation, attempt to follow invitations left by packet "of the same type" will abort packet execution.

Alternatively, we could state that packets can only leave invitations when they're on *ingress* routers, not on *egress*. This is only equivalent when one considers two domains that both support WASP, and that an invitation from a host in $A$ can be dropped on $B$'s ingress router rather than on $A$'s egress router, which would be a severe deployment blocker.

### 7.1.3 Validating Source Addresses

In the discussion above, we've been assuming that clients from $A$ and $B$ couldn't spoof their source address. In other words, that $A$ and $B$ enforce firewalling rules such that clients of $A$ never send packets that have an address from $B$ in their "source address" field. This is required so that we can assume that receiving packet $p$ with source address $S$ on interface $V$ implies that $V$ can safely be used to send packets with destination $S$.

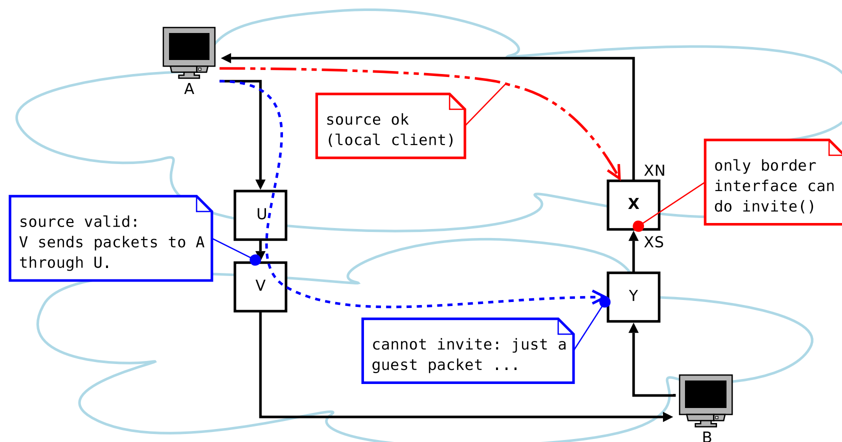



**Figure 7.2**: *Illustrating hot potato routing and troubles for invitation mechansim*

Unfortunately, several ISPs do not perform those minimal safety checks, and it is unlikely that everyone will eventually do so. As a result, it would be wise to ensure, when inserting invitation to $S$, that the receiving interface $V$ is actually the interface that we would use to send packets to $S$ as a kind of minimal authentication of the emitter. The importance of such checks is further increased when WASP router also operate on multicast addresses, as we will detail later in Sec. 7.2.5.

Note that, again, enforcing the "source validation" mechanism at the router level works poorly. In several cases it may occur that the network doesn't deliver packets from $S$ on router $R$ on the interface that $R$ uses to forward packets to $S$. For the purpose of load balancing, quality of service or any other trafic engineering mechanism, it could even happen that $R$ receives (or forwards) a packet from/to $S$ through multiple interfaces. We will thus instead distribute the responsibility of source validation on the edges of the domain. Each machine that receives an *invitation-capable* packet from another domain is required to confirm by the mean of a flag in the WASP header that the packet indeed originated from the expected interface.

It should be noted, however, that in order to validate a source address, the WASP component needs access to the inter-domain forwarding table. This is not a concern when the WASP component is located e.g. on a line card (and therefore is sharing memory with the IP forwarding component), but it may be a practical issue when considering an independent WASP filter installed before a border router.

### Invitations and Asymmetric Routing

The "invite/reroute" model presented above works fine as long as the routes from $A$ to $B$ and from $B$ to $A$ use the same routers. However, it is frequent on the Internet that routes are actually *asymmetric*, as the result of the "hot potato routing" principle. As depicted on Fig. 7.2, we suppose we have two large domains that have agreed on a peering relationship, one hosting $A$ and the other hosting $B$. Both domains are e.g. transatlantic bearer so when packets travel "horizontally", they need to be transported over long distances,

involving a higher operating cost. Since there is peering between the domains, however, delivering packet from one domain to the other (that is, travelling "vertically") is virtually free. Under these conditions, operators typically configure their routers so that packets that have to go to the peer network will go through the closest border router. The shortest path from $A$ to $B$ will thus cross the $U - V$ link while the path back from $B$ to $A$ will cross link $Y - X$.

In order to invite $B$, however, $A$ needs to leave an invitation on $X$ or $Y$'s border interfaces, but packets coming from $A$ usually don't cross those routers. Even if $A$ sends a message targeted to $X$ or to $Y$, this message will be delivered to application layer without going through the interface we're interested in. Moreover, in the case of $Y$, we have no guarantee that the inviting packet from $A$ will be received on the expected interface and $Y$ might conclude that this packet comes from a spoofed source. If instead we send our invitation for $B$ towards $X$, it will be received on an internal interface that does not allow invitation storage or usage.

The good thing is that WASP gives us the toolkit to detect the asymmetry as well as the address of $X$, the place where invitations should be installed. Yet, an additional mechanism is required to allow packets received on $X_N$ to be interpreted on $X_S$ rather than being immediately delivered to the upper layers of $X$. The protected invitations that are involved in multicast support in Sec. 7.2.4 could be a good candidate here.

## 7.2 Multicast to Small Group

Multicasting (many-to-many packet distribution) has been the centre of attention for number of research since Steve Deering's initial suggestion over 15 years ago. Its primary goal was to allow *any* source to send packets to a *group* address and let the network deliver the packets only to those nodes which subscribed to that group. With the initial multicast, as standardised by IETF, the source is no longer exposed to the (potentially huge) number of receivers, but several issues remain such as the allocation of globally unique group address and the (complex) protocol required in every router to map each group address to the corresponding set of interfaces.

For those reasons, even if multicast has been successfully deployed in national research networks, it is unlikely that the Internet will ever support a protocol that allows anyone to send a message to arbitrary groups. "Source-Specific Multicast" (SSM [Fenner06]) alleviates some of the problems related to group addresses by making the source address part of the group identification. The communication model is thus simplified to "one-to-many". This means that the receiver needs to know explicitly the address of the source to join (regular multicast could allow any number of sources and usually use a rendezvous mechanism to match sources and receivers), using a subset of "Protocol-Independent Multicast : Sparse Mode" messages[2] to inform routers of which (group address, source address) pair they would like to subscribe to.

[2]those messages where initially designed to allow receivers to switch to a source-specific tree to improve performance

### 7.2.1 Small Group Multicast

Yet, SSM does only address partially the problems of IP multicast. For core routers, it remains impossible to keep per-flow state that would be required if millions of users each want to run a multicast session with a couple of friends. This case of figure (also known as “one-to-few” distribution) is the target of the “Small Group Multicast” proposal [Boivie00]. Unlike other multicast flavours, SGM (and similar protocols such as explicit multicast [Boivie05] or IPv6 “Multiple Destinations” option [Imai02]) *do* expose the sender to the global group. SGM datagrams typically include the complete list of receivers in each message (and require virtually no state in core routers) and let the “branching routers” create duplicates of the packet as needed.

At each SGM-enabled router, that list is processed and partitioned to know which subset of the destination $S_i$ needs to be sent over each output interface $i$. This implies the router has to be designed to perform not only one IP table lookup per packet, but up to $n$ lookups (where $n$ is the number of destinations carried by the SGM packet).

While SGM removes the need for per-group state in the network core, this comes at the expense of a larger per-packet processing time, and added complexity that a 'regular' router will hardly handle. While Boivie et al. claim in [Allen03] that “*for a network processor such as the PowerNP, SGM is a simple matter*”, there is apparently no study of what SGM packet rate a network processor could handle nor “how small” the groups should be to avoid excessive use of the hardware lookup engines.

### 7.2.2 Application-level Multicast

Due to the difficulties to get a global multicast architecture working and deployed at the network layer, several solutions have been proposed to handle multicast at *application* level, making each receiver of a stream a *relayer* of that stream too such as in YOID (Your Own Internet Distribution) [Francis00], one of the pioneers in that area. More recent works such as SplitStream ([Castro03]) get further and ensures that each of the receivers contributes to a fair share of the retransmission effort and that the stream is still received correctly when a small portion of the nodes fail to deliver it properly.

Internet Indirection Infrastructure (*i3*, [Stoica02]) even makes application-level multicast retransmission transparent to the users by the mean of *triggers* installed on *indirection servers*. One of those triggers could well be used as the *group address* for the multicast retransmission and the sources will handle their packets to the closest i3 server, which will be in charge of locating the i3 server responsible of the destination trigger. Receivers then register themselves by adding their address to the list of destinations associated with the trigger in i3 servers.

In those schemes, the *source* of the stream is indeed off loaded compared to a pure unicast model, but the delay between initial packet emission and packet reception may increase more than we wish. Because packets are relayed by end-systems – perhaps behind asynchronous DSL – the retransmission time at one hop can become significant and the overall organisation of the distribution tree will play a significant role. Localisation-aware heuristics will be necessary for instance in the case of larger groups where several application-level relays are needed to keep the individual duplicating cost affordable at

each relay (otherwise the packet might well travel across oceans several times before it ultimately reaches its destination, leading to unacceptable delays for real-time applications).

### 7.2.3 Multicasting with ESP and Lightweight Modules

Even if ESP alone – on which WASP is based – has not the option of implementing a multicast function, K. Calvert et al. depict in [Calvert01] that multicast could be implemented using a simple *lightweight processing module* in addition to ESP.

The ephemeral state is used to perform the *probes computation* through which the source will identify the routers that play a strategic role in the distribution tree (e.g. those which will have to actively *split* the stream). ESP-enabled routers can perform simple computations such as "setup" which leave state indicating that a flow X goes through the router, and "collect" which, when used on S-Y path, allows to detect the routers that paths S-X and S-Y have in common.

The second component – *lightweight packet processing module* – is a small code package, strongly authenticated and loaded dynamically in the router that terminates in bounded time. Active networking research have presented numerous ways to support such code and modern network processor can efficiently support them. The authors advocate that a small number of lightweight modules could cover a large portion of application needs. In this case, the functionality required by the module is packet duplication. Each installed module consists of classifier rules that will invoke the processing instructions with specific arguments and parameters.

The information gathered with ESP probes indicates to the source where *dup()* lightweight modules should be installed. This module will catch stream packets and forward duplicates as needed. While this scheme has the advantage of inherently supporting topologies where not all nodes are capable of ESP or *dup()*lication, it is clear that, on 'branching' routers, per-flow state (e.g. classifying rule and forward list) is inevitable.

### 7.2.4 Building Small-Group Multicast with WASP

With WASP, we can replace the in-router per-flow state with tree-specific code in the packets. Once, e.g. a specific router $R$ has been identified as a potential branching point for a stream, packets of the stream can indeed contain WASP code that will perform specific actions when arriving on this node.

Similarly, WASP code can replace the need of a classifier by explicitly invoking the specialised function by means of a *unique key* for that function. The ability of WASP to rewrite packet destination addresses at network interfaces can here be advantageously used to replace the branching point address with a *node-local multicast address* (which is named through a key stored in the packet) that will generate a duplicate on every required interface and then, once the output interface has been reached, rewrite that multicast address into the address of the next branching point (also named through a key).

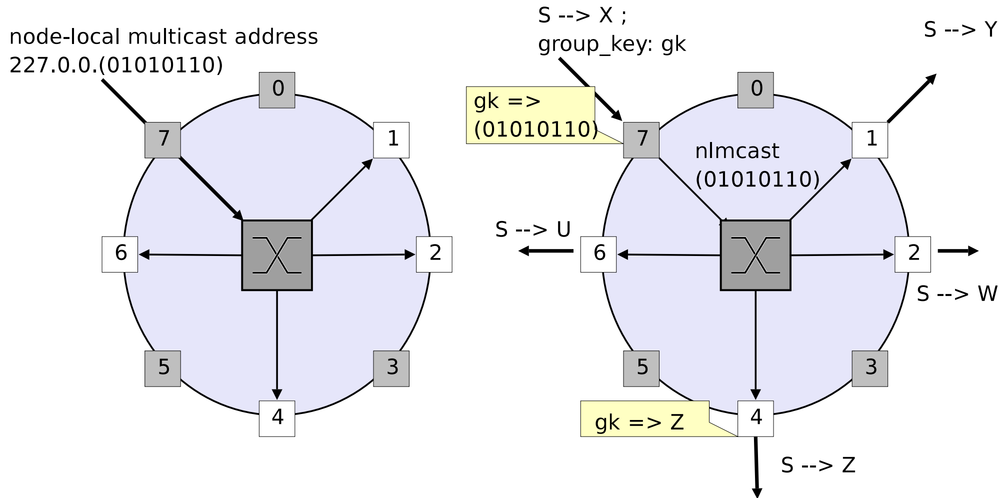



**Figure 7.3**: *Duplicating packets on a node using a local multicast address*

### Node-Local Multicast Addresses

Figure 7.3 illustrates the use of those node-local addresses with a router having 8 interfaces, meaning that we could encode on the last 8bits of a multicast address (say 227.0.0.x) the subset of interfaces that should receive a retransmission of the packet. E.g. 227.0.0.86 would be asking for retransmission on router interfaces 1,2,4 and 6 and 227.0.0.5 would be retransmitted on interfaces 0 and 2 only[3].

Of course, such addresses are not practical since the packet can only be handled by one router. That's where rerouting enters the game ... Upon reception of the packet at interface 7 (see Fig. 7.3.b), the WASP code will lookup the ephemeral store and resolve `group_key` into node-local address 227.0.0.86, which the switching logic of the router can interpret to duplicate packet on interfaces 1, 2, 4 and 6. At each of these interfaces, WASP post-processing happens and looks up the `group_key` in the interface-specific ephemeral store, where it will retrieve the address of the next branching router that has to process the packet.

The node-local multicast addresses, in this proposal, are never disclosed directly to other routers or end-systems. They are interesting for proof-of-concept implementations where WASP processing is added on line cards of an unmodified switch fabric. In a native implementation, the sender would use a *well-known key* that corresponds to the list of interfaces it wants to reach so that WASP processing can resolve this into a *virtual* node-local address for the switch fabric[4].

### An Example ...

Referring to figure 7.4, the message sent to members $B$, $C$ and $D$ carries WASP code used on the branching routers to take the following instructions:

[3]since $86 = 2^6 + 2^4 + 2^2 + 2^1$ and $5 = 2^2 + 2^0$

[4]to guarantee the expected behaviour of that well-known key, we could use *private* keys here

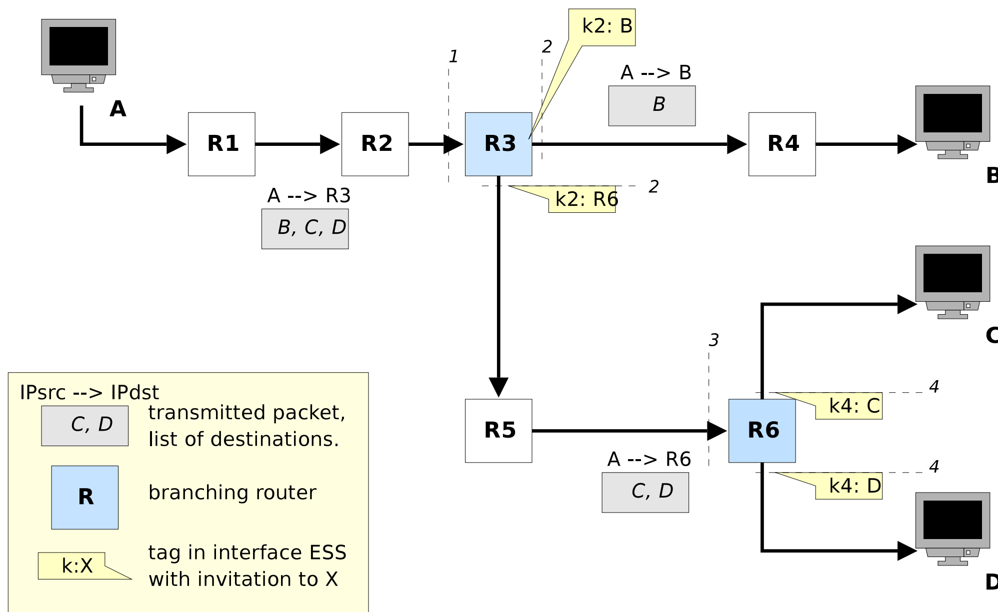



**Figure 7.4**: *Transmission of WASP-based multicast message to the small group "B,C,D"*

1. on its incoming interface, $R3$ will use key $k1$ to rewrite packet destination into the "node-local" multicast address corresponding to interfaces towards R4 and R5 (e.g. 227.0.0.20 if we assume the same 8-interfaces router as on Fig. 7.3).
2. the message is then delivered to those interfaces where we will do a second lookup and uses key $k2$ to follow invitations towards $B$ and $R6$ respectively.
3. when packet reaches incoming interface of $R6$, it uses key $k3$ to rewrite destination into the new node-local multicast address (e.g. 227.0.0.17) making the core enqueue packets on interfaces towards C and D.
4. destinations of the packets are finally rewritten into $C$ and $D$ at the output interfaces of $R6$.

### WASP-SGM without per-flow state

To enforce minimum network security, WASP cannot allow packets to reroute to arbitrary addresses. Instead, an *invitation* mechanism is used to ensure that destinations of rerouted packets *solicited* the rerouting and that, if the packet is already on an output interface, the new address will not cause trouble. The drawback of this approach is that, even if the source knows the new destination (for instance because it is written in the packet), it still *has* to use router-saved state because only those addresses will be trusted and allowed for rerouting by the router.

The reader will notice that the state stored in the router is however not stream-specific but that it can instead be reused by other streams. Each possible combination of output interfaces needs a 'rerouting invitation' so that packets can explicitly reroute to a local address saying "retransmit on interfaces 1, 4 and 5". If router has $N$ interfaces, $2^N$ such

entries might be required, which suggests an explicit way to name those combinations should instead be standardised.

Similarly, several flows that share a chain of branch point can reuse the "output renaming" entries, that is, a single rerouting entry for $R_6$ can be used on $R_3$ for every stream that first duplicates at $R_3$ and then at $R_6$, meaning that the required state is now depending on the number of *neighbours* the router has rather than on the number of end-system receiving multicast streams.

### 7.2.5 Pending Problems with WASP-SGM

**Network Friendliness ?**

The potential danger of node-local multicasting is to overflow the network with duplicates of a packet, by requesting forwarding on every output interface of the routers again and again. In the initial proposal for Small Group Multicast, such threat is avoided since all destination addresses are in the packet. The domain operator thus knows in advance how much load the packet might bring.

A possible fix for this problem is to include *credits* to the packet, indicating how many times it can be replicated. Routers will then have to maintain the 'credits' of each duplicate so that the sum of running packets' credits do not exceed the initial credits of the parent packet. Those credits exactly report how many remaining destinations a packet carries, and when a router will "split" the packet, WASP code will have to instruct how many credits are given to each duplicate. From the router's point of view, packet has an initial credit of $n$ destinations and it will use `reroute` with a key corresponding to a list of interfaces $(I_1, I_2, \ldots, I_k)$ and a list of child credits $(C_1, C_2, \ldots, C_k)$. What the router then has to do is to ensure the sum of $C_i$ doesn't exceed $n$ and then enqueue modified packets to $I_i$ with the corresponding new credit $C_i$. This can be done without modifying the instruction set of the WASP virtual processor, provided that the code for `reroute` interpretation detects the special case of a node-local multicast address and process credits accordingly.

The node-local multicast approach of WASP keeps the following advantages over a 'pure' packet cloning model as can be seen in SNAP:

- The switch fabric (or whatever technology interconnects interfaces) is used only once[5], while a collection of "clone" commands could potentially generate multiple packets that differs even in payload.
- A single router will never output more than one packet per interface when it receives a packet, while cloning could potentially generate many packets to be sent to the same destination.

[5]This assumes that the switch fabric is capable of delivering frames to a set of destinations. CSIX-L1 standards[Csix00] provide several ways to do this (all of which are optional), but we should stay aware that, generally speaking, it is not possible to send a single frame to an arbitrary set of ports. What is defined in CSIX-L1 are the following multicast modes: 1) deliver to arbitrary set for 16 contiguous ports; 2) deliver to two arbitrary ports; 3) allocate a 'group ID' that maps any arbitrary subset ($2^{21}$ groups available, registration mechanism not covered by the standard)

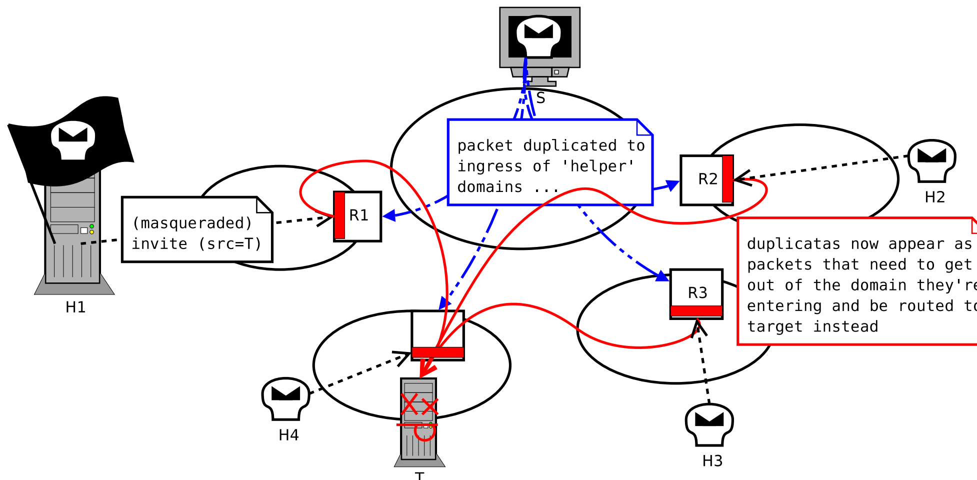



**Figure 7.5**: *Building a DDoS attack using WASP rerouting and packet duplication*

- A packet duplicate can only "leave" the router if proper invitation or validation token is known by the source.

### DDoS Toolkit ?

To exploit the packet duplication feature to overload a single destination, an attacker needs to be "invited" by a sufficiently high amount of fake destinations (that is, which will never receive the packets) and reroute to the actual target after duplicates have been generated. Those fake destinations, however, are only required to send a small amount of packets towards the initiator of the attack (that is, enough to maintain invitation state at the router, but not more), making thus packet duplication feature of WASP a potential threat if ISPs that allow them do not properly enforce a fair amount of credits per packet[6].

The success of the attack will not depend much on the *number* of attackers but more on how scattered they are on the network. It will also be necessary that those duplicates can be rerouted towards the actual target of the attack, which implies to find a WASP location $L$ where an invitation from the target $T$ can be followed. Under normal circumstances, such invitations can only be found on interfaces that points to the shortest path to $T$, but malicious hosts that use source spoofing could potentially install fake invitations. Fig. 7.5 illustrates that kind of attack and emphasises the need for a safer way to insert invitations.

Note that if we *validate* source addresses before we allow invitations to be left in routers (as suggested in section 7.1.3), the attack is no longer harmful. Moreover, the *validation* mechanism do not need to be implemented by all ISPs to be efficient provided that it is implemented in all WASP routers. Indeed, invitations from $H1$, $H2$ and $H3$ are now detected as fake by $R1$, $R2$ and $R3$ and duplicated packets reaching those routers can no longer be rerouted towards $T$. Only $H4$ has the opportunity to insert a fake invitation, but this router will only receive *one* instance of the packet, regardless of how many helpers

[6] Since this is intended to be for *small* groups, enforcing e.g. a maximum of 15 destinations per packet seems reasonable ...

like $H4$ are present in the same domain as $T$ (e.g. the branching routers generate one duplicate per output interface, not one per target address).

### 7.2.6 Interconnecting Multicast Islands

The idea of *Small Group Multicast* was to offer multicast delivery without exposing the routers to the per-session state and without requiring important network knowledge in the end-system. While rerouting provided an interesting way to achieve a similar goal using the WASP platform, the *invitations*, initially required to ensure proper behaviour of packets rerouting, force us to install (and refresh) state in the routers. This state can be shared by multiple sessions, but only if the routers themselves have issued invitations, which in turn implies that routers are aware of registrations they forward and are capable of sending an invitation to a peer router for the sake of minimizing state.

The presence of invitations also implies that the source must in first place discover which keys should be used on each branching router. This may require that the source knows much more about the network than we would like. We will indeed be able to use a rerouting entry only if the expected branching router is met.

Alternatively to the SGM use case, we might wish to use rerouting to translate destination addresses into a group address when packets enter a stub domain that natively support multicast. We should note, however, that the invitations mechanism is once again hardly compatible with that use case. In order to have packets destination translated from a internet-wide address $U$ into a local group address $M$, we are indeed required to issue an `invite` instruction from a packet using source address $M$.

Alternatively to the use case of small-group multicast, [Hjalmtysson04] and [Zhang06] suggested approaches where application-level *agents* in the network would self-organize into an overlay and help relaying packets between multicast-capable domain. Rather than trying to turn WASP into a self-sufficient solution to multicast delivery, it could be more interesting to opt for a hybrid approach where WASP would be used by ISP-operated agents to locate each other and synchronize their efforts in multicast distribution.

Rerouting remains interesting at the border of a multicast-capable domain to translate unicast addresses into a group address at wire speed, but it would then be the role of the local agent to install invitations (e.g. using a super-packet and protected tags) on behalf of its clients or based on negotiation with peer agents in other domains.

WASP could also be useful in detection of intermediate multicast-capable domain that have a compatible agent, using techniques such as MagNet (see 6.3). When such a "peer" domain is detected, we can set up a tunel relaying multicast packets through in-between unicast domains. Rather than letting the end-system drop invitations, we could guide it to its local agent who will in turn search for a peer agent and install proper rerouting entries in border routers' state stores.

The kind of approach described above would keep WASP router free of any controversary feature that could break the "friendiness" properties, and it could allow a network operator to upgrade the multicast "control" protocol without having to update routers (only the supporting agent would have to be modified). Still, it is clear that additional research is required here, preferably cooperating with teams having experience in multicast

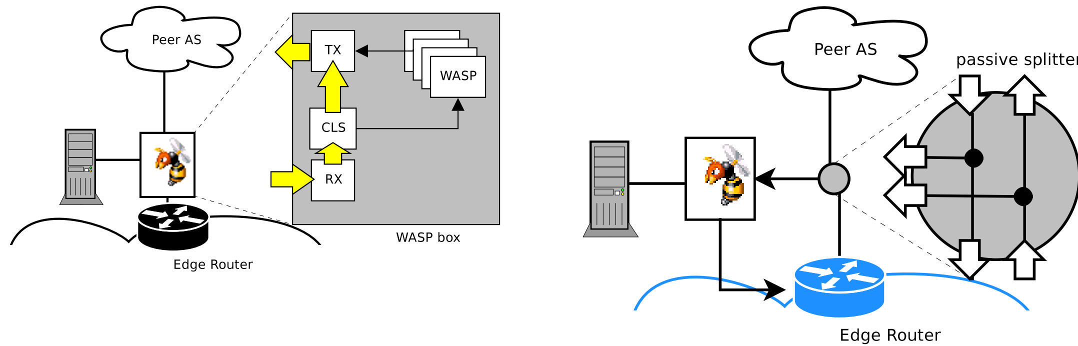



**Figure 7.6**: *Two techniques for extending a regular router with a wasp box (a) in a filter-like topology, (b) splitter-and-processor topology*

overlays, to estimate the potential benefits of the proposed approach and identify potential deployment issue of such an hybrid scheme, including what kind of modification would be required in end-system clients.

## 7.3 Deployment scenarios

### 7.3.1 WASP-aware line card

Introducing WASP as a part of the processing on a router line card is probably the most interesting approach for many of the extensions presented in this chapter. More specifically, when the network processor implementing WASP also hosts forwarding table lookup, we have more opportunity to integrate invitations checking, rerouting request lookups and the destination lookup. To some extent, the result of a "rerouting" could be not only picking a new destination, but also instructing what kind of operator-defined specific functional block should process the packet.

The down side is that better integration also means more interferences from WASP processing on the bare forwarding performance. A high load on the DRAM bus due to a high WASP activity might badly affect IP tables lookup.

### 7.3.2 WASP filter

The functionality is implemented through a "WASP box" that interrupts the router link, as represented in Fig. 7.6a. As mentionned before, this is the setup we implemented for the IXP2400 network processor.

Interrupting the wire with an experimental equipment of course raise a number of questions regarding the availability of the connection. We can however observe that the ENP2611 card has a number of features that isolates its behaviour from the hosting PC. Among other things, it has an external power source for the ENP card that makes it inde-

pendent from its host's power supply and the network processor can continue to operate even if the hosting machine crashes.

We can also observe that the "splitting" and "forwarding" functionalities are clearly separated from the active processing. WASP is implemented on separate micro-engines that can be restarted without affecting the receive, classify, forward process. Most of the "mission critical" code blocks come from a well-tested library. A special care will be required when designing and implementing the classifier block, which is mission-critical as well, but requires custom code and inter-operation with the WASP blocks[7].

### 7.3.3 WASP in a non-intrusive test bed system

Some ISPs and network operators might still be reluctant to insert an experimental device on their customer's data path. One might then wish to attach WASP behind a passive splitter (TAP) box, as depicted on Fig. 7.6b. The customers' traffic is then completely protected from a misbehaviour of the WASP box, but we introduce a radical change in the semantic of the WASP service.

1. It is no longer possible for the WASP box to *filter* packets. Even with the `DROP` opcode, a duplicate of the WASP packet has already been delivered by the passive splitter to the next hop. Consequently, applications like *concast*, active QoS enforcers will fail to work properly.
2. WASP is no longer capable of collecting information about a path in a single packet. Suppose we write a WASP program that reads the current node's address, stores it in packet data and forwards, a WASP box behind a passive splitter will actually *fork* such packet, one copy mentioning the forking device as being on the path, and the other copy not mentioning it.
3. by requesting the return of a WASP packet, we no longer guarantee that the packet doesn't go any further. As a result, if an end-system sends a packet to discover the closest instance of a service, it will suddenly receive one copy for every provider rather than a single packet coming from the closest device.

It is clear that we cannot allow a box with such a different semantics to process every WASP packet without at least warning the end-user that it will receive a different service. At least, the WASP packet format should be extended with a flag indicating whether or not a program is suitable for execution on a device that isn't capable of filtering packets.

Unfortunately, the semantic difference is only the smallest issue we have to deal with. The splitter device is typically not equipped to *inject* traffic on the wire, which means that if we want to forward a modified packet after WASP processing, it has to be submitted again to the edge router. If the program ended with `FORWARD` opcode, that also means the same passive splitter will duplicate the packet again and that the WASP box will see it a second time. It would thus be required to alter the WASP packet format to include the identity of the last WASP-capable device which processed a given packet so that we can avoid processing the same packet endlessly.

[7]For instance, it would be wise to ensure that delivering a packet to a WASP block is aborted (rather than delayed) if the core-to-wasp buffers are full.

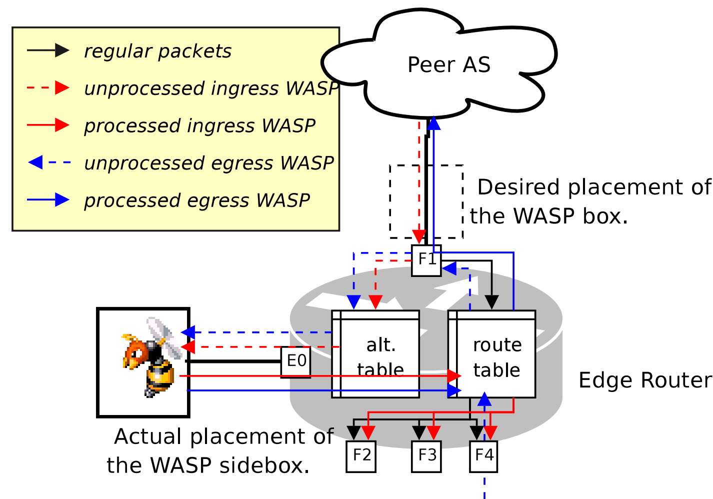



**Figure 7.7**: *WASP as a sidebox companion for the edge router, showing processing of ingress and egress WASP packets inside the router*

Finally, the “split-then-process” approach is a severe threat to the network safety. As we have discussed above, a FORWARD-terminated program actually result in a packet duplication on which we have no control, meaning that our simplest demonstration program sent along a chain of $n$ splitters will turn into a denial-of-service attempt of $2^n$ packets. In other words, by placing the WASP box like depicted on Fig. 7.6b, we’ve broken the WASP model and lost the invariant that WASP processing cannot result in an increase of the network load, on which most of WASP safety relies.

As a conclusion, placing a WASP box behind a passive splitter should be considered as a severe installation mistake.

### 7.3.4 Isolating WASP traffic on the router

An alternative to the use of a TAP box is to let the router’s firewall extract WASP packets from the regular flow. In the most favorable case, the firewall entries can redirect matching packet to an alternate routing table, which we will build such that all traffic is directed to the companion WASP box.

Figure 7.7 illustrates how we could configure such firewall rules to make a “side-box” behave as if it were a WASP filter. WASP packets received from the peer AS (in red) are separated from regular packets using a simple rule based on protocol number on interface $F1$ and an alternate route table that forwards all packets to the WASP box by default. The wasp box can then re-emit packets that need to be forwarded: since there is no rule to catch them on $E0$, they will simply use the regular route table and go on their way to the proper output interface.

We can apply a similar mechanism for packets to be transmitted to the peer AS (in blue on Fig. 7.7). An output firewall filter rule on $F1$ will intercept them and redirect them to the wasp box through the alternate routing table. However, after they have been processed by the WASP box, they will hit interface $F1$ again. We should then have *two* output rules on $F1$: a first one that catches packets seen for the first time and that will be

tagged before they are delivered to the WASP box, and a second rule that catches tagged packets and clear the tag before accepting them for transmission on the interface.

#### Unsupported redirection

The features mentioned above are available e.g. in the Juniper routers J,M,MX and T-series[juniper]. On other hardware, we might unfortunately lack the ability of redirecting packets based on firewall rules. Such routers could still partially support a WASP side-box if they are capable of dropping packets based on protocol version (e.g. CISCO routers supporting Flexible Packet Matching). The idea is then to use a passive splitter to deliver WASP packets (along with all the traffic) to the WASP box and discarding the WASP packets that directly come into the router to avoid the flaws we discussed in section 7.3.3.

#### Missing features

By placing the WASP box aside the router rather than on the main link, we however lost the ability to use WASP as a lightweight path monitoring tool. Indeed, since the box will not "see" all the traffic going trough the interface on which it is logically attached, it is unable to compute correct statistics on the number of received packets, drop ratio and instantaneous queue sizes.

Moreover, even firewalls that are capable of filtering "native" WASP packets may be unable to detect the WASP IP option used to piggyback WASP code on a regular TCP/UDP packet.

#### Multiple interfaces supported with one side-box

In figure 7.7, it is relatively easy for the WASP box to tell on behalf of which interface it should process packets. In the original ESP/WASP design, each interface is associated with its own Ephemeral State Store, and packets that take different interfaces cannot interfere with each other.

Since the WASP box only receives WASP packets here, we might want to implement several "virtual filter box" with a single side-box. This requires that we are capable of demultiplexing packets on the WASP box though they all came from the same physical interface. We will also need a way to tell which of the "execute on input interface" and "execute on output interface" bit should be checked for that interface.

When there is only one virtual filter, we can easily tell ingress and egress packets apart by means of the tag that is used by the output firewall filter to discriminate unprocessed from processed packets. Depending on router features, we could be restricted to e.g. DSCP[8] modification.

Encoding both the (un)processed state and the originating interface through DSCP values might become tricky and we might want to rather declare several alternate routing tables that would all direct traffic to the WASP box, but using different IP addresses for the WASP box. Each of those IP addresses would be resolved into a different MAC address,

[8] Differentiated Service Code Point – a 6-bit field in the "Type of Service" byte of IP header

allowing us to use the destination MAC address on the WASP box to infer the ESS we should operate on. This technique will unfortunately not only require several IP aliases, but also one alternate routing table per additional alias.

## 7.4 Conclusion

The option of changing packets' destination on the fly may allow the introduction of new services in the network. Through this chapter, we tried to adopt the point of view of a network operator and to suggest mechanisms that would permit such destination changes only in ways that couldn't break the rules that apply to normal packets.

We have unfortunately found ourselves limited by the fact that the current Internet cannot guarantee packets actually come from their source address, in which case our rerouting framework may be a severe threat of distributed denial-of-services attacks, especially when combined with multicast distribution.

The alternatives we can envision (see section 7.1.3 and 7.2.6) quickly require too much knowledge of the network either for the end-system or for the WASP box. In some cases, the actual implementation of the WASP functionnalilty on the router might even further restrict what is reasonably possible (e.g. a separated filter box may not have access to BGP tables).

This suggests that we are trying to solve at the network layer a problem that rather require cooperation of multiple layers to work properly. Sticking with the philosophy that WASP should not try to achieve complex things but should instead focus on what can be done at wire speed for cheap, the wisest move we could suggest is to allow packets to reroute according to invitations at WASP level, but to restrict setup of those invitations to trusted agents operated by the network owner.

The design of actual solutions involving WASP as a rerouting middleware for cooperating end-systems and operator agents require a better understanding of the existing mechanisms and will be dealt with in future work.

# Chapter 8

# Concluding Remarks, Future Directions

We proposed WASP, an active networking framework based on *Ephemeral State Store* that allows end-systems to install, retrieve and manipulate small pieces of information within the network. We have shown how, compared to the Ephemeral State Processing (ESP) router designed at University of Kentucky, the byte-code interpretation by a virtual processor (VPU) has enlarged the potential application field of WASP while keeping safety guarantees offered by ESP.

Unlike other active platforms, however, the language and the programming environment of WASP remain highly restricted. Compared to ANTS, for instance, which has the full expressiveness of JAVA language, WASP programs cannot exceed 256 bytes in length and the VPU doesn't allow any kind of loop. Yet, we have shown that useful services such as application-controlled dropping policies or jitter measurement can be built using the bytecode language to express operation over per-flow ephemeral state and generic statistics about the network card. Through this work, it appeared however that WASP gets even more powerful when it is used as *middleware* for other distributed applications such as equipment involved in a multicast session, hierarchical proxies for HTTP caching or agents supporting terminal mobility.

In addition to the "reference" x86 implementation, we provided and benchmarked an IXP2400 "proof-of-concept" implementation of the WASP interpreter as a Gigabit-filter box. We have shown that the performance we can get from WASP will greatly depend on the number of live keys in the store – and more importantly on the length of the chains colliding for a given entry in the hash table. Still, with moderate use of the state store, we were able to sustain 90% of the throughput of a pair of gigabit Ethernet ports with merely 40% of the processing power of the chip.

## 8.1 Towards a WASP Socket

The way we send and receive WASP packets in this work is only suitable for preliminary tests and measurements. Much like University of Kentucky provided an "ESP socket" option to attach count, compare, etc. programs to packets sent over a socket, we would

need system calls to have the kernel attaching WASP programs to packets and a way to send *standalone* WASP programs to a given machine without having to rely on raw sockets. The problem of retrieving data from packets received by an end-system also requires attention. Finally, application developers will need a library of validated WASP-based *transport protocols* so that once a "reliable multicast" socket is opened, all we have to do is send our data over the socket.

## 8.2 Rethinking the State Store?

Through this work, we have taken the state store "as is", without trying to modify its implementation or its semantics more than resizing its entries. However, our experiments with the IXP implementation show that the time required to walk a chain may quickly become critical under certain traffic patterns. Still, the amount of required SRAM needed to support full-speed operations gives the feeling that the performance gap between a multi-100baseT and a multi-gigabit equipment haven't really been anticipated when designing the ephemeral store. While the presence of hash pointers in SRAM may offer shorter "best case" when the use of the state store is low, scaling the store to maintain the "available for all" principle may require impractically high amount of (expensive) SRAM.

If we end up with a typical average chain length of e.g. 8 entries, we may even question the pertinence of such a pointers table and prefer a larger pointers table fully stored in DRAM. Evaluating the performance of such an approach against the existing implementation would be one of our priorities in future work. We also envision a hybrid approach where a chain (pointed from SRAM) could be "split" in several sub-chains (using a DRAM extension of the pointers table) when the chain length goes over a given threshold.

We are unfortunately strongly limited in the ways we can modify the state store if we want to keep a straightforward and lightweight cleaning process. Even arranging chains entries to follow a Most Recently Used order could lead to performance degradation due to the additional number of DRAM writes required to maintain the chain pointers.

Our latency and throughput experiments on the IXP implementation of WASP have also highlighted the need for a finer control of the entries creation rate. Not only a burst of entries creation will immediately slow down packets processing, but it also means that we will experience a sudden higher cleaning cost, which leads to longer buffering than what real-time applications may accept.

## 8.3 More on the "Best Effort" We Provide

There is an important (and unaddressed) decision concerning the "best-effort" nature of WASP/ESP that came to attention during section 5.7. For some protocols (such as robust collection of an important number of samples), it may not be acceptable that a router forwards a WASP packet without performing the computation (esp. if other packets of the same computation *have* been processed on that router).

On the other side, dropping packets that carry a WASP program *and* application-level data could be a bad idea. If the proper operation of the computation requires that all packets see the same set of routers, we could then prefer to *deactivate* the packet, clearing its execution bit, and simply forward it. That way, the end-system can still receive the packet, and the computation semantics is preserved.

Finally, some computation's output is independent of which specific routers do or don't process the packet, but dropping or deactivating the packet would have a significant drawback. This is for instance the case of multicast retransmission (see [Calvert02]) and service discovery application (see section 6.3).

Considering the possible combination, the best approach would probably consist of letting the application tell – via a proper control bit in WASP header – what is best to do with such "off-profile" packets.

## 8.4 The Role of WASP in Autonomic Networks

The recently introduced discipline of Autonomic Networks aims at evolving the Internet towards an architecture that could be both more resistant to outages and attacks and more efficiently managed, mostly through strong automation of configuration and optimisation tasks.

For many, the actual implication of "autonomic" paradigms on the actual network architecture is still fuzzy, but all acknowledge the need for self-configuring, self-optimising and self-healing devices and networks. By sensing its own state, and via learning and inference algorithms, the network management component should be able to detect changes in its own structure or in its "environment" (peering domains, submitted traffic, etc), and react accordingly to preserve high-level objectives.

To many aspects, the research done in active networks these last years may provide an interesting substrate for those kinds of problems, as we detailed in [Jelger06]. Among other things, the attempts to make active networks self-configuring and the mechanisms for code deployment can be useful building blocks for a new autonomic network architecture.

We believe that WASP (or a similar service) could be another key building block for these architectures, providing a simple mechanism for querying peer nodes' capabilities, discover helper agents in the network and perhaps even taking care of information gossipping or aggregation in a scalable fashion. We hope to have the opportunity to further explore these applications of programmable ephemeral state in the context of the ANA European initiative.

## 8.5 Towards User-Friendly Rerouting in WASP

Through this work, and especially through chapter 7, we've proposed applications based on an ever-updated virtual processor. At the time of writing, opcodes such as `reroute`, `invite`, `expose` and the corresponding mechanisms haven't been implemented yet. Thanks to the hierarchical nature of inter-domain routing, we have shown how rerouting

can be made loop-free and can adjust to existing traffic peering policies by means of *invitations* left in the ESS.

The success of applications based on rerouting will however strongly depend on whether the end-systems can ultimately trust those invitations or not. Most of the security of such rerouting-based applications depends on the fact that the tag used for rerouting is either unknown or unmodifiable by malicious third-parties. Enforcing such secret while making sure regular members of the application *know* the "secret" tag will require cryptographic aspects that we haven't investigated so far.

With a look back at all the technical difficulties we identified about rerouting (asymmetric routes, spoofed source addresses, inter-domain business rules), we should ask ourselves whether the proper abstraction for switching destination on the fly has been found. Having rerouting in WASP is appealing because WASP remains in lowest levels of packet processing. It could thus be applied on every packets crossing a router and it doesn't require identification of user flows – few other platforms can compete with that. More investigations are needed to see if the *invitation* mechanism could be made safer – e.g. allowing only invitation from super-packets and relying on domain-hosted services to install those invitations.

## 8.6 Benefits of WASP for Research Network Operators

Learning from experience and related discussions of other active platforms designers, we have tried through this work to take into account the expectations of network operators in our design. Guaranteeing that the presence of a WASP box in the network never becomes a nuisance for the operator has been our golden rule for many design choices.

However, it typically requires more than this to convince someone in charge of a production network. We tried to collect in this section arguments that could receive interest from research network operators such as Dante (operating the European GEANT research network).

#### Traffic statistics made automatically available to the researchers

The use of ephemeral counters helps distributed applications to measure their own traffic whenever required without the need for authorised access to a global statistics facility. However, as long as the key used for the measurement is kept undisclosed, third parties cannot eavesdrop others' traffic share. With the availability of per-interface global statistics, one can even estimate its relative share of the global traffic.

#### A more flexible platform for quality of service enforcement

Combining the presence of per-flow counters and precise timestamps from the routers, we can embed code that evaluates e.g. jitter or packet loss rate along a path and let the application adapt appropriately.

### Simplifying network programmability once for all

A research network operator is frequently solicited by network researchers wishing to implement their own solution by modifying code in the routers. This is unfortunately (for them, and probably hopefully for everyone else) not possible and the alternative of co-hosting an application-dedicated server with the router is a long, complicated and costly process. Moreover, it might even just be useless unless we also install firewalling rules and alternate route tables so that the traffic of the application is processed by the companion server.

Ephemeral entries in WASP can easily be used by a server (be it co-hosted with the router or running at another location) to advertise its presence to traffic from end-systems that might benefit from the new function. Operation around the core router is thus a one-host investment for any further request, but it also offers more transparency since only flows that *look* for the new service will be affected.

# Appendix A

# WASP Opcode Reference

| | | |
|---|---|---|
| $acc$ | the accumulator | used for most operations. |
| $ST_0 \ldots ST_15$ | stack content | |
| $TOP$ | Stack Pointer | $ST_{TOP}$ is the top-of stack ($ToS$). |
| $NF$ | Negative Flag | Highest bit of the last generated word is set. |
| $ZF$ | Zero Flag | Indicates the last generated word has all bits cleared. |
| $UF$ | Undefined Flag | Indicates the key used for last ESS operation was missing in the store |
| $X_B$ | index bank | Tells the memory bank used by the “X” window |
| $X_O$ | index offset | Tells the item pointed by the index register in $X_B$ |
| $X_S$ | data size | Tells the size (U8, U16, U32 or U64) of the current item in $X_B$ |
| $Y_B, Y_O, Y_S$ | alternate | same as $X_B$, $X_O$ and $X_S$, but for the alternate “Y” memory window |
| $imm$ | immediate | the value of immediate constant (for instructions supporting immediate values) |
| $PC$ | program counter | indicates the next opcode to be processed. |

immediate decode

| 00 | | 0 | operation |
|---|---|---|---|
| immediate8 | | | |

| | | | |
|---|---|---|---|
| BDEF | BUN-DEF | IMM | LAX |
| PULL | TRASH | SHL | SHR |
| LIX8 | LIX16 | LIX32 | LIX64 |

branch decode

| 00 | | 1 | iz | in | z | n |
|---|---|---|---|---|---|---|
| immediate8 | | | | | | |

memory decode

| 01 | y | x | c | operation |
|---|---|---|---|---|

| | | | |
|---|---|---|---|
| NOOP | IN-SERT | LOOK-UP | SWAP |
| LOAD | STOR | YLOD | YSTO |

ALU decode

| 10 | | ns | na | operation |
|---|---|---|---|---|

| | | | |
|---|---|---|---|
| ADD | SUB | OR | AND |
| XOR | NOT | min | max |

misc. decode

| 11 | operation |
|---|---|

| | | | |
|---|---|---|---|
| SSL | SSR | INC | DEC |
| SGN | SAX | PSH | POP |
| FWD | DROP | RET | ABRT |
| RND | MAP | EXT | |

note: only opcode page 0 of misc. operations is used so far. We still have 48 unassigned opcodes ...

## A.1 Control Operations

**ABORT** Current mapped ESS entries are *not* written back and packet "error" flag is set.

**FORWARD** Write back mapped entries. Packet will continue to its destination.

**DROP** Write back mapped entries. Packet is discarded by the node.

**RETURN** Write back mapped entries. Packet *source* and *destination* addresses are swapped. Packet *execute* flags are shifted, and packet *reflect* flag is set.

**REROUTE** $get()$ retrieves one word from ESS used as new destination for the packet. Mapped entries are written back. See section 7.1.2 for details.

All these operations ends packet evaluation. If the program executes past the end of actual code without encountering one of these operation, packet aborts with `VCOD_OUTOFBOUNDS` error.

## A.2 STACK Operations

**PUSH** $TOP \leftarrow TOP - 1$ ; $ST_{TOP} \leftarrow acc$
`VSTK_FULL` error occurs if $TOP$ is zero.

**POP** $acc \leftarrow ST_{TOP}$ ; $TOP \leftarrow TOP + 1$
`VSTK_EMPTY` occurs if $TOP$ is already larger than 15.

**TRASH** $TOP \leftarrow TOP + imm$
the resulting value of $TOP$ saturates at 16 (empty stack).

**PULL** $acc \leftarrow ST_{TOP+imm}$
`VSTK_OUTOFBOUNDS` error occurs if $TOP + imm$ is larger than 15.

## A.3 ALU Operations

Prior execution, $b$ is loaded with $ToS$ and $a$ with $acc$, then one of the following operations are taken.

**ADD** $b \leftarrow b + a$

**SUB** $b \leftarrow a - b$

**OR** $b \leftarrow a$ BITWISE OR $b$

**AND** $b \leftarrow a$ BITWISE AND $b$

**XOR** $b \leftarrow b$ EXCLUSIVE OR $b$

**NOT** $b \leftarrow$ BITWISE NOT $a$

The flags $NF$ and $ZF$ are modified to reflect the value in $b$. The result is then written back in $acc$ unless the `NOACC` modifier is set. The stack pointer is incremented (pop) unless the `NOSTK` modifier is set.

### A.3.1 Miscellaneous ALU Operations

These operations update $NF$ and $ZF$ flags according to the new value of the accumulator. `NOACC` and `NOSTK` modifiers do not apply here.

**SSL** shift $acc$ left by $X_S$ bits (e.g. 8, 16, 32 or 64)

**SSR** shift $acc$ right by $X_S$ bits

**INC** $acc \leftarrow acc + 1$

**DEC** $acc \leftarrow acc - 1$

**SGN** extends a $X_S$-bit signed value in $acc$ to 64-bit.
E.g. with $X_S = 8$ and $acc =$ `0xf1`, we would end up with $acc =$ `0xffff ffff ffff fff1`.

**SHL** shift $acc$ left by $imm$ bits.

**SHR** shift $acc$ right by $imm$ bits.

**IMM** $acc \leftarrow imm$

**RND** fills $acc$ with 64 random bits.

# A.4 Branch Operations

These operations alter the program sequence through $PC \leftarrow PC + imm$. No backward jumps are allowed: the resulting $PC$ is always greater than the previous one. The immediate value indicates the *number of microbytes to skip*. A jump that goes over program size aborts packet execution with `VCOD_JUMPTOOFAR` error.

**BDEF** branch if $UF$ is cleared.

**BUNDEF** branch if $UF$ is set.

The "generic" branch (any opcode with `VJMP_ISJMP` in the "immediate decode" family) has 4 bits interpreted in the following way.

1. if `IGNOREZERO` is set, skip to step 3.
2. don't branch if `ISZERO` doesn't match the value of $ZF$.
3. if `IGNORENEG` is set, skip to step 5.
4. don't branch if `ISNEG` doesn't match the value of $NF$.
5. branch unless stated otherwise previously.

Unconditional branches can be achieved by setting both `IGNOREZERO` and `IGNORENEG`. Note that the combination where both *ignore...* bits are cleared and both *is...* bits are set leads to a branch that is never taken.

# A.5 Memory Operations

## A.5.1 Memory Movement Operations

We will refer as $X$ (resp. $Y$) the memory location in bank $X_B$ (resp. $Y_B$) at offset $X_O$ (resp. $Y_O$) that is $X_S$ (resp. $Y_S$) bits long. $acc_{[0...X_S[}$ is the lowest $X_S$ bits of the accumulator. If `CLR` modifier is used, $acc$ is zeroed before the instruction takes place.

**STORE** $X \leftarrow acc_{[0...X_S[}$.
May fail with `VRAM_WRONGACCESS` if the bank is not writable.

**YSTORE** $Y \leftarrow acc_{[0...Y_S[}$.
May fail with `VESS_READONLY` if the mapped entry is not writable.

**LOAD** $acc \leftarrow X$.
$acc_{[X_S...63]}$ is cleared if $X_S < 64$.

**YLOAD** $acc \leftarrow Y$.

**NOOP** $acc \leftarrow acc$.

**INSERT** $tag \leftarrow X_{64}$ ; $put(tag, acc)$. $UF$ is updated.

**LOOKUP** $tag \leftarrow X_{64}$ ; $acc \leftarrow get(tag)$. $UF$ is updated.

**SWAP** $tag \leftarrow X_{64}$ ; $tmp \leftarrow get(tag)$ ; $put(tag, acc)$ ; $acc \leftarrow tmp$. $UF$ is updated.

After instruction is executed, `INX` modifier will advance $X_O$ by $X_S/8$ bytes in the bank. Respectively, `YNC` advances $Y_O$ by $Y_S/8$ bytes. If either $X_O$ or $Y_O$ goes over 32 bytes, it will wrap back at the start of the memory bank. In order to move $X$ register accross banks, a `LIX`*xx* instruction is required. `NOOP` | `INX` can be used if one wants to advance the $X$ register without performing a memory move.

## A.5.2 Index Register Manipulations

In the following, $bank(i)$ is an internal function retrieving the $i$th bank of the VPU's memory. Banks 0 to 3 map packet data and banks 4 to 7 map node environment variables. If one bank isn't available, the "dead" bank is mapped instead. Content of the dead bank is read-only and undefined. The $size(i)$ is an internal function decoding 0 into $U8$, 1 into $U16$, 2 into $U32$ and 3 into $U64$.

**MAP** $tag \leftarrow X_{64}$. $Y_B$ is linked to the corresponding entry in the ESS. $UF$ indicates whether this entry has just been created ($UF$ cleared) or was already there. $Y_O$ is reset and $X_O$ is automatically advanced by 8 bytes.

**SWYX** Swaps $X_B$ and $Y_B$. $X_O, Y_O, X_S$ and $Y_S$ are kept untouched.

**LIX*xx*** Load Immediate into X
$X_O \leftarrow imm_{0...4}$ ; $X_B \leftarrow bank(imm_{5...7})$ ; $X_S = xx$. We enforce that $X_O$ is a multiple of $xx/8$ bytes. E.g., after `LIX32(imm)`, $X$ is aligned on a 32-bit word in the bank.

**LAX** Load Accumulator into X
$X_O \leftarrow acc_{0...4}$ ; $X_B \leftarrow bank(imm_7 acc_{5...6})$ ; $X_S \leftarrow size(imm_{0...1})$. Note that the highest bit of the bank (indicating whether the bank belong to packet or node variables) is always taken from the immediate parameter rather than from the accumulator.

**SYSZ** Set Y SiZe
$X_S \leftarrow size(imm_{0...1})$ ; $Y_S \leftarrow size(imm_{2...3})$

# Appendix B

# WASP and ESP Packets Format

## B.1 Count Packet

```
COUNT (packet p)
  curVal = get (p.compValTag)
  if (curVal == ? )
    curVal = 0
  curVal = curVal + 1
  put (p.compValTag, curVal)
  if (curVal − p.thresh)
    forward p
  else
    discard p
```

(a) COUNT Pseudo code

| 32 bits | lword | xfer |
|---|---|---|
| Computation Value Tag (hi) | 5 | 10 |
| Computation Value Tag (lo) | | 11 |
| Threshold (immediate, hi) | 6 | 12 |
| Threshold (immediate, lo) | | 13 |

(b) COUNT Operands

```
vars key:long;

  LOOKUP|CLR, INC, INSERT, PSH;
  IMM(5),SUB,BL(1),FWD,DROP;
```

(c) WASP microbytes

## B.2 Compare Packet

```
COMPARE (packet p)
  curVal = get (p.compValTag)
  if (curVal == ? )
    put (p.compValTag, p.pktVal)
    forward p
  else if (p.op (curVal, p.pktVal))
    put (p.compValTag, p.pktVal)
    forward p
  else
    discard p
```

(a) COMPARE Pseudo code

| 32 bits | lword | xfer |
|---|---|---|
| Computation Value Tag (hi) | 5 | 10 |
| Computation Value Tag (lo) | | 11 |
| Current Packet Value (immediate, hi) | 6 | 12 |
| Current Packet Value (immediate, lo) | | 13 |
| Operation (immediate) | 7 | 14 |

(b) COMPARE Operands

```
vars key:long, val:long;

getval:  LOOKUP|INX,PSH,LOAD;
         BUNDEF(novalue);
check:   CMP,BG(novalue),DROP;
novalue: LIX64($key),INSERT,FWD;
```

(c) WASP microbytes

## B.3 Collect Packet

```
COLLECT (packet p)
   curVal = get (p.compValTag)
   if (curVal == ? )
      curVal = p.val
   else
      curVal = p.op (curVal, p.val)
   put (p.compValTag, curVal)
   chldCnt = get (p.chldCntTag)
   if (chldCnt == ? )
      abort
   chldCnt = chldCnt − 1
   put (p.chldCntTag, chldCnt)
   if (chldCnt == 0)
      p.val = curVal
      forward p
   else
      discard p
```

(a) COLLECT Pseudo code

| 32 bits | lword | xfer |
|---|---|---|
| Computation Value Tag (hi) | 5 | 10 |
| Computation Value Tag (lo) | | 11 |
| Child Counter Tag (hi) | 6 | 12 |
| Child Counter Tag (lo) | | 13 |
| Packet Value (immediate, hi) | 7 | 14 |
| Packet Value (immediate, lo) | | 15 |
| Operation (immediate) | 8 | 16 |

(b) COLLECT Op erands

```
vars key:long, val:long;

prepare:  MAP,LOAD,PSH,YLOD;
process:  ADD,YSTO|YNC,PSH;
cntchld:  YLOD,DEC,YSTO,BZ(+1),DROP;
forward:  POP,STOR,FWD;
```

(c) WASP microbytes

## B.4 Rchild Packet

```
RCHLD (packet p)
sibCnt = get (p.sibCntTag)
if (sibCnt == ? )
   sibCnt = 0
pktIdSeen = get (p.pktId)
if (pktIdSeen == ? )
   pktIdSeen = 1
   put (p.pktId, pktId)
   sibCnt = sibCnt + 1
   put (p.sibCntTag, sibCnt)
fwdCnt = get (p.fwdCntTag)
if (fwdCnt == ? )
   fwdCnt = 0
fwdCnt = fwdCnt + 1
put (p.fwdCntTag)
if (fwdCnt – p.fwdThresh)
   p.pktIdSeen = NODEID
   forward p
else
   discard p
```

(a) RCHLD Pseudo code

| 32 bits | lword | xfer |
|---|---|---|
| Packet Id (hi) | 5 | 10 |
| Packet Id (lo) | | 11 |
| Sibling Count Tag (hi) | 6 | 12 |
| Sibling Count Tag (lo) | | 13 |
| Forward Count Tag (hi) | 7 | 14 |
| Forward Count Tag (lo) | | 15 |
| Forward Threshold (immediate, hi) | 8 | 16 |
| Forward Threshold (immediate, lo) | | 17 |

(b) RCHLD Operands

```
vars key:long,  pktid:long;

      MAP,LOOKUP|CLR|YNC,BDEF(fck);
incr: INC,INSERT,YLOD,YSTO|YNC,JMP(1);
fck:  YNC;
      YLOD,PSH,IMM(3),CMP,BG(forw),DROP;
forw: POP,INC,YSTO,LIX32(@NodeID);
      LOAD|CLR,LIX64($pktid),STOR,FWD;
```

(c) WASP microbytes

# B.5 Rcollect Packet

```
RCOLLECT (packet p)
pktIdSeen = get (p.pktId)
if (pktIdSeen == ? )
   abort
sibCnt = get (p.sibCntTag)
if (sibCnt == ? )
   abort
if (pktIdSeen == 1)
   pktIdSeen = 0
   put (p.pktId, pktId)
   sibCnt = sibCnt − 1
   put (p.sibCntTag, sibCnt)
   compTot = get (p.compId)
   if (compTot == ? )
      compTot = p.val
   else
      compTot = p.op (compTot, p.val)
   put (p.compId, compTot)
else
   compTot = get (p.compId)
if (sibCnt == 0)
   fwdCnt = get (p.forwardCount)
   if (fwdCnt == ? ) fwdCnt = 0
   fwdCnt = fwdCnt + 1
   put (p.forwardCount, fwdCnt)
   if (fwdCnt <= p.fwdThresh)
      p.pktId = NODEID
      p.val = compTot
      forward p
   else discard
else discard
```

(a) RCOLLECT Pseudo code

| 32 bits | lword | xfer |
|---|---|---|
| Packet Id (hi) | 5 | 10 |
| Packet Id (lo) | | 11 |
| Sibling Count Tag (hi) | 6 | 12 |
| Sibling Count Tag (lo) | | 13 |
| Forward Count Tag (hi) | 7 | 14 |
| Forward Count Tag (lo) | | 15 |
| Computation Id Tag (hi) | 8 | 16 |
| Computation Id Tag (lo) | | 17 |
| Forward Threshold (immediate, hi) | 9 | 18 |
| Forward Threshold (immediate, lo) | | 19 |
| Packet Value (immediate. hi) | 10 | 20 |
| Packet Value (immediate, lo) | | 21 |
| Operation (immediate) | 11 | 22 |

(b) RCOLLECT Operands

# Appendix C

# Patches brought to ESP

The software package provided by Jiangbo Li[1] was in no way a "software release", but rather a "software escape". As such, we identified and fixed a collection of bugs and other mistakes that we report here.

## C.1 Erratum #69

According to erratum #69 in Intel's specifications update [IntelSU], some of the buffer elements of the MSF may corrupt data. In our context (128 bytes mpackets), elements number 4,9 and 13 are affected. We fixed `_packet_rx_free_all_rbuf_elements` so that it ignores those RBUF elements.

## C.2 Wrong CRC polynom

IXP hardware only implements the CRC polynom specified in Autodin/Ethernet and ATM AAL5 standards[2], while the Linux implementation (`lib/espcksum.c`) used CRC32C. For the sake of simplicity, we changed the software "reference" implementation to match hardware[3].

## C.3 ESS chains update

There is a miscalculation of the DRAM address in ESS chains update (+1 should be +8 to skip a whole longword in `super_find_create`) that causes ESS content to be trashed and exhausts ESS available space.

## C.4 No Queue Manager

The ESP application does not use any queue manager entity. This is usually not a problem as the only actual queue used is the packets free list and it needs no management beyond initialization. However, the queue manager is typically in charge of recycling buffers (esp. when several chunks need to be chained to hold a single packet). In some places (`dl_esp_core_drop` in the "core"

[1] esp-ixp2400-12-22-2004.tar.gz, md5sum `76a8de7fbf0ffb20ee9109a22f2a25c3`

[2] $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x^{1} + x^{0}$, [Kercheval94]

[3] We expect any network appliance to have hardware support of Ethernet CRC anyway

microengines), this has been handled by halting the processor and wait for debugger interaction, but in other places (`dl_sink` in the "classifier" microengine, and in RX microblock), we still enqueue those buffers on a scratch ring that is simply never processed.

The whole code has been patched to catch the (improbable) event of a complex buffer drop and interrupts execution if that happens, thereby guaranteeing that we won't leak buffers without notice.

## C.5 Overlapping Ring-Buffer

The size of ring-buffers for transmitting packets to the ESP rings was not properly encoded. Under high load, this could lead to messages (and buffer handles) dispatched to one context to overwrite an existing entry in another ring. This is probably the most evil programming error you could introduce on a network processor since it will (independently of the correctness of other components) drop some buffers randomly and introduce other buffers twice in the "free list". We fixed `esp_sram_ring_init` according to Table and enforced 512-entries buffer (16 KB) for each of the queues leading to ESP or WASP microblocks.

## C.6 Leaking Classifier

The classifier is not capable of detecting whether the `sram[put,...]` commands that enqueues a packet on a SRAM ring towards ESP or WASP microblock was successful. As a result, when load on a specific queue increases, the system will experience *buffers leakage* until the total number of buffers available in the system equals the size of the saturated queue(s).

Unlike *scratch* rings[4], SRAM rings do not provide an easy way for the microcode to determine *a priori* whether a "put" command will be successful. The program has first to wait for a signal indicating that the command has been completed and then check the result in a transfer register[5].

The current structure of the classifier do not allow us to easily fix this issue, but by duplicating the "get from scratch ring" code block, we managed to keep track of the *previous* buffer handle past the command that reads the next request, and check *then* that the previous buffer was correctly enqueued.Hopefully enough, transfer registers involved in the `sram[put]` and `scratch[get]` did not overlap.

## C.7 Trashing CRC on Header Update

The CRC instructions are a bit uncomfortable to deal with on the IXP 2400 network processor. Not only they require precise instruction latency between initialization, summing and read-back of the CRC remainder register, but there is only *one* `CRC_REMAINDER` register for the whole microengine, which means as soon as more than one thread on the ME is using CRC, we should no longer do external memory access without first saving the content of the CRC register in a GPR and restore it once the access is done.

---

[4] scratch ring state can be probed using `br_inp_state` instruction.

[5] see the notes on `sram_ring_put()` in [Johnson03] and section 3.2.56 of [IntelPRM] for additionnal information

There was a flaw in `_write_back_control_header` macro[6] where the header of the ESP packet was written first to DRAM after the header bytes have been checksummed, but without proper save/restore.

Note that a similar flaw remains in the packet fetching phase, but it only appears when the ESP packet is larger than 64 bytes, which never occurs with the current operations.

[6]in `esp-block/esp_core_util.uc`

# Appendix D

# Code Samples

## D.1 Microstore Reprogramming Benchmark

Listing D.1: IXP microstore reprogramming performance monitoring – microengine side

```
        .reg report,now,last,count
        local_csr_rd[timestamp_low]
        immed[last,0]
        immed[count,0]
        br!=ctx[0,just_sleep#]
stress#:
        .reg delta
        local_csr_rd[timestamp_low]
        immed[now,0]
        alu[delta,now,-,last]
        move(last,now)
        br_bset[delta,31,stress#]     // ignore timewraps
        alu[--,delta,-,4]
        bgt[report#]     // hmm. It took us long to loop.
        ctx_arb[voluntary]
        br[stress#]

// more than 64 cycles to run a loop of 9 instructions ?
// that sounds like we were interrupted. Let's report.
report#:
        move(report,delta)
        alu[count,count,+,1]
        br[stress#]

// only one thread is kept alive, to avoid interference
// at ctx_arb[].
just_sleep#:
        ctx_arb[kill]
        br[just_sleep#]
```

Listing D.2: IXP microstore reprogramming performance monitoring – reading back GPR

```
void ShowRegister(char* line)
{
  uint me,ctx,bank,reg;
  if (sscanf(line+2,"%d %d %d %x",&me,&ctx,&bank,&reg)!=4)
    fprintf(stderr,"usage: r <me> <ctx> <bank> <reg>\a\n");
  else {
    uint errcode=halMe_IsMeEnabled(me);
    uint enabled=(errcode==HALME_MEACTIVE)||(errcode==HALME_ENABLED);
    uint x=0xdeadbeef;
    fprintf(stderr,"reading ME=%d,CTX=%d, GPR-%c 0x%x...  ",
            me, ctx, bank?'B':'A', reg);
    if (enabled) halMe_Stop(me,0xff);
    errcode=halMe_GetRelDataReg(
        me,ctx,bank?IXP_GPB_REL:IXP_GPA_REL,reg, &x);
    if (enabled) halMe_Start(me,0xff);
    fprintf(stderr,"%x %s\n",x,UcLo_perror(errcode));
  }
}
```

Listing D.3: IXP microstore reprogramming performance monitoring – XScale side

```
void PutUwords(char* line)
{
  uint me,from,nb;
  if (sscanf(line+2,"%d %d %d",&me,&from,&nb)!=3)
    fprintf(stderr,"usage: U <me> <addr> <nb>\a\n");
  else {
    uint errcode=halMe_IsMeEnabled(me);
    uint enabled=(errcode==HALME_MEACTIVE)||(errcode==HALME_ENABLED);
    uword_T dumbcode[nb];
    uint i;
    for (i=0;i<nb;i++)
      dumbcode[i]=0xE000010000ull /* ctx_arb[kill] */;

    fprintf(stderr,"rewriting uwords %x..%x ME=%d...  ",from,from+nb,me);
    if (enabled) halMe_Stop(me,0xff);
    errcode=halMe_PutUwords(me,from,nb,dumbcode);
    if (enabled) halMe_Start(me,0xff);

    fprintf(stderr,"%x %s\n",UcLo_perror(errcode));
  }
}
```

# Bibliography


[Allan01] R.J. Allan and M. Ashworth : *"A Survey of Distributed Computing, Computational Grid, Meta-computing and Network Information Tools"*, Technical Report, CCLRC Daresbury Laboratory, 2001.
`http://www.ukhec.ac.uk/publications/reports/survey.pdf`

[Allen03] J.R. Allen, JR., et al. : *"IBM PowerNP Network Processor: Hardware, Software, and Applications"*, IBM Journal "Research & Development" vol. 47 No. 2 March/May 2003

[Amir98] E. Amir, S. Mc Canne, R. Katz : *"An Active Service Framework and its Application to Real-time Multimedia Transcoding"*, in Proc. of ACM SIGCOMM'98, Vancouver .

[Apple05] Apple Computer Inc. : *"MacOSX : Bonjour"*, Technology Brief, April 2005.

[Arlitt95] M. Arlitt and C. Williamson : *"Web server workload characterization: the search for invariants"*, in Proc. of ACM SIGMETRICS '95 .

[Avallone04] *"D-ITG v. 2.4 Manual"*, S. Avallone, A. Botta et al., University of Napoli, Dec. 2004,
`http://www.grid.unina.it/software/ITG`

[Baldi05] Mario Baldi and Fulvio Risso : *"Towards Effective Portability of Packet Handling Applications Across Heterogeneous Hardware Platforms"*, to appear in Proc. of 7th International Working Conference on Active and Programmable Networks (IWAN'05) .

[Baron05] C. Baron, Y. Luo and L. Bhuyan : *"Protocol Offloading Using an IXP2400 Network Processor"*, Intel IXA Summit 2005.

[Bassi02] A. Bassi, J-P. Gelas and L. Lefèvre : *"Tamanoir-IBP: Adding Storage to Active Networks"*, in Proc. of 4th IEEE workshop on Active Middleware Services (AMS'02) .

[Berson02] S. Berson, B. Braden, L. Ricciulli : *"Introduction to the ABONE"*, `http://www.isi.edu/abone/DOCUMENTS/ABoneIntro.pdf` , February 2002.

[Bhattach.98] S. Bhattacharjee, K. Calvert, E. Zegura : *"Self-Organizing Wide-Area Network Caches"*, in Proc. of INFOCOM 1998, pp. 600-608 .

[Bhattachrjee96] 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

[Bindels] *"The OS FAQ – How can i tell CPU speed?"*, P. Bindels, S. Martin et al. `http://www.osdev.org/wiki/Detecting_CPU_Speed` . Jan. 2006

[Bloom70] Burton Bloom : *"Space/time trade-off in hash coding with allowable errors"*, Communications of the ACM, 13(7): 442-426, July 1970.

[Boggs82] D.R. Boggs : *"Internet Broadcasting"*, Ph. D. thesis, Electircal Engineering Dept., Stanford, 1982.

[Boggs83] D.R. Boggs : *"Internet Broadcasting"*, Technical Report CSL-83-3, Xerox PARC Palo Alto, California.

[Boivie00] R. Boivie, N. Feldman, C. Metz : *"Small Group Multicast: A New Solution for Multicasting on the Internet"*, IEEE Internet Computing, may-june 2000 issue.

[Boivie05] R. Boivie, N. Feldman, Y. Imai, W. Livens, D. Ooms, and O. Paridaens, : *"Explicit multicast (xcast) basic specification"*, IETF Internet-Draft (draft-ooms-xcast-basic-spec-05.txt), Aug. 2003.

[Bolosky00] W. Bolosky, J. Douceur et al. : *"Feasability of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs"*, in Proc. of SIGMETRICS 2000, Santa Clara, USA, pp. 34-43 .

[Bond02] M. Bond, K. L. Calvert, J. Griffioen, et al. : *"ActiveCast: Toward Application-Friendly Active Network Services"*, in Proc. of DANCE 2002: 274-290

[Bos04] H. Bos and K. Huang : *"On the Feasibility of Using Network Processors for DNA Queries"*, in Proc. of Workshop on Network Processors & Applications NP3, Madrid, Spain, Feb, 2004.

[Boschi05] E. Boschi, M. Bossardt, T. Dübendorfer : *"Validating Inter-Domain SLAs with a Programmable Traffic Control System"*, in Proc. of IWAN'05

[Bossardt02] M. Bossardt, T. Egawa, H. Otsuki, B. Plattner : *"Integrated Service Deployment for Active Networks"*, in Proc. of International Working Conference on Active Networks (IWAN), 2002 LNCS 2546, pp. 74-86.

[Bossardt05] T. Dübendorfer, M. Bossardt and B. Plattner: *"Adaptive Distributed Traffic Control Service for DDoS Attack Mitigation"*. In Proc. of SSN 2005, April 2005, Denver, USA.

[Bowman95] C.M. Bowman, P. Danzig, D. Hardy et al.: *"Harvest: A scalable, customizable discovery and access system"*, Technical Report CU-CS-732-94, U. of Colorado - Boulder, 1995.

[Braynard02] R. Braynard, D. Kostić et al. : *"Opus: an Overlay Peer Utility Service"*, in Proc. of the 5th IEEE OPENARCH, pp 168-178, New York, June 2002.

[Caesar06] M. Caesar, M. Castro et al. : *"Virtual ring routing: network routing inspired by DHTs"*, in Proc. of ACM SIGCOMM, September 2006, Pisa, Italy, pp. 351 - 362 .

[Caesar06b] M. Caesar, T. Condie et al. : *"ROFL: Routing on Flat Labels"*, in Proc. of ACM SIGCOMM'06, Sept. 2006, Pisa, Italy, pp. 363 - 374 .

[Calvert01] S. Wen and J. Griffioen and K. Calvert : *"Building Multicast Services from Unicast Forwarding and Ephemeral State"*, in Proc. of IEEE OPENARCH'01 Anchorage, Alaska, USA, Apr. 2001.

[Calvert01b] : *"Concast: Design and Implementation of an Active Network Service"*, IEEE Journal on Selected Area in Communications, 19(3):426–437, March 2001.

[Calvert02] Kenneth L. Calvert, James N. Griffioen, and Su Wen : *"Lightweight network support for scalable end-to-end services"*, in Proc. of ACM SIGCOMM, 2002 .

[Calvert03] K.L. Calvert, J. Griffioen, N. Imam, J. Li : *"Challenges in Implementing an ESP Service"*, in Proc. of IWAN'03, Kyoto LNCS 2982, pp. 3-19.

[Calvert03w] *ESP implementation for Click and Linux routers*, The Activecast Research Group, 2003,
`http://protocols.netlab.uky.edu/~esp/download.html` .

[Calvert05w] *"ESP Instruction Pseudocode and Operands"* and *"ESP General Packet Header Specifics"*, K. Calvert et al., May 2005,
`http://protocols.netlab.uky.edu/~esp/document.html`

[Calvert99] Calvert, K. L., ed. : *"Architectural Framework for Active Networks"*, Version 1.0, Active Network Working Group, July 1999.

[Campbel02] A.T. Campbell, S.T. Chou, M.E. Kounavis, V.D. Stachtos and J. Vicente : *"NetBind: A Binding Tool for Constructing Data Paths in Network Processor-Based Routers"*, in Proc. of 5th International Conference on Open Architectures and Network Programming (OPENARCH'02)

[Castro03] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron and A. Singh : *"SplitStream: High-bandwidth multicast in a cooperative environment"*, in Proc. of SOSP'03, Lake Bolton, New York, October, 2003 .

[Castro02] M. Castro, P. Druschel : *"One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks"*, in Proc. of SIGOPS European Workshop, Saint-Emilion, France, 2002, pp. 140-145 .

[Cheshire05] Cheshire, S., Krochmal, M. : *"DNS-Based Service Discovery"*, Internet-Draft (work in progress), 2005.

[Cheshire05b] Cheshire, S., Krochmal : *"Performing DNS queries via IP Multicast"*, Internet Draft (work in progress) (2005).

[Clarke00] I. Clarke, O. Sandberg et al. *"Freenet: A Distributed Anonymous Information Storage and Retrieval System"*, in Proc. of Workshop on Design Issues in Anonymity and Unobservability, Berkeley, USA, July 25-26, 2000, LNCS 2009, pp. 46-66 .

[Cohen04w] *BitTorrent Protocol Specification*, Bram Cohen, 2004 `http://www.bittorrent.org/protocol.html`

[Csix00] "CSIX-L1: Common Switch Interface Specification-L1 1.0", Public Distribution, May 2000.

[DARPA99] L. Peterson (Editor). : *"NodeOS Interface Specification"*, DARPA AN NodeOS Working Group Draft, 1999.

[Decasper99] D. Decasper, G. Parulkar, S. Choi, et al. : *"A Scalable, High Performance Active Network Node"*, IEEE Network, volume 13, Jan 1999.

[Droms97] R. Droms : *" Dynamic Host Configuration Protocol"*, Bucknell University, March 1997. IETF RFC 2131.

[ethereal] Gerald Combs et al : *"Ethereal: A Network Protocol Analyzer."* `http://www.ethereal.com/`

[ethtool] David Miller et al. : *"ethtool - Ethernet diagnostic and tuning tool"*, `http://directory.fsf.org/All_Packages_in-_Directory/ethtool.html` , May 2005

[Fdida06] S. Fdida, I. Stavrakakis et al. "ANA Project Autonomic Network Architecture – Deliverable D2.1 First draft of routing design and service discovery", Sixth Framework Programme Priority FP6-2004-IST-4 Situated and Autonomic Communications (SAC) Project Number: FP6-IST-27489, `http://www.ana-project.org/autonomic/network/-deliverables.html` , Dec. 2006.

[Fenner06] B. Fenner, M. Handley, H. Holbrook and I. Kouvelas : *"Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised)"*, draft-ietf-pim-sm-v2-new-09.txt (work in progress)

[Floyd02w] Sally Floyd: *"Adaptive Web Caching"* `http://www.icir.org/floyd/web.html` , Sep. 2002

[Francis00] Paul Francis : *"Yoid: Extending the Internet Multicast Architecture"*, `www.aciri.org/yoid/docs/index.html` , April 2000

[Freedman06] M. Freedman, K. Lakshminarayanan and David Mazières *"OASIS: Anycast for Any Service"*, in Proc. of Networked Systems Design & Implementation, 2006, San Jose, CA, pp. 129 - 142

[Gelas02] J.P. Gelas, L. Lefèvre : *"Performance et dynamicité dans les réseaux : l'approche Tamanoir"*, in Proc. of JDIR 2002, Toulouse, France, Mars 2002 .

[Gelas] J-P. Gelas and L.Lefèvre : *"Gestion de flux TCP actifs"*, (unpublished work).

[George03] "Taming the IXP network processor", Lal George and Matthias Blume

[Gold05] B. Gold, A. Ailamaki L. Huston B. Falsafi : *"Accelerating Database Operators Using a Network Processor"*, in Proc. of 1st Int'l Workshop on Data Management on New Hardware (DaMoN), June 2005 .

[Gopal03] Gopal Racherlaa, Sridhar Radhakrishnanb, Chandra N. Sekharanc. : *"Performance evaluation of wireless TCP with rerouting in mobile networks"*, Computer Communications 26 (2003) 542 551.

[Gwertzman95] M. Seltzer and J. Gwertzman : "The case for geographical push caching", in Proc. of Hot Operating System, 1995 .

[Haas03] R. Haas, C. Jeffries et al. : *"Creating Advanced Functions on Network Processors: Experience and Perspectives"*, IEEE Network, 14 April 2003, pp. 46-54.

[Hawblitzel98] C. Hawblitzel, C-C. Chang, et al. "Implementing Multiple Protection Domains in Java", in Proc. of USENIX technical conference, June 1998 .

[Hicks01] M. Hicks, J. T. Moore, and S. Nettles: "Compiling PLAN to SNAP", in Proc. of 3rd International Working Conference on Active Networks, Sep. 2001 (IWAN'01)

[Hicks98] Hicks, Kakkar, Moore, Gunter and Nettles : "PLAN: A Packet Language for Active Networks" in Proc. of ACM SIGPLAN 1998 .

[HiFn04] *"HiFn 5NP4G Network Processor Product Brief"*, `http://www.hifn.com/uploadedFiles/Library/-Product_Briefs/5NP4G_pb_v1.pdf`

[Hjalmtysson04] Gísli Hjálmtýsson, Björn Brynjúlfsson and Ólafur R. Helgason, "Self-configuring Lightweight Internet Multicast." in proceedings of IEEE SMC 2004, Hague, Netherlands, September 2004.

[Hwang05] K. Hwang and Y.-K. Kwok et al. *"GridSec: Trusted Grid Computing with Security Binding and Self-Defense against Network Worms and DDoS Attacks"*. In International Workshop on Grid Computing Security and Resource Management (GSRM 05), in conjunction with the ICCS-2005, pages 187 195, 2005.

[IETF06] Cheshire and the IETF Zeroconf Working Group : *"Zero Configuration Networking (Zeroconf)"*,
`http://www.zeroconf.org/` .

[Imai02] Y. Imai : "Multiple Destination Option on IPv6 (MDO6)" , IETF Internet-Draft,
`http://www.ietf.org/internet-drafts/-`
`draft-imai-mdo6-02.txt`

[Imam03] Najati R. Imam : *"Implementation of an Ephemeral State Processor on the Intel IXP1200"*, thesis for master's degree, University of Kentucky, 2003.

[IntelAN] : *"Intel® IXP2800 Network Processor Optimizing RDRAM Performance: Analysis and Recommendations"*, Intel Application Note, August 2004

[IntelAP450] *"Interrupt Moderation Using Intel Gigabit Ethernet Controllers"*, Application Note (AP-450) Revision 1.1 September 2003 – Intel Press

[IntelAP453] *"Small Packet Traffic Performance Optimization for 8255x and 8254x Ethernet Controllers"*, Application Note (AP-453) – Intel Press

[IntelHRM] *"Intel® IXP2400 Network Processor, Hardware Reference Manual"*, Intel Press, July 2005.

[IntelPRM] *"Intel® IXP2400 and IXP2800 Network Processor, Programmer s Reference Manual"*, Intel Press, Order Number: 278746-019, July 2005

[IntelPB] *"Intel IXP2350 Product Brief"*, Intel Corporation,
`http://www.intel.com/design/network/prodbrf/303678.htm`

[IntelRdtsc] *"Using the RDTSC Instruction for Performance Monitoring"*, Intel Corporation, 1997,
`http://developer.intel.com/drg/pentiumII/appnotes/-`
`RDTSCPM1.HTM`

[IntelSU] *"Intel IXP2400 Network Processor – Specification Update"*, Intel Press, March 2004. Document Number: 301161-010,
`ftp://download.intel.com/design/network/specupdt/-`
`30116110.pdf`

[Jelasity06] M. Jelasity, A. Montresor and O. Babaoglu, *"The Bootstrapping Service"*, in Proc. of IEEE ICDCSW'06, Los Alamitos, CA, USA, July 2006, p. 11 .

[Jelger06] C. Jelger and S. Martin, "ANA Project Autonomic Network Architecture – Deliverable D.1.1 - State of the Art", Sixth Framework Programme Priority FP6-2004-IST-4 Situated and Autonomic Communications (SAC) Project Number: FP6-IST-27489,
`http://www.ana-project.org/autonomic/network/-deliverables.html`, Aug. 2006.

[Johnson02] E. Johnson and A. Kunze : *"IXP-1200 Programming"*, Intel Press 2002.

[Johnson03] E. Johnson and A. Kunze : *"IXP2400/2800 Programming – The Complete Microengine Coding Guide"*, Intel Press, April 2003

[juniper] *"JUNOS 8.2 Policy Framework Configuration Guide"*,
`http://www.juniper.net/techpubs/software/-junos/junos82/swconfig82-policy/html/-about-swconfig82-policy3.html#187130`

[Keller03] R. Keller and B. Plattner : *"Self-Configuring Active Services for Programmable Networks"*, in Proc. of IWAN 2003 LNCS 2982, pp. 137-150

[Kercheval94] Michael Yuen and Berry Kercheval, Sep. 1994 , *"CRC-32 Calculation, Test Cases and HEC Tutorial"*,
`http://www.cell-relay.com/cell-relay/publications/-software/CRC/32bitCRC.html`

[Keshav97] S. Keshav : *"An Engineering Approach to Computer Networking"*, Addison-Wesley.

[Keon05] E. Keon, J. Crowcroft et al. : *"A Survey and Comparison of Peer-to-Peer Overlay Network Schemes"*, Communications Surveys & Tutorials, IEEE, 2005, pp. 72–93.

[Kind02] A. Kind, R. Plekta and B. Stiller : "The potential of just-in-time compilation in active networks based on network processors", in Proc. of 5th Workshop on Open Architectures and Network Programming, June 2002 OPENARCH'02, pp. 79-90

[Kohler04] Mark Kohler : *"Introduction to Network Processors"*,
`http://www.netrino.com`.

[Kozierok01w] Charles M. Kozierok : *"PC Guide : Choosing your SDRAM"* April 2001,
`http://www.pcguide.com/art/sdramTiming-c.html`

[Kubiatowicz00] Kubiatowicz J., et al. : *"Oceanstore: An architecture for global-scale persistent storage"*, in Proc. of ACM ASPLOS IX, Cambridge, USA, Nov. 2000, pp. 190-201 .

[Labrecque06] Martin Labrecque : *"Towards a Compilation Infrastructure for Network Processors"*, Master Thesis at University of Toronto, 2006.

[Lee06] K. Lee and G. Coulson : *"Supporting Runtime Reconfiguration on Network Processors"*, in Proc. of Advanced Information Networking and Applications, 2006, Vol.1, Iss., 18-20 April 2006 pp. 721- 726

[Lefèvre02] L. Lefèvre and J-P. Gelas : *"Towards the design of an Active Grid"*, in Proc. of ICCS 2002, LNCS 2330, pp. 578-587, Amsterdam, The Netherlands, April 2002 .

[Lefèvre03] L. Lefèvre, J-M. Pierson, S. Guebli : *"Deployment of collaborative Web Caching with Active Networks"*, in Proc. of IWAN 2003 LNCS 2982 pp. 80-91

[Levis03] Ph. Levis, "Viral Code Propagation in Wireless Sensor Networks", Network Embedded System Technology Summer Retreat 2003.

[Limewire01w] *Gnutella protocol v0.4*, June 2001, `http://www9.limewire.com/developer/gnutella%20-protocol%200.4.pdf` .

[Lockwood03] J. Lockwood and J. Moscola et al. *"Application of Hardware Accelerated Extensible Network Nodes for Internet Worm and Virus Protection"*. In International Working Conference on Active Networks (IWAN 03), December 2003.

[Lopez95] A. Lpez–Ortiz and D.M. German : *"A multicollaborative push-caching http protocol for the WWW"*, poster at World Wide Web Conference 1995 (WWW5).

[Love05] Robert Love : *"Linux Kernel Development, Second Edition"*, Novell Press, ISBN 0-672-32720-1.

[Lu05] Jie Lu & Jie 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.

[Magoni02] D. Magoni and J. Pansiot. *"Network layer search service using oriented multicasting"*. In 21th IEEE INFOCOM, pages 1346 1355, 2002. New York.

[Martin02] Sylvain Martin and Guy Leduc : *"RADAR: Ring-Based Adaptive Discovery of Active Neighbour Routers"*, in Proc. of IWAN 2002 LNCS pp. 62-73

[Martin03] Sylvain Martin and Guy Leduc : *"A Dynamic Neighbourhood Discovery Protocol for Active Overlay Networks"*, in Proc. of IWAN 2003 LNCS 2982 pp. 151-162

[Martin05] Sylvain Martin and Guy Leduc : *"An Active Platform as Middleware for Services and Communities Discovery"*, in Proc. of International Conference on Computational Science 2005 LNCS 3516 (part 3) pp. 237-245.

[Martin05b] S. Martin and G. Leduc : *"Interpreted Active Packets for Ephemeral State Processing Routers"*, to appear in in Proc. of 7th Int. Working Conference on Active and Programmable Networks (IWAN), Sophia Antipolis 2005

[Martin06] S. Martin, P. Cascòn, HolyLich, L. Buytenhek et al. : *"ENP-Faq: The Hitchiker Guide to ENP-2611"*,
`http://ixp2xxx.sf.net/wiki/` .

[Martin06w] WASP implementation for Linux 2.6, prealpha release, S. Martin
`http://www.run.montefiore.ulg.ac.be/~martin/resources-/wasp-prealpha.tar.gz`

[Martin07] S. Martin and G. Leduc : *"Ephemeral State Assisted Discovery of Peer-to-peer Networks"*, to appear in in Proc. of 1st IEEE Workshop on Autonomic Communications and Network Management, Munich, May 2007 .

[Maymounkov02] P. Maymounkov and D. Mazieres : *"Kademlia: A peer-to-peer information system based on the XOR metric"*, in Proc. of IPTPS 2002, Cambridge, USA, March 7-8, 2002, Revised Papers, LNCS 2429, pp. 53-65 .

[mdgray03w] *"Spoofing the Wily Zip CRC"*, mdgray 2003,
`http://www.codeproject.com/cpp/crcspoof.asp`

[Merugu99] S. Merugu, S. Bhattacharjee et al., : *"Bowman and CANEs: Implementation of an Active Network"*, Invited paper at 37th Annual Allerton Conference, Monticello, IL, Sept 1999.

[Moore01] J. Moore and S. Nettles: *"Towards Practical Programmable Packets"*, in Proc. of 20th IEEE INFOCOM. Anchorage, Alaska, April 2001 .

[Moore02] Jonathan T. Moore : "Practical Active Packets", PhD Thesis, University of Pennsylvania, 2002.

[Moore02b] J.T. Moore, J.K. Moore, S. Nettles : *"Predictable, Lightweight Management Agents"*, in Proc. of IWAN02, zurich LNCS 2546, pp. 111-119

[Moore99] Jonathan T. Moore: *"Safe and Efficient Active Packets"* Technical Report MS-CIS-99-24, University of Pennsylvania, October 1999.

[NPForum03] 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.

[NSTinc] *$\mu C$ and $\mu L$ products download, Network Speed Technologies, inc.*
`http://www.network-speed.com/Products/index.html`

[Nygren99] E. Nygren, S. Garland, and M. Kaashoek : *"PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems"*, in Proc. of IEEE OPENARCH, pp. 78-89, New York, March 1999 .

[Ott00] M. Ott, G. Welling, S. Mathur : *"CLARA: A Cluster based Active Router Architecture"*, in Proc. of Hot Interconnects VIII, Aug. 2000 .

[Partridge93] C. Partridge, T. Mendez and W. Millikenn : *"Host Anycasting Service"*, IETF Request for Comments 1546.

[Plank01] J. Plank, M. Beck et al. : *"Managing data storage in the network"*, in Proc. of IEEE Internet Computing 5, sept. 2001 .

[Rekhter04] Y. Rekhter, T. Li, and S. Hares : *"A Border Gateway Protocol 4 (BGP-4)."*, Internet Draft draft-ietf-idr-bgp4-26.txt, October 2004.

[Rhea04] S. Rhea, D. Geels, et al.: *"Handling Churn in a DHT"*, in Proc. of USENIX Technical Conference, June 2004, pp. 127-140 .

[Rio04] , Miguel Rio, Mathieu Goutelle, Tom Kelly, Richard Hughes-Jones, Jean-Philippe Martin-Flatin, Yee-Ting Li. : *"A Map of the Networking Code in Linux Kernel 2.4.20"*, Technical Report DataTAG-2004-1, 31 March 2004

[Rorner96] T.H. Rorner, D. Lee, G.M. Voelker, et al. : *"The Structure and Performance of Interpreters"*, in Proc. of 7th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), Oct. 1996 .

[Rowstron01] A. Rowstron and P. Druschel : *"Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems"*, in Proc. of Middleware 2001 LNCS 2218, pp 329-

[Ruf05] L. Ruf, K. Farkas, H. Hug and B. Plattner : *"Network Services on Service Extensible Routers"*, in Proc. of 7th Int. Working Conference on Active and Programable Networks, nov. 05, Sophia Antipolis, France(IWAN'05) .

[Sacks05] L. Sacks, H. Sellapan et al. : *"On the manipulation of JPEG2000, in-flight, using active components on next generation satellites"*, in Proc. of IWAN'05 .

[Sanders01] M. Sanders, M. Keaton et al. : *"Active Reliable Multicast on CANEs: A Case Study"*, in Proc. of IEEE OpenArch 2001

[Schmid04] S. Schmid, L. Eggert, et al. *"TurfNet: An Architecture for Dynamically Composable Networks"*. In Proceedings of 1st IFIP TC6 WG6.6 Workshop on Autonomic Communication (WAC 2004), October 2004. Berlin, Germany.

[Schmid06] S. Schmid, M. Sifalakis and D. Hutchison, "Towards Autonomic Networks", in Proc. of 1st IFIP Conference on Autonomic Networking, Paris, France, Sept. 2006, pp. 1-11

[Schwartz98] B. Schwartz, A.W. Jackson, W.T. Strayer et al. : *"Smart Packets for Active Networks"*,
`http://citeseer.ist.psu.edu/schwartz98smart.html`

[Shalunov05w] *"thrulay, network capacity tester"*, S. Shalunov, Oct. 2005.
`http://www.internet2.edu/~shalunov/thrulay.`

[Sen02] S. Sen and J. Wang : *"Analysing peer-to-peer traffic across large networks"*, in Proc. of 2nd ACM SIGCOMM Workshop on Internet measurment, Marseille, France, 2002, pp. 137-150

[Shin01] Myung-Ki Shin, Yong-Jin Kim, Ki-Shik Park, and Sang-Ha Kim : *"Explicit Multicast Extension (Xcast+) for Efficient Multicast Packet Delivery"*, Electronics and Telecommunication Research Institute (ETRI) Journal, volume 23, number 4, December 2001.

[Sivakumar00] R. Sivakumar, S.W. Hanand V. Bharghavan : *"PROTEAN: A Scalable Architecture for Active Networks"*, in Proc. of OPENARCH'00

[Sivakumar99] R. Sivakumar, S. Ha, S. Han, V. Bharghavan : *"The Protean Active Router: Design and Implementation"*, in Proc. of The 14th IEEE Computer Communication Workshop IEEECCW'99.

[Sivakumar99b] R. Sivakumar, N. Venkitaraman, V. Bharghavan : *"The Protean Programmable Network Architecture: Design and Initial Experiences"* in Proc. of International Working Conference on Active Networks (IWAN), Berlin, Germany, 1999

[Song05] H. Song, S. Dharmapurikar, J. Turner, J. Lockwood : *" Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing"*, in Proc. of SIGCOMM 05, August 22 26, 2005, Philadelphia, Pennsylvania, USA .

[Spalink01] T. Spalink, S. Karlin, L. Peterson and Y. Gottlieb" : *"Building a Robust Software-Based Router Using Network Processors"*, in Proc. of Symposium on Operating Systems Principles 2001 pp. 216-229

[Stallings03] William Stallings : *"Organisation et Architecture de l'Ordinateur,* 6$^{e}$ *Edition"*, Pearson Education, ISBN 2-7440-7007-6.

[Sterbenz02] James P.G. Sterbenz : *"Intelligence in Future Broadband Networks: Challenges and Opportunities in High-Speed Active Networking"*, in Proc. of IEEE IZS 2002, Zürich, Feb. 2002 pp. 2-1 – 2-7

[Stoica01] Stoica , Morris et al. : *"Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications"*, in Proc. of ACM SigComm 2001 149-160

[Stoica02] I. Stoica, D. Adkins, S. Zhuang et al. : *"Internet indirection infrastructure"*, in Proc. of ACM SIGCOMM Computer Communication Review Volume 32 , Issue 4 (October 2002) .

[Tolly99] The Tolly Group : *"Cisco 12000 Series GSR POS: Performace Evaluation. No. 199128"*, Manasquan, NJ, Sep. 1999.

[Touch00] J. Touch and S. Hotz : *"Dynamic Internet Overlay Deployment and Management Using the X-Bone"* in Proc. of ICNP 2000, Osaka Japan, pp. 59-68.

[Tschudin98] A. Bansch, W. Effelsberg, C. Tschudin and V. Turau : *"Multicasting Multimedia Streams with Active Networks"*, in Proc. of IEEE Local Computer Network Conference LCN'98, Boston, MA, Oct 11-14, 1998, pp. 150-

[Tschudin03] C. Tschudin and R. Gold, *"Network Pointers"*, Computer Communication Review 33(1) pp. 23-28, 2003.

[Tullman01] Patrick Tullmann, Mike Hibler, and Jay Lepreau. : *"Janos: A Java-oriented OS for Active Networks"*, IEEE Journal on Selected Areas of Communication. Vol. 19, Issue 3, March 2001.

[Wehrle03] K. Wehrle, F. Pählke, H. Ritter, D. Müller, M. Bechler : *"Architecture Réseau Linux, Conception et implémentation des protocoles réseau du noyau Linux"*, Vuibert Informatique, ISBN 2-7117-4812-X

[Welte07] Harald Welte : *"Netfilter/iptables project homepage"*, `http://www.netfilter.org/`

[Wessels97] D. Wessels, K. Claffy : "Internet Cache Protocol (ICP), version 2", tech. rep., IETF Network Working Group, 1997. draft-wessels-icp-v2-03.txt.

[Wetherall01] D. Wetherall, A. Whitaker: *"ANTS - an Active Node Transfer System. version 2.0.2"* `http://www.cs.washington.edu/research/networking/ants/`

[Wetherall98] D. Wetherall, J. Guttag and D. Tennenhouse : *"ANTS - A Toolkit for Building and Dynamically Deploying Network Protocols"*, in Proc. of IEEE OPENARCH'98 .

[Wetherall99] Wetherall, D. : *"Active network vision and reality: lessons from a capsule-based system"*, Operating Systems Review, vol.33, ACM, Dec. 1999. p.64-79.

[Wethereall96] David J. Wetherall and David L. Tennenhouse : *"The Active IP Option"*, in Proc. of 7th ACM SIGOPS European Workshop, Sept. 1996 .

[Xie05] L. Xie, P. Smith, J. Sterbenz, and D. Hutchison. *"Building Resilient Networks using Programmable Networking Technologies"*. In 7th International Working Conference on Active and Programmable Networks (IWAN 05), 2005.

[Yamamoto03] Lidia Yamamoto : *"Adaptive Group Communication over Active Networks"*, Doctoral Thesis, University of Liège. Collection des Publications de la Faculté des Sciences Appliquées de l'Université de Liège, nr. 224, 2003.

[Yang05] X. Yang, D. Wetherall, and T. Anderson : *"A DoS-limiting Network Architecture"*, in Proc. of of ACM SIGCOMM 2005, Philadelphia, August 2005 .

[Yue06] "NPCryptBench: A Cryptographic Benchmark Suite for Network Processors", Y. Yue, C. Lin and Z.Tan, ACM SIGARCH Computer Architecture News, Vol. 34, No. 1, March 2006. pp. 49-56

[Zhang06] B. Zhang, S. Jamin, and L. Zhang. *"Universal IP multicast delivery"*. ACM Journal of Computer and Telecommunications Networking, Volume 50, Issue 6 (April 2006), pp. 781 - 806

[Zhao03] B. Zhao, L. Huang, J. Stribling et al. : *"Tapestry: A Resilient Global-Scale Overlay for Service Deployment"*, IEEE Journal on Selected Areas in Communications, 2003.

[Zhao05] L. Zhao, Y. Luo , L. Bhuyan and R. Iyer, : *"Design and Implementation of A Content-aware Switch using A Network Processor"*, in Proc. of 13th International Symposium on High Performance Interconnects (Hot-I05), Stanford, CA, August 2005