SmartMem

An Advanced Memory Subsystem for Networking Applications

Daniel Llorente Sancho

Technische Universität München
Lehrstuhl für Integrierte Systeme

SmartMem
An Advanced Memory Subsystem for Networking Applications

Daniel Llorente Sancho

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender: Univ.- Prof. Dr. Ing. Josef S. Kindersberger
Prüfer der Dissertation:
1. Univ.- Prof. Dr. sc.techn. Andreas Herkersdorf
2. Univ.- Prof. Dr. rer. nat. Doris Schmitt-Landsiedel

Die Dissertation wurde am 20.10.2011 bei der Technischen Universität München eingereicht und durch Fakultät für Elektrotechnik und Informationstechnik am 23.04.2012 angenommen.

Erklärung

Ich erkläre an Eides statt, dass ich die der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Promotionsprüfung vorgelegte Arbeit mit dem Titel: *“SmartMem. An Advanced Memory Subsystem for Networking Applications”* in Lehrstuhl für Integrierte Systeme unter der Anleitung und Betreuung durch Univ.- Prof. Dr. sc. techn. Andreas Herkersdorf ohne sonstige Hilfe erstellt und bei der Abfassung nur die gemäß § 6 Abs. 5 angegebenen Hilfsmittel benutzt habe.

(√) Ich habe keine Organisation eingeschaltet, die gegen Entgelt Betreuerinnen und Betreuer für die Anfertigung con Dissertationen sucht, oder die mir obliegenden Pflichten hinsichtlich der Prüfungsleistung für mich ganz oder teilweise erledigt.

(√) Ich habe die Dissertation in dieser oder ähnlicher Form in keinem anderen Prüfungsverfahren als Prüfungsleistung vorgelegt.

(   ) Die vollständige Dissertation wurde in ... veröffentlicht. Die Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München hat der Vorveröffentlichung zugestimmt.

(√) Ich habe den angestrebten Doktorgrad noch nicht erworben und bin nicht in einem früheren Promotionsverfahren für den angestrebten Doktorgrad endgültig gescheitert.

(   ) Ich habe bereits am ... bei der Fakultät für … der Hochschule …unter Vorlage einer Dissertation mit dem Thema ... die Zulassung zur Promotion beantragt mit dem Ergebnis: ...

Die Promotionsordnung der Technischen Universität München ist mir bekannt.

München, den 15. September 2011

Daniel Llorente Sancho

Abstract

Most of the forwarding capacity in today’s Routers and Switches is delivered by application-specific Network Processors. The performance of their Memory Subsystem is one of the main bottlenecks affecting the evolution of these critical processors and by extension constraining the capacity of current networks to deliver more services (at reduced costs) while offering ever higher speeds. In this context, SmartMem delivers key performance-enhancing improvements for building an advanced Memory Subsystem in Network Processors. It implements strong optimizations in throughput and data management while keeping very high storing efficiency.

The introduction of packet segmentation using multiple segment sizes is the central core of the project. This work presents two new packet segmentation algorithms that slice the packet using an optimal choice of segment sizes instead of only one size. As the segment size is the maximum amount of data that can be transferred seamlessly to the buffer, longer accesses to the memory increase performance considerably, even doubling data throughput in average for some packet mixes while obtaining very high storing efficiencies. This is achieved by selecting the best combination of segment sizes that minimize the number of segments per packet without jeopardizing storing efficiency in the memory.

In order to administer a packet buffer that contains multiple segment sizes, it is necessary to study a new buffer organization together with an adapted control structure. To tackle the variable requirements of segment sizes, we propose a dynamic memory organization, where the buffer is organized in blocks. These large blocks are then assigned dynamically to each portion according to their usage levels, having in addition a mechanism that maximizes the availability of free blocks. The new method achieves an excellent resource allocation response that satisfactorily meets the continuously-changing packet patterns to be found in Internet traffic.

One of the major contributions of this work is achieved because the SLA segmentation algorithm enables constraining the maximum number of segments per packet to two, hence control data management can be drastically simplified. With only two segments, the packet descriptor contains all the information necessary to reassemble the packet in the output. This removes the need for Linked Lists and its large requirement of control operations. Now, only one bit is necessary per segment to mark the segment status as free or busy. More importantly, the introduction of bitmap-based control structures replacing state-of-the-art linked lists leads to a ten-fold Control Buffer size reduction. Such a small buffer can be comfortably implemented on SRAM consequently further accelerating control update performance.

A final contribution of this work is the study of a Packet Processor’s Local Buffer (PPLB). The PPLB investigates the promising concept of data injection in a processor’s local memory. This scheme has two aims: avoiding short and thus inefficient accesses to the Packet Buffer and furthermore preventing CPU stalling due to cache misses. With the PPLB, cache misses never occur as packet data is preemptively pushed into the CPU’s local buffer, ready for the CPU to immediately begin with the packet processing.

The previous concepts have been realized in the SmartMem Buffer Manager (SBM) prototype and the PPLB buffer. These units have been coded in VHDL and tested on an FPGA-based platform to confirm the performance improvement thus validating the benefits of using multiple segment sizes for packet segmentation. Insights from a comprehensive SystemC simulator have been considered in the design of the accelerator’s architecture.

Abstrakt

Die meiste Prozessorleistung in den heutigen Routern und Switches wird durch anwendungsspezifische Netzwerk-Prozessoren (NP) erbracht. Die Leistungsfähigkeit ihrer Speichersubsysteme ist einer der wichtigsten Engpässe von NP. Die Verbesserung dieser Prozessoren ist kritisch für die Erweiterung der Kapazität des Netzwerkes, um mehrere Dienstleistungen zu bieten (zu reduzierten Kosten) und um einen höheren Durchsatz zu erreichen. Im Hinblick darauf liefert SmartMem wichtige leistungssteigernde Verbesserungen für das Speichersubsystem von NP. Diese Optimierungen im Durchsatz und in der Kontrolldatenverwaltung werden mit einer sehr hohen Speicherungseffizienz erreicht.

Die Einführung von Paketsegmentierung mit mehreren Segmentgrößen ist der zentrale Punkt dieses Forschungsprojektes. In der vorliegenden Arbeit werden zwei neue Paketsegmentierungsalgorithmen vorgestellt, die das Paket mit einer optimalen Auswahl der Segmentgröße anstatt nur einer Segmentgröße unterteilen. Die Segmentgröße entspricht der maximalen Anzahl an Daten, die nahtlos in den Puffer übertragen werden kann. Durch längere Zugriffe auf den Speicher wird die Leistung des Speichers erheblich erhöht und zudem wird eine höhere Speicherungseffizienz erreicht.

Für die Verwaltung des Paketpuffers mit mehreren Segmentgrößen ist es notwendig, eine neue Pufferorganisation zusammen mit angepassten Kontrollstrukturen zu entwickeln. Zur Bewältigung der variablen Anforderungen an die Segmentgrößen wird eine dynamische Speicherorganisation vorgeschlagen, wobei der Paketpuffer in Blöcke geteilt wird. Diese großen Blöcke werden dann dynamisch an jede Segmentgröße je nach Bedarf zugeordnet, zusammen mit einem Mechanismus, der die Verfügbarkeit von freien Blöcken maximiert. Die neue Methode erzielt eine hervorragende Ressourcenzuordnung, die die ständig wechselnden Paketmuster des Internetverkehrs erfolgreich einpasst.

Eines der wichtigsten Ergebnisse dieser Arbeit ist erreicht, weil der SLA Segmentierungsalgorithmus die maximale Anzahl der Segmente pro Paket auf zwei einschränkt. Damit wird ermöglicht, die Kontrolldaten drastisch zu vereinfachen. Mit nur zwei Segmenten enthält der Paket-Deskriptor alle notwendigen Informationen, um das Paket in der Ausgangsseite wieder zusammenzubauen. Dadurch entfällt die Notwendigkeit für verkettete Listen und somit der Bedarf an Kontrollstrukturen. Nun wird nur ein Bit pro Segment benötigt, um den Segmentstatus als frei oder besetzt zu kennzeichnen. Noch wichtiger ist, dass die Einführung von Bitmap-basierten Kontrollstrukturen zum Ersatz von state-of-the-art verketteten Listen führt, damit eine zehnfache Verkleinerung der Kontrollpuffergröße ausgeführt wird.

Ein letzter Beitrag dieser Arbeit ist die Untersuchung des Packet Processor Local Buffer (PPLB). Die PPLB untersucht das vielversprechende Konzept der Dateninjektion in einem lokalen Speicher des CPU. Diese Architektur hat zwei Ziele: Vermeidung von kurzen und damit ineffizienten Zugriffen auf den Paketpuffer und außerdem Verhinderung von Cache Misses in der CPU. Mit dem PPLB liegen Paketdaten für die CPU auf dem PPLB bereit, sodass das Packet Processing sofort anfangen kann.

Die bisher vorgestellten Konzepte waren im SmartMem Buffer Manager (SBM) Prototyp und im PPLB realisiert. Diese Einheiten wurden in VHDL codiert und auf einer FPGA-basierten Plattform getestet. Die gemessenen Leistungsergebnisse bestätigen die Vorteile der Segmentierungsalgorithmen. Erkenntnisse aus SystemC Simulators sind für die Entwicklung der SBM Architektur berücksichtigt worden.

Die bisherigen Konzepte haben in den SmartMem Buffer Manager (SBM) Prototyp und die PPLB Puffer realisiert worden. Diese Einheiten haben in VHDL wurden codiert und getestet auf einem FPGA-basierte Plattform, um die Performance-Verbesserung somit Validierung der Vorteile der Verwendung mehrerer Segment Größen für Paket-Segmentierung zu bestätigen. Erkenntnisse aus einer umfassenden SystemC Simulators sind in die Konstruktion des Beschleunigers Architektur berücksichtigt.

Index

1 INTRODUCTION 2

1.1 DATA NETWORKS, FROM BPS TO GBPS 2
1.2 ROUTERS AND SWITCHES 5
1.3 AN OVERVIEW OF NETWORK PROCESSORS 7
1.3.1 Functionality 7
1.3.2 Advanced applications 8
1.3.3 NP Architecture 8
1.3.4 Performance bottlenecks and memory constraints 12
1.4 CONTRIBUTIONS OF THIS WORK 14

2 STATE-OF-THE-ART 16

2.1 COMMERCIAL NETWORK PROCESSORS 16
2.1.1 Brief historical introduction to Network Processors 16
2.1.2 The IXP family 17
2.1.3 IBM's PowerNP Family 20
2.1.4 Motorola/Freescale 21
2.1.5 EZChip 22
2.1.6 Cisco's Network Processors 23
2.1.7 LSI's Network Processors 24
2.1.8 General trends in Network Processors 25
2.2 THE MEMORY SUBSYSTEM IN NETWORK PROCESSORS 27
2.3 THE STRUCTURE OF THE PACKET BUFFER 31
2.3.1 Terminology 31
2.3.2 The need of Packet Segmentation 32
2.3.3 Control structures to manage the Packet Buffer 34
2.3.3.1 Linked Lists 34
2.3.3.2 Buffer administration in other computing applications 37
2.3.4 Packet Buffer administration in a per-flow basis 38
2.3.5 The Buffer Manager accelerator in NPs 41
2.4 THE MEMORY WALL: THE BANDWIDTH BOTTLENECK 42
2.4.1 DRAM technology 42
2.4.2 The influence of bank interleaving and access length over latency 43
2.5 MECHANISMS TO IMPROVE DRAM ACCESSING 46
2.5.1 Ladders 46
2.5.2 Hammers 46
2.6 THE CONCEPT OF CACHE INJECTION IN NETWORKING 49
2.7 STATE-OF-THE-ART SUMMARY 50

3 MULTIPLE SEGMENT-SIZE PACKET SEGMENTATION 52

3.1 INTRODUCTION AND METHODOLOGY 52
3.1.1 Motivation 52
3.1.1.1 Performance and memory efficiency with Fixed Size Segmentation 53
3.2 MULTIPLE-SIZE PACKET SEGMENTATION 56
3.2.1 Memory Efficiency Optimized Algorithm (MEO) 56
3.2.2 Segment number Limited Algorithm (SLA) 58
3.3 PACKET SEGMENTATION ALGORITHM ANALYSIS 60
3.3.1 Methodology and design flow 60
3.3.2 Packet stimuli used during the experiments 62
3.3.3 Memory Efficiency Optimized Algorithm performance 64
3.3.4 Segment number Limited Algorithm performance 68
3.3.5 Conclusions 71
3.4 SYSTEM-LEVEL EXPLORATION 73
3.4.1 Model Description 74
3.4.1.1 PLB Bus 76
3.4.1.2 MAC interfaces 76
3.4.1.3 Buffer Manager Model 77

3.4.1.4 The Processor's Cluster 77
3.4.1.5 Output Queuing 78
3.4.1.6 DDR-SDRAM Memory Controller 79
3.4.2 *Simulation results* *81*
3.4.2.1 Simulation of performance and storing efficiency 81
3.4.2.2 Impact of memory types 85
3.4.2.3 Study of the CPU impact over the Packet Buffer performance 87
3.4.2.4 Burst-length constraint on the bus 88
3.4.2.5 Direct-IF between the Buffer Manager and the memory controller 89
3.4.2.6 Conclusions from the system-level simulations 91

4 PACKET BUFFER ORGANIZATION WITH MULTIPLE SEGMENT SIZES 92

4.1 MOTIVATION 92
4.2 DIMENSIONING THE SEGMENT POOLS 93
4.2.1 *Average segment requirements* *93*
4.2.2 *Dynamic segment requirements* *95*
4.3 MEMORY ALLOCATION FOR DYNAMIC BUFFER ORGANIZATION 98
4.3.1 *Segmentation and coalescence of memory blocks* *99*
4.3.2 *Gravity Centers* *100*
4.3.3 *Block-based memory organization* *103*
4.3.4 *Conclusions* *104*
4.4 CONTROL STRUCTURES FOR ADMINISTERING MULTIPLE SEGMENT POOLS 105
4.4.1 *Bitmaps and linked lists of blocks* *107*
4.4.2 *Operations with bitmap-based control structures* *109*
4.5 CONCLUSIONS 114

5 THE PACKET PROCESSOR'S LOCAL BUFFER 116

5.1 PACKET INJECTION IN THE LOCAL BUFFER OF THE CPU 118
5.2 INTERNAL DATA FLOW 121
5.3 PACKET PROCESSOR'S LOCAL BUFFER ARCHITECTURE 123
5.3.1 *Functionality* *123*
5.3.2 *Buffer Manager Unit* *126*
5.3.3 *Classification Engine* *126*
5.4 MEMORY SIZE REQUIREMENT OF THE CPU QUEUES 127
5.5 BENEFITS OF THE PACKET PROCESSOR LOCAL BUFFER ON DRAM ACCESSING 131
5.5.1 *Access reordering to avoid bank conflicts on DRAM* *131*
5.5.2 *Optimized address mapping* *133*
5.5.3 *Bank conflict-aware segment allocation* *135*
5.6 SUMMARY 137

6 THE SMARTMEM BUFFER MANAGER 138

6.1 SMARTMEM BUFFER MANAGER IN STANDALONE MODE 139
6.1.1 *Functionality* *139*
6.1.2 *SmartMem Buffer Manager architecture* *140*
6.1.2.1 Address Manager Unit 142
6.1.2.2 RX Unit 144
6.1.2.3 TX Unit 147
6.2 SMARTMEM BUFFER MANAGER WITH FLEXPATH 149
6.2.1 *Motivation the integration of the Buffer Manager in the FlexPath platform* *149*
6.2.2 *FlexPath description* *149*
6.2.2.1 Internal data flow 151
6.2.3 *SmartMem Buffer Manager design for compatibility with FlexPath* *153*
6.3 COMPATIBILITY WITH THE PACKET PROCESSOR LOCAL BUFFER AND THE MULTIPORT MEMORY CONTROLLER 156
6.3.1 *Adaptation of the architecture to the Packet Processor Local Buffer* *156*
6.3.2 *Compatibility with the Multi-Port Memory Controller* *158*
6.3.3 *RX Architecture with the Multi-Port Memory Controller* *159*
6.4 SYNTHESIS RESULTS - RESOURCES AND CLOCK FREQUENCY 162
6.5 MEASUREMENTS WITH THE PROTOTYPE 165
6.5.1 *Prototyping platform description* *165*
6.5.2 *Organization of the experiments* *167*
6.5.3 *Measurements* *169*
6.5.3.1 System reference: Fixed-Size Segmentation 64-Bytes and Linked Lists 169

6.5.3.2 Study of packet segmentation with Fixed-Size Segmentation and the SLA algorithm 170
6.5.3.3 SmartMem Buffer Manager with Bus and Direct-IF to Multi-Port Memory Controller 172
6.5.3.4 Prototype performance with different Internet traffic mixes 175
6.5.3.5 Packet Processor Local Buffer performance 177
6.5.3.6 The Address Manager in standalone mode 180
6.5.3.7 Performance results of FlexPath and SmartMem 181
6.6 SUMMARY 182
7 CONCLUSIONS AND OUTLOOK 184
8 ACKNOWLEDGEMENTS 188
9 REFERENCES 190

Index of figures

Figure 1: Projected traffic increase in Internet ... 3
Figure 2: Cisco Nexus 7000 Series Internet Router, source [102] ... 5
Figure 3: A geographical view of T-Online’s backbone network in Germany (2004), source [103] ... 6
Figure 4: Functional blocks of a generic NP architecture ... 9
Figure 5: Parallel pipelining architecture in the Cisco 1000 Toaster, source [104] ... 11
Figure 6: Average processing time per packet for a multi-core processor ... 12
Figure 7: Line speed increasing for access, edge and core routers ... 17
Figure 8: Intel's IXP 1200, source [12] ... 18
Figure 9: Block diagram of the Freescale's P4080, source [94] ... 22
Figure 10: Cisco QuantumFlow Processor Used on Cisco ASR 1000 Embedded Services Processor, source [104] ... 24
Figure 11: APP3300 Block Diagram, source [105] ... 25
Figure 12: Criterion for sizing the Packet Buffer according to the probability of packet accumulation, source [44] ... 28
Figure 13: Linked-list-based control structures ... 35
Figure 14: Administering a Packet Buffer with multiple segment sizes using linked lists ... 35
Figure 15: UNIX’s sk_buff structure for storing data packets ... 38
Figure 16: Method proposed by García-Vidal et al. for conflict-free storing in packet switchers, source [58] ... 39
Figure 17: Memory scheme of Iyer et al., with tail and head caching and per-flow memory administration ... 39
Figure 18: Microscopic view of a state-of-the-art (2009) 50-nm DRAM cell from Micron ... 42
Figure 19: Access example in a DDR-400 memory, with row/column activation commands and pre-charge ... 44
Figure 20: Bank interleaving helps to improve DRAM accessing ... 48
Figure 21: Influence of segment size with FSS over performance, with stimuli consisting of one packet size. In the figure, traffics with 64, 128, 256, 512, 1K, 1500 –byte packets are represented. ... 53
Figure 22: Internal fragmentation loss for 37 different PCAP files, which proves that increasing the segment size decreases memory efficiency. ... 54
Figure 23: Iterative selection of segment sizes for packet allocation ... 56
Figure 24: segmentation of a 1481-byte packet with MEO and F=0 ... 57
Figure 25: segmentation of a 1481-byte packet with MEO and F=0,2 ... 57
Figure 26: Criteria for selecting segment sizes by limiting the number of possible segments per packet ... 58
Figure 27: segmentation of a 1481-byte packet with SLA, M=2 and N=3 ... 59
Figure 28: Design approach and level of modeling abstraction ... 60
Figure 29: quality parameter evaluation ... 63
Figure 30: MEO algorithm performance for OC-48 traffic ... 65
Figure 31: MEO algorithm performance for Adv. IMIX traffic ... 66
Figure 32: Variation of memory efficiency and average number of segments with the F factor for MEO ... 67
Figure 33: SLA algorithm performance for OC-48 traffic ... 68
Figure 34: Selection of SLA algorithm’s performance for Adv. IMIX traffic ... 69
Figure 35: Enlargement of the area of interest (upper-left corner) with Pareto curve ... 70
Figure 36: Performance comparison of MEO, SLA referenced to FSS ... 71
Figure 37: Abstraction level of SmaSyMo ... 74

Figure 38: Block diagram of the SystemC model SmaSyMo, dotted lines display setup configuration alternatives 75
Figure 39: Packet delay observed in an OC-48 router (left) compared to the delay simulated in the model (right) 78
Figure 40: Detailed block diagram of the memory controller model, two alternatives are depicted for the connection to the bus (Xilinx-IPIF or LIS-IPIF). 79
Figure 41: Trade-off between storage efficiency and throughput obtained for various segmentation algorithms with backbone traffic 82
Figure 42: Throughput and memory efficiency results for up- and downstream traffic 82
Figure 43: Standard deviation of dynamic throughput for various algorithms 83
Figure 44: Reducing the average number of segments leads to higher performance 84
Figure 45: Throughput obtained for different Packet Buffer mappings, control mapped in a separate dynamic memory 85
Figure 46: Throughput obtained for different Control Buffer mappings 86
Figure 47: Throughput obtained for different CPU cluster access schemes 87
Figure 48: Allowing unconstrained transfers on the bus accelerates the system 88
Figure 49: Architectural variant with direct-IF between the BM and the memory controller. 89
Figure 50: Improvement with the Direct-IF architectural variant 90
Figure 51: Graphical comparison of segment requirements. 94
Figure 52: Port contention and the subsequent Packet Buffering increase 96
Figure 53: Packet Buffer usage during episodes of memory saturation in an access router with static and dynamic segment pool management 97
Figure 54: Memory allocation in the coalescence method, packet D is divided in two segments. 99
Figure 55: Memory discarding in the coalescence method, packet A is retrieved from two separate memory locations. 99
Figure 56: Homogeneous allocation of segments (right) compared to random allocation (left) 100
Figure 57: Floating Gravity Centers allow for a better adaptation to changing segment requirements. The different colors represent areas allocated to different segment-size areas. 101
Figure 58: Comparison between segmentation-and-coalescence (up) and block-based methods (down) 103
Figure 59: Control structures based on linked lists with multiple pools. 105
Figure 60: Bitmap-based memory organization 108
Figure 61: Control structures based on bitmaps and blocks with multiple pools 109
Figure 62: Bitmap-based structures supporting dynamic resizing of segment pools 110
Figure 63: Control Buffer requirements. Comparison between state-of-the-art linked list and the proposed bitmap method. Notice the logarithmical scales used in the graphic. 113
Figure 64: Packet buffering immediately after packet arrival 117
Figure 65: Processor Buffer Packet Buffer 117
Figure 66: Data flow comparison between three Memory Subsystem architectures 118
Figure 67: access requests to the Packet Buffer in a state-of-the-art architecture 119
Figure 68: Access pattern to memory with the PPLB, notice the simplification in the Packet Buffer interface 119
Figure 69: Full packets first stored in the Header Buffer and once processed in the Packet Buffer 120
Figure 70: Memory Subsystem architecture with the Packet Processor’s Local Buffer 122
Figure 71: Block diagram of the Packet Processor Local Buffer 125
Figure 72: Maximum fill level with 2 CPUs – 4 Kb of Level-1 local buffer 128
Figure 73: Maximum fill level with 2 CPUs – 8 Kb of Level-1 local buffer 128

Figure 74: Maximum fill level with 4 CPUs – 4 Kb of Level-1 local buffer......................... 129
Figure 75: Maximum fill level with 4 CPUs – 8 Kb of Level-1 local buffer......................... 129
Figure 76: Probability of a bank conflict with access reordering 132
Figure 77: Direct memory mapping on DRAM 133
Figure 78: Virtual Row addressing method with embedded bank interleaving in the memory block 133
Figure 79: Access efficiency comparison with Virtual Row address mapping 135
Figure 80: Rescheduling of accesses to the Packet Buffer... 135
Figure 81: Block diagram for the implementation of Active Memory Allocation 137
Figure 82: SmartMem Buffer Manager internal architecture... 141
Figure 83: format of commands accepted by the AM.. 143
Figure 84: Block diagram of the RX Unit 146
Figure 85: Block diagram of the TX Unit. 148
Figure 86: Overview diagram of the FlexPath NP 151
Figure 87: timing diagram of the interface between the SBM’s TX unit and the Post-Processor 154
Figure 88: Packet Descriptor structure 155
Figure 89: Packet Structure with the PPLB 157
Figure 90: Data flow in the RX Unit in compatibility mode with the PPLB......................... 158
Figure 91: RX Unit in compatibility mode with the PPLB 160
Figure 92: Contribution of the three main BM units to the total resource requirement of the BM.. 162
Figure 93: Comparison of Look-up Tables and FPGA slices required by the SBM in different system configurations. 164
Figure 94: Test prototyping platform 166
Figure 95: Measurements with SmartMem 168
Figure 96: Maximum throughput with FSS-64B and Linked Lists (BM v1.0)...................... 170
Figure 97: Performance comparison of SLA and FSS with three segment sizes................... 171
Figure 98: SBM with Direct IF to MPMC. CPU loopback and CPU header processing 173
Figure 99: Packet rates measured 174
Figure 100: SBM performance with different internet traffic mixes 175
Figure 101: End-to-end packet latency for different traffic mixes... 176
Figure 102: Performance with the PPLB with Direct IF to the memory controller............... 178
Figure 103: Performance with the PPLB and multiple traffic mixes as stimuli 179
Figure 104: End-to-end packet latency with the PPLB, without processing latency. 179
Figure 105: Maximum address discard rate with the Address Manager in standalone mode 180
Figure 106: Performance of the FlexPath platform with the SBM. 181

Index of tables

Table 1: Example of two different packet segmentation results for a 1392 Byte Packet and 128-512-1024 Byte segment sizes using the MEO algorithm.. 57
Table 2: Traffic pattern distribution of the studied PCAP files 62
Table 3: Best performing algorithms: best memory efficiency, best average number of segments and the two algorithms with the best trade-off between both parameters........ 72
Table 4: Model setup for the simulations... 81
Table 5: Average segment requirements with four segment sizes... 93
Table 6: Control Buffer size with different block sizes 112
Table 7: Configuration Table parameters... 156
Table 8: Total HW resources and maximum possible clock frequency required for the BM implementation on a Virtex IV FPGA 162
Table 9: SBM resource requirement for different system configurations.............................. 163

Index of equations

Equation / Page number
(1) 27
(2) 40
(3) 40
(4) 58
(5) 63
(6) 63
(7) 102
(8) 131
(9) 132
(10) 154

Foreword

Creativity and innovation disappeared from several human activities in the 19th century when traditional workshops were replaced by the modern factory. Though nowadays many repetitive tasks have being outsourced to robots and other automats, many industries still seek for profit by means of stealing the employee of its capacity for thinking (and thus of error) reducing the human being in the process to a perfectible part of the production/service chain. Luckily enough, the research activity at Universities is still in the beginning of the century an island of humanism brightly contrasting with expanding oceans of Hedge Funds, investment banks and real estate developers. It is true that we investigators contribute with our findings to the creation of new products at reduced cost, but praising the Market God is not the main thought we have in mind as the alarm clock rings at seven o'clock in the morning. On the contrary, what really moves us is the savage pleasure of an every-day crossing of that last wild border we use to refer to as state-of-the-art knowledge.

None of this was by far in my mind as I was approaching the outskirts of Munich six years ago while driving an ancient Seat Ibiza, equipped with enough olive oil for surviving whatever may come upon me and without the slightest idea of how tasty "*käsespätzle*" are. Neither had I had the audacity of imaging that I would eventually present my dissertation for a PhD promotion in the best of the technical universities of "*Das Land der Ingenieure*". The paths of life have such uncertainties, sometimes surprisingly beautiful and self-fulfilling.

At that point, my knowledge of Germany and their people was non-negligible though quite biased by the after-café stories of my grandpa, who lost his father while fighting for a better world. I was expecting a diligent land, full of well-educated but distant human beings who speak something completely strange for me, even after months of intensive language learning. Nothing could be farther from reality. What I actually found was a group of colleges among which I never felt myself alienated and whom extensive insight, capacity for detailed analysis and endless will for discussing about technical issues produced in me, and still does today, a sincere admiration.

Es war eine Ehre fünf Jahre von meinem Leben mit meinen LIS Kollegen arbeiten zu können. Ich habe mit je und Mühe versucht, die Gelegenheit an der TUM promovieren auszunützen. Als ich jetzt nach neue Länder weiter umziehen muss, die eingepackten Erinnerungen und Gefühle hier gesammelt könnten schwierig besser werden. Vielleicht das erste Wort, dass ich auf Deutsch lernte, hilft mir diese Gedanken einzudeutigen: *Danke*.

Bayern is really one of the best places under the sun, or rather under the grey sky. But still it is impossible to dismiss a feeling of privilege for having the opportunity to live in *Minga* for six years. If Berlin is poor but sexy, Munich is affluent and equally stylish. With the exception of Catalonia, I do not feel at home anywhere as here. *Dahoam is dahoam*!

Finally, the different contributions to this work are summarized at the end of the document but in this foreword I would like to mention specially Herr Thomas Wild. Thomas has been my supervisor first during my Diploma and then along my PhD Thesis. His advice and patience correcting and improving my publications have left in me a profound feeling of gratitude. Without his insight, this thesis would not be in your hands.

1 Introduction

Networking is a complex art which has been particularly productive in technical advances in the recent decades, all fueled by the increasing number of internet users and on-line services. In particular, the amount of information transmitted tends to double every two years, which produces a strong requirement of processing capacity in the network nodes. A node is similar to a crossroad where, according to different parameters, packets must be routed towards their final destination. This operation is done by switches and routers, which need to provide enough processing capacity to meet the line speed. Switches operate essentially based on layer 2 information, most of the times they simply flood all ports except the one the packet is received from. Routers are more complex as they know about the topology of the network so packets can be more effectively transmitted to the next hop. In all routers, the packet forwarding operation is done at layer 3, where information about the source/destination IP address is to be found. The part of the router in charge of forwarding packets is known as the data plane while opposite, the control plane is a high level control system where routing protocols are executed and topology tables elaborated.

The characteristics of these operations imply a strong requirement of performance together with a great deal of programming flexibility. For that reason, an application-specific set of processors has been developed to deliver high throughput while supporting multiple software applications. Network Processors are widely implemented in multiple networking applications, with different characteristics and for different purposes but in essence, they all fulfill the same purpose: flexibly forwarding packets at a high speed. The rest of this chapter overviews the technological basis that underpins modern data networks and Network Processors, together with introducing the Memory Subsystem that this work aims to improve.

1.1 Data Networks, from bps to Gbps

Even though it lacks the romanticism of astronomy or the religious implications of biotechnology, designing the architecture of a processor is a challenging art always in the pursuit of higher performance while keeping low resource usage inasmuch as possible. Transistor integration on a chip, as predicted by Gordon Moore in 1965, approximately has duplicated every eighteen months during the last 40 years. This poses a tremendous amount of resources at the system designer disposal, who struggles for structuring billions of transistors in a coherent architecture that efficiently performs a processing task.

In 1947, John Bardeen and Walter Brattain discovered the transistor while working at Bell Telephone Laboratories, which secured for them the Nobel Prize in 1956. Only two years after that discovery, the German engineer Werner Jacobi, from Siemens AG, integrated five of these transistors on a single silicon to produce an amplificatory device. But it was Jack Kilby and Robert Noyce who settled the technological basis of what today is known as integrated circuits and so the chip was born (again Kilby was granted with the Nobel Prize in 2000). All of them were completely oblivious of the revolution they were to unleash. Their breakthroughs were rapidly improved by successive generations of engineers following the above mentioned Moore law, striving for reducing integration technology down to the current (2010) state-of-the-art ~22 nanometres, a technological race that so far, sixty year later, does not show signs of losing steam [4]. Although some physical constraints are already in sight, electronic engineers have often in the past succeeding in overcoming them, so we can be positive about the fact that integrated circuits' performance will keep improving.

The introduction of glass fiber to link two distant network nodes was a major technological breakthrough that allowed transmitting at data rates up to 1 Tbps per optic channel [2]. Besides, in one single fiber it is possible to merge several of those optic channels using a technique known as Wavelength-Division Multiplexing (WDM), which has been successfully tested with up to 160 wavelengths per optical fibber [1]. Such an impressive transport capacity is clearly ahead of traffic requirements provided that only relatively recent multimedia applications consume large amounts of bandwidth. This is best understood if we look at the data rates of current routers, as defined in the Synchronous Optical Network standard (SONET) in multiples of 51,84 Mbit/s or Optical Carrier (OC-x). The power horse of current backbone networks are routers servicing line rates of 2,5 Gbps (OC-48) and 10 Gbps (OC-192). In some North Atlantic submarine, cables of 40 Gbps (OC-768) are already in use and in the foreseeable future applications of 100 Gbps will enter the market [5].

The continuous upgrading of capacity in data networks has been driven by the constant increase of traffic and users since the very beginning of Internet [80]. Such trend is poised to continue in the foreseeable future [8] so the data transport infrastructure will need to cope with traffic increases of up to 300% in only five years. Such a tremendous pressure to improve transport capacity will require of a huge modernization effort to overhaul the current Internet bottlenecks, not only upgrading network capacity but also redefining some of the current Internet paradigms [9]. Proposed solutions go from a general reengineering of Internet to more local enhancements.

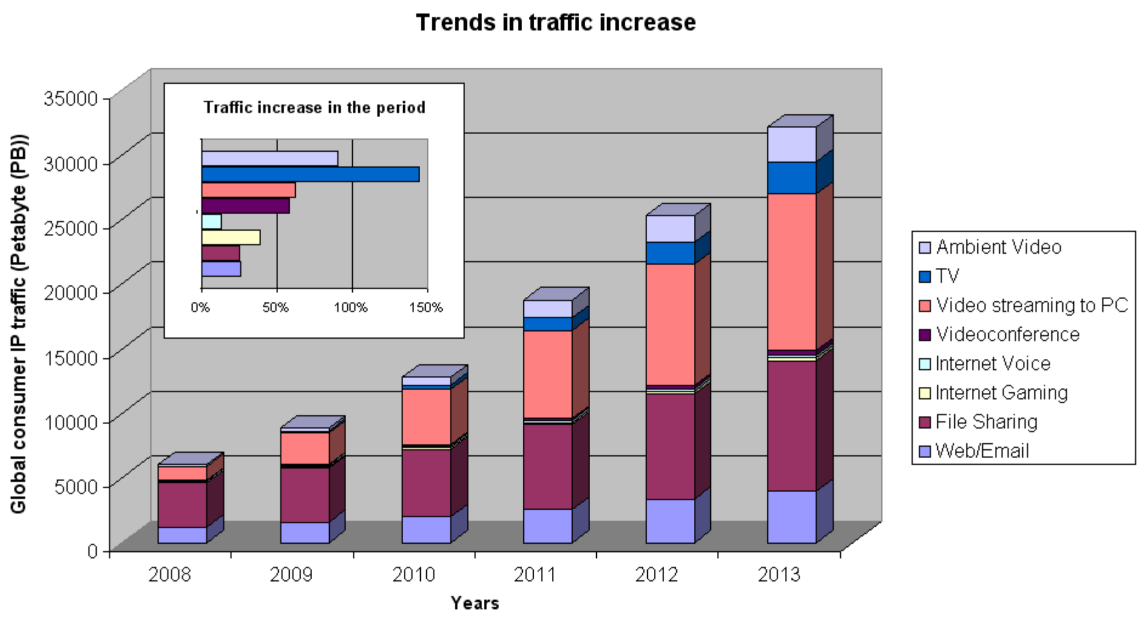


Figure 1: Projected traffic increase in Internet

Probably, the only realistic solution in a world with millions of Kilometers of deployed optic fibber and many billions of dollars invested in routing equipment is a step-by-step improving with backward compatibility. A new dawn for the net is just too expensive and, contradictory as it may seem in computing, this time a reset is not the trick that can solve all issues.

Networks consist not only of links but of nodes as well, where routers and switches forward packets to the corresponding next hop. A basic IP routing application looks up for the next node address in its routing table, modifies some control flags (like the time-to-life counter) and transmits the packet to the corresponding output port. However simple this may seem, the trend is to increase the complexity of transport protocols with demanding Quality-of-Service (QoS) policing or deep packet inspection (e.g. virus scanning), which requires more processing capacity in the router. To make it more challenging, the router designer not only has to meet a growing demand of processing resources but he also has to ensure compatibility with the remarkable range of network protocols present in nowadays networks.

This configures a scenario where optical links comfortably provide enough transport capacity while network nodes struggle to keep pace with this optical bandwidth. As only very basic optic switching has been demonstrated so far [2], routers and switches still will rely in the foreseeable future on electronic technology to keep the capacity of packet forwarding increasing. Resuming, it is of paramount importance to improve and upgrade the capacity in nodes for ensuring that the most important of the recent inventions, the Internet, does not stall.

At that point, the discussion leads to the question of which art of processors better fits the specifics of networking. Most usually, the taxonomy of integrated circuits is done according to the range of applications they target as well as the performance they deliver. General Purpose Processors (GPP) such Intel's Pentium or AMD's Athlon families are designed for general applications, having as main goal offering flexibility even if a good deal of performance is sacrificed. Opposite, Application Specific Integrated Circuits (ASICs) are specific-purpose devices optimized to deliver maximum speed when executing one particular application. Inconveniently, they are expensive to develop being only profitable in mass production. Even though both of them have been used by router manufacturers to some extend, neither fulfill adequately the current requirements of flexibility and fast packet processing in high-speed networks. For these reasons, a new type of device was required capable of delivering flexibility and high performance, which leaded to the introduction of Network Processors.

1.2 Routers and switches

Internet is a web of networks that carry digital data using multiple transport protocols with a large variety in complexity. In this context, two main tasks are performed on the network node: routing the packet to the corresponding next hop and bridging possible protocol differences to allow networks with different characteristics stay compatible.

When traffic remain in a same small network and do not require further processing, like in the case of a Local Area Network (LAN) consisting of switches, packets are simply broadcasted to all output ports without doing much more than checking data integrity and basic packet filtering. However, in more complex LANs using more advanced switches or routers as well as in Wide Area Networks (WAN) broadcasting does not suffice to forward a packet to its proper destination, because if all traffic were transmitted to all users, the network would collapse. To avoid that, a router forwards each packet only to its proper destination (unicast) using a transport protocol stack like TCP/IP, which becomes necessary to identify the elements of the Network and thus to convey the packets to the right address destination. The router takes advantage of routing protocols in charge of maintaining its local routing table, where look-ups are performed to find out the most suitable next hop. In case of not finding a default route in the routing table, dynamic routing protocols as Open Shortest Path First (OSPF) can establish adjacencies with neighboring routers in the same area and exchange their routing tables to enlarge their knowledge of the network.

The intelligence of packet processing in the node is implemented using a wide range of methods, from simple low-speed software-based routing applications to complex high-end routers that switch packets at high speed between multiple links while bridging networks with different transmission technologies (Frame Relay, Ethernet, SONET…). We can broadly classify routers according to their function in the network. Access Routers are the front-end of the network and provide access to final subscribers. A typical Internet Service Provider (ISP) implements multiple aggregation points physically close the users (last mile) and owns (or rents) a transport network including several Enterprise Routers that transports aggregated traffic from multiple subscribers to the backbone network. There, high-end Core Routers forward traffic to every corner in the world with the Border Gateway Protocol (BGP) which identifies all major existing networks. Physically, advanced routers are typically a rack with one or several Network Cards inserted in it, all connected through a high-speed Switch Fabric.

Figure 2: Cisco Nexus 7000 Series Internet Router, source [102]

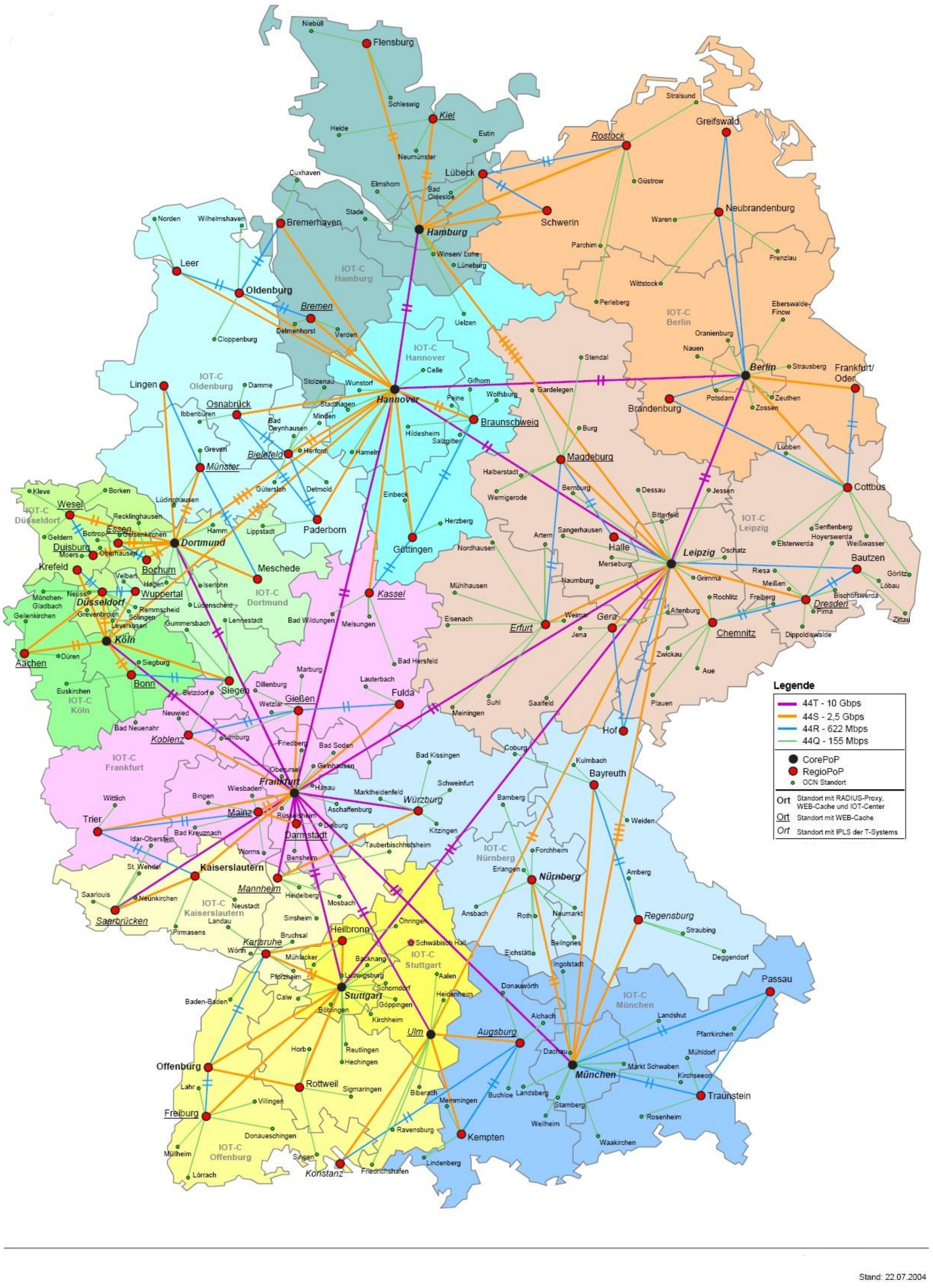


Figure 3: A geographical view of T-Online's backbone network in Germany (2004), source [103]

1.3 An overview of Network Processors

Back in the late 90's of the twentieth century, the industry had to address the mentioned problem of expanding requirements of processing power while simultaneously keeping protocol programmability versatile. At the moment, IP over SONET was been deployed to ensure efficient, cost-effective and high-speed communication in backbone networks [90]. In parallel, more traditional network devices like routers and switches were being joined by a new set of network equipment that provided functions like load balancing among several servers, network monitoring or security-related applications like firewalls or intrusion-detection systems. It was obvious that a unit capable of providing all these new services on a single device would gain rapid acceptance among carriers willing to maintain the technological edge without compromising their costs. This lead to the introduction of Network Processors (NP) which are application-specific processors designed for versatile packet processing at very high speed.

1.3.1 Functionality

From the point of view of the functionality, NPs are optimized for packet/cell forwarding applications. Even though current networks support a large amount of protocols, the most widespread protocol stack is the combination of the Transmission Control Protocol (TCP) together with the Internet Protocol (IP) transmitted over Ethernet or Multiprotocol Label Switching (MPLS) networks. In the Ethernet/IP case, the typical task of a router consists of simple IP forwarding, an operation that requires the following processing steps:

- Remove the link layer header
- Find the destination IP address in the IP header
- Look up the destination Table to determine the IP address of the next hop
- Determine link layer address of the next hop
- Add link layer header to packet
- Queue packet for sending
- Send or drop selectively the packet (if link is congested)

The frame's header is removed in the Medium Access Control (MAC) layer while a Direct Memory Access (DMA) unit performs packet data storing in the main memory under the supervision of a dedicated Packet Processor Engine (PPE) also in charge of memory allocation. A Packet Descriptor containing pointers to data is eventually generated and queued for a short wait until the Processing Cluster fetches it and executes the corresponding forwarding application. There, multiple PPEs modify the appropriate header fields; most importantly the destination address is updated with the results of a look-up in Ternary Content Addressable Memories (TCAM), which returns the next node address in case of an address hit.

After processing, packets are queued in accordance to the priority policies defined by the carrier. Packet output priority scheduling in NPs [91] may be from a simple round-robin mechanism to a very complex packet classification policy where different transmission quotas are assigned per user and traffic type, per traffic type, or even per active flow, which may require managing hundreds of thousands of queues and a remarkable processing intelligence.

The Look-up table is populated by routing protocols [6], which can be classified into two categories. In static routing protocols, the administrator is in charge of updating the contents of the table manually. Alternatively, dynamic routing protocols actively exchange information which allows adaptation in case the topology of the network changes. The simplest dynamic protocol exchanges the complete routing table periodically, thus temporarily overloading the network, as for example with the Routing Internet Protocol (RIP). More advanced dynamic routing protocols like Open Shortest Path First (OSPF) only update the link state whenever a topology change has occurred or after a look-up miss. In this second case, the protocol requests neighboring routers for a suitable next hop for the unmatched destination address.

If no result matches the destination address, the PPE generates an interrupt to the control plane, which usually runs on a separate General Purpose Processor (GPP). The control plane is also informed in case a malformed packet or one with its Time-To-Life field set to zero is detected by a PPE. In both cases packets are discarded and if the network supports the Internet Control Message Protocol (ICMP) an ICMP error message is sent to the packet source as well. A silent discard occurs when no ICMP message is generated.

1.3.2 Advanced applications

This relatively straightforward and repetitive list of actions suffices for the bulk of traffic and thus represents most of the workload for a Network Processor. Nevertheless, there is a trend to include more application complexity when forwarding a packet. Security is the main driving force behind the extra processing requirements, especially when packets are encrypted using the IPSec protocol. IPSec uses three cryptographic algorithms: HMAC-SHA1 for integrity protection and TripleDES-CBC together with AES-CBC for confidentiality, any of them consuming a big deal of processing power for en/decryption. As even a short burst of IPSec packets may hamper the normal operation of a NP, last-generation NPs include Crypto-cores in order to offload such heavy task from the CPUs. Some high-end devices achieve on-the-fly IPSec processing at line rates up to 10 Gbps [7].

Most recently, some routers are also in charge of discovering known virus-matching patterns in order to rapidly prevent the spreading of infections or preventing unfair use of bandwidth. These are just but an example of Deep Packet Inspection (DPI) techniques, which promise to provide the means to detect illegal materials like child pornography or copyright-protected exchange. More worryingly, the ability to closely monitoring the Internet enables carriers and governments to infringe privacy with near-complete impunity. DPI has been reported as already in use to intercept the communications of terrorist organizations as well as to control political opposition in authoritarian countries like China and Iran [70].

1.3.3 NP Architecture

The architecture of commercial NPs has some common characteristics, for instances the inclusion of multiple cores (either in a processing cluster or pipelined) or the shared memory buffer, but largely differs in the realization, as every manufacturer includes different intercommunication infrastructure or hardware accelerators to better focus on specific router tasks. On a single unit, a wide variety of components are to be found, starting with multiple packet processors together with a set of hardware accelerators responsible for processing intensive functions like data moving (DMA), en/decryption or traffic management. All these components communicate with each other through an array of buses, interconnection matrixes or direct interfaces. With such a large number and variety of architectural components on the same chip, NPs are a book-case example of System-on-Chip (SoC) units.

To gain insight into the complex architecture of an NP, it is a good approach to classify the multiple NP's components in four functional areas (see Figure 4), to proceed then discussing separately which bottlenecks need to be tackled of and what sweet spots are the most appropriate for their design trade-offs. As the old say goes, divide and rule.

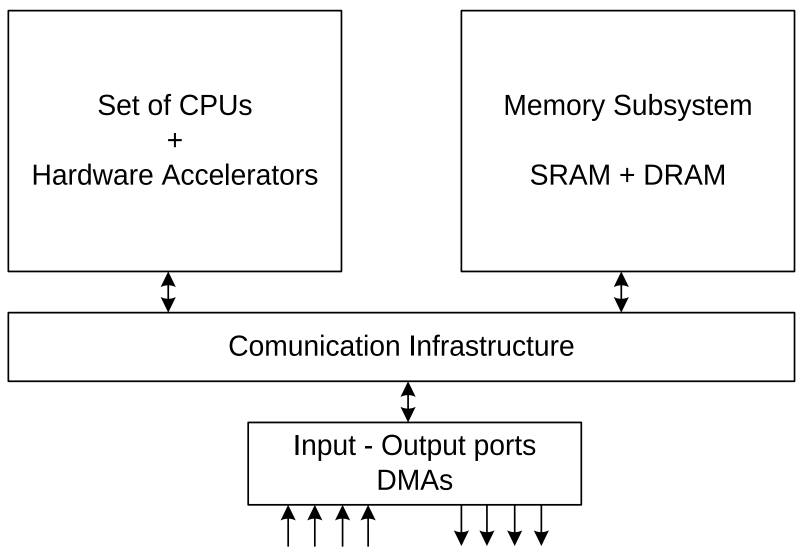


Figure 4: Functional blocks of a generic NP architecture

- First and most critical, all NPs include a processing core with multiple microengines or PPEs for data plane processing, optionally supervised by a more sophisticated processor for the control plane. Data plane functionality includes all essential operations for the packet forwarding, which are repetitive and done at high speed, thus ideally performed by many Reduced Instruction Set Processors (RISCs). In comparison, the control plane includes non time-critical functions such as updating routing tables, priority policing and congestion management. As programmability is clearly a must for the control plane, a General Purpose Processor is usually included to undertake the task while the forwarding capabilities of PPEs are complemented with a wide range of purpose-specific hardware accelerators. The number and function of these accelerators varies considerably among commercial applications. However, the need to offload tasks like IPSec decoding or packet queuing from the PPEs increases with the line wire, in an effort to liberate resources for packet processing.

- The system's communication infrastructure is in charge of connecting the different devices in the system. It must be dimensioned to provide enough bandwidth, resolve collisions and avoid long access times. In NPs, it consists either of a set of multiple buses for data transmission and control message parsing or of an interconnection matrix.

- The IO ports are the interfaces between the data links and the processor, being common to include from four to hundreds of ports while servicing different line speeds and applications. While a backbone router usually has a low number of ports, and access point usually services many users, in many cases with oversubscription. The most widespread interface for core processors is the Fast Ethernet standard Media Independent Interface (xMII). Particularly well-known are its sub-standards GMII (for gigabit Ethernet) and XAUI (for 10 gigabit Ethernet). A typical Fast Ethernet adapter presents two components, the Physical Layer Interface (PHY) that interfaces with the physical link (for example a CAT-5 cable) and the Media Access Controller (MAC) connecting the upper layers. Although synchronous data transmission is the current forerunner for Wide Area Networks (WAN), cell-based Asynchronous Transfer Mode (ATM) data links still make up for a remarkable part of the data transport infrastructure in SONET (American) and Synchronous Digital Hierarchy (SDH, European) networks, as well as in low-speed access links using ADSL. In this case, asynchronous data links are nowadays predominant in aggregation nodes placed on the edge of the network, for instance DSL Access Multiplexing (DSLAM), while synchronous Broadband Integrated Services Digital Network (Broadband ISDN) come only afterwards in a second place.

- Large Packet Buffering is necessary in current NPs in order to store packets during processing, to avoid packet loss during short episodes of port contention (when traffics from several inputs are multiplexed into one output port) and finally to allow prioritizing traffic according to Quality of Service policies. In that way, the Memory Subsystem is in charge of storing/fetching data from/to the in/output ports and the packet buffer. It plays a central role when designing the processor because of the high line rates it must meet together with the large buffer size required. Considering those constraints, the main Packet Buffer is mapped on off-chip DRAM, thus severely limiting bandwidth available, while faster SRAM is reserved for storing control data (linked lists, routing tables…) and cache memory, therefore becoming the size of control structures a critical factor.

From the point of view of the programming model, we may distinguish between Run-to-Completion (RTC) and Pipelined processors [22], being RTC processing the most widespread solution. Parallelization is a must in order to support high performance whereas high processing effort per packet is required. In RTC, one core fetches a packet from the input queue and performs most of the tasks required for packet forwarding. In that way, several packets can thus be processed in parallel hence making easier to balance the processing effort by dedicating some cores specifically to high priority queues while simultaneously reserving some CPUs for best-effort processing [47]. Inconveniently in RTC NPs, the packet sequence may be modified due to latency variations during processing hence requiring packet reordering before transmission [25]. Moreover, all cores run the whole IP stack, which consumes more instruction memory than pipelined CPUs, which execute simpler applications. Among its main advantages, RTC makes software programming easier as the processing effort in each core must not be carefully evaluate as in a pipelined structure, where a single slow stage can slow down the performance of the whole pipeline. Examples of RTC NPs are IBM’s PowerNP family or AMCC’s nP.

Pipelined processors are those where multiple cores or threads intervene in the processing of a single packet, which go trough multiple steps until completion. Advantageously, this architecture delivers a deterministic packet rate, as all packets going through the pipeline experience the same processing time, thus preventing out-of-order problems. On the list of drawbacks, it must be annotated that programming is more complex since tasks must me be equally distributed to avoid one step bottlenecking the whole processing chain. A good example of pipelined architecture is Cisco’s 10000 Toaster (Figure 5).

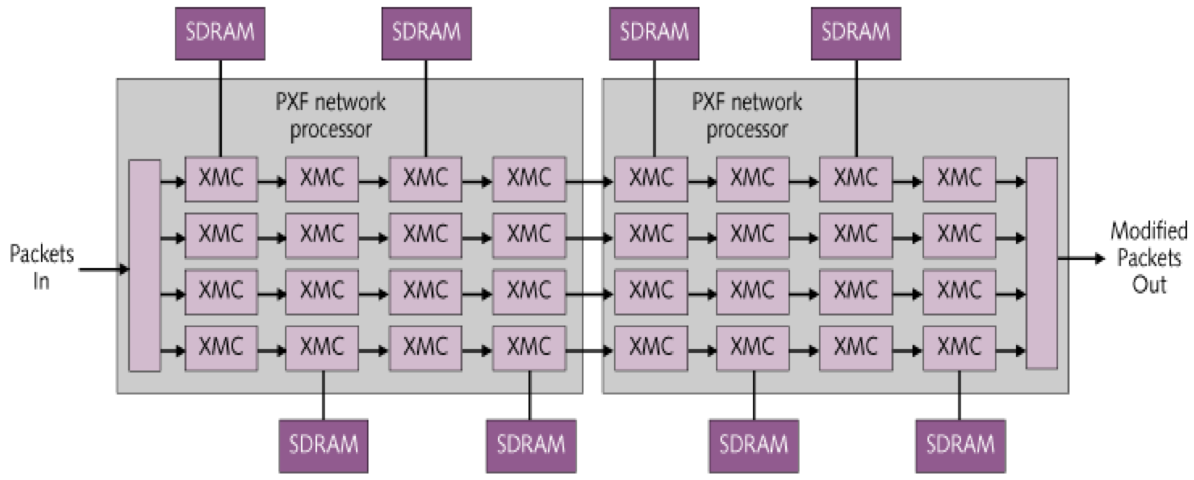


Figure 5: Parallel pipelining architecture in the Cisco 1000 Toaster, source [104]

As we have seen both processing architectures (cluster and pipelined) address different needs of packet processing, being difficult to produce a decisive conclusion on which architecture better suits all requirements of packet processing. Probably for that reason, manufacturers will probably keep using both for their designs in the future. Finally mentioning that the methods introduced in this scientific paper deliver satisfactory results regardless of the architecture of the processing core.

1.3.4 Performance bottlenecks and memory constraints

Current NPs experience two main system bottlenecks:

1. The most critical problem in Network Processor is the short processing time available to process every single packet due to the high packet rates to be processed. As a piece of example, each CPUs in a 64-core cluster dealing with a line rate of 100 Gbps has little more than 300 ns to forward every packet. This time has to be pondered considering the processor's frequency, say 2 GHz, which makes 600 clock cycles per packet. This strongly constraints the complexity of the software application that may run on the core. Figure 6 shows the processing time (in ns) available for a realistic number of CPUs and link rates, which is a de-facto roadmap for choosing the minimum number of cores required for the required processing effort per packet. As a piece of example, 32 cores servicing a 100 Gbps line would have un average 152 ns of processing time per packet while a cluster with 64 cores would allow up to 305 ns.

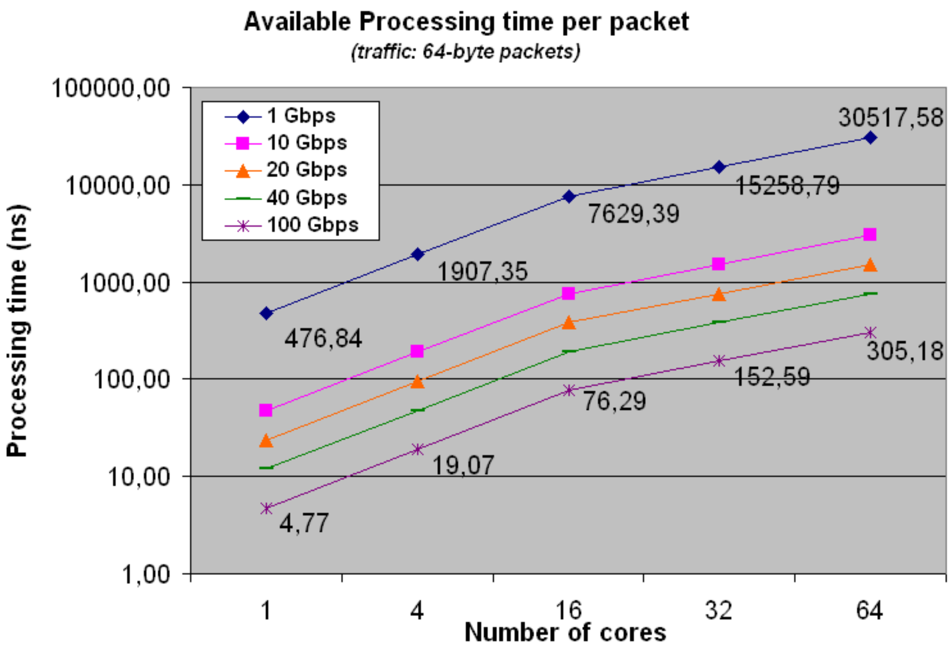


Figure 6: Average processing time per packet for a multi-core processor

2. A second challenge when designing NPs is the lack of memory bandwidth (BW) when accessing the Packet Buffer. As the input traffic rate (from now on, *R*) must be stored and retrieved at least once to/from the buffer, the total BW required from the memory device is at least *2R*. To make it worse, a portion or the complete packet is accessed during processing, which elevates the final memory interface's requirement up to *4R* in the worst case. Some researchers even include the 65-Byte problem [92] to push up this figure still further. This situation appears when the processor receives a burst of 65-Byte packets. Single bytes cannot be transferred efficiently to DRAM because the access granularity is either 4 or 8 words. Hence for a 64-bit interface, the minimum burst length is 32/64 Bytes. Hence storing a single Byte requires 31 or 63 Bytes of data padding. Theoretically this pushes the load in the memory interface up to *6R*-*8R* for a short time interval. In this work, the 65-Byte problem is

not considered a realistic scenario, as the probability of having 70-Byte[1] Ethernet frames in the traffic mix is 3,5% [10]. With these numbers, a stream of ten back-to-back 70-Byte packets may appear with a low probability of $2{,}75^{-5}$, what for a rate of 40 Kpps means once every second.

Finally and in addition to the bottlenecks above described, NPs are required to perform complex Quality-of-Service (QoS) and load balancing functions. Both concepts require demanding queue management, which in several cases implies traffic policing at flow level. Over high-load periods, millions of flows may be active simultaneously thus requiring an additional management effort from the processor cluster and the Queue Manager. This has been identified as a potential bottleneck in some first-generation commercial devices, for example the IXP1200 [13].

[1] 70 Bytes is by large the Ethernet frame size with the highest appearance probability in the problematic interval of 65 to 94 Bytes.

1.4 Contributions of this work

In the following chapters, the thesis proceeds by introducing, evaluating and finally implementing a new set of techniques for the improvement of the Memory Subsystem in NPs:

1. A new packet segmentation strategy that reduces the number of segments, increases access performance to the memory while delivering storing efficiency well over 90%. The system interconnection infrastructure (buses, interconnect fabric…) also clearly benefits from a load reduction.

2. A new method for administering a packet buffer containing several segment sizes. This not only reduces the amount of control operations but more importantly reduces ten-fold the requirement of control memory, usually implemented in expensive SRAM.

3. A packet injection system together with a processor's local buffer for storing the packet data. This concept allows for higher performance of the CPU-core by eliminating the latency when the processor accesses packet data. It also reduces the bandwidth requirement from the Packet Buffer, specially for packet patterns with large concentrations of small packets.

All of them are of the highest importance for the global NP's performance. But precisely because of the centrality of the Memory Subsystem in a NP, the analysis would be incomplete without also considering the implications of these new techniques in other parts of the system that may be critically affected by the alteration of the system's architecture. For instance, reducing the average latency of the Packet Buffer by transferring data in longer bursts of data increases the bus performance but negatively affects CPU's access time to the bus, eventually slowing down packet processing. Such undesired secondary effects are considered and when necessary traded off against the obtained benefits.

2 State-of-the-art

Since the first Network Processors appeared in 1998-9 they have been steadily increasing their performance and processing capacity. Many large manufacturers like Intel, IBM or Cisco have presented their products over this period, each with a different conceptual approach. It is very useful to review the evolution of commercial units over the last decade to understand the functionality and challenges that the Memory Subsystem faces nowadays, which is the main research field of this work. Besides, this chapter also focus on the academic publications that address the specific challenges of the memory subsystem, which are mainly related to the low throughput that can be obtained from dynamic memories. In addition, we will also review the state-of-the-art methods for efficiently administering the Packet Buffer, where millions of packets need to be stored, queued and retrieved at high speed, especially paying attention to the last research trends in packet buffering: per-flow storing. After this chapter, the work will proceed to introduce the contributions and investigations that were done during the SmartMem project.

2.1 Commercial Network Processors

2.1.1 Brief historical introduction to Network Processors

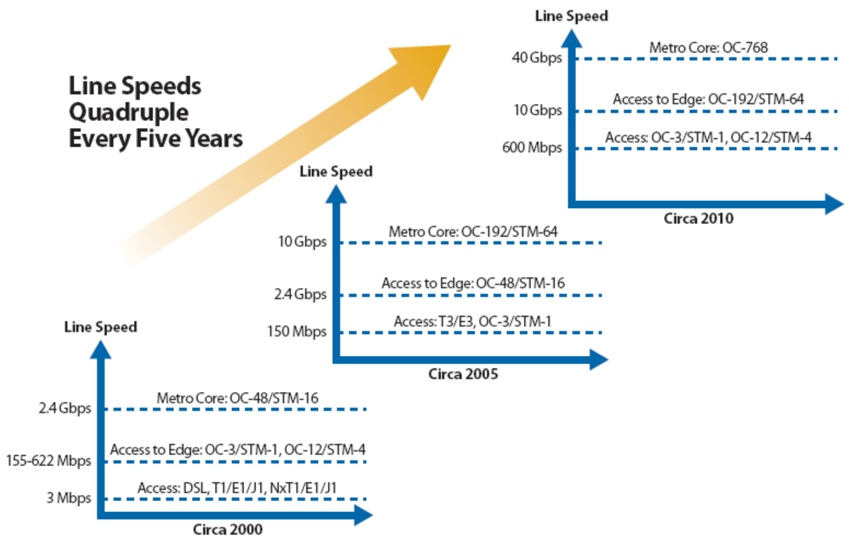
The first wave of high-end networking-specific processors entered the scene with the turn of the century, targeting 1 Gbps line rates [13]. At that point, networking seemed a promising niche of market worth billions of dollars [18], so the big moguls of the silicon industry (Intel, IBM, Motorola…) were eager to get their share by strongly investing in NPs to deliver a competitive products oriented to high-capacity WAN networks, that at the moment were being deployed by carriers like AT&T, British Telecom or Deutsche Telecom.

The overoptimistic growth forecasted produced a technological “gold rush” that brought a large number of established companies and start-ups to the market. The academic world supplied new chip architectures and paradigms for the new processors while the Network Processing Forum (NPF) was created to deliver the standards the industry was asking for. NPF’s most remarkable agreements were the Look-Aside Interface (LA), that lead to the generalization of Content Addressable Memories, as well as several Application Programming Interface (API) for software services like IPv4/v6, DiffServ or IPSec.

After the dot-com bubbled busted, 2001 saw how the NASDAQ index loss roughly 70% of its value. What initially was expected to be a market worth billions of dollars, resulted to be one of *only* hundreds of millions. In 2008, the global sales of Network Processor manufacturers reached $301 million [20] after robust but moderate year-on-year two-figure increases in the first years of the 21st century. In the face of this, some of the biggest vendors gave up and sold their NP divisions, as for example Intel did with its IXP family (now Netronome), IBM (now HiFn) or Motorola (now Freescale) while a large list of smaller manufacturers simply disappeared [19]. The NPF lost some of its initial impulse and was finally merged into the Optical Internetworking Forum in June 2006.

However, networking is still quite a lively business. Access infrastructure is slowly migrating from cupper wire to optic fiber and thus from narrow to broadband connections.

Access data rates, for both wire and wireless networks continue to climb and approach hundreds of Megabits. These two factors are pushing up for faster access nodes, which are getting similar to metro-class routers with link-aggregation features. Simultaneously, backbone networks are also scaling in order to accommodate the traffic increase. 10 Gbps links make up for the power horse of current networks, 40 Gbps networks are already implemented in some transoceanic links in the North Atlantic whereas 100 Gbps-capable architectures have been already introduced [21]. All in all, NPs have for sure a bright future, if only because the increase of internet traffic and the introduction of newer complex services will keep driving the need of processing capacity in the routers, both in the access infrastructure, edge and core routers (Figure 7).



Line Speeds Quadruple Every Five Years
Line Speed
40 Gbps
Metro Core: OC-768
10 Gbps
Access to Edge: OC-192/STM-64
600 Mbps
Access: OC-3/STM-1, OC-12/STM-4
Circa 2010
Line Speed
10 Gbps
Metro Core: OC-192/STM-64
2.4 Gbps
Access to Edge: OC-48/STM-16
150 Mbps
Access: T3/E3, OC-3/STM-1
Circa 2005
Line Speed
2.4 Gbps
Metro Core: OC-48/STM-16
155-622 Mbps
Access to Edge: OC-3/STM-1, OC-12/STM-4
3 Mbps
Access: DSL, T1/E1/J1, NxT1/E1/J1
Circa 2000

Figure 7: Line speed increasing for access, edge and core routers

In the following, this chapter overviews some chip examples belonging to different generations that appeared during the short ten years-life of NPs. Without the aim of being exhaustive, this work refers to the best known NPs, highlighting their most remarkable features.

2.1.2 The IXP family

Included in the first wave of NPs, Intel Corporation presented its successful IXP1200 [12] that rapidly became the reference architecture in several scientific publications, which make it a de-facto reference probably because of the detailed documentation availability. The family was later extended with the IXP2400 [26], which was designed to cope with OC-48 lines (2,5 Gbps) and the IXP2800 [27] that could meet OC-192 rates (10 Gbps).

The IXP1200 integrated a StrongARM processor for control plane functions as well as four/six multi-threaded RISC microengines (ME) working at 166, 200 or 232 MHz. A single SDRAM channel connected the processor to a 256-MB off-chip dynamic memory complemented by another 8-MB off-chip SRAM memory. Both memories were administered by a SW-based buffer manager running on one of the cores. The picture was completed with a set of internal buses and a peripheral IX bus to interface with the IO ports (see Figure 8). Curiously, the only on-chip memory available was a 4-KByte Scratchpad RAM used for CPU intercommunication and global data. In that way, no caching was implemented to reduce memory access latency though a large number of transfer registers were placed between the CPU and the Packet Buffer to foregone the cache absence, together with multi-threaded microengines to hide the access latency to DRAM.

The basics of this architecture were maintained in the second wave of IXP's NP family, this time with extended capabilities to target OC-48 and OC-192 line rates as well as to better cope with specific applications like encrypted traffic. The IXP2400 included eight 32-bit programmable microengines (each of which with 8 threads and a 640-word local memory) organized in two clusters. An additional QDR-SRAM interface was necessary to provide 3.2 Gbyte/s of BW for control data updating, which allowed for addressing up to 128 Mbytes of off-chip SRAM, with its corresponding high power consumption and cost in off-chip memory. These high requirements for the control buffer are in great part consequence of managing the memory buffer using linked lists of pointers.

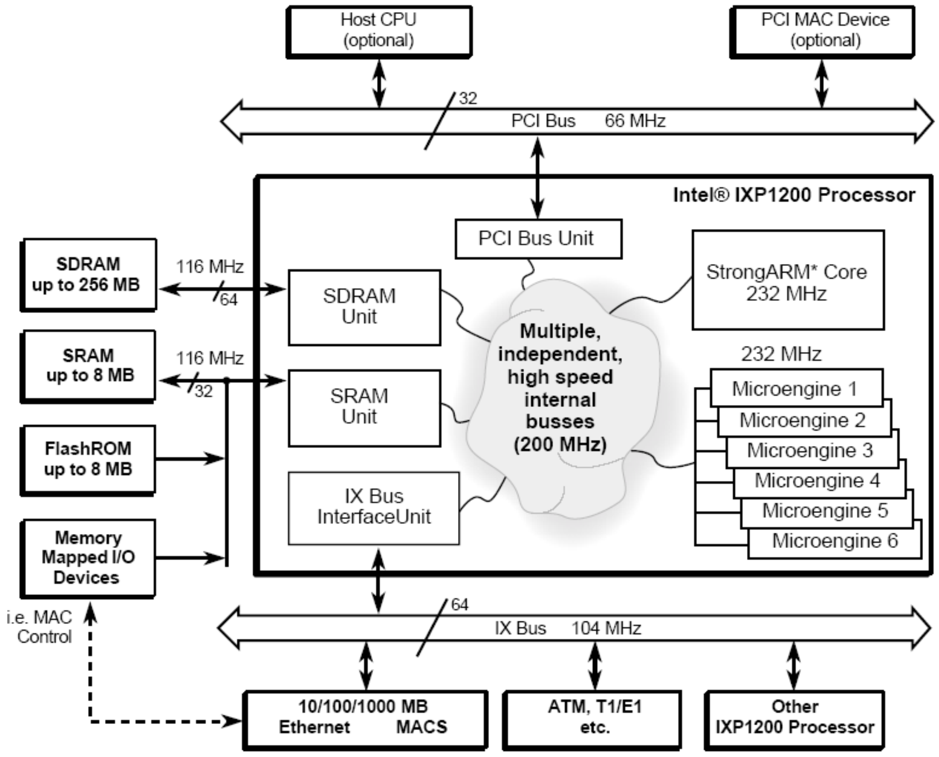


Figure 8: Intel's IXP 1200, source [12]

The Memory Subsystem was again further upgraded in the IXP-2800, with up to three RAMBus dynamic memory channels (peak BW of 1.6 Gbytes/s per channel) and four QDR-SRAM interfaces (with an aggregated peak BW of 6,4 Gbytes/s), which pushes the NP power requirement up to 25.5 Watts when working at 1,4 GHz. The number of microengines doubled that of the IXP-2400 but, in combination with a higher frequency, delivered 60 millions of operation per second, four times more than its predecessor. With these specifications, Intel claimed that the IXP2800 was capable of delivering payload processing at 10-Gbps line rates, even in the worse case when receiving a short-packet burst. Other variants of that processor were developed to target encryption applications, specifically the IXP2850, which was essentially the IXP2800 mattress upgraded with Crypto-cores capable of providing IPSec en/decryption at 10 Gbps (DES, SHA-1, AES algorithms). After presenting the IXP2850, Intel sold his NP family to the Netronome start-up under the agreement that Netronome would further develop the IXP franchise while keeping compatibility backwards with the IXP NPs [29].

Recently, Netronome announced its first Network Flow Processor, the NFP-32xx [30], a third generation NP targeting 20 Gbps line rates with 40 multi-threaded programmable MEs. The NFP-32xx is QoS-oriented where flows, instead of packets, receive differentiated attention, each flow being assigned with different forwarding priority, security requirements, and bandwidth. Netronome proposes a heterogeneous architecture that sounds familiar when compared to the former IXP2850: one GPP processor for control plane, multiple MEs (now Packet Processor Engines) for Layer 2 up to Layer 7 forwarding and finally security-specific accelerators for IPSec operations.

2.1.3 IBM's PowerNP Family

IBM introduced in 2000 a Network Processor that at the moment was probably the best device available architecturally as well as in terms of performance (4 Gbps), the NP4GS3 "Rainier" [17]. For data plane functions, a 16-strong Picoprocessor cluster was on place. Each engine was equipped with two threads running at 133 MHz. Besides, a PowerPC 405 was in charge of control plane functions and could interface with external units through a 32-bit PCI bus that enabled extended flexibility. The IO interface was flexible as well, as most optical data transmission standards were supported (from OC-3c to OC-48c). For Ethernet compatibility, four Gigabit Ethernet could be selected or alternatively forty Fast Ethernet Ports. From the point of view of the memory subsystem, a 128-KB on-chip data buffer were shared by all processors while an advanced Queue Manager offloaded the tasks of packet segmentation and reassembly from the CPUs while simultaneously administering ingress and egress buffers without the need of CPU supervision.

Some other particular characteristics differentiate it from the IXP family:

- Scalability factor: up to 64 NP4GS3 can communicate through the same Switch Fabric, thus easily enabling for a large router with up to 1024 ports.

- Hardware-accelerator approach: Packet forwarding in the Picoprocessors is assisted by several co-processors, for example to perform table searching, register access scheduling and flow control among several tasks. This design policy allows the "Rainier" to deliver deterministic performance in many traffic scenarios, which for instance is not the case of SW-centric IXP family when these processors administer a high number of flows.

- The memory subsystem: it is independent in the ingress and the egress paths. In the ingress (traffic flows from input towards switch fabric), only small queuing is required due to the high BW of the switch fabric. On the egress side (from the switch fabric to the output ports), DDR-SDRAM is the choice for accommodating a larger queue system, capable of buffering big amounts of traffic in case of port contention. Two independent Ingress/Egress Traffic Managers undertake autonomously packet reassembly, en/dequeing of packets and QoS policing.

Following the NP4GS3, IBM introduced in 2003 the second generation of its NP franchise, the NP4GX, which was the first NP that could meet line rates of 10 Gbps. In fact, the system architecture is pretty much the same though as principal novelties we may distinguish a superscalar PowerPC 440 and the smaller 0,13 µm integration technology that allows for higher operation frequency (500 MHz). This gives the NP a deeper processing capacity and in accordance, the instruction memory had to be enlarged to 64 KB to make room for more advanced SW applications.

The networking business of IBM ceased and its related patents were sold to HiFn in 2004, soon after the NP4GX entered the market. HiFn reoriented the PowerPC family towards security applications, somewhat neglecting the throughput race. His 5NP4G [31] only targets 4 Gbps line rates but it increases the processing capacity up to 2,128 MIPS to allow for more sophisticated packet forwarding protocols. Time-to-market and chip cost reduction seems to be main improvements in this last member of the PowerPC family, which recently in 2009 pass to Exar Corporation after its acquisition of HiFn.

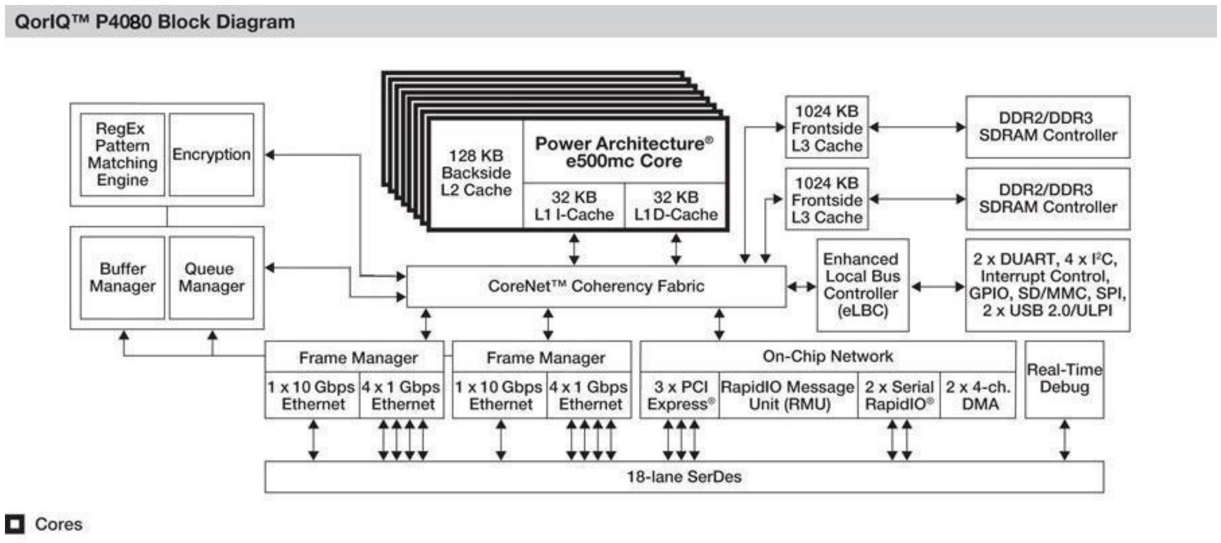
2.1.4 Motorola/Freescale

Motorola was one of the big companies that developed its own NP architecture, the C-Port family, consisting of the C-5e OC-48 and the C-3e OC-12 NPs [33]. The design philosophy was closer to that of IBM’s PowerPC, with multiple Channel Processors assisted by a set of hardware accelerators performing classification, traffic management and interestingly a Buffer Manager Unit (BMU). Focusing on the C-5e, 16 Channel Processors (CP) deliver processing flexibility and 4500 MIPS. A CP is composed of a RISC core (supporting a variant of the MIPS instruction set), with two (ingress/egress) Serial Data Processors, which are programmable pipelines that assist the CPU with common packet processing tasks like Cycle Redundancy Checks (CRC) or protocol en/decoding.

Motorola is one of the only manufacturers (together with Intel) that discloses details about the NP’ intercommunication infrastructure: the C-5e has two internal high-capacity busses, one for packet headers and another differentiated for payloads, which are used to transfer data from the IO ports to the off-chip SDRAM memory, a process that is supervised by a Buffer Manager. Besides, a global bus is included to enable message passing and control communication between cores and accelerators. Finally, the Table Lookup Unit (TLU) not only speeds the search in the Lookup table but also helps as flexible high-speed classification unit, capable of performing 46 millions of IPv4 searches per second while being flexible in the actual classification algorithm used [34][35].

Recently, Motorola split all microelectronic activities and created the Freescale spin-off that continues giving support to the C-Port NPs. Freescale’s QorIQ P4080 [94] is its latest 45-nm technology NP (Figure 9). It is a medium-range unit allowing for two 10 Giga-Ethernet links while consuming as low as 30 Watts even at full performance. Eight RISC cores, called Power Architecture e500mc, empower the packet data plane each implementing large caching structures, with a 32-KB Level 1 together with a 128-KB Level 2 cache. In addition, the SRAM requirement is further enlarged in order to accommodate a 2-MB shared L3 platform cache placed in front of the main memory. The intercommunication infrastructure allows for up to 800 Gbps of BW while connecting the CPU cluster to two 64-bit DDR2 (or alternatively) DDR3 SDRAM memory controllers with interleaving features.

A whole army of accelerators is on-duty to improve packet parsing, classification and data distribution. The Queue Manager undertakes packet scheduling, packet sequencing, and congestion management as well as memory allocation. Somewhat surprising is the fact that the functions of encryption and pattern matching are included together with the Queue Manager. Data transferring between the IO ports and the Packet Buffer is done using two 4-channel DMA engines.



QorIQ™ P4080 Block Diagram
RegEx Pattern Matching Engine
Encryption
Buffer Manager
Queue Manager
128 KB Backside L2 Cache
Power Architecture® e500mc Core
32 KB L1 I-Cache
32 KB L1 D-Cache
1024 KB Frontside L3 Cache
1024 KB Frontside L3 Cache
DDR2/DDR3 SDRAM Controller
DDR2/DDR3 SDRAM Controller
CoreNet™ Coherency Fabric
Enhanced Local Bus Controller (eLBC)
2 x DUART, 4 x I²C, Interrupt Control, GPIO, SD/MMC, SPI, 2 x USB 2.0/ULPI
Frame Manager
1 x 10 Gbps Ethernet
4 x 1 Gbps Ethernet
Frame Manager
1 x 10 Gbps Ethernet
4 x 1 Gbps Ethernet
On-Chip Network
3 x PCI Express®
RapidIO Message Unit (RMU)
2 x Serial RapidIO®
2 x 4-ch. DMA
Real-Time Debug
18-lane SerDes
Cores

Figure 9: Block diagram of the Freescale's P4080, source [94]

2.1.5 EZChip

EZChip technologies is a fabless semiconductor company based in Israel that was already present in the first wave of NPs and, remarkably enough, is still today in the market delivering leading networking solutions even for the high-end range. It is worth mentioning the successful NP-2 [36], one of the first chips targeting 20 Gbps and its follower, the NP-3, for the 30-Gbps range. The architecture details have not been disclosed but it is capable of simultaneously maintaining millions of flows while addressing a Packet Buffer of one GB, together with claiming a particular specialization in advanced traffic shaping policies. It is revealing that while targeting such high speeds, the NP only provides buffer capacity for one Gb, which does not correspond with the traditional thumb-of-rule to calculate the buffer size. In other words, the Packet Buffer is proportionally much smaller than in other NPs.

EZChip NP's are empowered by a cluster of TOPCore engines, which are not RISC cores but application-specific pipelined processors where parsing, lookups and packet modification are done step by step. Several pipelines are implemented in parallel in order to scale performance to the desired rate and thus high throughput is achieved, while packet latency during processing is reduced nearly seven-fold [37].

The last member of this processors family is the NP-4 [105] appeared during 2010. It delivers up to 100 Gbps in half-duplex mode (only ingress or egress) incorporating DDR3-SDRAM memory for the Packet Buffer. The NP-4 focuses on voice and video streaming and is capable of maintaining millions of TCP sessions simulatenously.

2.1.6 Cisco’s Network Processors

Cisco debuted late in the NP scene (2004) with its Cisco Silicon Packet Processor, securing a remarkable market position even having missed the first wave of NPs. Its follower, the Cisco QuantumFlow Processor [39] was a thoroughly developed NP, product of a considerable investment effort worth $250 million. This processor was specifically developed for being deployed in edge routers. When it was presented in 2008, its performance was beyond their closest competitors, delivering between 10 and 100 Gbps with 40 Tensilica cores that were able to handle each up to four threads [38], which allows the simultaneous processing of 160 packets. Performance depends though on the actual processing effort, only achieving the maximum line rate for basic IP or MPLS packet forwarding.

With the QuantumFlow NP (Figure 10), Cisco claims to be filling a flexibility gap that previous NPs did not. For example, the configurability of the pipeline is broader (and thus more network protocols are supported) and options in traffic management and classification are expanded by supporting up to 128K queues. For administering such high number of queues, hardware queue management is mandatory thus reserving more-flexible PPEs for protocol processing, as IBM did with its PowerNP, together with requiring a large control buffer.

Regarding the load balancing in the processing cluster, the QuantumFlow includes a hybrid system where packets are mostly executed in a Run-to-Completion mode but can be as well reassigned to a second processing thread or hardware accelerator if needed, all with a high degree of flexibility. This task assignment is controlled by an execution manager, which can be programmed in multiple ways thus allowing the reservation of processing power for specific traffics. In other words, this architecture can be viewed as a matrix of processing units fed by a configurable task dispatcher. This allows for example executing the IP stack on a PPE thread and then encrypting the packet with a CrytoCore.

Last but not least, the Memory Subsystem is divided in two parts: one for the PPE cluster and one for queuing and output buffering. Both have their dedicated SRAM and DRAM external interfaces with the corresponding hardware accelerators managing the interfaces autonomously (without CPU intervention).

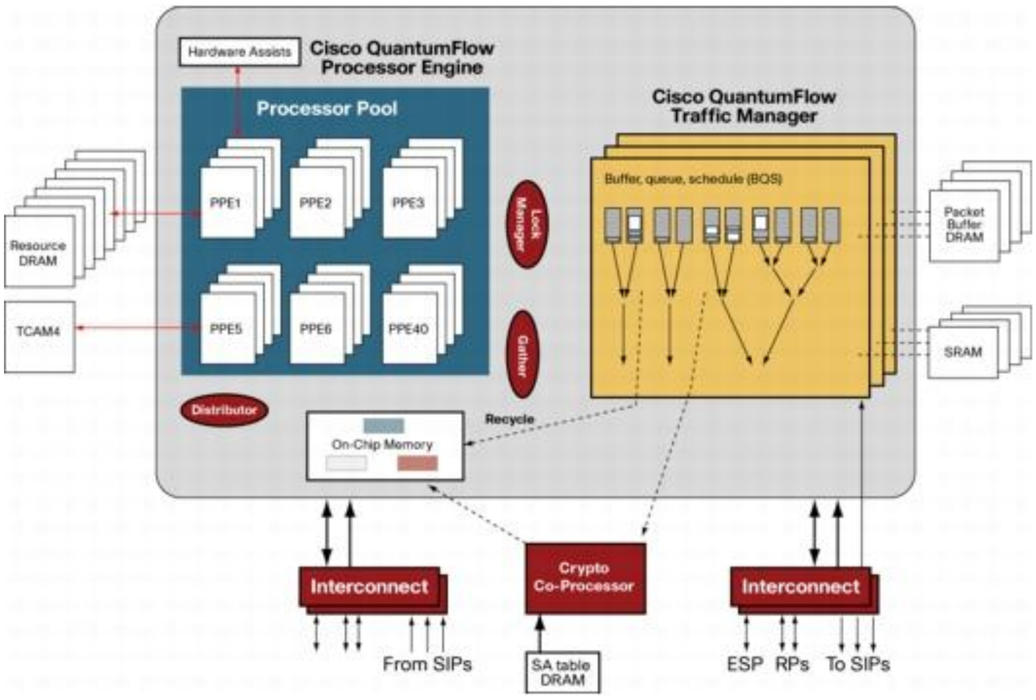


Figure 10: Cisco QuantumFlow Processor Used on Cisco ASR 1000 Embedded Services Processor, source [104]

2.1.7 LSI's Network Processors

LSI (originally Agere) is an American manufacturer of networking solutions that has in the PayloadPlus NP family its flagship product line. Their units target medium wire-speed networks by supplying NPs ranging from 300 Mbits/s to 5 Gbits/s, essentially for simple fast layer 2 switching. These cost-competitive units are mainly oriented at carrier Ethernet networks while supporting VLAN, link aggregation, packet policing and traffic management as well as many security features.

To focus in their latest chip, the APP3300 (Figure 11) enables low-cost packet switching and is mainly oriented towards DSLAM aggregation nodes as it supports several DSL variants like VDSL2, VDSL, ADSL2+, ADSL2 and plain ADSL on a per port basis. Architecturally, the system is formed out of two chips, one for fast-packet forwarding module (Data Plane) and the second Control Plane, based on two general purpose ARM11 cores as well as a Crypto-core. The data plane is monitored through a high-speed management port.

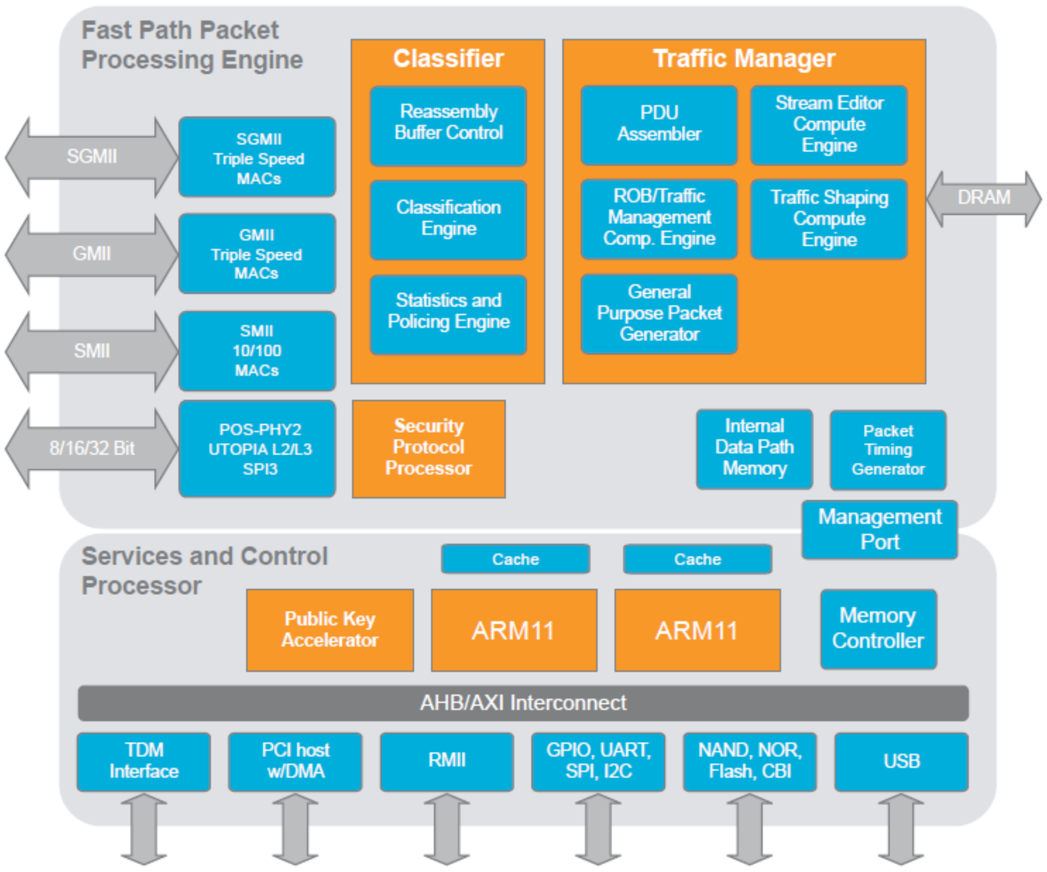


Figure 11: APP3300 Block Diagram, source [105]

2.1.8 General trends in Network Processors

- Line rates serviced keep quadrupling every five years.

- Protocol and application complexity keep increasing as well, which asks for more intelligence in the routers especially when the box is placed at the network edge and thus implementing traffic classification, policing and/or encryption.

- Header processing, as required for layer 2-3 MPLS and IP forwarding, is being joined by payload processing applications that enable more complex layer 4-7 applications like DPI. These applications have a bright future in a broad range of new fields like security, traffic monitoring and virus inspection.

- The need of more processing capacity is being addressed by increasing the number of cores at faster clock frequencies while being assisted by an increasing number of application-specific HW accelerators. In the case of pipelined processors, the number of stages and parallel pipelines tend to grow to thus enable a more powerful processing.

- Packet Buffer sizes are still increasing although not any more linearly with the line rate, both because scientific research has proved it unnecessary as because a growing buffer size cannot be cost-effectively sustained with the current memory technology.

- In addition to packet processing, flow processing is gaining acceptance. In flow processing a packet is firstly identified as belonging to a particular flow and then assigned with previously stored processing results. This saves accesses to the look-up table as all packets belonging to the same flow are conveyed to the same output, which also easies traffic shaping. Supporting complex QoS agreements per user enables user-specific traffic shaping (bandwidth and latency) thus increasing revenue.

- Partially as a consequence of the previous, the complexity of queue administration and the number of queues to maintain (already in the order of hundreds of thousands) are clearly raising. This leads to the specialization of NPs, some dedicated to traffic management while others to very high-speed plain packet forwarding.

- Voice and gaming are emerging as two network applications with strict latency requirements. These are just but two examples of Differentiated Services or DiffServ. For such time-critical traffics, packet identification and flexible queuing must be enforced and accordingly billed.

- Energy consumption is increasingly been observed as a major concern because the exponential growth of power requirements in each new NP generation, which cannot be sustained in the long run. Thus, new units have appeared targeting medium-power applications. In order to curve down power growth, off-chip static memory requirement must be downsized together with allowing for the implementing of other energy-saving techniques like dynamically adapting the system’s clock frequency to traffic load.

- Hardware virtualization provides support for the implementation of different Operating Systems on the same silicon. A common case is when for instances, a real-time OS (RTOS) is used for data plane whereas an embedded Unix-based OS runs on the control processors, thus easing the interface with the programmer. This trend arises several questions about inter-OS communication or shared resources management.

2.2 The Memory Subsystem in Network Processors

There are three main buffers in a NP. The biggest is the Packet Buffer, where packets are stored during packet processing (which usually requires short buffering) and more critically during port contention episodes, which may require a large memory to prevent packet loss. In addition, the Packet Buffer is used to queue packets for traffic shaping purposes. Second, the Control Buffer stores the control structures used for administering the Packet Buffer as well as the structures to manage the high number of queues a typical NP supports. Finally, large Look-up Tables contain the default routes where to forward the incoming packets and are mapped on Content Addressable Memories (CAMs).

Two basic storing technologies are used in networking: one-transistor Dynamic Random Access Memory (DRAM) and six-transistor Static Random Access Memory (SRAM). Three electrical characteristics define their performance, namely their power consumption, integration density and access latency. While DRAM density is around 28 times bigger than SRAM and its power consumption is about 250 times that of SRAM, row activation latencies in the order of 40 ns make DRAM a slower option in comparison to the approximately 4 ns of access time in a SRAM device [42].

Considering these memory technologies, designers usually find themselves confronted with a trade-off between buffer size and performance. Either the buffer is implemented using low-performance large DRAM devices or relying on high-performance but costly and power-hungry SRAM resources. In that sense, the buffer size has a direct influence on performance because when it cannot be accommodated in SRAM, a DRAM solution becomes unavoidable hence slowing the buffer operation. With current requirements tending towards one GByte, DRAM is the only realistic option to accommodate the Packet Buffer. Though other complex devices like RAMBus, which allows low-latency accessing to DRAM, have been used in networking, they have been discarded due to its manufacturing costs and proprietary fees.

The Packet Buffer size is calculated following a rule-of-thumb *(1)* where Round-Trip-Time (RTT) stands for the approximate latency a packet incurs when going from the source to the sink when crossing the network and back (hence the factor 2) while C stands for the line rate. The formula comes from early estimations for a router forwarding packets of single flow and in the hypothesis that the TCP was the only transport protocol [95]. According to this rule, a router servicing four OC-192 links (40 Gbps) for a commonly-accepted RTT value of 250 ms requires over 1200 MB. The use of DRAM with such values becomes unavoidable.

$$B_{size} = 2 \cdot RTT \cdot C$$

(1)

As a matter of fact, this provision is clearly oversized. TCP regulates the transmission speed according to the number of reception acknowledges it receives within a defined time window. The accepted theory states that implementing enough buffer capacity in the node allows the source to reduce its output flow in the event of network congestion. This prevents packet loss hence avoiding triggering retransmissions that overload the network [96].

Though this way of calculating the buffer size has become a kind of non-written standard in the industry, numerous publications have observed empirically as well as demonstrated analytically that modern NPs handling thousands (when not millions) of flows never make use of more than 10% of the buffer capacity [44][45]. This comes as a product of the smoothing effect of packet-accumulation probability of millions of flows (see Figure 12), where the worse case is highly improbable. However there is a built-in resistance among carriers to accept NPs with smaller memories, albeit some third-generation NPs specify somewhat smaller buffers, as mentioned above in the case of the EZChip NP-3.

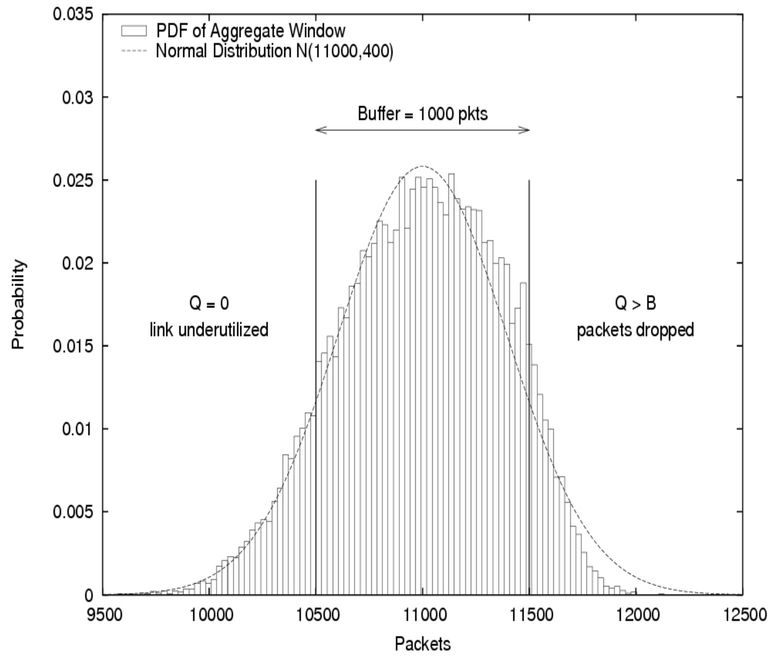


Figure 12: Criterion for sizing the Packet Buffer according to the probability of packet accumulation, source [44]

So far, we have discussed the reasoning behind using DRAM for the Packet Buffer without tackling the bandwidth problem. In NPs, data is transferred first from IO ports to the memory. Afterwards the packet header is retrieved by the CPU in case of normal processing while the full packet is read by applications requiring the packet payload as well. Processed data are then written back in the Packet Buffer to finally be retrieved for transmission. With this model the bandwidth required from the Packet Buffer oscillates between *2R* and *4R*, being *R* the accumulated input line rate. Remember that this rate is eventually higher in case of a burst of short packets with small sizes, 64 Bytes for instances. In such scenario, padding-data is needed to complete the minimum burst size due to the big access granularity of current DDRx-SDRAM memories and so the total data rate may be even larger.

The BW requirement of the Packet Buffer is further increased if context-associated data is used for keeping intermediate processing results, which also must be stored in memory. This can be done in a separate context buffer administered separately [40] or back-to-back appended to packet data (either as header or trailer) [46]. Anyhow, the burden of administering context data is particularly significant when the incoming traffic pattern consist of small frames, as context data becomes a significant part of the total amount of data.

Current DRAM technologies struggle to deliver BW as high as *4R* or even *6R* in high-end applications, as a consequence the Packet Buffer emerges as one of the main performance bottlenecks in NPs. Real access efficiency in DRAM is usually below 50% of the theoretical maximum throughput, for reasons later addressed in this chapter. Only by including multiple memory channels (and their corresponding external interfaces) can the Packet Buffer deliver enough BW. This solution cannot scale with the line rate for several reasons (number of interfaces, power consumption, load balancing complexity…) so buffer designers find themselves in a race to propose alternative solutions. The current forerunners are per-flow o per-queue storing schemes that improve access efficiency, although such systems still have to introduce convincing solutions related to the excessive SRAM requirement, before becoming mainstream in NPs.

But performance is just one half of the problem, being power consumption and the number of memory channels also matter of concern. Memory resources must be carefully allocated to reduce both the amount of external memory chips required (thus reducing the NP pinout) and the ever increasing power dissipation, especially SRAM, since a 4-MByte chip consumes in the order of 1W. To illustrate the need of constraining power consumption in current routers, let us focus on the Juniper M120 [43], a powerful router for backbone routing which can accommodate up to four network cards delivering a maximum performance of 120 Gbps. For such standard equipment, admittedly high performing, the power supply installed is capable of delivering 2,1 MW. To better reference this high consumption, this supposes 0,06% of the total energy generation at the huge Tricastin four-reactor nuclear facility (France), which supplies a maximum of 3,820 MW [97].

Behind this high consumption in backbone routers, memory is a major energy claimer for two reasons: first because of the amount of power dissipated in a memory with high activity, and second due to the need of dissipating all that heat with fans or other cooling methods that again consume power. All in all, in our previous Juniper example, memory accounts for a non-despicable 11% of the total chip consumption [43].

For the mentioned power consumption reasons as well as to reduce the number of off-chip memory channels, this thesis endorses a reduction in the Packet Buffer size. Such a design decision also brings advantages by lessening the amount of control data and thus the requirement of SRAM. Control data is a generic group that encloses all those structures necessary for keeping track of free memory, linking packet segments (segment descriptors), packet descriptors or queue descriptors. A main challenge for any administering method consists in constraining the size of control structures hence reducing the SRAM requirement. Keep in mind that DRAM memories are highly inefficient for control data because they are not suited for multiple short memory accesses, as required for control update.

Although reducing the Packet Buffer size diminishes the number of segment descriptors, a more decisive improvement in the size of control structures can be achieved by optimizing them. Smarter control structures, for example without linked lists, can lessen not only the Control Buffer size but also the effort of control data updating. Even an on-chip SRAM solution for the Control Buffer begins to be feasible insofar as the control structures are reduced from MBs down to hundreds of KBs, which in addition removes the need for off-chip SRAM memory. Due to the importance of such an architectural improvement, this work presents a buffer administration system that achieves that goal.

In addition to control structures requirement, the NP requires on-chip SRAM for storing the SW application together and other temporal data generated during processing. Because such a memory is quite expensive, the size of the SW application must be constrained to hundreds of KBs. Alternatively, some processors include instruction caches and then keep the SW application stored in DRAM, which allows for more complex protocol processing.

The routing table is the last but by no means the least source of challenges for the designer of the memory subsystem. The SW application needs to look-up millions of entries to find an adequate next hop destination address. The look-up must be executed fast and been finished in a deterministic number of cycles in order not to stall the CPU operation. When the key is small, a simple SRAM can be used as the key corresponds directly to the physical memory address where the destination address is kept. However, it is usual to use the five fields of the IP-header, the so-called IP five-tuple (source address, destination address, protocol type, source port and destination port), as a key for the look-up table to find the destination address so the size of that table must be much larger [48].

In order to reduce the number of entries in the look-up table, a Hash function can be used to constraint the size of the look-up key, although this can generate address collisions that need to be solved. The hash value is then used for looking up in a Ternary Content Addressable Memory (TCAM). TCAMs enable having a single destination address for a broad group of keys, which is very useful to assign the same destination to all packets directed towards for instances address 194.xxx.xxx.xxx. When in addition to the 5-tuple more parameters need to be considered for packet classification, a more sophisticated method consisting in parsing a decision tree becomes a better design choice due to the increased number of address collisions and the logarithmical-nature of TCAMs scalability [35].

Although routing tables require in a typical NP a big portion of the total memory resources, they are out of the scope of this work because of their independent implementation with respect to the main memory subsystem. They do not interfere with the Packet Buffer operation and so they do not affect the claims in our investigations.

2.3 The structure of the Packet Buffer

2.3.1 Terminology

In the following chapters, the text often refers to a block of memory using a myriad of terms that may be puzzling for the reader but are necessary for precise qualifying. For the sake of clarity, it follows a short listing of the terms most often used to refer to memory portions, sorted out from the smallest memory size up to the largest.

Segment: small portion of memory, usually between 32 and 2048 Bytes. Multiple segments may be required for storing a single packet.

Memory block: a memory block is a broad term that refers generically to a small portion of memory however without clear-cut definition of function or association to any particular management method. The term Block is inherited from high-level programming languages as C++ or Java, where the object/structure is allocated to one memory block.

Memory portion: In essence it is the same as a pool of segments but with the particularity that the size is fixed and encloses completely the available buffer space. A memory portion occupies a continuous physical range in the memory. Moreover, they are associated with Buddy Memory Allocation[2] in opposition to the paging method. The term is often used by Intel and IBM when describing the Packet Buffer organization in their second-generation commercial NPs. In comparison, the term pool is preferred for the description of our methods because it is more appropriate for memory areas with dynamic sizes.

Memory pool: A pool of segments is a subdivision of the Packet Buffer containing several pre-allocated segments. Each pool can only contain one type/size of segment though the number of segments can be either statically defined or dynamically resized upon requirement. For dynamic expansion of a particular pool, it may be necessary to allocated additional memory space and for that reason it is not necessarily mapped on a physically-continuous space.

Memory page: A page is a 2-4 Kb memory area mapped on a single memory-row in DRAM devices. Cache lines are mapped on a same row thus reducing bank conflicts and increasing performance when using an open-row policy (i.e. no automatic pre-charge). It is the mainstream memory organization used by operating systems in GPP.

Memory heap: this term refers specifically to a dynamically-assigned portion of memory during program run-time. A heap is returned to the free status either after a free command or when a garbage collector sets it back to the free list. A garbage collector is a program that parses the memory searching for unused memory blocks wrongly discarded.

Buffer: This is a big portion of memory reserved for a particular set of data like packets, control data or processor instructions. We may thus refer to them using Big Case, for instances the Packet Buffer, the Control Buffer or the Instruction Buffer. In that way, the Packet Buffer alludes to the whole memory area where packet data are kept.

[2] Buddy systems allocate a long set of same-sized objects in memory even if they are not required to thus reduce memory fragmentation [99]. Usually only powers of 2 are allowed. It is a quite old segmentation method already proposed in the 1960s [98].

Address range: a range of addresses refers to a physically-contiguous block of addressing space, not only necessarily memory. A single buffer is usually assigned with a single range of addresses.

Memory device: a memory device is the silicon hardware that provides the circuitry to keep data. In that sense, SRAM/DRAM devices are always the external chips or the internal on-chip resources providing the physical support for data storage.

2.3.2 The need of Packet Segmentation

Organizing the Packet Buffer is trivial in a packet switching device where the packet order remains exactly the same in the input as in the output: the buffer organization can actually be as simple as collection of queues, one per output port. Inconveniently in NPs, packets may follow several parallel internal data paths crossing the NP while experiencing variable queuing times that decisively modify the outgoing packet order. This reordering is magnified by traffic-shaping policies, ever increasing in complexity, which give higher priority to some flows over others and thus the packet order in the output may end up completely altered from the order in the input. A queue organization does not work in NPs.

Using no buffer organization at all does not work either, as it leads to running out of large contiguous portions of free memory. This happens when short packets are transmitted, leaving behind a fragmented memory space lacking big continuous spaces, the ones feasible for storing incoming large packets. Only by using high-effort defragmentation methods or garbage collectors [54] can memory been compacted again albeit at the price of copying data internally and thus increasing the memory bandwidth requirement. This is called external fragmentation loss (internal refers to unused space inside an allocated segment) and can be successfully avoided by segmenting the memory space available into fixed-size pages, according to terminology used in GPP operating systems like Linux or DOS/Windows. In GPP, the page size is usually equivalent to the memory's row size (2-4 KB) in order to improve performance by avoiding multiple row-activation penalties. As in NPs the page size used is much smaller, memory blocks are commonly referred to as segments.

The size of the segment is critical for different reasons. Very large segment sizes are not used in networking applications because the typical packet pattern is different than that of GPP applications. For instances, Ethernet frames are between 64 and 1518 Bytes long [51], including MAC source and destination addresses, Ethernet type field and CRC trailer. In these values the preamble and frame delimiter fields are not included, as they are only required for synchronization purposes and thus discarded in the physical layer. To accommodate these sizes, a smaller segment size is usually chosen to foregone wasting large portions of memory when storing the smallest packet (low internal fragmentation).

The preferred size in the first generation of commercial NPs like the IXP12xx series, PowerNP NP4GS3, Freescale's C-Port C-5 or in the experimental PRO3 prototype [41] was 64 bytes. Two main factors led to this decision: (1) small segments offer a fair storing efficiency [50] and (2) in NPs servicing ATM networks, where cell size is 53-byte long, it is convenient to use segments rounded-up to the next power of two.

Inconveniently, the use of short segments to store large packets is inefficient because the large number of segments a large packet is segmented into, up to a maximum of 24 for a

1518-Byte packet. In [53] we observed an inverse linear relation between the segment size and the Memory Subsystem performance. This comes as a consequence of the increased effort of managing many small segments as well as the reduced effectiveness of transferring data in short bursts to/from dynamic memory.

De/allocating a new segment requires updating the corresponding control structures and thus accessing the Control Buffer. The more often this operation is needed per packet, the higher the BW demanded from the control memory. But the most critical effect is reducing the access efficiency to the Packet Buffer, typically mapped on DDR-SDRAM devices. Due to high row-activation latency, these devices favor storing/fetching long bursts of data to/from a single memory row. As the Packet Buffer is usually a key bottleneck in NPs, short accesses should be avoided inasmuch as possible. Besides, each segment is usually assigned with a one-to-one associated control descriptor. That descriptor can get in some systems as big as four or eight bytes, therefore pushing up the Control Buffer size requirement what, as we have seen, must be expressly avoided too.

Newer devices utilize more complicated schemes. The Intel IXP24xx and IXP28xx families allow for dividing the Packet Buffer in up to three partitions. The partitions coexist, each with a different segment size, though each packet must be segmented using **only one** segment size and hence end up completely stored in the same partition. The sizes selected for the three partitions are 64, 128 and 256 Bytes, but not all traffics are stored using that scheme, with several just being stored in the main 64-Byte partition. Besides, as partition's sizes are static there is no guaranty that when one of the partitions is overrun, new packet must be stored in larger portions with available space. The Intel approach is similar to an IBM patent where the Packet Buffer is divided into two partitions, each of which corresponds to a segment size. The packet is then segmented according to its length and then stored to the appropriate partition [55]. Even though multiple segment sizes are available, packets cannot be segmented using segment sizes from two different buffers what would achieve much better storing efficiency if we consider that a 257-Byte packet wastes 255 Bytes of the second 256-B segment.

As mentioned above, segmenting packets into 64-byte segments leads to a performance reduction due to the fact that it necessitates multiple bus and memory transactions per packet. In contrast, larger segment sizes can benefit from reduced bus arbitration and higher DRAM performance since they can take advantage of higher spatial row locality [56]. The inflexible solutions that are found in current devices can only take advantage of this by sacrificing memory efficiency.

This work presents a solution that achieves both. On the one hand, storing efficiencies close to that of 64-Byte segmentation and on the other, a very low number of segments per packet (well below 2 in average) thus improving memory accessing. Using multiple segment lengths and advanced packet segmentation algorithms we can achieve it without compromising the memory utilization. The space allocated to each segment size is dynamically adjusted upon requirement. Using multiple segment sizes for packet storing presents several challenges: which segment sizes are better suited for the traffic mix present in internet, how to manage the multiple segment buffers, which control structures are appropriated, which is the impact on performance, how many segments are optimal for a single packet, and a long etc. To the best of our knowledge, no scientific work has even suggested the topic for networking, let alone presenting a thorough study about that question. This is then the central topic of this Ph.D. Thesis, though not the only one as, during the investigations, multiple length segmentation revealed itself as key for additional advantageous mechanisms for administering the Packet Buffer.

2.3.3 Control structures to manage the Packet Buffer

The control data structures must be carefully designed to minimize the size of SRAM and number of control operations. In NPs, we need to consider mainly as control data the destination address stored in the Look-up Table, intermediate processing results (e.g. context data), buffer-associated control structures (e.g. linked lists), packet descriptors and queue descriptors. Numerous studies are oblivious of the implications that, for example, the storing of a small 64-Byte packet with 32-Bytes of associated context has upon the memory throughput. In a similar way, parsing a list of 24 pointers for every single 1518-Byte packet is a very demanding operation, which is required when using 64-Byte segmentation. The number of segments in the Packet Buffer is indeed a critical parameter for the NP performance. Despite this critical impact, the different overloads that control data introduce are often either not thoroughly studied, patchily addressed or simply directly ignored in scientific literature. Here on the contrary, control data structures for Packet Buffer management receive prime attention in this work.

2.3.3.1 Linked Lists

Packets are stored segmented in the buffer to prevent buffer fragmentation. Maintaining a linked list of pointers and storing them in one-to-one associated segment-descriptors is by far the preferred method to keep track of those segments in NPs [41]. Segment descriptors may not only contain pointers but also other control data such the bytes-valid field, the end-of-packet flag along with occupation status. Linked lists are widely used because of their simplicity, effectiveness and flexibility to modify the memory area assigned to one packet. As their principal drawback stands the effort of parsing/updating a large amount of pointers, let alone the huge amount of control memory resources needed to store the multitude of segment addresses.

Figure 14 represents schematically a typical Packet Buffer organized in 64-Byte segments. Each data segment (left) has an associated descriptor containing a pointer to the next descriptor (second column by the left). This pointer contains the **logical** address instead of the much larger **physical** address. All segments are originally linked by a single “free list” originally pointing all segments in the memory. When the segmentation algorithm requests new addresses, free positions can be easily reached by reading the first position in the free list. The segments assigned to an incoming packet are then split to another list whose head and tail pointers are kept in the Packet Descriptor (PD). A typical size for those descriptors is 4 / 8 Bytes hence the amount of control data is either a 6,25 / 12,5% of the data buffer size (for 64-Bytes segments).

The queue system can be built either by queuing PDs or using a second level of linked lists as shown in the picture. The second method has the advantage of flexibility and simplicity to move packets between queues, a most desirable feature taking into account that the packet descriptor is transferred several times from queue to queue in a NP. In its turn, each queue requires a descriptor pointing to the first and last PD in the queue. Queue descriptors also require memory space, although this is of minor concern as the PD identifier is much smaller than that of data segments.

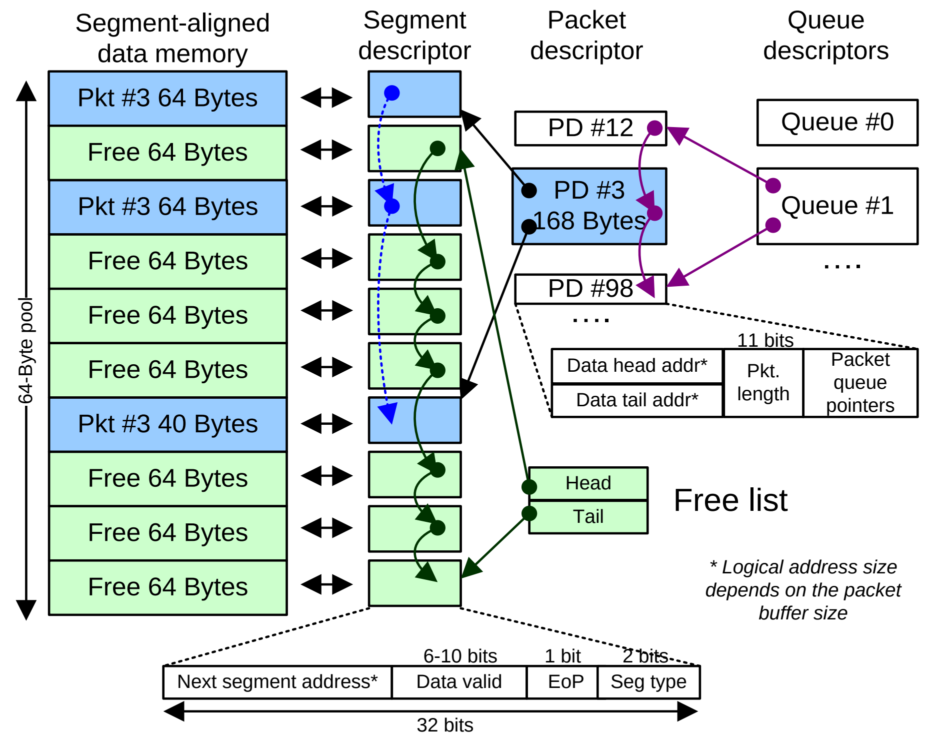


Figure 13: Linked-list-based control structures.

Figure 14 on the contrary represents schematically a typical Packet Buffer with two separated static buffers for different segment sizes. In some second generation NPs, specifically Intel's IXP2800, packet memory is divided into three portions, each with a different segment size (the sizes chosen are 64, 128 and 256 Bytes). The number of segments required for medium and large packets is thus substantially decreased. The control structures are quite similar to those used with one portion, only that one additional free list is needed per buffer together with some sort of control information indicating the segmentation used.

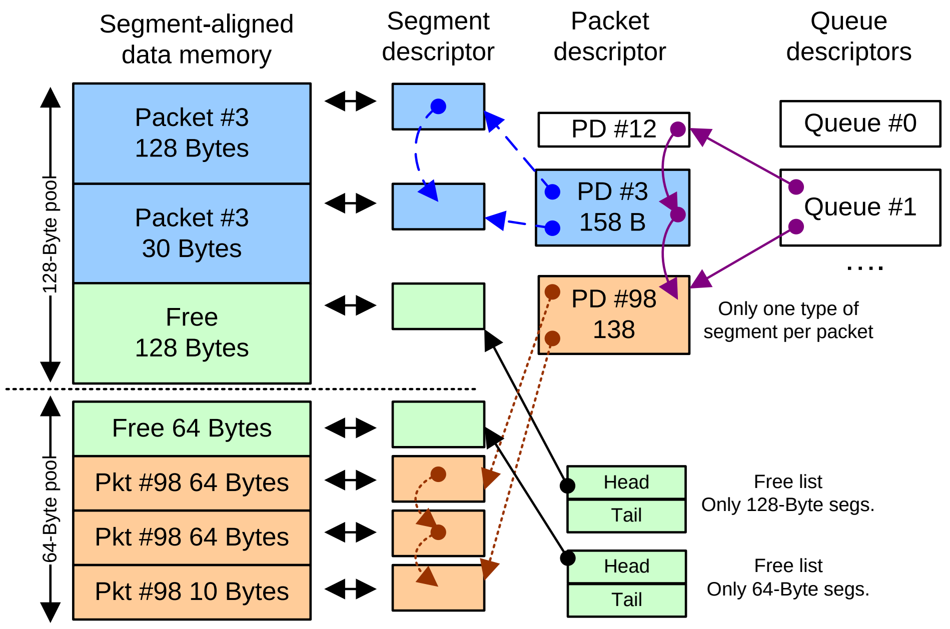


Figure 14: Administering a Packet Buffer with multiple segment sizes using linked lists

Even with multiple portions and low storing efficiency, single linked lists are quite inefficient in case packet resizing is necessary. This is a frequent operation in edge nodes when packet processing adds a new header or removes encryption at the end of an IPSec tunnel. Any operation that implies shortening or enlarging the packet length can only be accomplished by removing some segments. In a *single* linked list (with only **one** pointer towards the next segment) it is impossible to parse the linked list backwards. This is necessary to identify the previous segment of a node. If a node must be removed (say #4), the previous element (#2) in the list must be redirected to point the following element (#5), which can only be done keeping a double linked list pointing both forwards and backwards or alternatively parsing the list from the beginning. This last method is non-deterministic and extreme inefficient for long lists.

Double-direction parsing is possible by expanding segment descriptors to include another pointer directed to the previous segment in the chain [41]. Inconveniently, double linked lists need enlarged segment descriptors that further increase the demand of control memory. This leads almost inevitably to a much slower DRAM-based implementation of the Control Buffer or otherwise including several SRAM channels.

We should not forget that performance is a critical factor when designing the Control Buffer. All those pointers and descriptors must be updated with every new packet operation (allocation, de-allocation, resizing, PD switching between queues…). In order to provide enough speed for control data accessing, nearly the only solution is to map the Control Buffer on pricey SRAM, either on-chip or with multiple off-chip SRAM channels. Off-chip SRAM is somewhat slower than its on-chip variant because of the delay introduced by the circuitry necessary to access the off-chip memory (tri-state pads). In comparison, on-chip memory's performance depends only on the technology used. As a piece of example, Intel's current fab-edge process of 32-nm SRAM technology entered production in Q4 2009, claiming error-free operation at 4 GHz [68] and thus delivering more than enough throughput for the Control Buffer. This excellent performance must be pondered with the high cost and power consumption of a large buffer implementation in SRAM. To put it in values, the 2-Mbyte QDR-SRAM at 167-MHz (the off-chip device used in the IXP 2400) delivers 333 Mbps per pin but requires a sustained power of 1,12 W.

Alternatively, a header or footer can be allocated in the segment together with packet data and thus allow space for the control descriptors. This saves static memory resources at the price of slowing down the updating of control data and the Packet Buffer, as shown in this [40] study of different memory subsystems.

To sum up, the size of control structures becomes much more critical than that of the Packet Buffer as the required performance of updating control operations need SRAM-based buffering. At the moment, an unsustainable trend links the growth of line rates with the size of the Packet Buffer, and through it, the size of control memories. For that reason, it is of critical importance to decouple line rate increasing from SRAM requirement. Moreover, using external static memory cannot be avoided because the amount of on-chip static memory at the designer's disposal is constraint by the cost of chip area along with its heat dissipation.

Even though the central point of SmartMem regards packet segmentation using multiple segment sizes, early during the study became clear that reducing the requirement in number of segments per packet leads to strong reductions in the size of the Control Buffer as well as in the effort required for upgrading control data. Chapter 4 provides insight into a bitmap-based method that meets the administration requisites of a buffer accommodating multiple segment-sizes namely in an efficient way (see Chapter 4).

2.3.3.2 Buffer administration in other computing applications

To the best of our knowledge, no concrete publication studies the implementation of a buffer with different segment-sizes in the context of networking and consequently there is no academic research on the administration of such a buffer. However, the area of General Purpose Processing (GPP) is prolific in literature and mechanisms to dynamically manage variable-size blocks of memory. Though not recent, this study [69] is a most recommended gateway to low-level memory management in General Purpose Processing, as it includes a comprehensive collection of references as well as a clear taxonomy of methods. Here we review some of the most interesting ones, which can be applied to any storing device (Hard Drive, DRAM…) containing objects with variable sizes.

The most basic low-level mechanism consists of a single/double linked list chaining free blocks regardless of their size (used ones are not tracked), a slow technique that requires parsing the list to find the desired block length. Segregating blocks per size and tracking them using separate lists accelerates the search but requires of a huge number of lists, in consequence several similar-sized blocks are often grouped. In some cases, the block address is used as a criterion to sort the list, with the aim of concentrating live objects in a particular part of the memory.

The allowed block sizes may be constrained to the powers of two or to the values of the Fibonacci series, which not only eases block splitting and coalescing but also allows again reducing the number of lists. Another popular mechanism uses bitmaps to mark whether the block is free or busy and later track it with a hierarchical tree or chain blocks of bits with linked lists.

It is also of interest giving some pointers about how non-embedded operating systems administer the Packet Buffer in software-based routing applications, however slow performing. Linux for instance builds chains of packets using a double-linked list structure [52]. Incoming packets are assigned to a 2-Kb segment, regardless of its length. Such segments have an associated structure called *sk_buff* containing several pointers to the different headers and trailers inside the IP packet. Then, the *sk_buff* is linked together with other structures to construct the packet queue. A final queue descriptor, the sk_buff_head, contains a pointer to the first and the last member of the list as well as a counter with the packets on the list (Figure 15). Differently to the double linked list in Figure 14, *sk_buff* structures contain a pointer to the queue descriptor to facilitate finding the queue descriptor from a packet reference. In addition, functions as *trim(), push()* or *pull()* provide a greatly simplified administration interface to the programmer, allowing for example packet shortening/enlargement with a simple call to function *trim*. In spite of its simplicity, the Unix method does not suit NP's specifications due mainly to the large internal fragmentation incurred when small packets are stored in such large buffers.

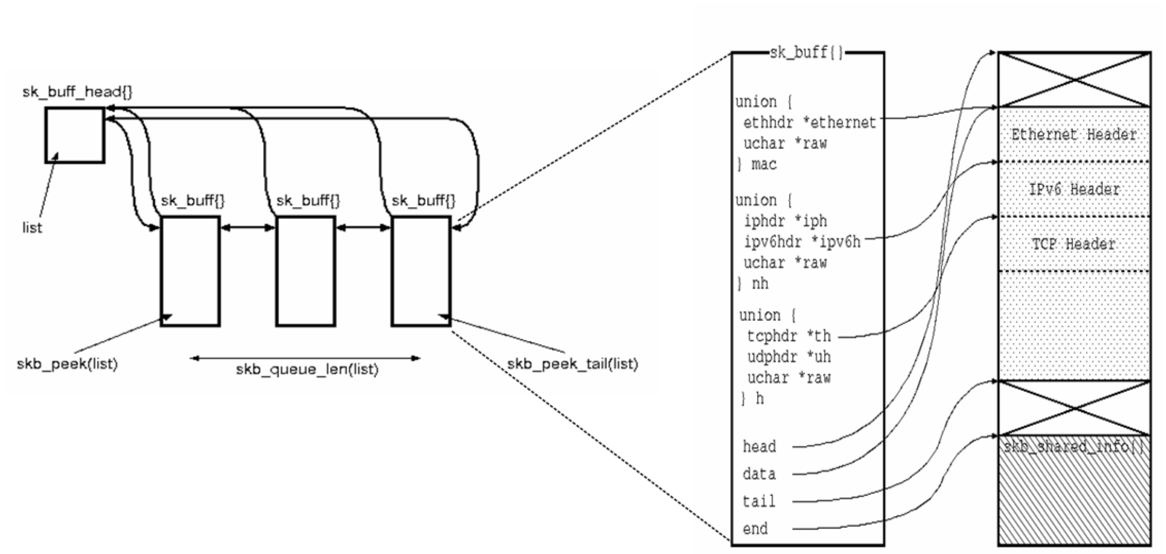


Figure 15: UNIX's sk_buff structure for storing data packets

2.3.4 Packet Buffer administration in a per-flow basis

Recently, several scientific publications have revisited the idea of organizing the NP's Packet Buffer in a per-flow basis. It consists in administering the Packet Buffer as a **collection of queues**, one per data flow, instead of segmenting the packet and storing its parts without considering the physical position of the data in memory. In this section, we review whether this practice offers a suitable solution for tackling the many challenges of the Memory Subsystem and if not, which are its shortcomings.

A flow is defined as a unidirectional sequence of packets among the user and the targeted server, with all of its packets sharing the same 5-tuple values. Every new flow can be assigned with its own memory space (fix or dynamically resized), which facilitates storing all packets belonging to the same flow in a FIFO-structured buffer. Therefore, it is possible to access the memory in a deterministic way that completely prevents bank conflicts or in other words, it is possible to maximize the access throughput by reading/writing large blocks of data in the memory instead of accessing the memory in a per-packet basis. In addition, no packet segments neither its associated control structures are required.

The mechanism is actually a hybrid memory system where the tail and the head of the flow queues are kept in SRAM. Whenever a tail overflows a threshold, its data is transferred to DRAM. In a similar way, if the head of the FIFO is below a defined threshold, the scheduler refills it fetching from DRAM or directly from the tail. The main problem of such a system is the large SRAM requirement, which scale linearly with the number of flows. First in 2006, March and Corbal [58] established the SRAM requirement for the tail as $Q \cdot b$, where Q is the number of queues while b stands for access time to DRAM. Moreover, if considering a memory with M banks where multiple accesses can be pipelined (Figure 16), according to them this requisite goes down proportional to M.

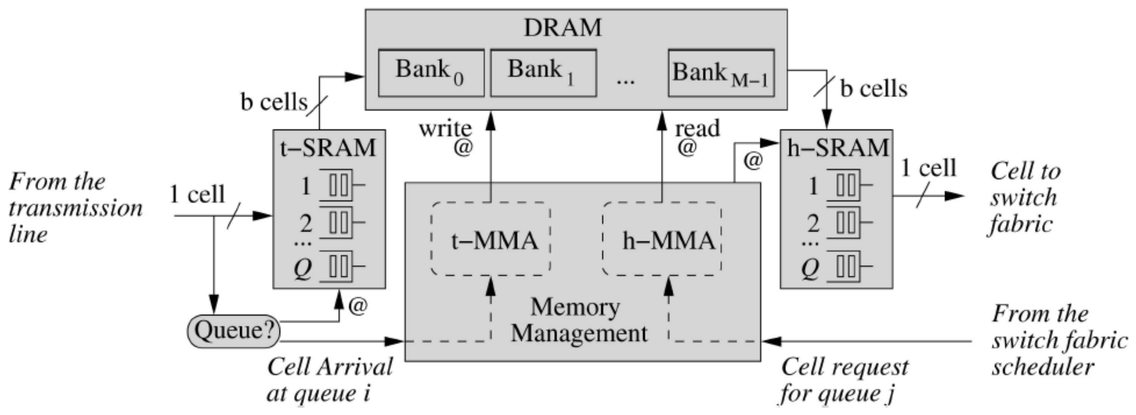


Figure 16: Method proposed by García-Vidal et al. for conflict-free storing in packet switchers, source [58]

Iyer et al. [57] accepted the linear relation between *Q* and the SRAM size for the tail queue, though demonstrating that *b* is actually the block size transferred to the memory and so equivalent to the access granularity. The result applies only when the memory space can be reallocated dynamically to each tail, which requires an additional mechanism for memory administration as for instances a linked list. If a dynamic management for the tail cache is considered too complex and so the tail is administered statically, the SRAM requirement grows according to $Q \cdot b \cdot (3+lnQ)$.

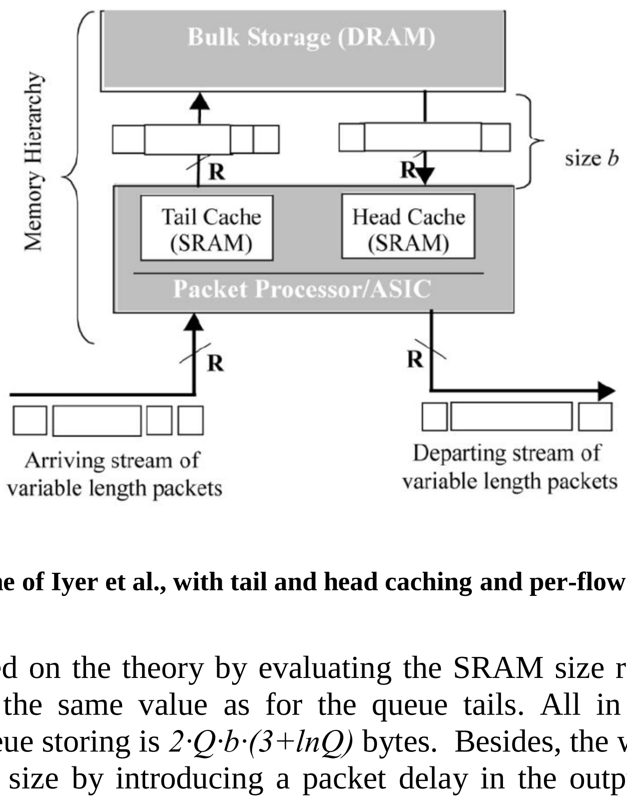


Figure 17: Memory scheme of Iyer et al., with tail and head caching and per-flow memory administration

Iyer et al. elaborated on the theory by evaluating the SRAM size required for the queue head, while obtaining the same value as for the queue tails. All in all, the total SRAM requirement for per-queue storing is $2 \cdot Q \cdot b \cdot (3+lnQ)$ bytes. Besides, the work considers how to reduce the head queue size by introducing a packet delay in the output, then preemptively fetching from DRAM those queue heads to be transmitted next. Such a **look-ahead buffer** allows a finer operation in the read scheduler hence obtaining a better adjustment. The first head queue due to underflow is thus prioritized when fetching from DRAM, which enables a slight reduction in the SRAM space allocated to the head cache though it requires a small queue for the look-ahead buffer containing the next memory requests. Finally, they concluded that when the size of the look-ahead buffer increases, there is a reduction of the head's queue size to a minimum of *Q·(b-1)* with a maximum packet latency of *Q·(b-1)+1* time slots. Inconveniently, this is still depending on *Q* so even if queues are small, it is still not possible to implement the thousands of queues required in current routers for supporting QoS.

Resuming, Iyer et al. propose a scheme where the fast-memory requirement was defined as in *(2)*, with memory blocks of *b* size while the total number of queues in the system is *Q*. According to that, if we consider a system with 1000 queues and a memory block 128-Bytes long the SRAM final SRAM requirement is established in 1,4 Mbytes (considering both head and tail memory). Even for a relatively low number of queues, the SRAM memory is quite big.

$$SRAM_{size} = Q \cdot b \cdot (3 + \ln Q) + Q \cdot b$$

(2)

By using the look-ahead buffer together with the queue head, this requirement is reduced. In that way, a trade-off between packet delay introduced and the size of header queues is established, being the new memory requirement redefined as in *(3)*. With the same values and considering maximum delay, the authors claim that the SRAM requirement can be reduced down to 256 Kbytes.

$$SRAM_{size} = Q \cdot (b-1) + Q \cdot b$$

(3)

Wang et al. [59] recently added to the discussion demonstrating that the actual SRAM requirement can be further reduced in the tail by a half because flows are ergodic and stationary, which is generally true as traffic flows from different sessions can be modeled as independent stochastic processes. They proposed a variant of the previously described per-flow storing where only one queue is maintained in SRAM, both in the head and the tail instead of *Q*. The main difference with previous schemes consists in that now, packets are transferred to DRAM individually instead of in blocks of *b* data. *Q* queues are however maintained in DRAM in a per-flow basis though now split among several memory devices. This enables fetching several packets from the same flow (and thus continuously) whenever the scheduler grants that flow access to the output. The mechanism relies however in the fact that a single flow has a long transmission time (the flow’s throughput is low) so the transmission is not interrupted when the Scheduler switches between flows quickly enough. Still, it allows storing packets continuously in DRAM, though without clearly specifying a proper memory mapping in order for bank conflicts to be completely avoided.

2.3.5 The Buffer Manager accelerator in NPs

Several tasks of packet processing are iterative and need high processing requirements without requiring a great deal of flexibility, making them perfect candidates for being transferred from the CPU to specialized hardware accelerators. This trend is fuelled both by the scarcity of processing resources and by the repetitive nature of packet processing. Two striking examples of that are security functions like IPsec encryption, which are assumed by Cryto-cores, and more relevant for the topic here discussed, the Buffer Manager (BM), which autonomously stores and fetches data in/from the Packet Buffer while taking over Packet Buffer administration.

Administering the system's memory consists in allocating, discarding and resizing memory buffers upon requirement. Memory administration is an extensive topic common to all computing branches, but it is particularly demanding in networking. The nature of packet processing, where data is continuously flowing, makes the updating of control data a high-effort operation. Moreover, administering free memory and discarding allocated buffers is repetitive and can overload the CPU processors. According to the accepted HW/SW co-design theory, static and repetitive tasks at high speed make ideal candidates for being implemented in a HW accelerator and thus offload the CPU.

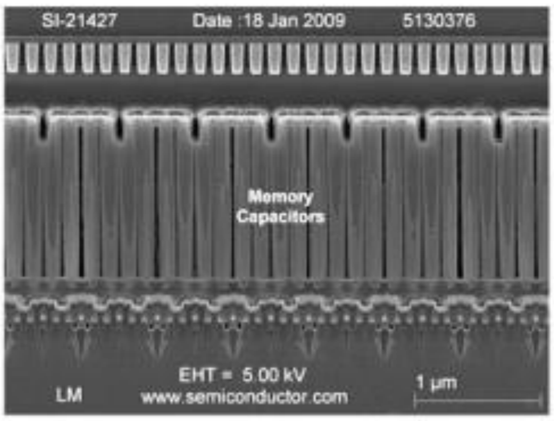
The supervision of data transfers between IO devices and the Packet Buffer can be offloaded from the CPU as well. One way is instructing Direct Memory Access (DMA) devices to move data but this solution still requires CPU supervision and is inefficient as data is transferred twice over the bus, even when scatter & gather DMAs are employed, thus increasing the bus load. When the BM performs the complete packet storing/fetching, data is transferred only once over the bus. Gries et al. [11] observed a forwarding performance loss in the Intel's IXP-1200 when dealing with traffic composed of short packets, where a dedicated PPE is in charge of data moving. The heavy CPU overload produced by the effort of transferring packet data was identified as the source of this slow performance. We confirmed the large effort of packet storing/fetching with a VHDL-coded demonstrator running on a FPGA-based prototyping platform [40].

A good example of a Buffer Manager for hardware-assisted buffer administration was implemented in the PRO3 NP [41]. Vlakos et al. described a system where the CPU is only interrupted when the packet has been completely segmented, memory has been allocated and data transferred to the buffer. On the transmission side, the function is symmetrical: a Traffic Manager indicates to the Buffer Manager which is the proper order to transmit packets in accordance to the corresponding QoS policy. Then, the BM fetches the segments belonging to the outgoing packet from memory, reassembles the packet and transmits it to the corresponding output port. A final factor that recommends the use of BMs regards packet segmentation. By using a BM accelerator, advanced packet segmentation algorithms, as the ones presented in chapter 0, can be implemented without requiring additional CPU processing resources and with a very minor increase in chip area. For those reasons, a **BM is considered the most suitable platform** for implementing and testing the new techniques and methods introduced in this work rather than using a SW-based approach. This is done even after considering the lower effort of coding such algorithms in C++ in comparison to the more demanding implementation and debugging process of a hardware unit. Hence, the mechanisms discussed will be implemented in the SmartMem Buffer Manager prototype, which is a VHDL-coded HW accelerator that deliver high performance while achieving average storing efficiencies over 90%, as well as requiring a 10-fold smaller Control Buffer than state-of-the-art applications.

2.4 The memory wall: the bandwidth bottleneck

2.4.1 DRAM technology

That DRAM is the preferred memory for bulk storage in most computing applications has its roots on a simple fact: cost. DRAM technology has nowadays the edge in VLSI technologies due to its regular structure that allows for a larger integration on a silicon wafer. For instances, one Gbit DDR2 50-nm chips are already in production [63]. With larger integration comes lower cost and with lower cost, use becomes widespread and economy of scale appears. Finally, this feeds back again by reducing production costs in a self-reinforcement process. Other DRAM solutions, like Rambus (RDRAM) which provide reduced-latency access have been successfully tested in networking environments (for example the IXP-2800) however due to their price and licensing fees it has been replaced by commodity DRAM in following devices to in that way curve down line-card costs [64].



Cross section of the memory cell array imaged by SEM

Figure 18: Microscopic view of a state-of-the-art (2009) 50-nm DRAM cell from Micron

Inconveniently, high access latency prevents DRAM devices to reach the maximum theoretical throughput as this requires a succession of long burst accesses without bank conflicts, which cannot be guaranteed. DRAM has longer activation latency than SRAM due to its much simpler construction. It is based on a MOSFET transistor that conceals the charge of a capacitor that may either hold a ~10^{-15} Farad charge ('1') or be discharged ('0'). The total charge depends on the technology, so as it gets smaller, so goes the number of electrons the capacitor may hold. Whenever the transistor's gate is active, the value stored can be written or read, but an access always erases its value (destructive accessing). A row of sense amplifiers is needed to sense the small charge in the cell and then convert it to a logical value, which takes some time depending on the voltage applied (the higher, the faster). Voltage is actually shrinking in order to reduce leaks and thus consumption, which allows further integration. Time reductions cannot be achieved by increasing voltage, so they are achieved by improving the metal in the connection or allowing for larger intensity in the sense amplifiers. Anyhow, no big reduction in access times is expected in commodity DRAM.

The first method that memory designers devised to **hide** high memory latency in dynamic memory devices consisted in organizing the memory in sections or banks. Each bank can be activated/pre-charged in parallel though only one can be accessed simultaneously hence the latency of bank A can be hidden while bank B is being accessed. Once the data transfer from B is done and while B is undergoing pre-charging, data from A can be seamless transferred. The system works nicely except when two simultaneous requests are due to the same bank, which generates a bank conflict and makes latency "unhideable" hence reducing performance. In fact, most of the methods to improve memory efficiency target bank conflict reduction.

The current most widespread DRAM technology, Double Data Rate (DDR), was first introduced in 2000 when the international standardization organization JEDEC first published DDR-SDRAM standard JESD79. DDR is aimed at throughput increase based on the fact that while one complete memory row (1-2 KB) is activated for each access, only one word (4-8 Bytes) can be transferred with the rising edge of the clock. Adding a second clock shifted 180° degrees at the same frequency enables transferring already active data both with the rising and the falling flank of the clock. Inconveniently, this increases the minimum access to four words and thus granularity cannot be smaller than four times the bus width.

To add in complexity, the Burst Length parameter (BL) is usually set to 8 to in that way avoid the insertion of control commands, which overloads the command bus. This simplifies the scheduling operation when simultaneously activating multiple banks. The main drawback of having BL set to 8 is the further increase in the access granularity hence potentially leading to considerable overloads when padding data is required. This effect is in the origin of the previously discussed 65-Byte problem (see 1.3.4) as well as the reason for many NPs to include multiple narrower memory interfaces instead of one single and wider off-chip memory interface.

2.4.2 The influence of bank interleaving and access length over latency

This thesis gives now a closer look at the effects of bank interleaving and access length in DDR-DRAM, in an attempt to evaluate latency and thus describe how to improve access efficiency. Because the size of data is given, bank interleaving has been the preferred way of memory controller designers to come to terms with the increasing access latency in dynamic memories.

In Figure 19, a very simple access pattern to a DDR-400 memory is depicted. In the first example, two consecutive four-word data bursts are written in bank A. This produces a bank conflict and generates long latencies that reduce the BW of the memory down to 18% of its maximal theoretical throughput. Such a case may occur in a NP when two consecutive requests from the CPU need to fetch two 32-byte headers randomly stored on the same memory bank (*Figure 19, uppermost example*). If the same accesses were 8-word long, the same latency timing would apply, though in this case twice as much data is transferred and thus the efficiency jumps up to 30%, which is still a low value *(Figure 19, second from above*). In this example, a seamless access avoids latency completely by writing a longer burst of data. As a row is usually 2-4 Kbytes long, seamless accessing actually achieves the best access efficiencies in DDR memories and hence the reason behind the performance improvement of longer data segments presented in chapter 4. But such long accesses cannot be always guarantied (packet pattern with small packets) and in fact never occur when the packet segments are small (*Figure 19, third from above*).

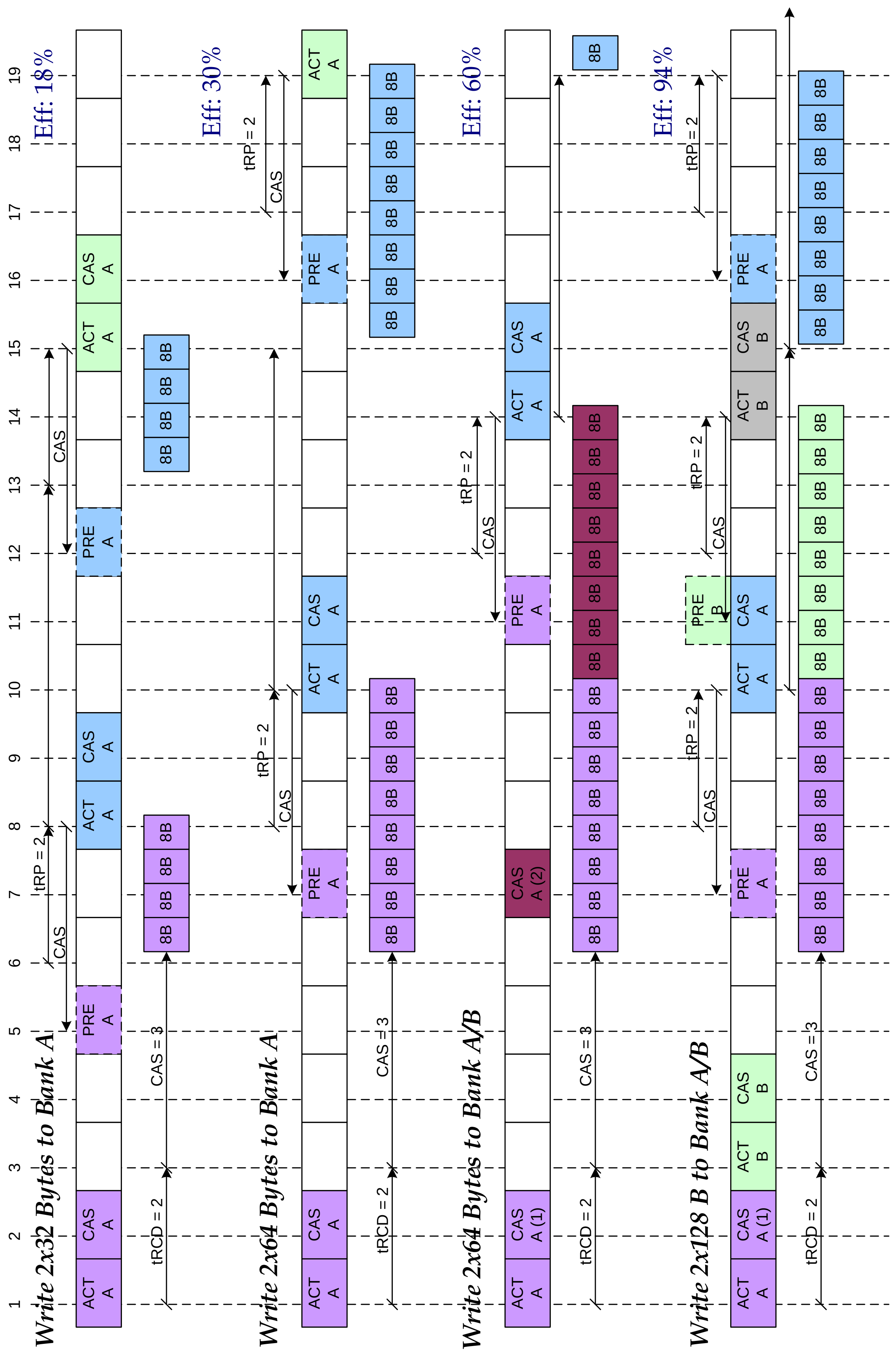


Figure 19: Access example in a DDR-400 memory, with row/column activation commands and pre-charge

Having multiple banks helps to improve the efficiency figure as the bank activation latency may be hidden by another active bank transferring data. In the last case depicted, we observe such timing with an interleaving factor of two (access pattern AB-AB-AB). The interleaving reduces latency still further and hence efficiency increases up to 94%. The number of banks in modern memories is usually eight or sixteen, so the probability of bank conflicts is actually low. However bank interleaving has the drawback that it is not intrinsically deterministic and so the final throughput depends on the access pattern. In a worse-case scenario (short accesses to the same bank), BW is decisively worsen.

One factor out of consideration in this study of throughput efficiency is row refreshing. Refreshing is required due to leakages in the isolation of the capacitor that keeps the bit value. Every 32 or 64 ms the cell must be read and pre-charged again, otherwise the logic levels cannot be ensured and data loss may occur. If during a refresh, there is a request to the same bank, the access must wait until refresh is done. Usually, a memory refreshing is executed in a periodic way without further considerations. Some advanced systems propose introducing smart-refreshing, which consists in queuing refresh accesses and interleaving them together within normal data accesses hence reducing the collision probability. A more advanced method is aware of the last accesses to a memory position, keeping a timeout table for each position and refreshes only those rows with its timer closer to a security threshold [87].

Resuming, DDR-DRAM throughput varies strongly depending on the access pattern. Access length and bank interleaving are the two main factors influencing the final efficiency figure, while memory refreshing may reduce somewhat the total memory throughput.

This introduction about DRAM technology ends with quick referencing how to reproduce the memory behavior in the investigations. Modeling DRAM memory is an essential part in every study about NP architectures. For the sake of simplicity, most designers model the DRAM interface with a fixed activation latency followed by the data transferring. Actually, it is much more complex and this complexity affects performance evaluation. During this thesis, DDR-DRAM technology is central for the understanding of how a method might or might not improve the overall throughput obtained from memory. To improve the quality of the analysis and its conclusions, we use a sophisticate memory model including bank-conflict behavior together with observations from the prototyping platform, as we will see in the next chapter.

2.5 Mechanisms to improve DRAM accessing

Several scientific publications tackle the insufficient memory bandwidth in the context of networking. In an interesting review of methods for overcoming the memory wall in NPs, Mudigonda et al. [65] from the University of Texas at Austin proposed a classification that we follow suit due to its remarkable elegant differentiation. They consider first those methods that attempt to hide the activation latency to gain in throughput (thus *ladders* of the memory wall). Alternatively, a second group of techniques directly go to the source of the problem which claim a reduction of access latency, a group that is plastically named as *hammers*.

2.5.1 Ladders

Ladders exploit packet parallelism in packet processors. Multiple flows can be active simultaneously so multi-threaded engines can switch execution among different packets whenever a thread stalls for example due to a cache miss. As we have seen, most commercial NPs with microengines include multi-threaded processors. Their results are however constrained by low memory bandwidth because even though multi-threading contributes to accelerate processing, it does not improve memory throughput in the slightest. Moreover, managing multiple threads requires additional memory resources, registers, thread-schedulers, etc. Regardless of the number of threads, there is no such a thing as perfect switching between threads as some cycles are always lost, two in the case of IXP-2800. Finally, data must be protected to avoid simultaneous accessing from two different threads to the same data chunk. This is only achieved when packet processing is completely independent, which is not the case in for instances some deep-packet processing applications.

Multithreading is actually outperformed by other ladders, notoriously by asynchronous memory accesses. This method is an upgrade of multithreading and consists in generating **multiple** memory requests before changing to another thread thus reducing the switching penalty. In any case, the throughput limitation also constraints asynchronous memory accesses, which together with the programming complexity (programmers must indicate manually the trace of memory requests to be issued) prevents this method of becoming mainstreaming.

Ladder methods are better evaluated attending to their achievements in processor utilization and processing latency reduction rather than in terms of memory throughput improvement.

2.5.2 Hammers

Hammer methods go to the root of the problem trying to **minimize** access latency or to **reduce** the number of requests, instead of simply avoiding the stalling effects in the CPU. Notoriously, data-caching is the most common *hammer*, being always used in GPP processors. That is however not the case in networking due to the low row-locality of packet processing applications due to packet segmentation using short segments (typically 64 Bytes). Time locality (i.e. recurrent requests to the same data) is very low in networking applications because packets arrive and depart continuously. Several units, like the IXP family, implement a large set of registers without any caching scheme, while those NPs relying on adaptations of existing RISC processors usually include caching as an inheritance from previous applications. Caching methods from GPP just do not deliver the same degree of efficiency when used in packet processing.

The above is also true for the several variations of caches specifically designed for NPs, for instance Wide-word caches. This caching method consists in requesting longer-than-usual cache lines from DRAM to exploit seamless data transmission. DDR memories have low granularity so larger accesses are in principle beneficial but the improvement is limited again by low space locality (the desired data is not stored in the same row).

A second caching method consists in keeping previous computation results in the cache in order to avoid not only the memory requests but the complete packet processing. The header-processing results of two packets belonging to the same flow are identical (the five-tuple value) so processing can be utterly avoided if the incoming packet matches a previous entry in the cache/look-up table and then updated with a stored pre-processed header. The concept is also known as per-flow or flow-aware processing and has been recently applied by Lawrence G. Roberts, the original designer of ARPANet, to develop his FR-1000 Network Processor [66]. The goals of such processor go far beyond overcoming the memory wall as they mainly pretend accelerating forwarding by discarding low-priority traffic like P2P. Anyhow, flow processing clearly reduces the number of accesses from the processor to the Packet Buffer along with unloading the CPU. Flow-processing seems a quite reasonable approach to several bottlenecks in currents routers though inconveniently, managing millions of flows require large look-up tables that are usually implemented using high-cost TCAMs. This makes scalability a hard issue to solve in flow-processing. A possible way to approach the scalability problem may consist in using the Longest Prefix Match (LPM) algorithm, which reduces considerably the TCAM size by assigning a group of multiple addresses switching to the same destination with a single entry in the look-up table and in the event of a multiple address match, assigns the longest match.

Another well-known caching technique from GPP adopted in NPs consists in designing hierarchical caches. Multiple-level caching has been implemented in networking applications but leaving up to the programmer the actual mapping of particular address ranges on defined portions of the cache. This is called Exposed Memory Hierarchies [67] and has its background in the context of low spatial-locality. The drawback is naturally the programming complexity for the application and the expansive chip area correspondingly dedicated to scratch-pad SRAM.

Rescheduling memory requests to minimize access latency has a very promising future in networking to improve the access efficiency of DDR memories. In the context of the PRO3 project, Papaefstathiou et. al [61] proposed first to classify accesses in a per-bank basis to afterwards reschedule these accesses to achieve better interleaving hence avoiding bank conflicts and maximizing throughput. With this simple operation, they observed a bank-conflict reduction of 33,8% for a memory with 8 banks when randomly reading and writing or even better, 56% reduction for 16-bank memories. Later, this thesis elaborates on the topic presenting an exhaustive study that evaluates the probability of bank collision with access reordering as well as other mapping mechanisms, together with investigating latency trends for upcoming memory devices.

In [68], the authors discuss another approach consisting in a two-level memory hierarchy in the memory subsystem. It is actually a SRAM-based intermediate buffer that caches the last requested rows in order to serve further petitions for such memory positions without repeating the DRAM access. The results reported show apparently low cache miss rates thus offloading the DRAM memory interface. Even though the intermediate buffer is located next to the DRAM-buffer instead of local to the CPUs, its operation reassembles that of a CPU cache.

For that reason, it reproduces data-caching problems in NPs such as lack of deterministic results, strong dependence with the cache association policy and finally but most importantly, bringing little performance improvement due to low time/space locality.

A final structure that is mentioned is the Virtual Pipelined Memory with deterministic latency recently presented by Agrawal et al. [60]. In fact, it is an advanced memory controller that takes advantage of the previous mentioned mechanisms of per-bank queuing and reordering of accesses while trying to prevent bank conflicts hence maximizing throughput. Moreover, redundant requests are introduced if a bank conflict cannot be avoided to make latency constant. Additionally, data to be written are hold in a intermediate buffer until it can be transferred to memory thus accelerating completion from the point of view of the master unit (e.g. the CPU). But most original is the Circular Delay Buffer, a SRAM-based intermediate buffer that holds requests while they are prioritized to ensure a deterministic answer time. Last but not least, the work acknowledges that bank-conflict probability is extremely low when accesses are correctly interleaved, even in worse case scenarios.

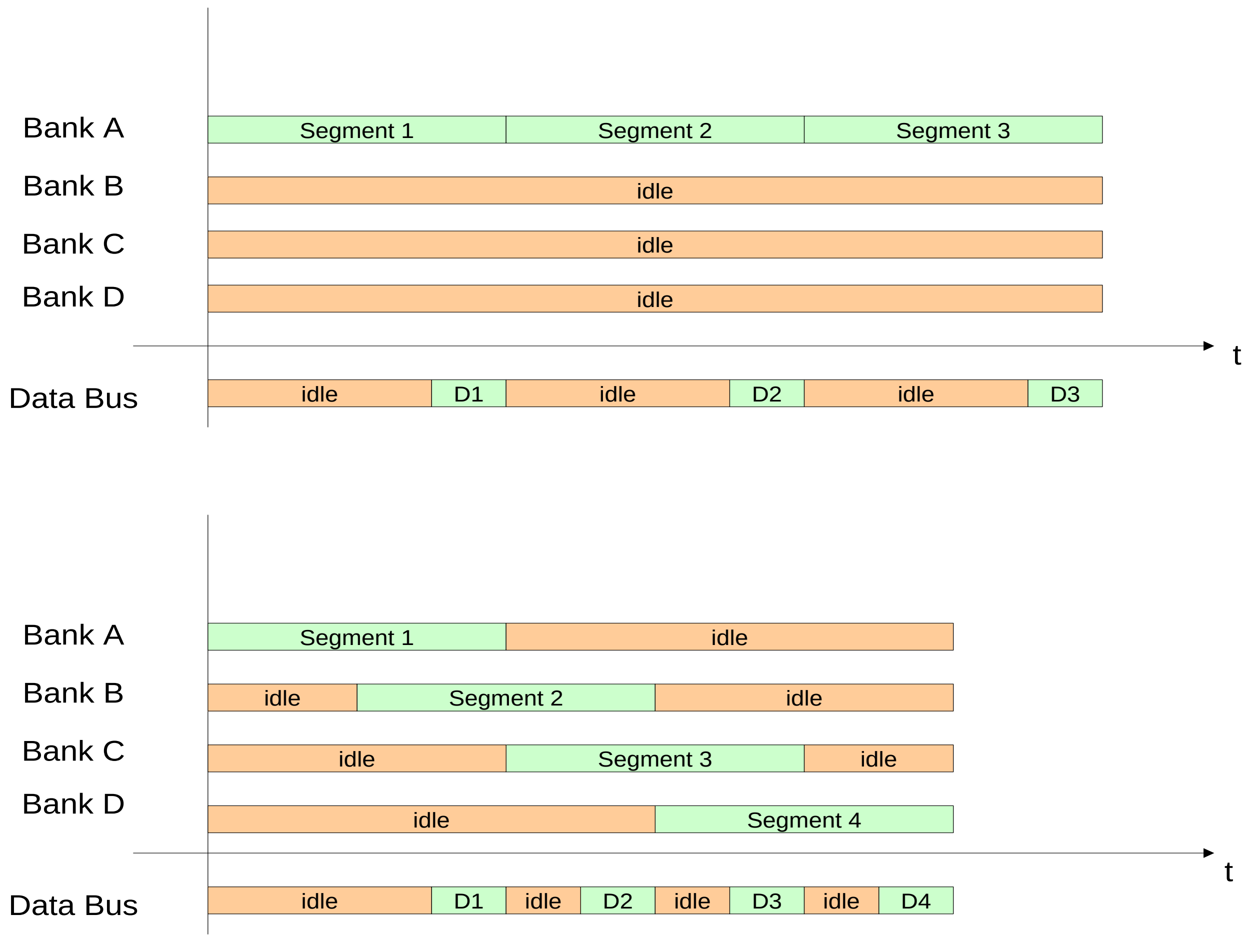


Figure 20: Bank interleaving helps to improve DRAM accessing

2.6 The concept of Cache Injection in networking

In all NPs mentioned up to now, the CPU is in charge of pulling data from the memory into its cache/registers and, once processing is done, flushing it back to the Packet Buffer. As packet data has low row locality, high cache miss rates are common place in packet processing applications. Ideally, the next packet to be processed by a particular core would be available in the cache or in the CPU local memory. In such were the case, the CPU would not stall while fetching the packet from memory. Pushing data into the cache is a technique known as Cache Injection (CI) that has its origins in multi/many-core processing, where it has been introduced to bridge the performance gap between multi-core processing capacity and the BW of the shared memory [82].

Essentially, CI is an evolution of early concepts like pre-fetching, where future cache misses are predicted and loaded in advance. In Read Snarfing for instances all caches always snoops data from the bus regardless of the data destination to in that way keep cache coherence in all caches in the cluster. With the Data Forwarding method, one core indicates others when to update an invalid cache block after it has been modified for the same cache coherence purpose. Finally in Cache Injection a control SW is in charge of opening "listening" windows. Whenever a transaction is initiated on the bus that falls into any of the address ranges listed in the Local Injection Table, data is snooped into the local cache, independently of the source/destination device.

The translation of this data push philosophy to a working device in Network Processors remains, to the best of our knowledge, unexplored. However Huggahalli et al. recently presented their Direct Cache Access system [83], a system-level exploration of cache injection for a packet forwarding application with a generic processor cluster, memory and IOs. DCA is based on the fact that all traffic between IO ports and memory crosses the chipset, where it can be monitored and snooped into the processor's cache. They reported a reduction of read cache misses up to 19% in two well-known TCP-based testbenches (SPECWeb99 and TPC-W) whereas a worsening in the hit rate was observed in a third test (TPC-C). A second important conclusion was that cache sizes for such processing should be as high as 4 MB while targeting 10 Gbps line rates, being smaller values critical for performance.

Upcoming NPs will increasingly rely on many-core clusters to provide the processing resources that modern routers require, tending towards massive parallel architectures such as that of Graphic Processor Units (GPUs). As the Memory Subsystem of traditional NPs is not adequate to fit that trend, new strategies like Packet Injection are most needed in order to tackle the memory bottleneck and allow NP's performance to keep pace with the breathtaking speed growth of network traffic. A novel system that explores data injection in networking applications will be discussed in chapter 5.

2.7 State-of-the-art summary

The Packet Buffer has been identified as a main bottleneck in NPs due to low efficiency when accessing DRAM-based buffers. Memory latency is a critical factor in most processors, although more critical in Network Processors than in General Purpose Processing. This is because caches memories are much more effective in GPP than in NPs as packet processing applications have a very low data locality. Hence longer cache lines or even multi-level hierarchical caches are not very effective at hiding memory latency in NPs. In that way and taking into account that GPP is the main market target for memory manufacturers, they keep developing new devices that only marginally reduce access latency while networking-specific low latency memories are only delivered at a considerably higher price. As a consequence, several NP developers still prefer cheaper slower commodity DRAM than more faster and priced low-latency devices. To overcome the bottleneck, designers introduce additional memory channels thus increasing power consumption as well as the chip pinout of the silicon while creating many load-balancing problems on the dual-channel memories.

At the core of the problem lays the size-requirement of the Packet Buffer that according to standard practice is coupled to the line rate to be serviced. Though several scientific publications argue against over-sized Packet Buffers, most carriers hardly accept this *less-can-be-more* concept and still rely on the old rule-of-thumb that binds speed rate with the buffer size. Whatever the outcome of this debate, the size of the Packet Buffer is going to remain in the Gbyte range, which with current memory technology cannot be implemented otherwise than with DRAM.

In DRAM devices, performance decreases when short requests are issued to different memory rows in a same bank. Row activation and then pre-charge latencies become dominant over data transmission and thus, not surprisingly, efficiency can drop down to 20-30% of the theoretical maximum throughput. Such short accesses are originated for example when packets are segmented in small segments, typically 64-bytes long. Moreover, the CPU tend to access the data it requires for processing precisely generating this short and ineffective accesses hence worsening the problem.

Some proposed solutions do not tackle the problem effectively. For instances per-flow data storing, which packages data to generate longer accesses to memory, cannot become a mainstream solution mainly due to its high SRAM requirements and implementation complexity. Several other methods have been proposed to overcome the memory wall in DRAM-based Packet Buffers. They can be broadly classified as ladders and hammers attending to their capacity to hide or reduce the memory latency. While some of them are quite effective, several trade-offs need to be considered for their implementation. It is however clear that none provide a comprehensive solution that fits all networking applications and thus new methods need to be introduced to gain in efficiency when accessing DRAM memory.

It is in that context that the method of multiple segment-sizes for packet segmentation is introduced. Among other benefits, the new segmentation scheme greatly improves the performance of DRAM-based Packet Buffers for networking applications.

3 Multiple segment-size packet segmentation

3.1 Introduction and methodology

3.1.1 Motivation

Organizing the Packet Buffer using Fixed Size Segmentation (FSS) is standard practice in commercial NPs in spite of its many shortcomings in terms of bandwidth and storing efficiency. The problem can be quickly understood if we compare packet storing with the trunk of a car just before the annual summer rush to the beach. A struggled husband confronts the troubles of fitting the large family's luggage into the not-so-large berlina car. In such a situation, having several small suitcases is preferred because of the flexibility to better squeeze them in the different-shaped remaining free holes in the trunk, although it implies managing a huge amount of small objects.

Actually, a non-inconsiderably amount of the available place is consumed by the suitcases themselves rather than by the actual objects to transport. Switching to the opposite strategy (i.e using larger suitcases) is neither the solution as a sun-umbrella, the baby carriage and the likes are all multiple-sized different-shaped items that do not fit in efficiently. Sure you can replace your VW-golf by a minivan or even a SUV, which is actually what several families do as their members come of age, but admittedly it is not the cheapest of the solutions. As any couple with kids knows, the answer to the riddle is fitting first the big suitcases in the trunk and then filling the remaining free holes with small objects.

If above we substitute trunk by a Packet Buffer, suitcases by packet segments and the struggled husband by a Buffer Manager, we come close to the problematic of selecting one suiting segment size for organizing the Packet Buffer. Accessing DRAM in short bursts of data is inefficient because of row-activation latencies. A small segment size, typically 64 Bytes, achieves excellent storing efficiency although it requires administering a large number of segments with the corresponding overload in terms of control buffering (descriptors). As we keep increasing the segment size, the number of segments per packet gets smaller but so does the storing efficiency, while internal fragmentation is reduced. Some bandwidth increase is obtained at the cost of losing in memory efficiency, a trade-off several engineers are ready to go with. However, no designer can justify a space loss of up to 80% of the memory, which is the case with the 2-KB segment size as we will see later. The first purpose of this chapter thus is to quantify whether there is a sweet spot for this trade-off.

In order to service faster line rates, NPs implement bigger Packet Buffers, which requires upgrading single-interface memories with multiple dual-channel DRAM devices. It is the architectural design choice equivalent to the enlarged family replacing its compact car by a spacious SUV, with the consequent increases in chip-packaging costs and energy consumption. It seems more adequate to allow for packet segmentation with multiple segment sizes and thus improving performance while simultaneously keeping high storage efficiency. This is precisely the central point in this thesis: introducing multiple segment-size packet segmentation, to thus store large packets in big memory blocks while the small packets are fitted in the small free memory spaces.

3.1.1.1 Performance and memory efficiency with Fixed Size Segmentation

With FSS and by tuning the segment size, it is possible to obtain high memory bandwidth or efficient memory storing, but not both simultaneously. In a previous experiment [40], the effect of tuning the segment size with FSS was thoroughly studied. The results (see figure below) reported a performance enhancement particularly remarkable when replacing 64-byte by 128-byte segments, roughly twice the throughput. After that point, enlarging the segment size yields higher performance than its immediate smaller segment although not as much as between 64 and 128 bytes. The memory controller in this experiment does not admit bursts longer than 128-Byte so requires multiple transfers for longer segments. This already illustrates a key aspect to consider when evaluating DRAM accessing: many factors may have an influence on the burst size and thus on the efficiency when writing/reading data to memory. In general, the longer the data burst, the better the performance. **Larger segment sizes benefit performance mainly because they enable larger burst accesses to the packet buffer.**

We can find additional proof that such benefits come as a product of using larger segments in the fact that **only** those traffics with packets large enough to take advantage of the bigger segment actually achieve a higher throughput. In other words, the throughput measured is identical in all scenarios with 64-byte traffic regardless of the segmentation chosen, opposite to the case with 1500-bytes packets, when a throughput increase is observed whenever the segment is enlarged.

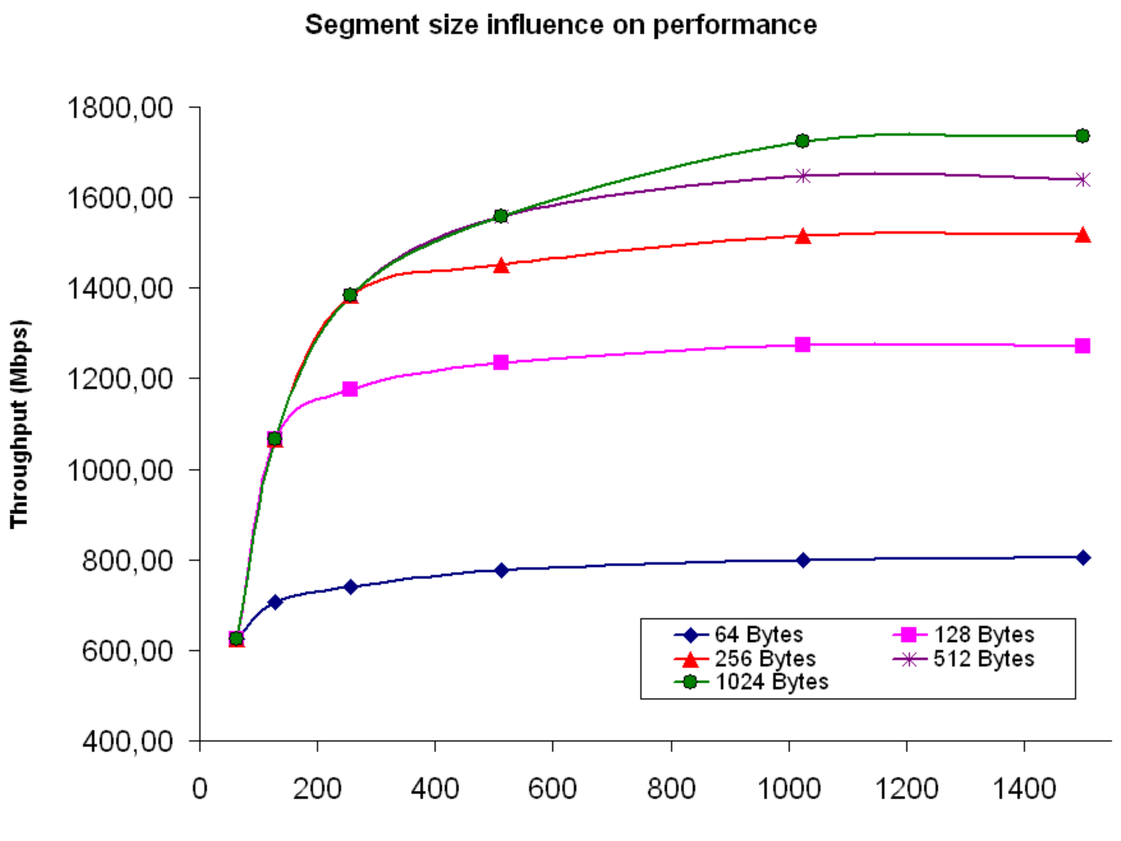


Figure 21: Influence of segment size with FSS over performance, with stimuli consisting of one packet size. In the figure, traffics with 64, 128, 256, 512, 1K, 1500 –byte packets are represented.

But performance is only one parameter to consider, storage efficiency needs to be evaluated as well. To do that, multiple traces recorded on different network nodes were segmented using different sizes and their memory efficiency evaluated. These traces cover a wide range of traffics and packet patterns (backbone, uplink and downlink) with more than 21 million packets evaluated. For exploring storing efficiency with FSS, measurements on the board have a limited scientific value as they are dependent on the traffic pattern used, which may not correlate adequately to that of the Internet. A better approach consist in using several traffic patterns recorded on different points of transport networks have been tested using a C++ tool that calculates internal fragmentation. Long traffic records are available for downloading on the CAIDA server [74][75]. Such PCAP files contain a record of packets sniffed on different points of the transport network which can be processed in C using the WinPCAP library [76]. In addition, the Advanced IMIX traffic mix [10] is a useful approximation to the traffic pattern present in nowadays networks, with a correlation to real traffic above 99% and for that reason a useful approximation.

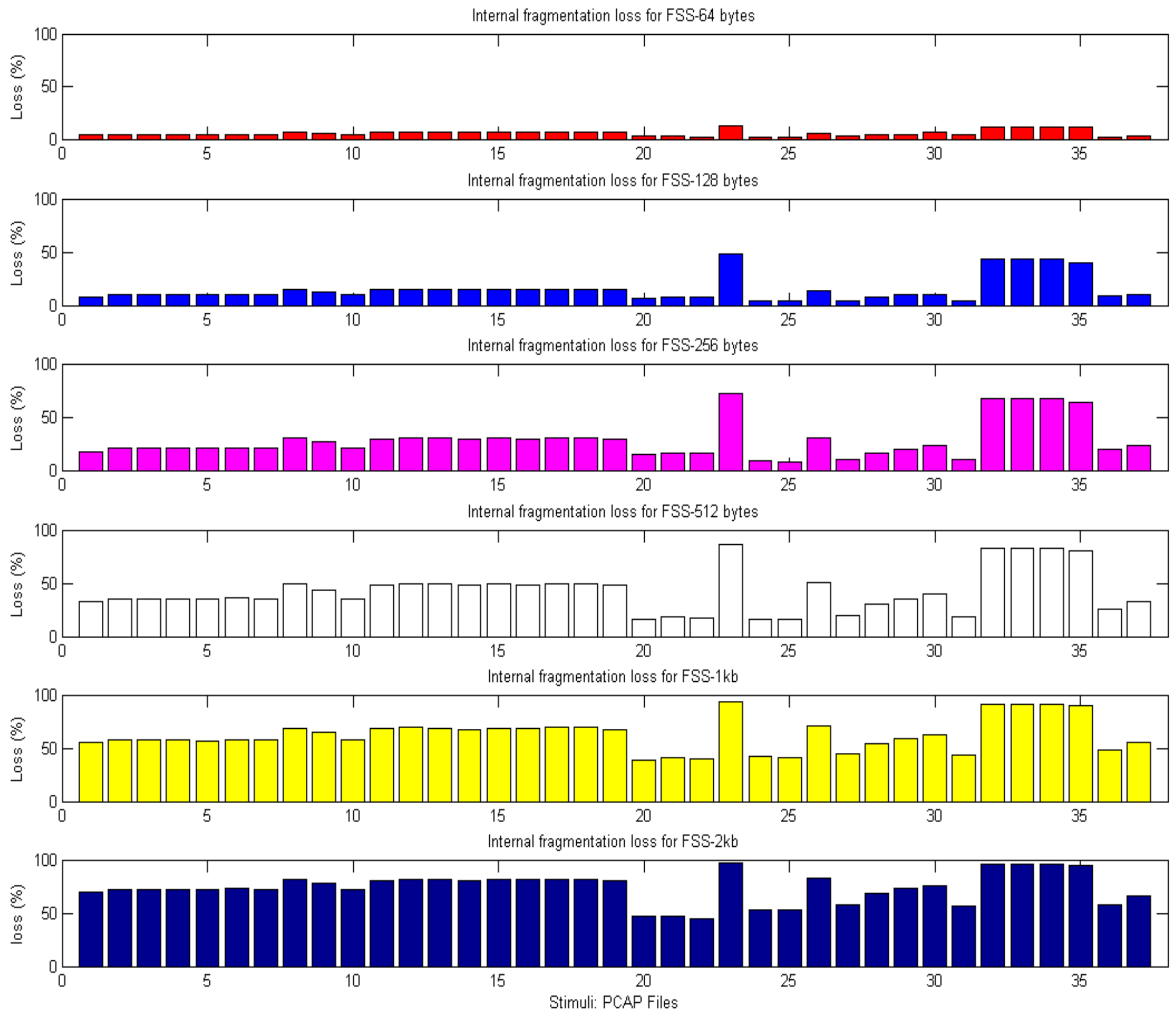


Figure 22: Internal fragmentation loss for 37 different PCAP files[3], which proves that increasing the segment size decreases memory efficiency.

[3] Details about the PCAP files used in the experiment can be consulted in Table 2.

The results (Figure 22) reported an inverse relation between segment size and memory efficiency. The bigger the segment used to segment the packets, the worse the usage of available memory space. In order to find a sweet spot in the trade-off between storing efficiency and throughput, we need to check the 128 and 256-byte segment sizes, the only sizes that could obtain nice results with both criteria. In the case of 128 bytes, the fragmentation loss goes from 5% to 48% whereas in with 256-byte segments the figures go from 8% to 73%. In general, those traffics with a higher proportion of small packets generate lower efficiencies than those with lots of large packets. The impact of such a random and uncontrollable parameter as traffic pattern in the efficiency of the Memory Subsystem is considerable with FSS. This is considered inacceptable for a deterministic design, even before considering the lower throughput obtained with these two sizes in the previous experiment, which was clearly below that obtained with the 1kb and 2kb segment sizes.

In conclusion, FSS packet segmentation cannot simultaneously achieve acceptable storing efficiency and high performance, nor with 128 bytes neither when using 256-byte segments. The only solution to achieve both consists in upgrading the Memory Subsystem to allow segmenting packets using multiple segment sizes. This upgrade requires a thorough study of which packet sizes are the most convenient as well as proposing new segmentation algorithms. Besides, the Packet Buffer capable of administering different-sized objects has a different set of requirements and so a new range of design challenges that will require careful consideration.

3.2 Multiple-size packet segmentation

As we have seen, FSS schemes are limited to either optimizing memory efficiency or minimizing the number of segments per packet, which can lead to an excessive amount of control information or memory waste. In this section, **two algorithms** are proposed that utilize a finite number of segment sizes to divide a packet into multiple segments each of which might be of a different segment size. The segment sizes are defined during the system configuration. The goal is to bridge the gap between the number of segments per packet and memory efficiency by allowing an algorithm to achieve high performance in both parameters concurrently. This will allow an NP to achieve higher throughput through reduced control overhead and improved bus and memory bandwidth utilization.

3.2.1 Memory Efficiency Optimized Algorithm (MEO)

The first of the two algorithms proposed here aims at optimizing the memory efficiency achieved by selecting segments that better fit each packet. Using this algorithm a packet is segmented into several segments of different size from a selection of possible segment sizes, in contrast to the approach by Intel [27], which divide each packet into segments of one size, selected from a pool of several possible alternatives.

The algorithm processes each packet, testing all the available segment sizes from biggest to smallest until one is found which is smaller than the packet length. This will be the largest segment which can be completely filled with data from this packet. Then a segment of that size is created, its size is subtracted from the packet length and the process is repeated for the remaining packet length until the entire packet has been stored. The figure below illustrates the criteria for segmenting a packet. S_i denotes the segment size being tested while S_{i-1} and S_{i+1} are the immediate smaller and larger segment sizes respectively. The dashed boundary indicates the area in which the data amount must be located for the segment S_i to be selected.

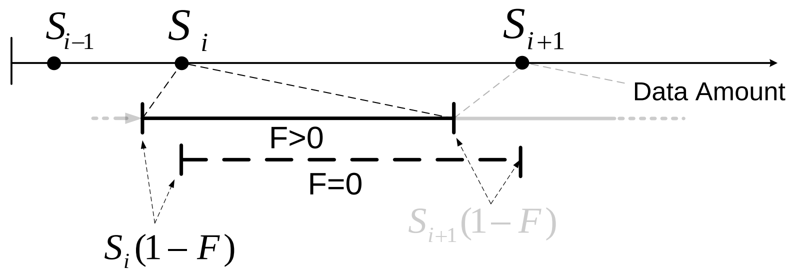


Figure 23: Iterative selection of segment sizes for packet allocation

The parameter F, called the ranging factor, can be used to shift the segmentation selection boundaries thus allowing for the dynamic manipulation of the trade-off between memory efficiency and segment number by loosely regulating the number of segments the algorithm commits for each packet. The greater the ranging factor, the more the lower bound expands. The upper bound of S_i is defined by the lower bound of S_{i+1}.

Ranging factor	F=0	F=0.3
1st segment	1024	1024
2nd segment	128	512
3rd segment	128	-
4th segment	128	-
Memory Efficiency	98.8 %	92 %

Table 1: Example of two different packet segmentation results for a 1392 Byte Packet and 128-512-1024 Byte segment sizes using the MEO algorithm

Table 1 provides an example of this tuning of F for a packet size of 1392 and available segment sizes of 128, 512 and 1024 bytes. Here, the increase of the range factor means that instead of allocating three 128 byte segments we allocate one 512 byte segment instead. This is because the remainder length range with F=0 is from 512 to 1024 bytes whereas with F=0,3 it is from 358 to 716 bytes. The increase of F thus results in reduced number of segments per packet but also in reduced memory efficiency. In another example, nine segments are required when using the MEO algorithm for a 1481-byte packet with F set at 0 (Figure 24). If instead this parameter is redefined to 0,2 then a 512-byte segment replaces the eight small segments, which is preferred to increase performance in spite of storing efficiency stays constant (Figure 25).

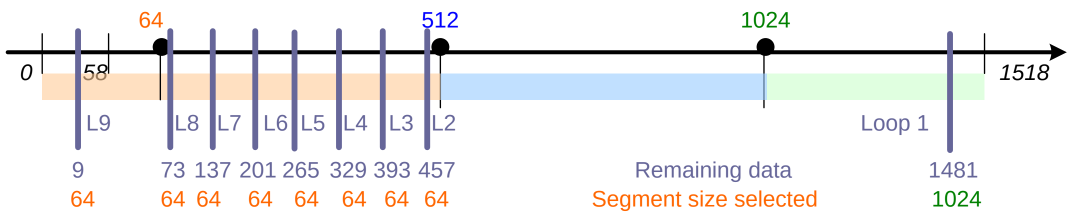


Figure 24: segmentation of a 1481-byte packet with MEO and F=0

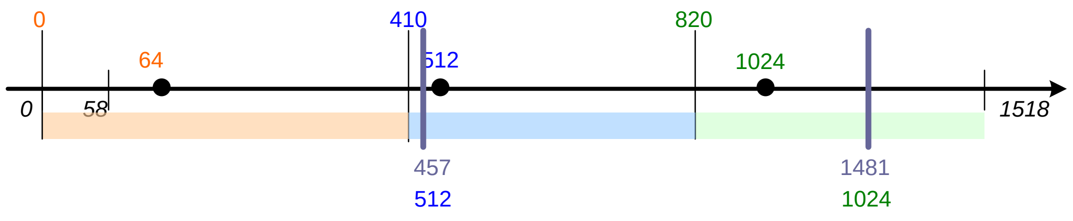


Figure 25: segmentation of a 1481-byte packet with MEO and F=0,2

3.2.2 Segment number Limited Algorithm (SLA)

The algorithm described in the previous section attempts to optimize memory efficiency without however providing a hard bound on the number of segments in which it can divide a packet. Such a limitation can be useful in order to enable a more deterministic approach to better optimize control data structures (e.g. to avoid linked lists to store packet segment descriptors).

$$Number_{combinations} = \binom{N+M-1}{N-1} = \frac{(N+M-1)!}{M!\,(N-1)!}$$

(4)

In the SLA algorithm a packet is still divided into segments selected from a predefined pool of segment sizes, however the implementation of the selection process is different. Assume that *N* is the number of segments sizes available to the segmentation algorithm, while *M* is the maximum number of segments per packet the algorithm is allowed to allocate. Here all possible combinations between all of the available segment sizes for the specific values of N and M are calculated. The total number of points is obtained calculating the combinations with repetition for N and M according to *(4)***.**

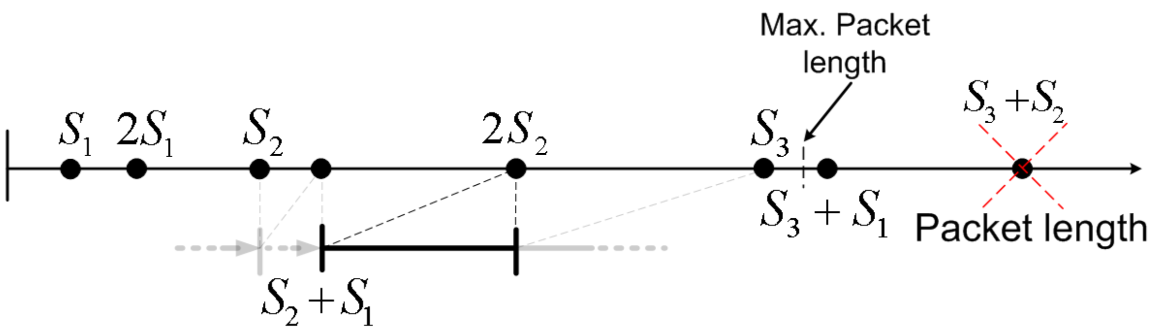


Figure 26: Criteria for selecting segment sizes by limiting the number of possible segments per packet

Figure 84 illustrates this for 3 segment sizes and a maximum of 2 segments per packet allowed (M=2, N=3). In this case the following combinations are available: { S_1, 0, 0}, {0, S_2, 0}, {0, 0 , S_3}, { S_1, S_2, 0}, { S_1,0, S_3}, {0, S_2, S_3}, { $2S_1$,0}, {0, $2S_2$,0}, {0, 0 , $2S_3$}. Some results which are larger than the maximum packet length are discarded. Only the one which is immediately larger than the maximum packet length needs to be considered. After calculating all possible combinations within the limitations we define, we then compare the packet length to all of the calculated results in order to determine the combination that is immediately larger than the segment size. This ensures better efficiency than MEO while limiting the maximum number of segments to the defined number.

For the same 1481-byte packet, the SLA algorithm finds always the best match in the first iteration because all possible values are already calculated. In this case, the closest combination of segment sizes is 1kB together with 512 bytes.

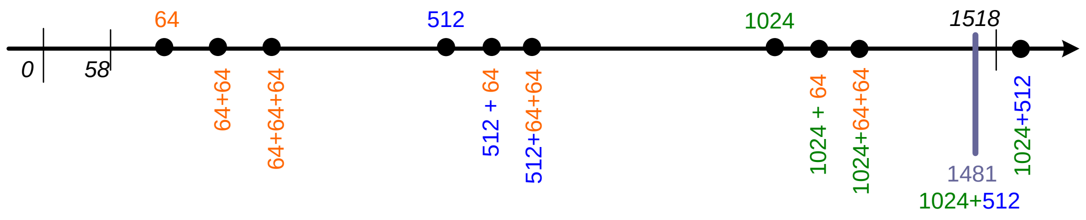


Figure 27: segmentation of a 1481-byte packet with SLA, M=2 and N=3

3.3 Packet Segmentation Algorithm Analysis

3.3.1 Methodology and design flow

This study proceeds now to investigate multiple segment size segmentation in a three-staged approach because of the extended number of parameters to tune and evaluate, so the new algorithms are approached at different levels of abstraction, as Figure 28 illustrates.

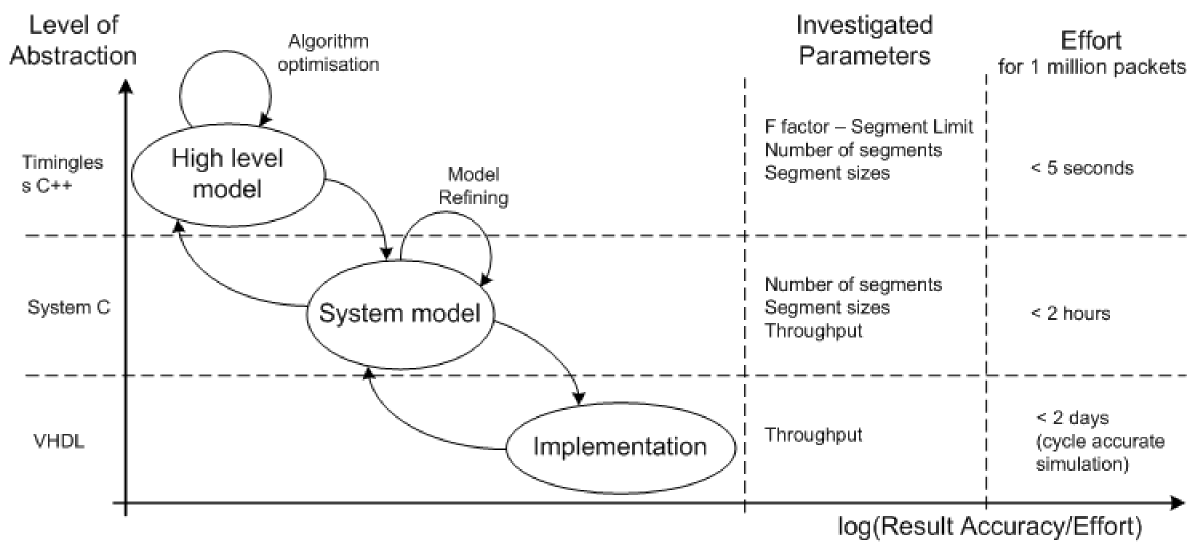


Figure 28: Design approach and level of modeling abstraction

Each step provides higher simulation accuracy but also requires increased effort to model, as well as longer simulation times to execute. The goal was thus to explore a large number of options using a fast high level model (written in C++) and a large variety of stimuli (e.g. PCAP files), then move into a much more detailed system level model (in SystemC) and finally when the selection of the various algorithm parameters has been sufficiently narrowed down, to implement the Buffer Manager in synthesizable VHDL.

The significant number of parameters (number of segment sizes and segment size values) that may be tuned in any of the two algorithms creates a huge number of different variations (a few hundred) that need to be considered. To counter this, a high level model which behaviorally captures the Memory Subsystem was developed. Its purpose is to provide high speed evaluation of each algorithm without significant compromises in result accuracy.

- The first stage in the design approach provides a preliminary estimation of algorithms performance and a selection of the fittest solutions. Here, the quality parameters used are average number of segments per packet and storing efficiency, both of which require low computation power to be analyzed. This allows an extensive exploration of all possible segment combinations so, with these results, we can narrow down the best candidates for a more detailed exploration.

- The second stage in the investigation aims at implementing the most promising algorithms variations, but upgrading the model to extract performance data regarding the entire system and not only the algorithms itself. With this system-level model, we observe a close correlation between average segments per packet and throughput, which validates the approach followed in the early stage of the study. The SmartMem SystemC Model (SmaSyMo) captures the entire Memory Subsystem while focusing on its performance and the functionality of the Memory Management Unit. It combines transaction-level accuracy for the Buffer Manager and the CPUs with a cycle-accurate modeling for critical performance bottlenecks like the bus and the memory controller. Several degrees of freedom allow a comprehensive exploration of factors that affect performance like memory devices, buffer mapping, number of memory interfaces, transfer length and destination address, memory controller policy or internal interconnect architecture. Moreover, the Buffer Manager architecture's functionality is accurately modeled, including packet segmentation algorithm, control structures, free-address lookup, packet reassembly and internal transactions. The model utilizes a CPU cluster for packet processing. Each CPU contains its own local cache and is connected to the rest of the system via a common data bus. To emulate the CPU load on the bus and memory, a basic processing delay together with different memory request patterns depending on the application type were considered. Then, Queue Manager delays are introduced matching common queuing latency distributions. Delay is based on measurements taken from real routers found in [71, 72] and corroborated in [73], showing that packet delays across a router can be approximated with a Weibull distribution.

- The results from the above algorithms allow not only selecting the appropriate parameters for the segmentation algorithm but also exploring the architecture for the SmartMem Buffer Manager. In order to keep a linear approach, the prototype implementation together with its results is only discussed once all its constituent internal elements have been properly introduced (chapter 0).

3.3.2 Packet stimuli used during the experiments

A set of PCAP files from core network OC-48 lines is chosen as well as slower access network ones, which for the sake of clearness is used through this work to easier compare results from different experiments. In total, the files contain nearly 22 million packets of varying packet sizes and traffic patterns. Table 2 provides the origin, number of packets and packet size distribution for these files.

ID	PCAP file	Num. Pkts	Packet size distribution		
			40-319	320-639	>640
1	20040219-105000-a	3.809.426	56,0392	6,169958	37,79084
2	bcn_OC48_36k	36.853	55,3035	10,93805	33,75845
3	bcn_OC48_37k	37.411	55,03729	11,09299	33,86972
4	bcn_OC48_37k_a	37.872	54,95617	10,88931	34,15452
5	bcn_OC48_37_b	37.985	54,66895	11,01487	34,31618
6	bcn_OC48_38k	37.551	55,65231	11,03832	33,30937
7	bcn_OC48_7000	7.238	55,0152	10,92843	34,05637
8	CAIDA2_225k	226.255	70,57435	8,851959	20,57369
9	CAIDA2_275k	274.544	65,2329	9,415249	25,35186
10	CAIDA2_600k	675.191	55,07716	11,0431	33,87975
11	CAIDA2_Dec07_566k	11.318.834	69,94217	8,384998	21,67283
12	CAIDA2_OC48_37k	37.524	70,9306	8,9063	20,1631
13	CAIDA2_OC48_38k	38.303	70,74642	8,495418	20,75817
14	CAIDA2_OC48_39k	39.461	69,45845	8,775753	21,76579
15	CAIDA_OC48_19k	19.581	70,53777	8,932128	20,53011
16	CAIDA_OC48_20k	19.822	69,50863	9,539905	20,95147
17	CAIDA_OC48_37k	37.524	70,9306	8,9063	20,1631
18	CAIDA_OC48_38k	38.472	71,04388	8,668642	20,28748
19	CAIDA_OC48_39k	39.461	69,45845	8,775753	21,76579
20	downstream_101k	101.939	6,569615	16,34115	77,08924
21	downstream_Jan08	24.492	16,84632	1,972073	81,18161
22	Feb_16h_downstream	124.526	11,56305	2,662095	85,77486
23	Feb_16h_upstream	83.447	98,59312	0,438602	0,968279
24	HTTP_videostream	88.498	36,97824	0,414699	62,60706
25	HTTP_videostream_short_4	50.286	36,45547	0,351987	63,19254
26	OC48_Cat_20k	20.001	76,49618	1,889906	21,61392
27	SVL-LAX-20050319-101000-0	2.362.021	43,43873	2,071446	54,48982
28	torrent	76.702	54,41579	5,100258	40,48395
29	torrent_downstream	38.939	60,46637	5,012969	34,52066
30	trackmania	24.806	68,78981	1,165041	30,04515
31	trackmania_downstream	12.558	38,9871	1,791687	59,22121
32	upstream_29k	29.322	94,87416	1,66769	3,458154
33	upstream_39k	39.014	94,55324	1,940329	3,506434
34	upstream_59k	59.881	94,54418	1,953875	3,501946
35	upstream_Jan08	15.568	92,72225	2,935509	4,34224
36	user_traffic_Feb_21_11h	1.839.511	34,90993	0,047023	65,04305
37	user_traffic_Feb_21_16h	214.875	48,20198	1,713089	50,08493
	Total	21.975.694			
	Max		98,5931	16,3411	85,7749
	Min		6,5696	0,0470	0,9683

Table 2: Traffic pattern distribution of the studied PCAP files

The number and sizes of segments available to the segmentation algorithm has a direct effect on the storing efficiency and average number of segments per packet it achieves. A preliminary selection of segment size combinations can be achieved by simple observation of the packet length histogram. Packet sizes can be broadly categorized into small (less than 100 bytes), a medium (around 500 bytes) and a large (more than 640 bytes) packet size. This means that to provide acceptable coverage for this packet size range, we need at least three packet segment sizes, each of which should cover one of these areas. This hypothesis states that solutions with less than 3 segment sizes should not provide comparable performance, while the ones with more than 4 segment sizes should not provide any significant improvement since the packet size spectrum should already be adequately covered by solutions with a smaller number of appropriate segment sizes.

$$\bar{S}_{packet} = \frac{\sum_{i=1}^{N} s_i}{N_{packets}}$$

(5)

$$E_{mem}\,[\%] = \frac{\sum_{i=1}^{D} p_i}{\sum_{i=1}^{D} m_i} \cdot 100$$

(6)

As stated previously, the most important parameters to evaluate the performance of a segmentation algorithm are the memory efficiency and the number of segments per packet, as defined in *(5)* and *(6)* respectively. To evaluate them we graph the memory efficiency achieved versus the average number of segments for each algorithm variation obtained when analyzing an IMIX traffic pattern and the OC-48 files described in Table 2 (average). Thus the optimal solution would be in the upper left corner of the graph (as shown in Figure 29). For clarity reasons the graphs presented do not include all possible combinations of segments, but only the most interesting ones (for both MEO and SLA). Additionally we include the results of the fixed size segmentation algorithms as a reference.

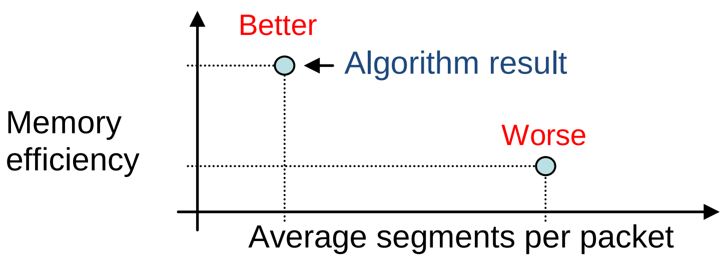


Figure 29: quality parameter evaluation

With this in mind, the new algorithms are tested against different traffic mixes and their results reproduced in the following. For the exploration, at least 3 segment sizes are always available being these sizes powers of two in order to simplify later implementations in digital systems. No other constraint was in place, all other combinations of segment sizes and algorithm parameters like M, N and F were studied systematically. However only the better performing sets of parameters are here reproduced due to lack of place to display the millions of points obtained.

3.3.3 Memory Efficiency Optimized Algorithm performance

The high-level exploration of the MEO algorithm investigates all combinations of segment sizes as well as many values for the ranging factor *F* by using a low-effort fast C++ application that calculates storing efficiency and average number of segments per packet. The best results have been filtered in the displays for the sake of better understanding the main trends without masquerading them behind a cloud of points.

In Figure 30 and Figure 31, these results are depicted according to the algorithm configuration used. First, when the ranging factor *F* is set to 0, the storing efficiency obtained was in general excellent for the selected segment sizes though at the price of increasing the average number of segments per packet. Only when the *F* factor is tuned upwards in order to use larger segments, the performance obtained approaches the left axis whereas only slightly decreasing efficiency (ranging factor of 0,2-0,5). The experiment was repeated first applying the F factor on the low side and afterwards on both sides of the selection range, with little difference in the outcome.

Interestingly, in nearly all cases MEO performs much better than the state-of-the-art best possible FSS configuration. This is again the case if segmentation is performed using any of the six segment sizes available (64, 128… up to 2kb) but limiting segmentation only to one segment size per packet. In that case, the number of segments is improved however with the correspondent penalty in terms of efficiency. Finally when trying different combinations of the 64-128-256-byte segments while constraining again the algorithm to one segment type per packet, both quality parameters appear to worsen. This final test reproduces the buffer organization of second-generation NPs described in the prior art, which even being superior over FSS still clearly perform worse than MEO, specially in terms of efficiency.

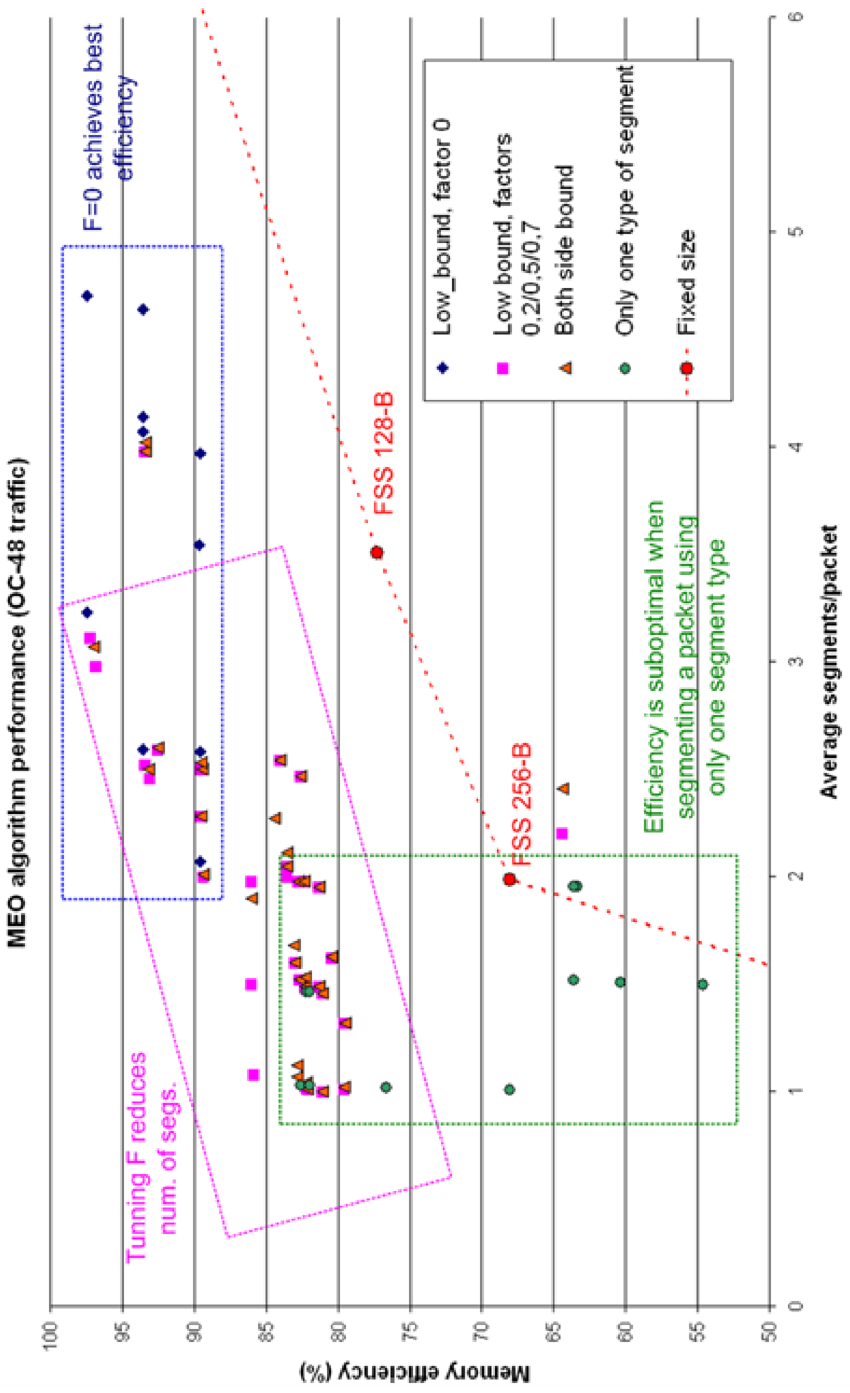


Figure 30: MEO algorithm performance for OC-48 traffic

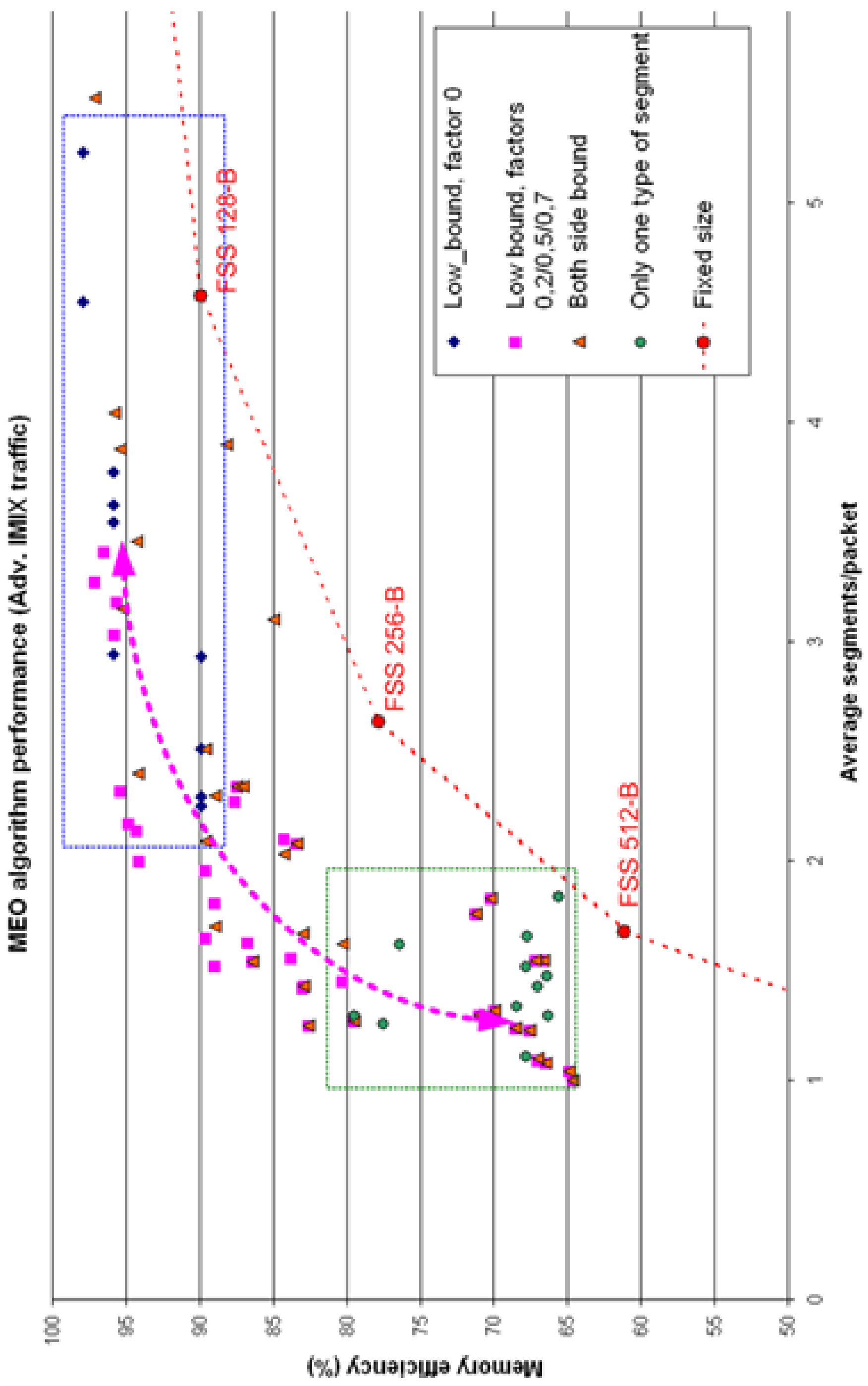


Figure 31: MEO algorithm performance for Adv. IMIX traffic

These conclusions are confirmed when repeating the experiment stimulating the system with an advanced IMIX traffic pattern. Again MEO with a ranging factor of 0,2-0,5 is superior to all other schemes tested, being the only one capable of bridging the gap between high efficiency and low number of segments. There are slight variations on the results obtained compared to the previously-tested OC-48 traffic, due to the different packet sizes found in Advanced IMIX. For instance, efficiency is in general somewhat higher while the number of segments required increases, which is a consequence of having larger packets in the traffic pattern.

Now we concentrate on tuning the *F* factor as accurately as possible. Modifying the range factor in the memory efficiency optimized algorithm provides an easy way to alter the balance of the memory efficiency vs. number of segments per packet trade-off so as to cover a wide area of system requirements. To deepen in the effect of the F factor, we have simulated the algorithm with *F* values varying from 0 to 1 in 0,1 increments, using the three segment size combinations that achieved the most promising results in the previous analysis.

Figure 32 illustrates the results from these simulations with OC-48 traffic.

An increase of the F factor leads to a worsening in storing efficiency however reducing the number of segments required. On the contrary by reducing the F to values close to 0, we obtain a lower number of segments per packet together with lower efficiency. A value of 0.3 (fourth point) seems to provide the best trade-off. When using combinations with large segment sizes this trend is distorted due to the fact that the expanded segmentation range borders begin overlapping and in worst cases even completely cover one another. This can be observed with the 256-512-1Kb and the 512-1Kb-2Kb sets that anyhow achieve poor storing efficiency hence will already be discarded and not included in the next simulation stage.

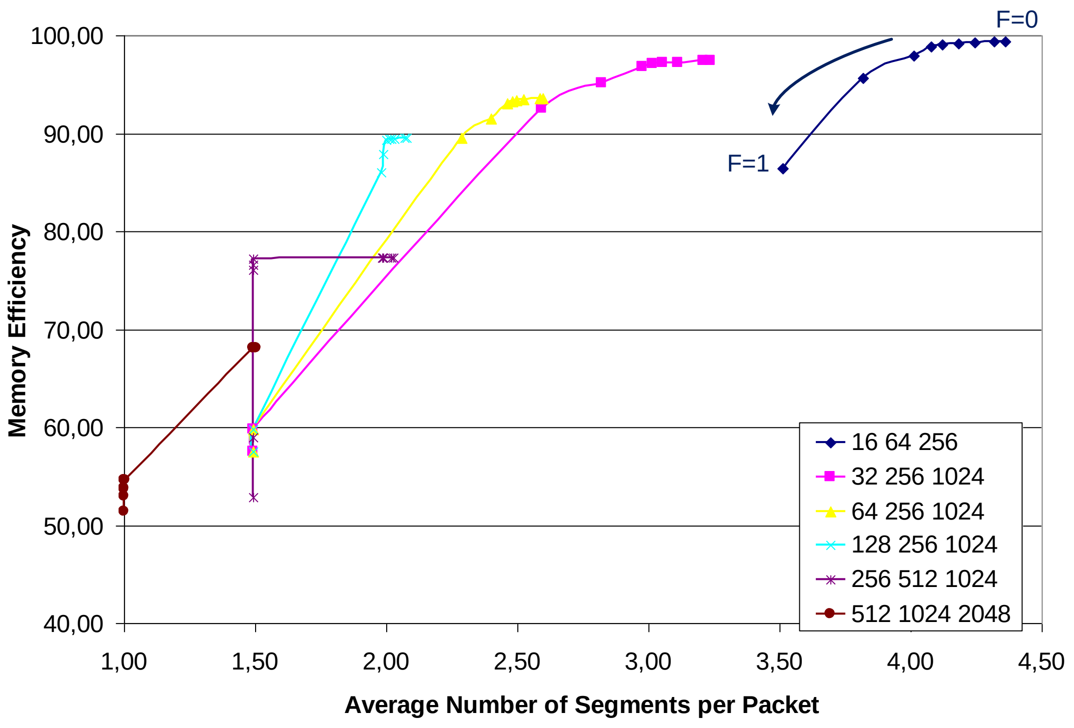


Figure 32: Variation of memory efficiency and average number of segments with the F factor for MEO

3.3.4 Segment number Limited Algorithm performance

The high-level exploration is now done for the SLA algorithm, again studying all combinations of segment sizes for all *M* (maximum number of segments per packet) values below six. The experiment sheds light into the advantage of adding more segment sizes to the pool of choices of the SLA algorithm concluding, not unexpectedly, that the more sizes available the better the performance results (Figure 33).

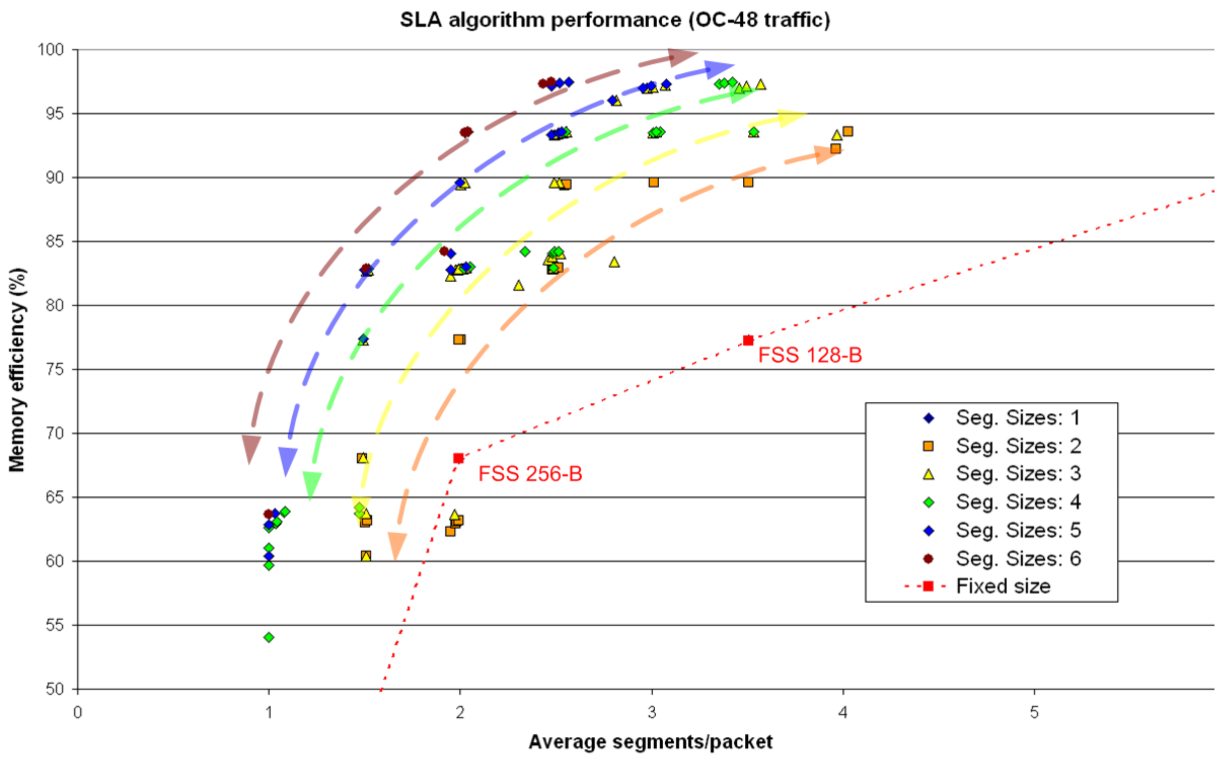


Figure 33: SLA algorithm performance for OC-48 traffic

The main conclusion that can be extracted from the results is that SLA also overcomes FSS performance whatever the segment sizes chosen except when *M* is set to one, in which case SLA segmentation effectively behaves as FSS. When observing the curves, we can distinguish a collection of points with the same number of segments in the pool forming a half-arc shape. The targeted upper-left area can thus be approached by increasing the number of segments in the pool, but such an improvement needs to be traded off with the increasing difficulty of managing a Packet Buffer containing several types of segments. Anyhow, even for low *N* values, SLA effectively removes the need to choose between high storing efficiency and low number of segments, thus delivering a satisfactory trade-off between both criteria.

As before, the experiment is repeated for an Advanced IMIX traffic mix (Figure 34) to confirm the previous results. Efficiency is very high for several combinations of segments especially for those sets that include the 64-Byte segment, which can be clearly observed in the efficiency leap of those points with that size in its pool. The half-arc shapes are again recognizable though with the mentioned step in-between. Once again, SLA allows avoiding the choice between efficiency and number of segments per packet by offering both.

Figure 35 depicts an enlargement of the area of interest on the upper-left corner of the exploration space, to better identify the high concentration of best performing points. This is helpful to determine which are the best sets and thus candidates for the final hardware implementation. We can observe that most of the points belong to configurations with five or even all the segment sizes available. Though this was expected, interestingly enough four of the algorithms in the desired area have *N* equal to three or four, having all of them the known combination of small-medium-large segment sizes.

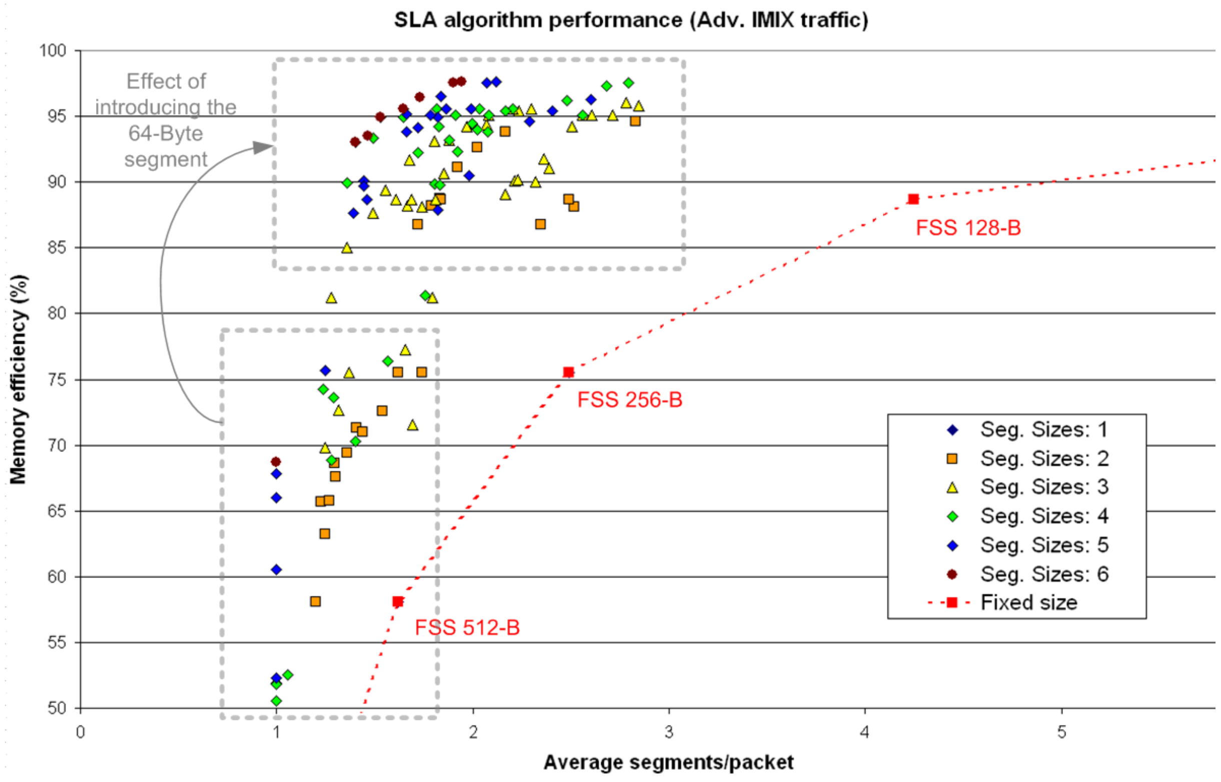


Figure 34: Selection of SLA algorithm's performance for Adv. IMIX traffic

Such satisfactory results are even more encouraging when observing their *M* parameter. Contrary to what could be expected, constraining the number of segments per packet does not necessarily lead to lower performance. Quite the contrary, if a Pareto analysis is drawn on the best performing points we observe that all of them have the *M* value set to three or even two, which is crucial for reducing the complexity of packet administration as we will see in Chapter 4. This leads to the conclusion that constraining *M* with an appropriate selection of segment sizes allows achieving the best results in both quality parameters and at the same time ensuring a low hard bound in the maximum number of segments per packet.

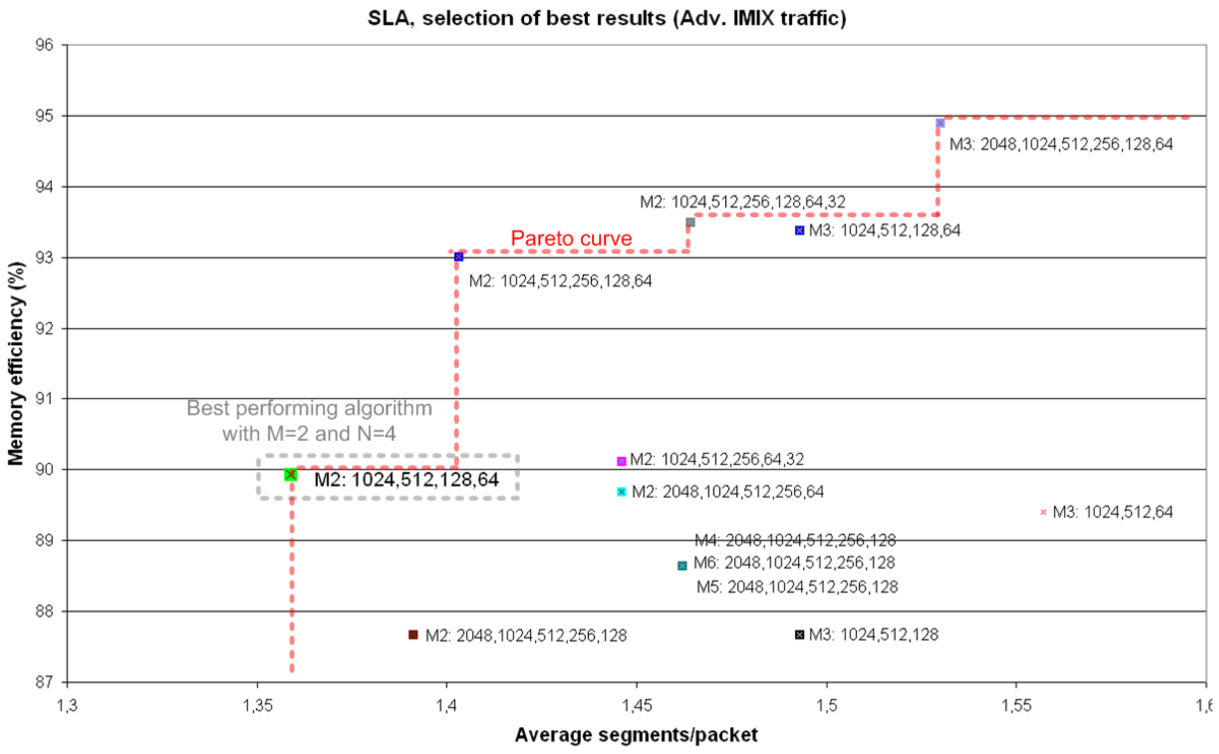


Figure 35: Enlargement of the area of interest (upper-left corner) with Pareto curve

3.3.5 Conclusions

Both the MEO and SLA achieve hugely improved results in comparison with the fixed segmentation scheme for all shown combinations. In general MEO achieves better efficiency results while SLA constraints the number of segments more efficiently though both algorithm results converge to the area of interest (upper left corner) for the appropriate selection of segment sizes and parameter values (F and M respectively) as depicted in Figure 36.

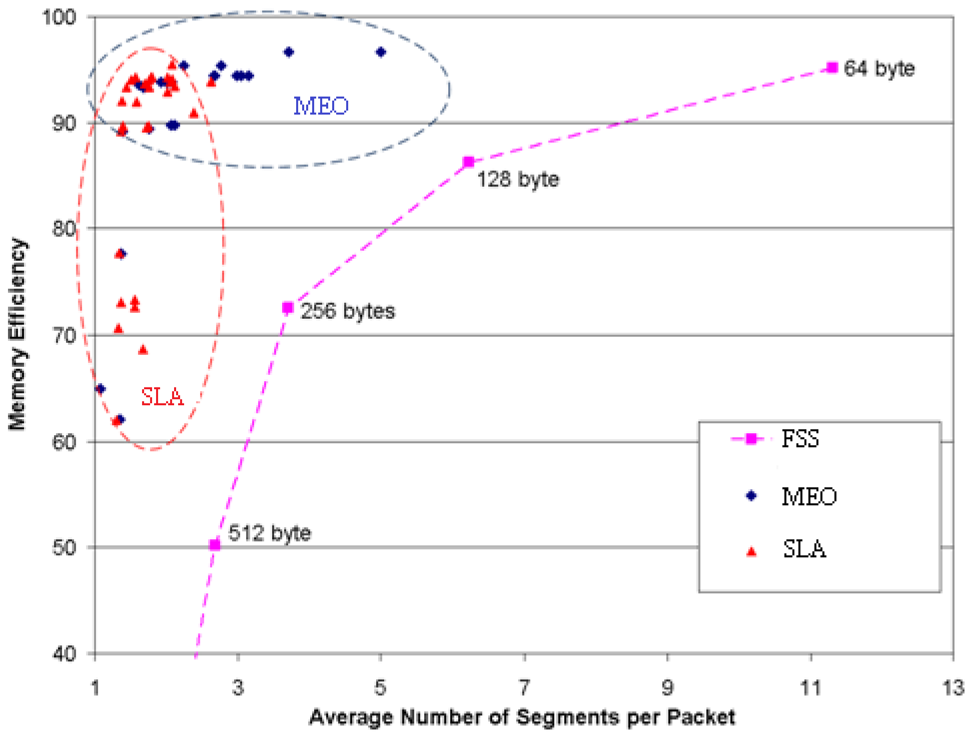


Figure 36: Performance comparison of MEO, SLA referenced to FSS

Table 3 provides a list of the best performing solutions from both algorithms. The table verifies the design goals for the two algorithms. For both three and four segment sizes MEO reaches higher memory efficiency than SLA. As the segment number limit in the SLA becomes stricter, efficiency drops below 70%. It is important to note that for bigger N values the SLA algorithm tends to provide similar results as the MEO algorithm. Thus the most interesting parameters here are a relatively low segment number limit, which offer somewhat decreased memory efficiency but with a lower average segment number and a hard limit on the maximum number of segments in which a packet may be split.

Regarding the selection of segment sizes, our original assumptions are confirmed. We see that when **using three segment sizes** with a small- medium-large segment size combination the best results are obtained, where the large segment size is almost always 1024 bytes (using 2048 bytes results in a significant drop in memory efficiency) and the medium size is most cases 256 bytes, which enables us to cover medium sized packets with 2 or 3 segments, while providing enough granularity to efficiently cover other packet sizes as well. As small segment size, 64 bytes represents the most reasonable choice as the 32-byte segment brings only marginal improvement in memory efficiency but with a significant increase in the number of segments generated while the 128-byte segment results in an overall reduction of memory efficiency.

Based on these results we can select some variations of the two algorithms to test in greater detail with our system-level model. More specifically the MEO algorithm is used with both three (64-256-1024 bytes) and four (64-128-256-1024 bytes) segment sizes and an F of 0,3 as these proved to be the best all-around performers in the previous investigation. From the SLA algorithm we select a four (64-128-512-1024 bytes) segment-size variant with a maximum segment number of two. Since the performance of both algorithms tends to converge as the limit in the number of segment number relaxes, we chose a variant with a maximum of two segment sizes per packet because this provides both excellent memory efficiency and minimization of control overhead.

All segment combinations and parameters of MEO and SLA have been considered, now narrowed down to 3 candidates for later implementation (in green). The results of this investigation greatly help to identify the best candidates for an in-deep analysis with more accurate simulation means. In addition to these three candidates, we will test the SLA algorithm with five segment sizes (64-128-256-512-1024) and constrained to only one segment per packet. This is a very interesting setup for SLA, as it ensures that only one segment per packet is required and so very helpful for the reduction of control data.

Algori thm	Segment Size Combination	M	F	Mem. Efficiency	Avg. no. of Segments
MEO	256-512-1024	-	0,3	77,59	1,37
	64-256-1024	-	0,3	93,08	2,05
	64-128-512-1024	-	0,3	93,08	1,83
	32-64-256-1024	-	0	96,63	3,71
SLA	32-64-256-1024	4	-	95,54	2,09
	64-128-512-1024	2	-	92,05	1,38
	64-128-512-1024	5	-	94,33	1,58
	256-512-1024	2	-	77,70	1,35

Table 3: Best performing algorithms: best memory efficiency, best average number of segments and the two algorithms with the best trade-off between both parameters

3.4 System-level exploration

This section introduces the SmartMem System-level Model (**SmaSyMo**), used to further refine the evaluation of system performance and gain insight into the influence of additional functional and architectural parameters, such as memory type, memory access pattern and system topology, on the new segmentation algorithms. It also allows investigating suitable Memory Subsystem architectures for the later implementation of the SmartMem Buffer Manager.

The implementation of a model like SmaSyMo is effort-demanding and thus it is of the highest importance to evaluate beforehand whether the benefits that such an investigation may yield justify the resources invested. A **top-down** design approach allows an early detection of key bottlenecks and helps to find sweet spots for the design trade-offs, which enables constraining the exploration space for more refined simulations. Precisely, SmartMem requires of several architectural decisions and thus a systematic exploration of the great variety of parameters to properly tune them. If such a exploration were done at Register Transfer Level (RTL), it would consume a prohibitive amount of programming time. For that reason, we need a model with a higher degree of abstraction as for example a trace-based simulation. In addition to the parameter exploration, our model pursues validating the algorithm results obtained in the high-level exploration presented before, but this time with a much more detailed scenario. For instances, the average number of segments per packet will be replaced by the actual throughput obtained by the memory subsystem.

SmaSyMo captures the functionality of the typical Memory Subsystem architecture in a NP, including the buffer manager, the interconnection infrastructure (bus and direct interfaces), packet processing, output queuing as well as the packet and control buffers. Different devices are modeled with a variable degree of abstraction. For instances, the bus and DDR memory controller are critical for accurate throughput simulation and for that reason have been carefully modeled at cycle-level. Opposite, the influence of the behavior of the BM and the processor cluster on the system's performance can be adequately captured at a transaction-level.

The model uses SystemC [78] as modeling language, an option that requires low modeling effort while providing accurate results with reasonable simulation effort. SystemC offers real productivity gains by letting engineers design both the hardware and software components as these components would be later implemented, with variable degree of abstraction. By using higher levels of abstraction, the identification of critical bottlenecks is possible early in the design process as well as it is getting insights into the system trade-offs together with verifying the design conceptually. But most important, this is done in an easy way with simple C programming hence with lower programming effort than VHDL. SystemC meets perfectly the system evaluation requirements for a system-level exploration of SmartMem.

SmaSyMo is inspired, but not based or compatible with, the previous simulator Trace-based Architecture Performance Evaluation with SystemC (TAPES). TAPES captures a network processor architecture using traces of the internal transactions required in a system with a common bus, multiple cores and a shared memory. Interestingly, the application-example using TAPES confirms that the use of a Buffer Manager in NPs greatly helps to improve the throughput up to a 270% with respect to a pure-SW solution [79], which backs the decision of using a BM for demonstrating our segmentation concepts, as exposed earlier in this chapter.

3.4.1 Model Description

As mentioned above, SmaSyMo follows a two-abstraction level approach for modeling the components in the system (Figure 37). This is done to compensate simulation effort and performance accuracy, what can be adequately achieved by increasing precision in the simulation of the system's bottlenecks. In SmaSyMo, these bottlenecks have been identified as the share resources (especially the memory and bus), which is a common case in multi-core systems. A slow memory device is selected in order to better observe the variations of performance due to the segmentation algorithm, otherwise results with a fast memory would be masquerade by other bottlenecks in the system, specially the bus. The two abstraction levels present in SmaSyMo are:

- ➢ As inter-connection infrastructures, SmaSymo incorporates a cycle-accurate model of a full-duplex 64-bit Peripheral Local Bus (PLB). This is the bus interface that Xilinx includes for core connection in the standard library of its development environment EDK, offering performance, flexibility and collision arbitration between requests. As the bus load is expected to be quite high, it has been modeled with a great deal of accuracy and synchronized using its corresponding RTL model. The same is true for the memory controller, which has been modeled on the basis of the Xilinx's DDR-200 controller, though in that case allowing room for modifications to better study the behavior of faster memory devices.

- ➢ The rest of the components are event-based and incorporate functionality at transaction level. This means that each block-function is assigned with a pre-defined hardware delay that corresponds to that of the latter hardware implementation. Such a level of abstraction allows fast simulation as well as a satisfactory accuracy level, being the goal more the study of the functionality and the hardware architecture exploration rather than the evaluation of parameters like throughput or workload.

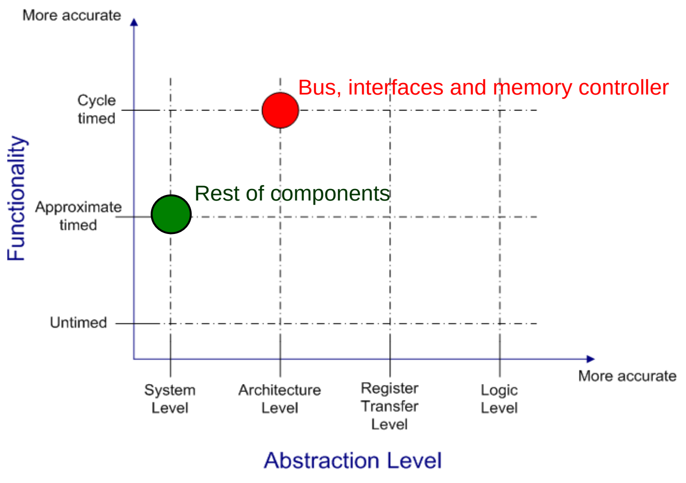


Figure 37: Abstraction level of SmaSyMo

In Figure 38, the block diagram of the implemented model is shown. At its centre lay the Buffer Manager, with some of its internal units also depicted. The bus connects the different master and slave devices in-between while a shared DDR-based buffer models the memory of the system. Two architectural variants are displayed: the direct-IF between BM and Packet Buffer as well as the local Control Buffer instead of a bus-connected one.

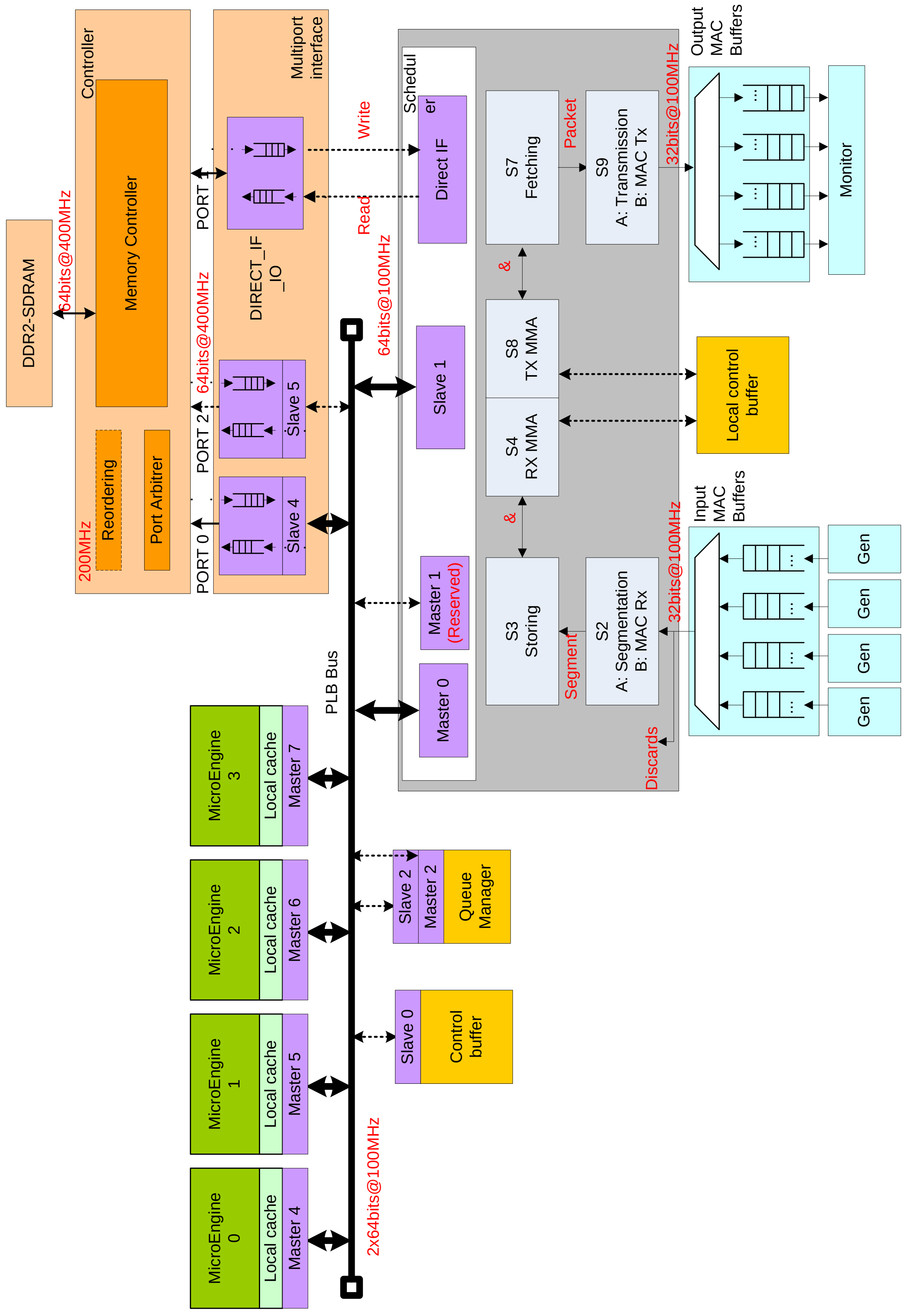


Figure 38: Block diagram of the SystemC model SmaSyMo, dotted lines display setup configuration alternatives

3.4.1.1 **PLB Bus**

The PLB bus consists of two 64-bit buses (read and write) working in full-duplex mode. Master modules have the capability to initiate data transfers over the bus towards a slave device. A typical example of bus transaction would be a memory responding to CPU read requests. There, access to the read bus is granted after the arbiter schedules all read requests considering higher priority to those with lower master number. Then the slave must acknowledge the request address as belonging to its range and send back a variable amount of data.

The actual time to accomplish the transfer depends on the capability of the slave to initiate the transfer, which can be long if for example DRAM latencies need to be considered. When data has finally been completely transmitted, the slave generates a completion signal, which frees the channel for next transmissions. Data can be transmitted simultaneously on the write and read channels at a maximum speed of 6,4 Gbps each though limited by the fact that only one bus arbitration per cycle is possible.

When a master is accessing a busy slave, the arbiter allows other requests to access the bus. If one request completion surpasses a given timeout, the arbiter triggers a bus re-arbitration and cancels the blocking transmission. The PLB also supports address pipelining, which consists in arbitrating a second request while the previous is being processed thus improving performance by enabling back-to-back data transfers. The bus does not support transfer aborts as it is considered unnecessary for the model as this error appears when the slave cannot answer a request after acknowledging the bus address, which is never the case in the simulations.

For efficient bus arbitration, a trade-off between performance and delays generated when accessing the bus must be considered regarding the maximum burst length allowed in the bus. In order to evaluate this constraint, a test will be done first with a 128-Byte transfer limitation (the default value in Xilinx IFs) and then enabling unlimited burst transfers. This will be tested with the different algorithms to measure its impact on segmentation.

3.4.1.2 **MAC interfaces**

Up to four independent GMAC-ports can be set up simultaneously, each with an independent source of traffic. The stimuli used include PCAP files, advanced IMIX and random packet generation, which suffices to capture representatively the behavior of routers when dealing with common internet traffic scenarios (user traffic, backbone, etc). As it is necessary to generate enough frames to saturate the system, the generator actuates on the inter-arrival time between packets recorded in the PCAP files, which is shortened or enlarged correspondingly to achieve the desired stimulus rate. In the output, four MAC buffers have been accurately modeled to reproduce the transmission delay of a packet when transmitted over an Ethernet link. It must be annotated that the operation of the MACs in any case influences the system performance, as they deliver an aggregated BW of four Gbps.

3.4.1.3 **Buffer Manager Model**

The BM receives variable-length packets from the MAC layer, to perform then packet segmentation in fixed-length segments or alternatively using the proposed segmentation algorithms. The incoming dataflow is first buffered internally in a local buffer. In parallel, the Segmentation Unit calculates packet segmentation and passes this control data on to the Storing Unit. Before storing segments in the Packet Buffer, the Storing Unit requests free addresses of the corresponding segment type from the RX unit.

In this unit, a memory management unit is capable of allocating multiple-sized segments in memory, allowing for a dynamic resizing of the segment pools by allocating same-size memory blocks upon demand. This strategy consists in dividing the available memory space in smaller blocks (2 Kb) and allocating only same-size segments in the block. When the level of the address cache goes under a defined threshold for a specific address type, the mechanism allocates a new 2 Kb-block to the required address pool. This design decision will be further discussed in section 4.3.3.

Once the Storing Unit receives the corresponding free segment addresses, it stores the packet in the Packet Buffer and finally generates a Packet Descriptor. Afterwards, it triggers an interrupt to the Processor Cluster where the first available CPU can access the PD queue in the BM over the PLB Bus and fetch the PD.

On the transmission side, the reception of a PD together with the appropriate TX command triggers packet transmission. The Fetching Unit (FU) reads the data segments belonging to the packet and reassembles it. At that point, the used addresses are sent to TX-MMA for discarding, thus allowing for later address reuse. The packet data is transmitted to the next stage of the transmission pipeline, the Transmit Unit that interfaces with the MAC buffer while the FU fetches another packet in parallel. Between these two units, an intermediate buffer has been allocated to thus keep the TX-Unit feed with new tasks. This solution reports beneficial results in the sense that increases the TX-Unit’s performance when the data fetching operation becomes irregular due to excessive bus load.

3.4.1.4 **The Processor’s Cluster**

When a packet has been fully stored in memory, the buffer manager triggers an interrupt to the first available microengine hence achieving best-effort CPU balance (a register keeps track of the CPU status). The CPU fetches the packet descriptor and starts a forwarding application, yet one modeled with high degree of abstraction, characterized by its CPI parameter (cycles per instruction) and number of instructions to be executed per packet, which allows easier characterization of the application running on the CPU regardless of its functionality. In the following simulations, packets are always assigned with a fixed processing effort as we want to results being independent of the SW-application.

For modeling the access pattern to the Packet Buffer, it is assumed that each new packet produces a cache miss, as new packet data cannot be already in the cache hence always triggering the corresponding cache line read/write to the Packet Buffer. Each packet produces one read- and one write-access to the Packet Buffer from the CPU. Each access-length is 64 Bytes long when modeling header-processing applications whereas for payload processing the whole packet is retrieved/written-back for packet processing.

3.4.1.5 Output Queuing

After processing is done and results have been fully written back in the memory, the corresponding Packet Descriptor is transferred to the Output Queue Manager over the bus, where it waits its turn for transmission. After some time, the QM sends the PD to the BM, which triggers fetching and transmission to the corresponding output port.

The QM can be modeled in two different ways. The simplest consists in allocating a fixed number of queues and scheduling them in a strict round-robin way according to port availability. The drawback of this approach is that it does not capture the behavior of a real router where QoS classification distorts the transmission priority and port contention produces transitory buffering growth. In that way, the Packet Buffer is always almost empty, hence generating uninteresting/unrealistic results.

For that reason, the queue delay has been simulated according to the Weibull distribution, as suggested by Papagiannaki et al. in [44]. Their measurements of packet delay showed that the time to cross the router is in the order of 10µs -10ms, with predominance of the range between 100 µs and 1 ms. In order to better capture this delay, each incoming packet is assigned with a random delay when it arrives to the Queue Manager, roughly approximating a Weibull curve. As packet collision occur when two or more packets are assigned with a same departure slot, additional delays affect the final delay a packet experience in the queues. To prevent that, the average additional delay, defined as the quadratic error between expected and observed delay, is evaluated and subtracted from the Weibull generator using a feedback function that finally properly approximates the real Weibull curve. The delay simulated has been depicted in Figure 41 (right) and compared with crossing time measurements by the Papagiannaki group (left).

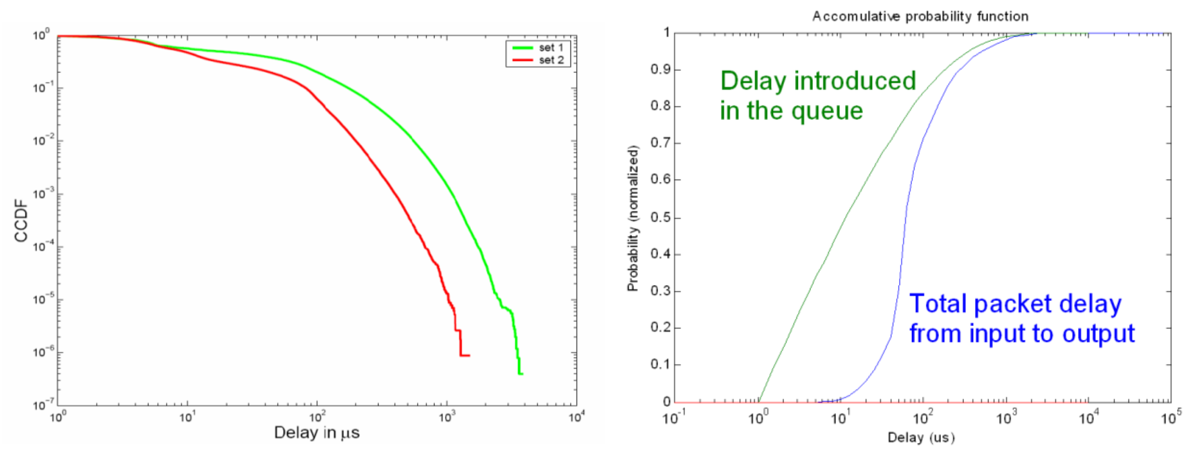


Figure 39: Packet delay observed in an OC-48 router (left) compared to the delay simulated in the model (right)

3.4.1.6 DDR-SDRAM Memory Controller

An essential part of every Memory Subsystem is the memory device the Packet Buffer is mapped on. For SmaSyMo, multiple DDR-SDRAM devices compliant with the JESD79-2E standard have been modeled. Their performance is compared with that obtained with another model, this capturing the behavior of the Xilinx's DDR-200 controller, which is available in Xilinx's development platform EDK. This second device offers the advantage of easy calibration by comparing the model results to RTL simulations to thus gaining in modeling accuracy.

The controller can be configured as single port, as by default in most simulations, or alternatively with multiport capabilities. In this second mode, the Buffer Manager can be connected directly to the controller hence bypassing the PLB bus in order to off-load it, which yields a throughput increase along with reduced access latency (see results below).

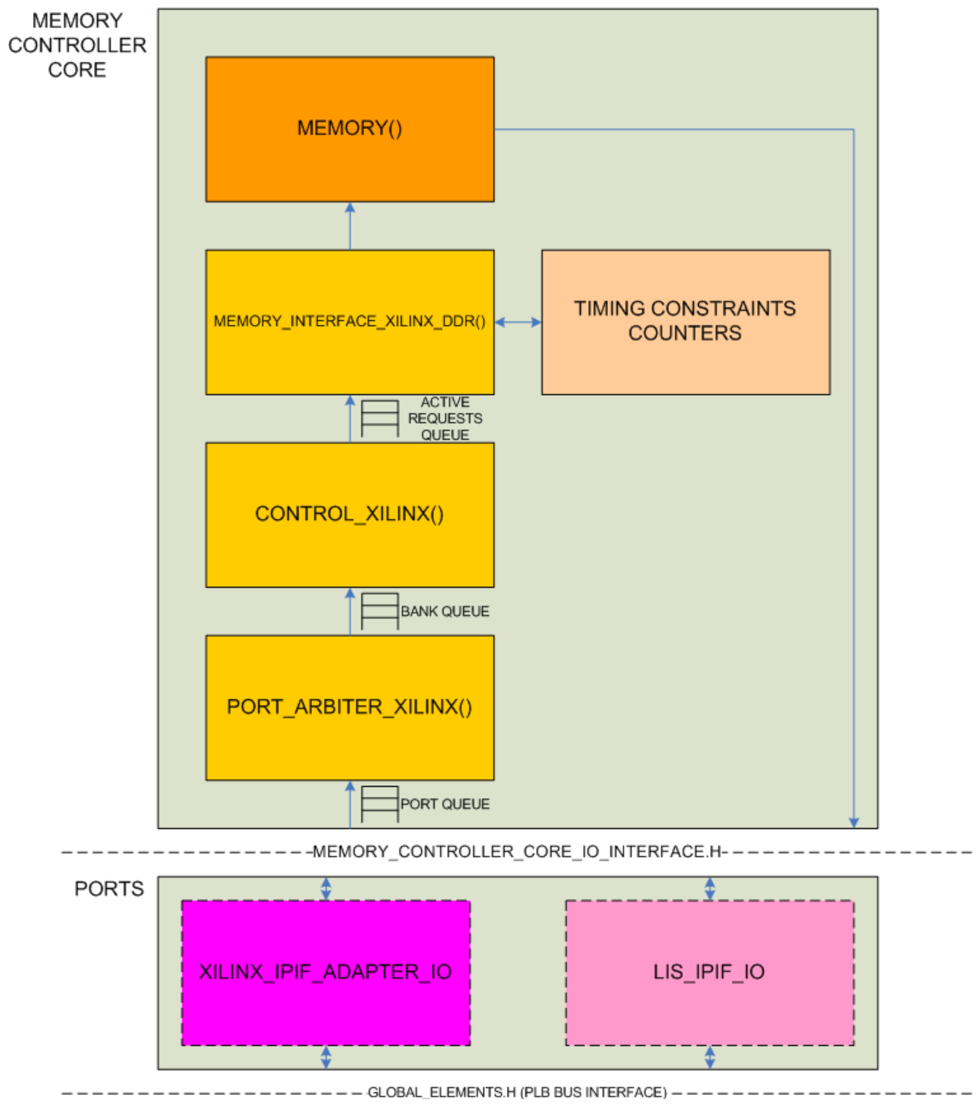


Figure 40: Detailed block diagram of the memory controller model, two alternatives are depicted for the connection to the bus (Xilinx-IPIF or LIS-IPIF).

The model of the memory controller consists of five principal blocks. A clear interface is inserted between them, making the insertion of variants an easy operation (for instances using different interface types with the bus or different DDR options). These are the five main blocks, beginning with those closer to the bus and up to the memory:

- Port_Arbiter_Xilinx: reads the requests queue where the input ports store memory requests (port_queue), and passes them directly to the next stage of the model (bank_queue). It provides compatibility with the PLB bus as well as the direct IF modus and essentially schedules simultaneous requests to the memory.
- Control_Xilinx: classifies the input requests per bank, to allow a later reordering of request for the purpose of investigating performance improvement with access-reordering mechanism. By default, this feature is deactivated.
- Timing constraints counters: several counter functions are included to control the different timing constraints of the DDR memory, providing timeout signals to the other functions of the controller.
- Memory_interface_xilinx_ddr: it is the core of the controller, as it implements the state machine controlling access to the memory. It triggers the corresponding memory commands (row activation, pre-charge row…) as required. The timing is controlled by enable signals provided by the constraints counters.
- Memory: it receives commands from the memory_interface_xilinx_ddr function to then return data acknowledgement as the standard specifies. No packet data is stored.

3.4.2 Simulation results

3.4.2.1 Simulation of performance and storing efficiency

Let us focus now in the simulation results obtained with SmaSyMo with real backbone traffic, results for file 2, 3 and 4 in Table 2 are presented here as representative (average value). Other OC-48 files in Table 2 were tested as well together with IMIX traffic, reporting only marginal differences and so not adding any additional scientific value to the investigation. This is because the packet pattern in the backbone is normally made out of small, medium and large packets in different proportions, with little variation. It is more interesting to observe upstream and downstream traffic, which indeed present a different traffic pattern. All in all, over 200-k packets were used in this system-level simulation, requiring many hours of simulation time. The system setup is presented in Table 4.

Buffer Manager	RX-TX Full-duplex
Packet Buffer	DDR-SDRAM 200
Control Buffer	SRAM / PLB Bus
Buffer access	Over PLB Bus
Bus–BM interface	2 LIS-IPIF Masters

Table 4: Model setup for the simulations

As Figure 41 shows, the FSS segmentation reference results in a linear trade-off (six segment sizes tested), with the 64-byte value achieving excellent memory efficiency but low throughput while the 2048-byte segment obtains poor efficiency with best throughput. All other segment sizes with FSS obtain results in-between, with 256-byte obtaining the best compromise between both. This is in accordance to the observations in the first experiments of this chapter. Remarkably, the segment size selected has a great impact on throughput, being the FSS-2kb throughput almost twice the value simulated for FSS-64. Again, this FSS behavior matches that seen in the previous experiments.

When focusing on the new segmentation algorithms, both MEO and SLA achieve superior throughput than any FSS configuration, again in accordance with the high-level study presented before. In general, MEO achieves near optimal efficiency (95%) although lower performance than SLA, in which case throughput is as high as 2 Gbps. The MEO algorithm with three segment sizes (1024, 256 and 64 Bytes) achieves approximately 100 Mb/s of throughput more in comparison to the 256-Byte FSS, while achieving 15% more storage efficiency. The SLA algorithm performances with four sizes (64, 128, 512 and 1Kb) is 15% higher than FSS-256 (approx. 300 Mb/s more) while its storing efficiencies hit an excellent 94%, only marginally worse than that observed for the FSS and 64-byte segments.

A more critical lecture could be done of these results. If FSS-256 achieves only 15% less performance than SLA while delivering reasonable storing efficiency why investing in revamping the Memory Subsystem if gains are moderate? Although this is true, traffic pattern is not always identical. It is possible that transitory patterns with concentration of packets of a specific size may cause a storage efficiency drop if the segment sizes used are not appropriately selected. To demonstrate this, the backbone stimulus is replaced by user's upstream and downstream traffic which contains a higher concentration of small packets.

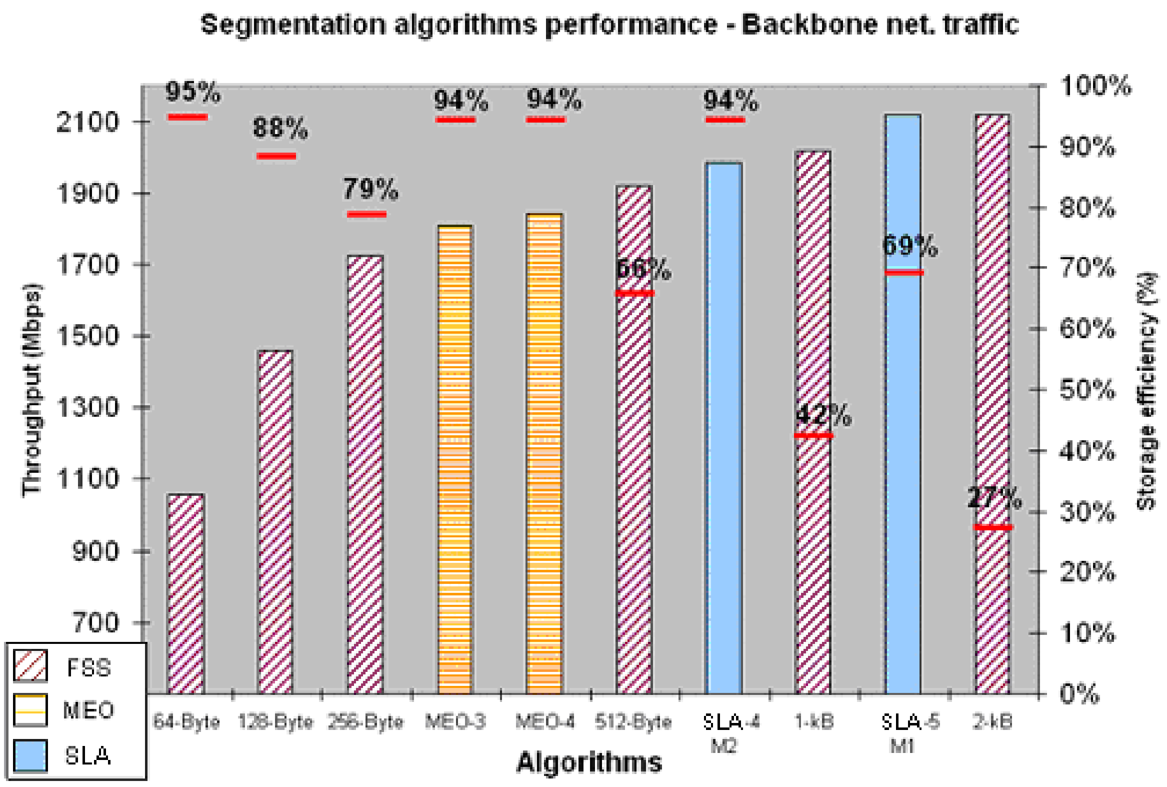


Figure 41: Trade-off between storage efficiency and throughput obtained for various segmentation algorithms with backbone traffic

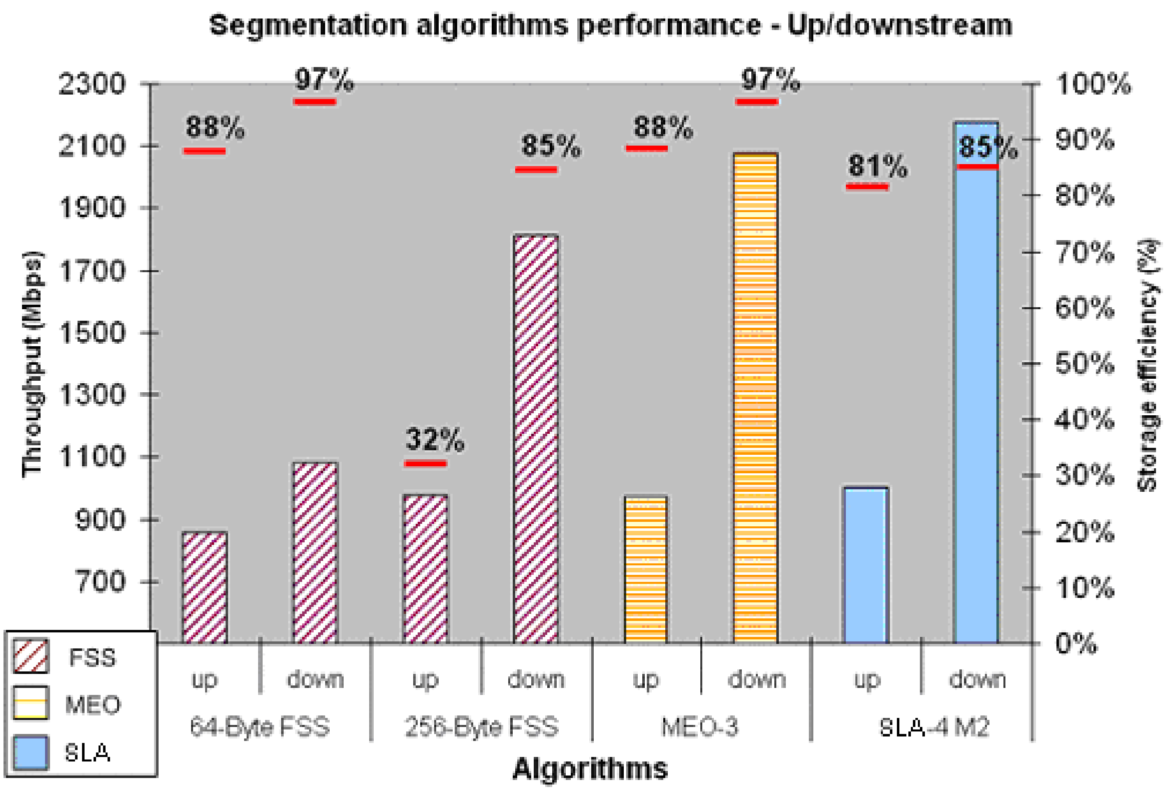


Figure 42: Throughput and memory efficiency results for up- and downstream traffic

Now as Figure 42 illustrates, storage efficiency drops down to 32% if only 256-Byte segments are available, while for example SLA-4 M2 hits a satisfactory 81%. On the other hand throughput is predictably lower when a larger amount of smaller packets have to be processed. Even in this case our algorithms achieve a slightly better performance, about 100 Mb/s higher. The new segmentation algorithms prove again superior when the traffic pattern contains an unusual amount of large packets, which is typical of downstream traffic in last-mile access links. Results show again better performance because large packets are more effectively transferred to the memory in long bursts rather than split in several requests.

These simulation results confirm SLA-N4-M2 (64, 128, 512 and 1kb) as the best suitable choice for hardware implementation.

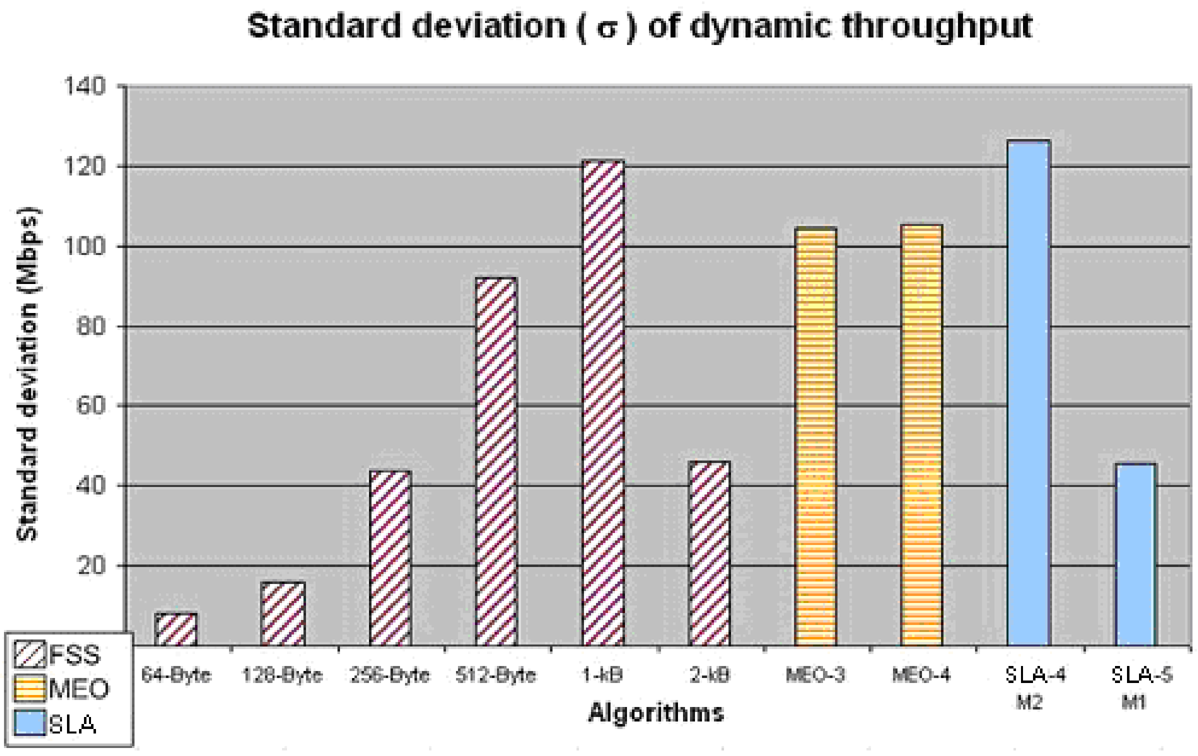


Figure 43: Standard deviation of dynamic throughput for various algorithms

In order to complement the average values presented above, Figure 49 depicts the standard deviation (σ) of the dynamic throughput obtained when simulating different PCAP files with backbone traffic. It is a useful way to evaluate whether strong variations of performance can be expected depending on the stimuli and the segmentation algorithm. As the figure illustrates, those segmentations with lower average number of segments per packet experience more performance oscillations. The reason for that lays in the generation of longer data bursts with bigger segments, which take more time to complete. One particular master occupies the common resource for a longer time thus delaying the access to the bus and impacting the performance of for example the CPU, who is also attached to the same bus. Exceptions are the segmentations where all packets are always stored using one segment, which present a fairly steady throughput. The explanation is that since there is only one segment to be transferred, the random latencies that factor in to the total latency do not play such a major role and consequently, variations in memory access latency have lower influence over the final dynamic performance.

The deviation observed for the SLA-4-M2 algorithm is only about 120 Mbps (5% in relative terms), which is slightly higher than that of MEO and FSS. It is though a reasonable value in absolute terms and thus not considered an inconvenient for recommending its use in the HW prototype.

Finally, simulation results validate another assumption of the high-level model, which is that there is an inverse linear relation between the average number of segments and the throughput obtained. This can be clearly been observed in Figure 44, where the algorithms simulated with SmaSyMo are depicted comparing the throughput obtained and the average number of segments required per packet.

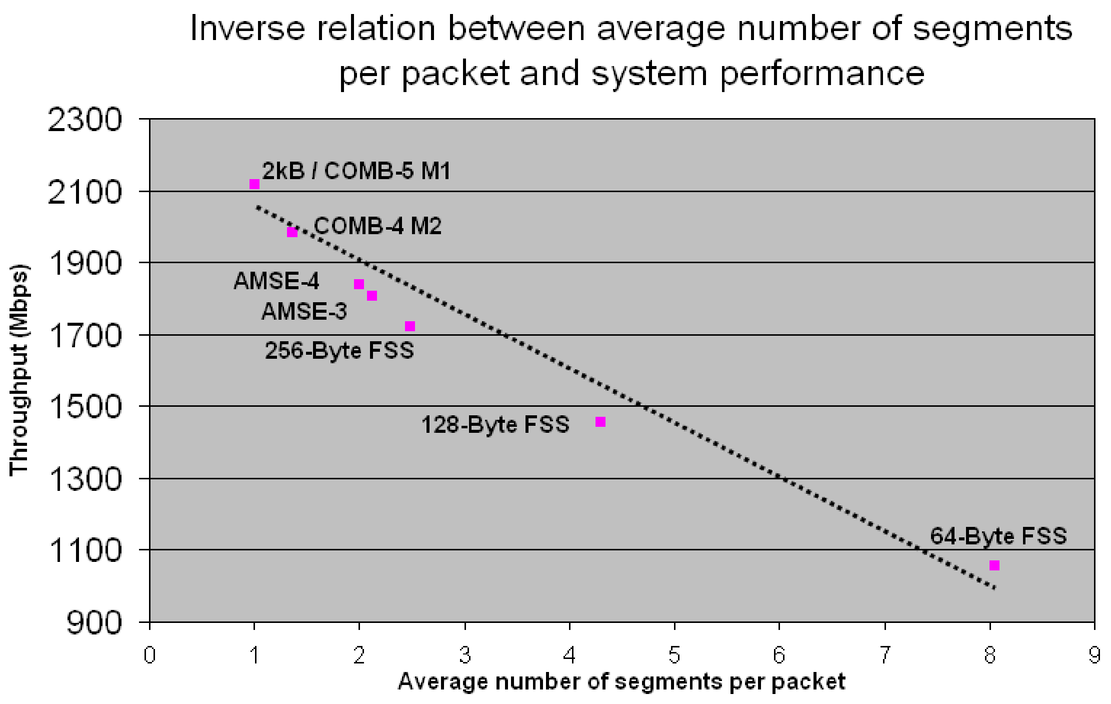


Figure 44: Reducing the average number of segments leads to higher performance

3.4.2.2 Impact of memory types

A further step in our investigation is to determine the impact of various memory technologies and system topologies on the algorithm's performance. Since the Packet Buffer in NPs is usually implemented using commodity DDR-SDRAM chips, it is interesting to evaluate how the performance of various DDR technologies and speed grades affects the overall system throughput in conjunction with the selected algorithm, the SLA-M2-N4.

Figure 45 summarizes the results of this investigation which demonstrate that faster memory technologies bring only marginal improvements to the overall system performance, regardless of the segmentation scheme used. This can be attributed to the fact that the latency of a memory access depends only partially on the memory contribution to the total transfer completion time, being bus access delays a factor to take into consideration. Moving the Packet Buffer to on-chip SRAM is more effective, since SRAM latency is significantly lower than any DRAM device (10 ns in the model). However, current fabrication technologies do not allow for the integration of the necessary amount of memory for the Packet Buffer if we want to map it on SRAM.

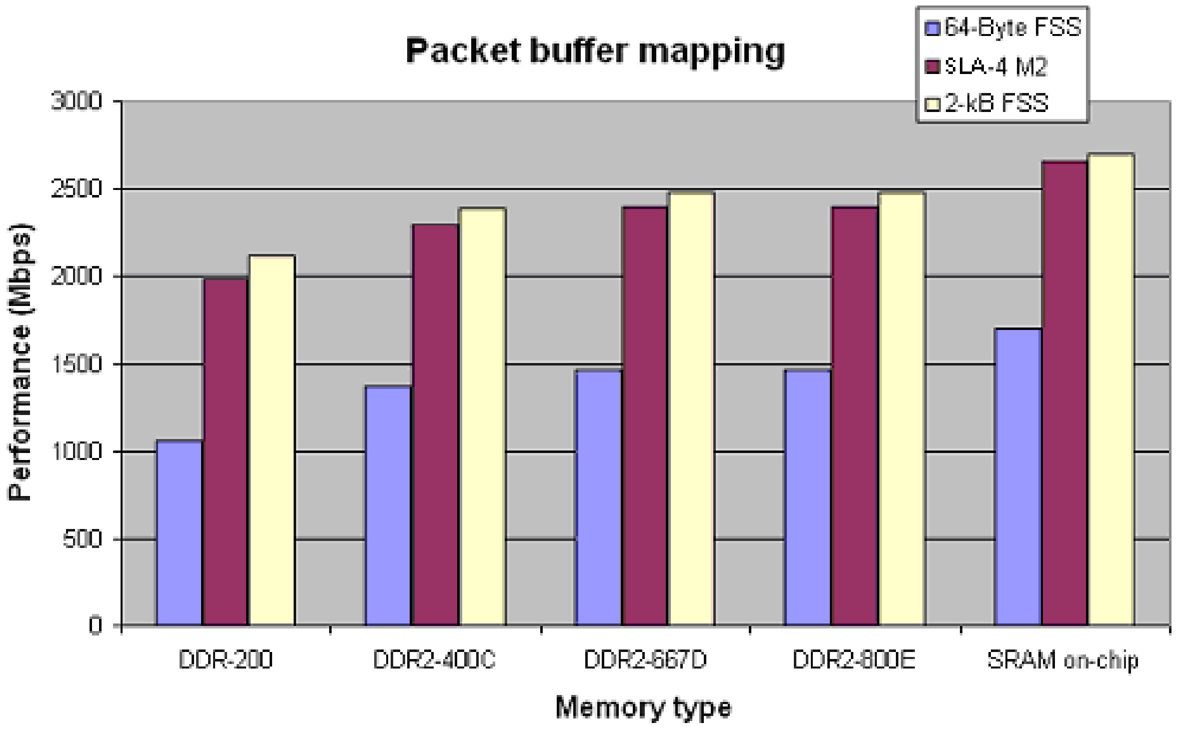


Figure 45: Throughput obtained for different Packet Buffer mappings, control mapped in a separate dynamic memory

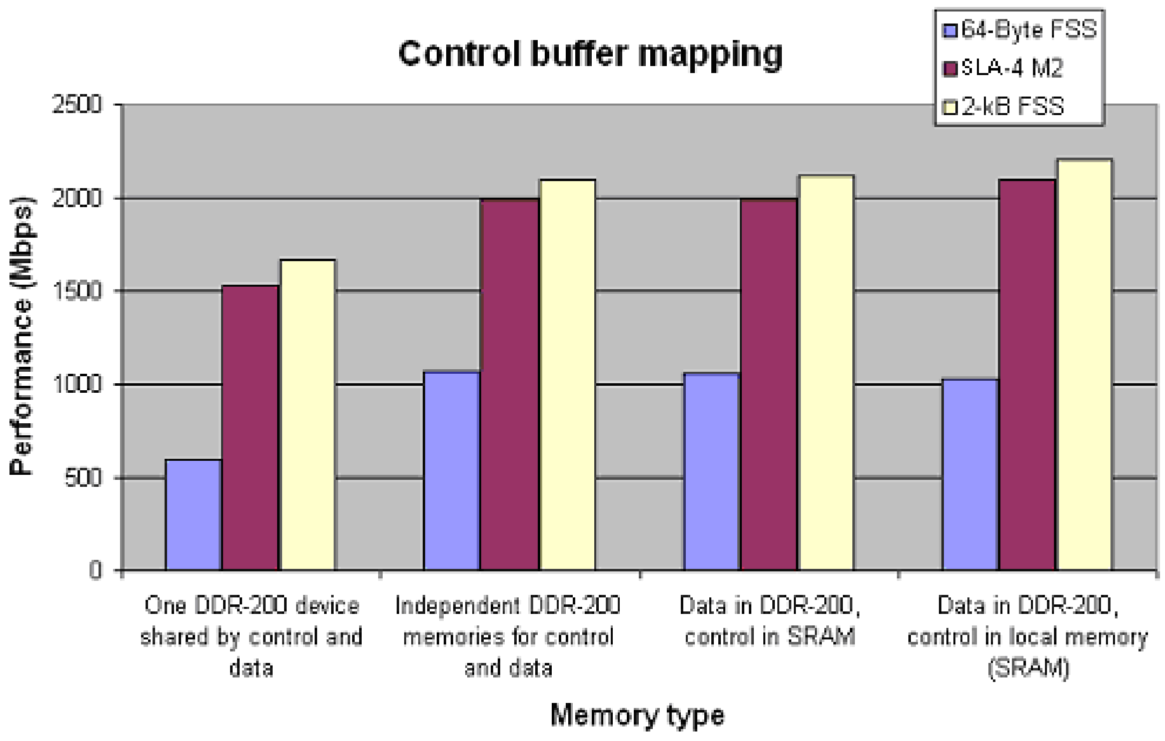


Figure 46: Throughput obtained for different Control Buffer mappings

Also of interest is the impact of various memory architectures which is illustrated in Figure 46. In contrast to the reference scenario where the Packet Buffer (DRAM memory) and the Control Buffer (SRAM memory) were attached to the same bus, we try utilizing one common DRAM buffer for both packet and control data, which brings a significant performance reduction regardless of the segmentation algorithm used. Moving the Control Buffer to a separate DRAM memory has no effect on system throughput, however it is possible that packets experience additional latency due to slower control update.

Finally we mapped the Control Buffer on a SRAM device local to the BM (no bus IF). In this case we get only minor improvement in throughput. These last results display a significant deviation with previous experiments on the hardware prototyping platform, where significant benefits where observed as a product of accessing the Control Buffer from the BM through a direct interface. The reason for this deviation is found in the bus load. When the Packet Buffer is the system's bottleneck instead of the BM's interface to the bus, control data updating does not produce any additional penalty in the BM operation. In the model, we are using separate masters for RX and TX precisely to avoid this bottleneck. All in all, the direct interface to the Control Buffer from the BM will be maintained in the HW implementation, to thus offload the PLB bus and accelerate the update of control data once a faster memory controller is implemented.

Additionally these results confirm that using longer segments proportionally increases performance independently of memory type.

3.4.2.3 Study of the CPU impact over the Packet Buffer performance

The CPU retrieves/stores information from the Packet Buffer and by doing that places additional strain on the inter-connection infrastructure and on the memory interface. Packet processing requires in many cases simple IP forwarding, which means that the packet header should be fetched from memory and the updated header written back once processing is done. However more complex packet processing applications require the entire packet data, which has to fetched from and written back to the memory. To analyze the effects of the CPU when it accesses the Packet Buffer, Figure 47 shows the results of an experiment where different parts of the packet are read and written-back by the CPU. We are after the throughput obtained when only the header is read and written by the CPU (header processing) and when the complete packet is required by the application (payload processing). Simulations are done for the selected SLA algorithm and three FSS algorithms (OC-48 traffic again) as reference. This sheds light into the question whether the previous results of multiple-size packet segmentation are somehow affected when the CPU interacts with the Packet Buffer.

First of all we confirm the general trend observed in previous simulations: longer segments lead to better performance, even when the CPU generates only short accesses per packet (far right case). We can also distinguish a pattern where performance decreases when a higher proportion of packets require payload processing. Obviously, this is due to the extra data traffic on the bus as well as on the memory interface, which pushes the BW requirement in the Packet Buffer up to four times the line rate (4R). Interestingly, SLA-4-M2 performance decreases are equal to all segmentation schemes, hence demonstrating that regardless of the processing access pattern the algorithm provides significant improved performance.

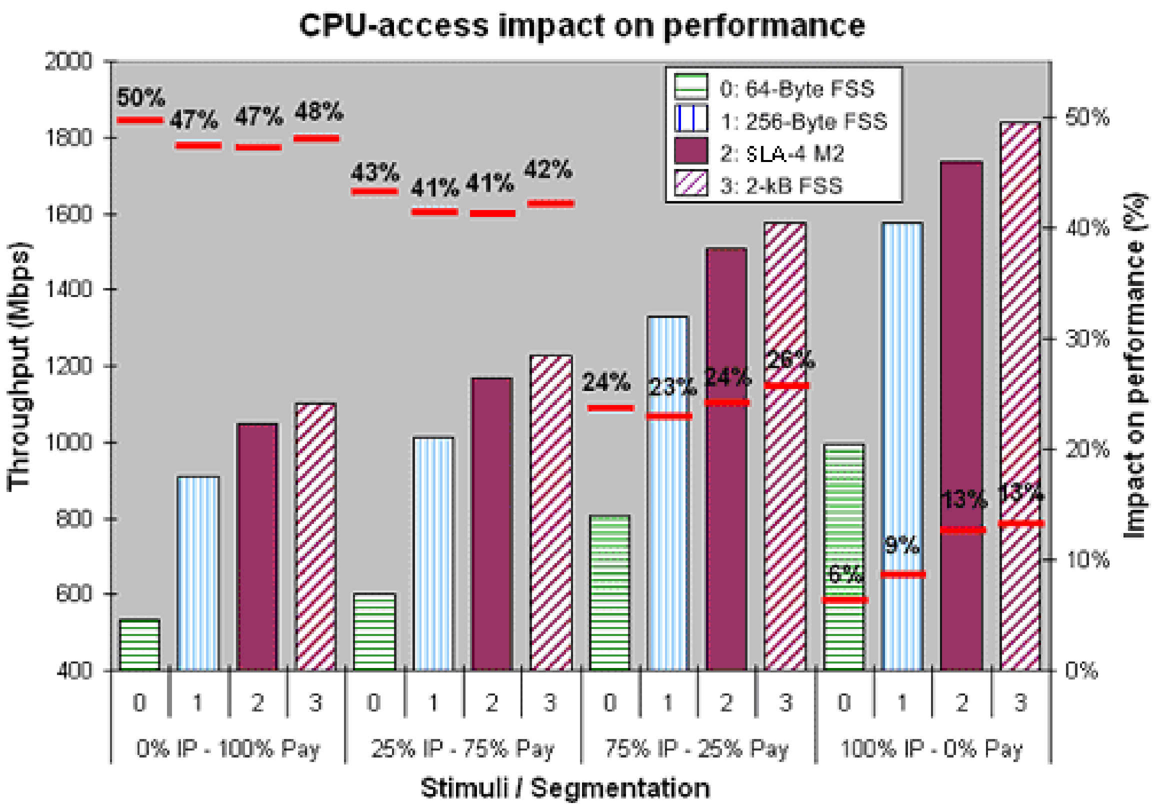


Figure 47: Throughput obtained for different CPU cluster access schemes

3.4.2.4 **Burst-length constraint on the bus**

A final parameter investigated was the maximum burst length allowed by the PLB bus. It is a good practice limiting the burst length on the bus to avoid master starvation due to long data transfers by other masters. When we introduced this limitation in our model, we observed a performance degradation close to 25% for the SLA algorithm, a similar figure than that of the 2-Kb FSS (27,63%). The bus constraint does not affect the final throughput only when the segmentation algorithm does not generate transfers longer than the burst limit, as for example with FSS-64B. These results discourage constraining the maximum bus transfer when the Memory Subsystem is clearly the system's bottleneck. Opposite when the CPUs are slower than memory (which is not the usual case), it is recommended to reduce the length of the burst as longer time to access the bus can stall a non-multi-threaded processor. We can conclude that the burst length is a parameter that strongly impacts the Memory Subsystem performance.

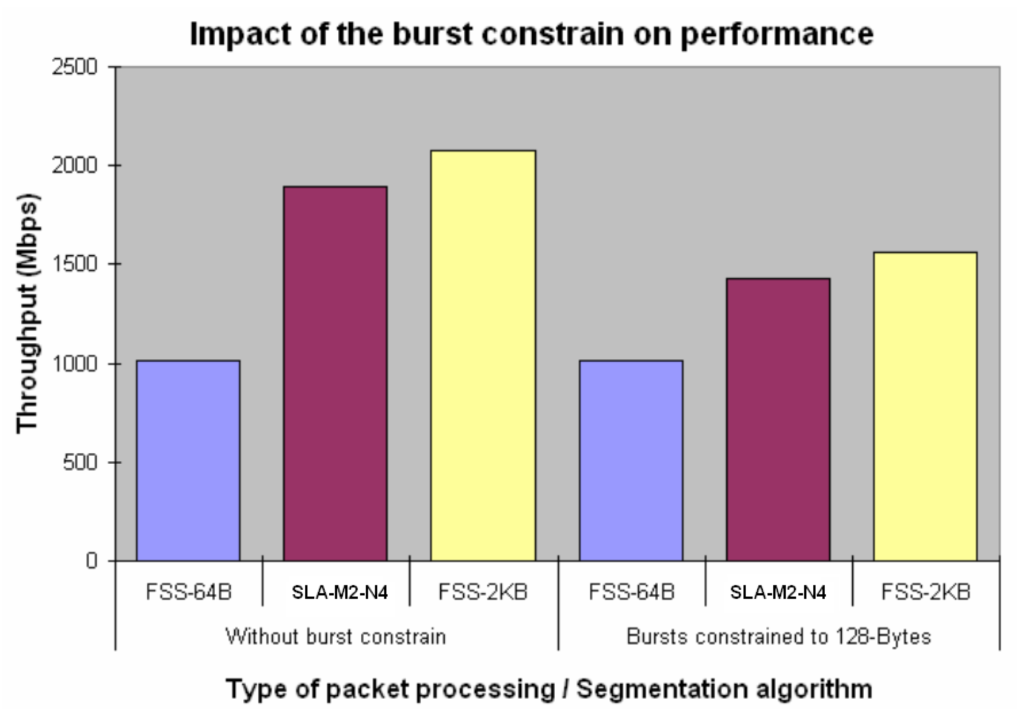


Figure 48: Allowing unconstrained transfers on the bus accelerates the system

3.4.2.5 Direct-IF between the Buffer Manager and the memory controller

Figure 49 shows an alternative communication strategy between the Buffer Manager and the Memory Controller. It consists in replacing the master interfaces to the PLB bus by a direct connection to a multi-port memory controller, thus effectively by-passing possible bottlenecks in the bus. For sure, data requests from the CPU still collide with those originated in the BM when they arrive at the memory controller, which requires scheduling and in that way generating delays. However this approach shifts the bottleneck to the DDR-SDRAM memory, a device offering higher throughputs than the bus as it works at higher frequency. For that reason, higher performances can be expected.

The model results (Figure 50) confirm the superiority of the direct-IF scheme over the all-over-the-bus approach. The diagram presents the throughput obtained with both architectures under study, alternatively for three reference segmentations when stimulating the system with advanced IMIX traffic.

The introduction of the direct-IF arguably yields a remarkable performance improvement, 100% throughput with the FSS-64 and 88% with the SLA algorithm. Effectively, this throughput increase comes in addition to that obtained with the introduction of multiple segment size segmentation, so **including the Direct-IF in the final hardware architecture** successfully adds performance on top of the benefits of the new packet segmentation algorithm.

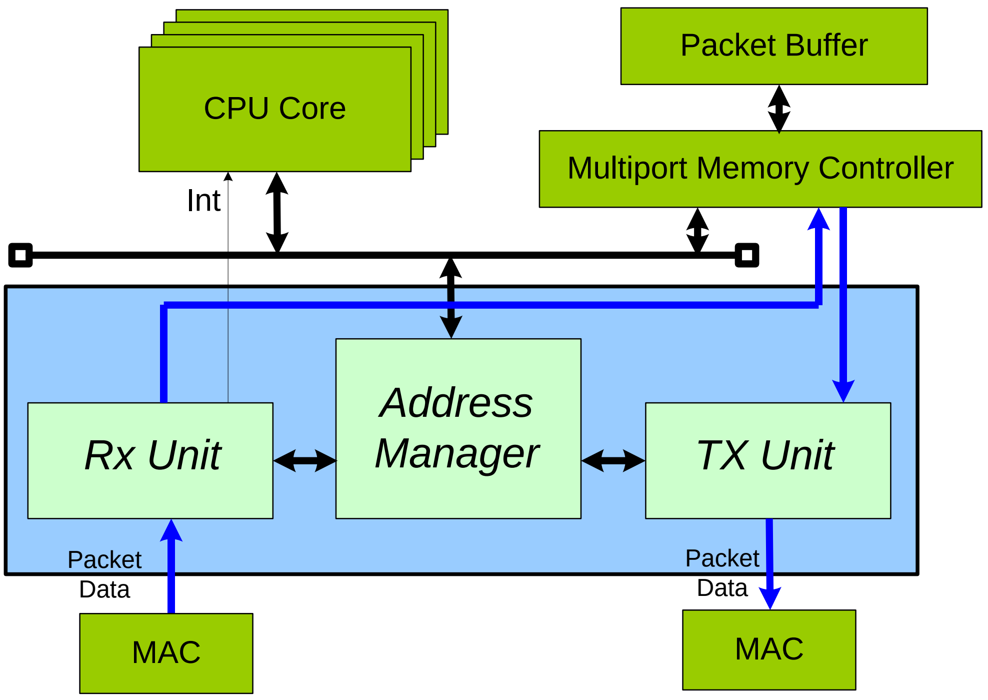


Figure 49: Architectural variant with direct-IF between the BM and the memory controller.

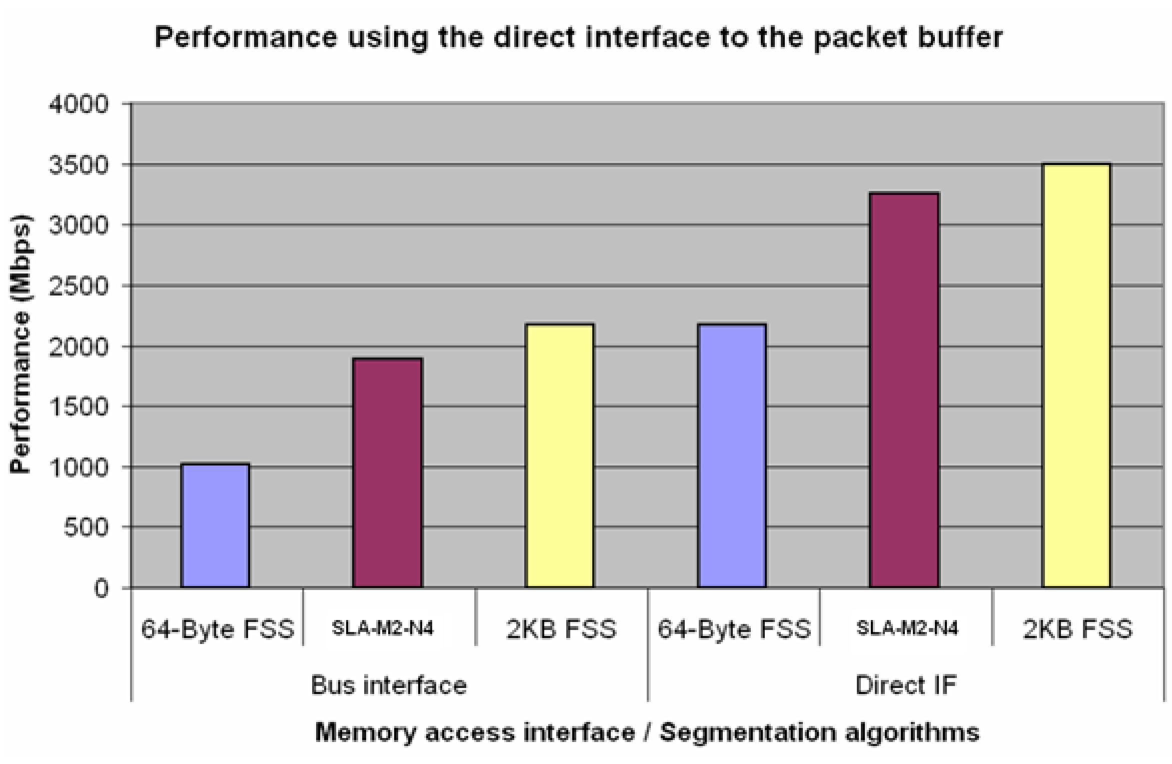


Figure 50: Improvement with the Direct-IF architectural variant

3.4.2.6 Conclusions from the system-level simulations

In this section, we have evaluated two novel, advanced segmentation algorithms for Network Processors that aim to decisively increase system throughput in comparison with current state-of-the-art FSS segmentation schemes. This is achieved by finding a sweet spot between the number of segments per packet and memory efficiency using the SystemC-based SmaSyMo model.

It confirms that algorithm configurations with 3 or 4 segment sizes like 64-256-1024 and 64-128-512-1024 provide excellent all around performance with both proposed algorithms (MEO and SLA). They achieve significantly increased throughput in comparison to the traditionally used fixed size segmentation scheme (FSS), while maintaining very high memory efficiency (in all experiments above 81%). In terms of throughput, both MEO and SLA achieve solid results only slightly under FSS-2kB, around 5% lower. **Only because SLA provides a hard bound on the number of segments per packet** without decreasing the high quality of the results in terms of efficiency and throughput, SLA is preferred over MEO.

Hence the provisional selection of SLA with four segment sizes (64, 128, 512 and 1024) and two segments per packet (M=2) is validated for implementation. From now, this algorithm configuration is assumed as the *default* configuration in the following analysis that evaluates buffer organization and its associated control structures.

Additionally we studied the impact of several memory types and system topologies on the algorithms and validated their superiority in a variety of conditions. We observed that using faster memory types brings relative small improvements in system performance and that decoupling the control from the data buffer leads to significantly higher throughput. Finally the impact of increased CPU load on the memory was demonstrated without affecting the superior performance offered by the proposed algorithms.

The interface the BM uses to communicate with the memory controller is of critical importance to accelerate data transfers between the BM and the Packet Buffer. The solution where a direct interface and a Multi-port Memory controller replace the bus achieves performance improvements of 88% with the selected segmentation algorithm. In case the bus solution together with a single-port memory controller is preferred, the study provides insights that constraining the maximum burst length leads to a penalty in terms of system performance.

Once the segmentation for the SmartMem Buffer Manager prototype has been established, it is necessary to proceed to review feasible buffer organization schemes for administering a Packet Buffer which now will contain multiple-sized segments. Next chapter tackles this question, focusing first on whether assigning fixed-size portions of the memory to each segment size suffices (static management) or otherwise dynamic memory allocation is unavoidable for preventing external memory fragmentation. As the exploration will demonstrate, dynamic administration is most recommended to increase the buffer utilization, which is critical during congestion periods. This poses new questions about how to manage such a dynamic Packet Buffer, which are addressed by proposing new suiting control structures.

4 Packet Buffer organization with multiple segment sizes

4.1 Motivation

The investigation picks up at the point where the previous chapter ended: the SLA algorithm with four segment sizes (64, 128, 512 and 1 KB) delivers the best combination of storing efficiency and memory access performance. Now first it is necessary to investigate the general memory requirements for each one of these segment sizes when segmenting real traffic. We need to answer questions regarding the share of the buffer claimed by each segment size as well as whether these requirements are static. If not, we must study the behavior of the different segment pools and how their size can be dynamically modified depending on the actual traffic pattern.

These questions have a critical impact on the efficient management and organization of the Packet Buffer. If memory is allocated statically to the segment pools, there is a risk that one or more of these pools are depleted while others still have free space. The goal of the first experiment in this chapter is thus to demonstrate whether a static memory allocation is suitable for multiple segment pools. The experiments investigate whether packet segmentation results vary considerably depending on the traffic pattern, which would exhaust different segment pools alternatively. As the results will show, only resizing the segment pools upon traffic requirement allows for an efficient memory storing. Dynamic pool management can be done by different means, so this chapter reviews methods already in use in other areas of computer science and check for a suitable one for SmartMem. In case none of the existing methods satisfies SmartMem requirements, a new way to manage the packet buffer must be introduced.

A second issue closely linked to packet administration, but by no means the simplest one, concerns the control structures needed for dynamic pool administration. One option is the adaptation of traditional linked lists to the new buffer requirement. This solution may be desirable because it is a simple and well-known solution for buffering in NPs, but has the inconvenient of large control buffer requirements.

Finally, we will see a complete new set of control structures that not only accomplish the required functionality but also introduce many benefits compared to state-of-the-arts methods of administering the Packet Buffer. By taking advantage of the hard bound of two segments per packet, only two memory pointers are required per packet, which removes the need of building chains of pointers. The results demonstrate that the new method not only achieves a huge reduction in the Control Buffer size requirements, as mentioned above, but also decreases the number of accesses to update control data, which in its turn reduces the load on the control memory interface. In the end, using multiple segment sizes not only accelerates memory throughput but also logarithmically reduces the size of the control buffer.

4.2 Dimensioning the segment pools

In this section, two investigations shed light on the requirement of segments when processing different traffic patterns using the algorithm SLA-M2-N4:

- First, traffic files recorded at different locations in the network are inspected to obtain their **average** segment requirements for a long period of time.

- Afterwards, some of the previous files are simulated at system level with SmaSyMo in order to gain insight into the **dynamic** behavior of the segment requirements, so to study whether strong variations with respect to the average may be expected and which consequences this may have.

4.2.1 Average segment requirements

In the first investigation, a plurality of files recorded from different network links is analyzed to quantify the average segment-pool size without timing considerations. Results are shown in Table 5 and in Figure 57, broken down by link type (backbone/user traffic) with comments and clarifications about the source. In total, we took a sample of 80.3 million packets from diverse link types monitored at different locations. The backbone files analyzed (in blue) can be freely obtained here [75], though some require authorization.

File	**Pkt**	**Source**	**Average pool size (%)**			
			64	**128**	**512**	**1024**
20050319-101000-0	3.8 M	OC192	2.8	4.5	31.7	62.4
20041109-232000-0	3.2 M	OC192	1.0	0.5	31.9	**66.6**
20041110-000000-1	1.1 M	OC192	5.8	1.0	31.1	62.2
20040130-132500-0	0.8 M	Gbit Eth	2.8	5.6	32.5	59.1
20031229-223000	1.8 M	Gbit Eth	2.0	**20**	29.9	48.1
20040219-105000-a	3.8 M	Gbit Eth Cat	3.3	5.5	32.4	58.8
20040219-125000-a	0.5 M	Gbit Eth Cat	3.3	5.5	32.4	58.8
20030115-103400-1	24 M	OC-48c	8.4	3.6	33.3	54.7
20030115-105900-0	6.6 M	OC-48c	3.8	2.6	33.5	60.4
20020814-090000-0	22 M	OC-48c	4.7	4.0	35.1	56.2
20030424-005500-1	9.1 M	OC-48c	6.3	4.6	34.4	54.8
Feb_21_08_16h	210 k	Mix user	3.4	1.2	31.8	63.6
Feb_21_08_11h	1.8 M	Mix user	2.2	**0.3**	31.9	65.6
Online_gaming	25 k	Mix user	3.5	10	28.3	58.2
P2P_traffic	21 k	Mix user	3.8	3.9	32.3	60.0
HTTP_Videostream	140 k	Mix user	2.2	0.6	32.3	65.0
User_traf_mix_1_d	101 k	Downstream	**0.3**	0.4	**36.6**	62.8
User_traf _mix_2_d	102 k	Downstream	0.5	0.4	33.1	66.0
User_traf _mix_3_d	45 k	Downstream	0.5	1.2	33.1	65.3
User_traf _mix_4_u	83 k	Upstream	**65**	18	**4.9**	**11.4**
User_traf _mix_5_u	69 k	Upstream	51	5.8	11.6	31.3
User_traf _mix_6_u	39 k	Upstream	51	5.5	12.1	31.2

Table 5: Average segment requirements with four segment sizes

File	Pkts	Source
20050319-101000-0	3.8 M	OC192 CENIC HPR
20041109-232000-0	3.2 M	OC-192 NLRZ Down
20041110-000000-1	1.1 M	OC-192 NLRZ Uplink
20040130-132500-0	0.8 M	Gbit Eth San Diego I
20031229-223000	1.8 M	Gbit Eth NCAR
20040219-105000-a	3.8 M	Gbit Eth Cat R&D Net
20040219-125000-a	0.5 M	Gbit Eth Cat R&D Net
20030115-103400-1	24 M	OC-48c CAIDA Net
20030115-105900-0	6.6 M	OC-48c CAIDA Net
20020814-090000-0	22 M	OC-48c CAIDA Net
20030424-005500-1	9.1 M	OC-48c CAIDA Net
Feb_21_08_16h	210 k	User traffic
Feb_21_08_11h	1.8 M	User traffic
Online_gaming	25 k	User traffic
P2P_traffic	21 k	User traffic
HTTP_Videostream	140 k	User traffic
User_mix_1_d	101 k	Downstream traffic
User_mix_2_d	102 k	Downstream traffic
User_mix_3_d	45 k	Downstream traffic
User_mix_4_u	83 k	Upstream user traffic
User_mix_5_u	69 k	Upstream user traffic
User_mix_6_u	39 k	Upstream user traffic

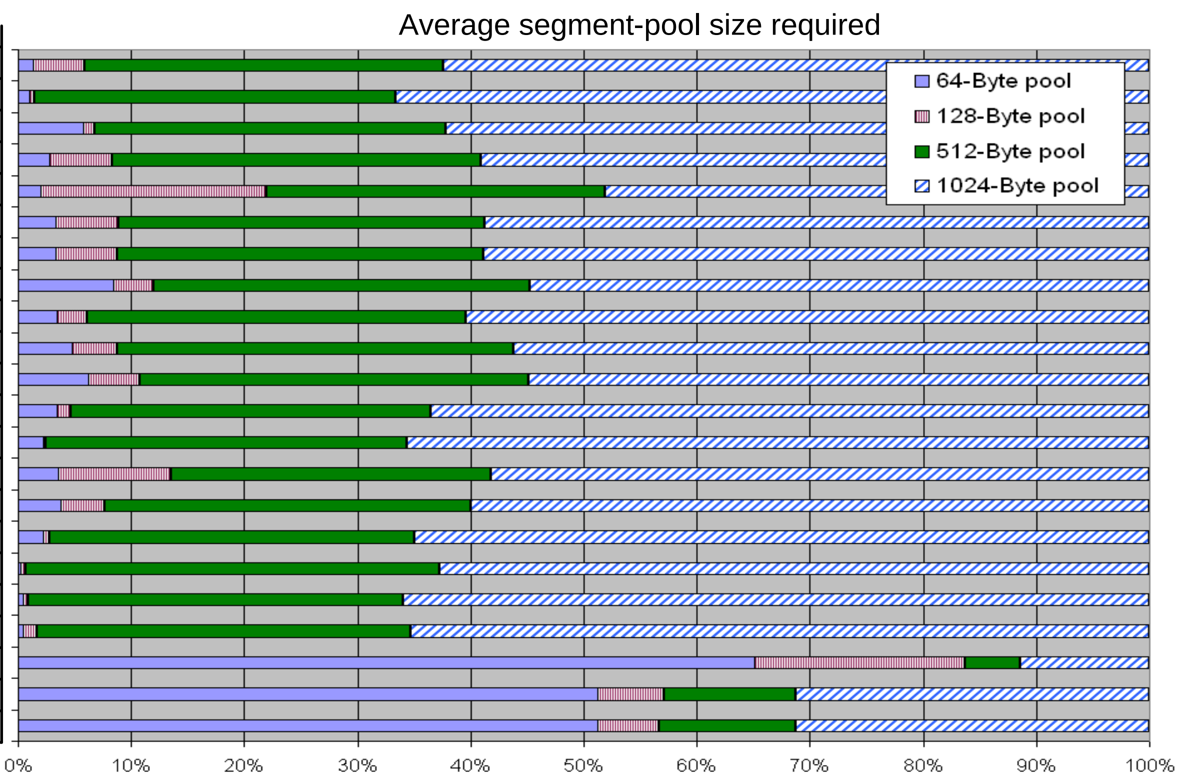


Figure 51: Graphical comparison of segment requirements.

As it may be observed, there are only slight variations in the results between the different backbone traffic files studied *(+uppermost eleven files*). In all cases the biggest memory share corresponds to the 1-KB pool (48-66%), followed in requirement by the 512-Byte pool that asks for 31-35% of the buffer space. The smallest segment sizes are less demanding on storage resources, requiring altogether a maximum of 22%. No discrepancies were found when looking separately at the uplink and downlink flows of the same router.

The study would be incomplete without paying a focused attention to traffic generated by internet users that access the network through slower connections. At the access point, special-purpose NPs multiplex for instance several DSL lines into high capacity links to afterwards forward this upstream traffic to the transport network. Symmetrically, downstream data flows from the backbone network to the users. We have sampled user traffic generated during the execution of those network applications responsible from the main amount of traffic in the Internet (i.e. P2P, video streaming, etc). This is done to study not only average traffic patterns but also to gain insight into more specific traffic patterns that may temporarily change the behavior of the buffer.

Again, we study whether these packet patterns generate distributions discrepant with those observed in backbone traffic. We observe only minor changes, which meets the expectations as the bulk of Internet traffic consists of aggregated user traffic. However, after dissecting the traffic in downstream and upstream and analyzing them separately, we observe major shifts of memory shares between the two largest and the two smallest segments, in both directions. Whereas downstream traffic consists mainly of large packets transporting the data required by the user, upstream comprises mostly small packets, common for requesting and acknowledging data transmissions.

In conclusion, the average recounting of pool sizes clearly demonstrates **poor storing efficiencies when using static segment pool** resizing. It is difficult to see how memory can be allocated in advance and still deal efficiently with the strong variations observed between backbone, downstream and upstream traffics (Figure 51). This statement will be further confirmed when observing the dynamic behavior.

4.2.2 Dynamic segment requirements

In order to properly investigate whether the segment requirements produced by a specific traffic change substantially over time, we need to observe congestion periods where the Packet Buffer may overflow. In modern routers and switches not implementing QoS, queuing tends to be minimal during normal operation thus reducing packet delay, so the actual packet buffering taking place in the router is usually low. Packet Buffering is required in case of port contention and as a temporary solution until the transport protocol reduces the flow transmission rate. This is produced when traffic from several inputs merge into the same output port. During these intervals, the memory must be capable of absorbing the queue growth so as to meet the actual Service Level Agreement the carrier is bounded to. Using again the SmaSyMo model, episodes of port contention with a variable duration of 10-20 ms are generated. This is enough to overflow a small Packet Buffer with OC-48 rates and observe the usage levels of each segment pool in saturation. Incoming traffic from two/three input ports are multiplexed into a single output that cannot cope with it temporarily, increasing the output queue size.

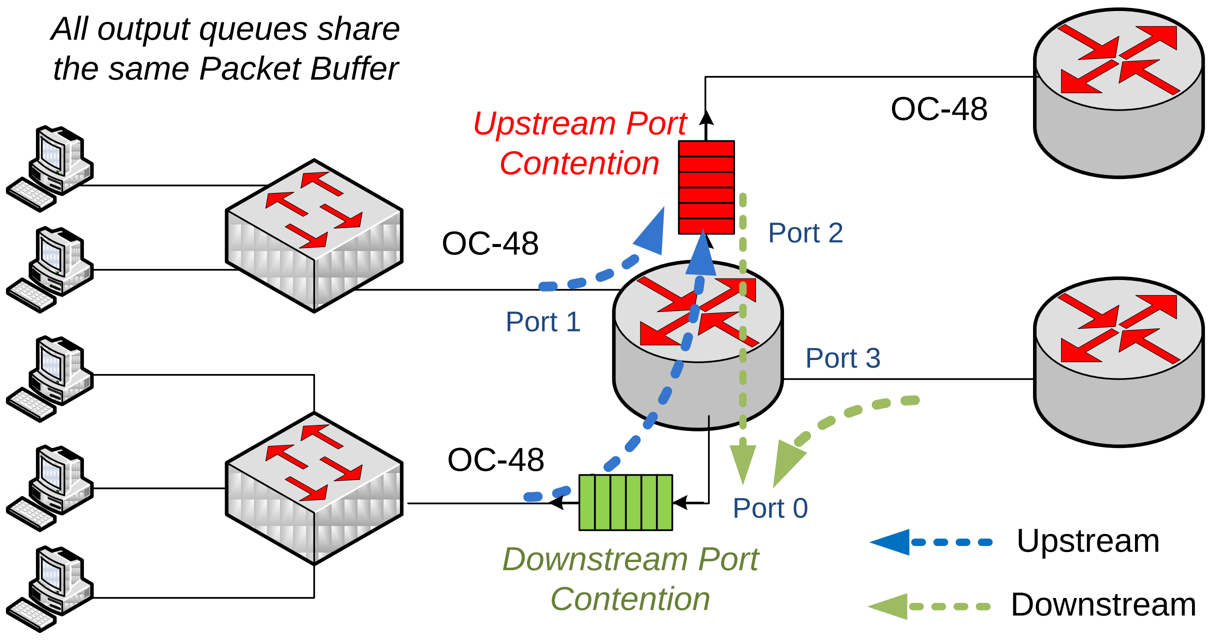


Figure 52: Port contention and the subsequent Packet Buffering increase

This experiment (Figure 52) is done twice, first feeding the system with backbone traffic and second configuring the router to work as access point. Each segment pool is statically assigned with a share of the Packet Buffer, in accordance with the observations from previous section: 54%, 32%, 8% and 8% of available memory for the 1024, 512, 128 and 64-Byte segment-pools respectively. In a second experiment, a function allocates free blocks of memory to the segment pool upon requirement, which allows investigating whether a dynamic management achieves superior buffering efficiency (Figure 53).

The experiment with backbone traffic reports neither substantial segment requirement variations nor poor memory usage in those contention episodes long enough to fill up the packet memory. We obtain memory efficiencies of over 90% with most of the files, with only minor differences in the pool saturation levels.

The access router case yields more variations. We stimulate ports zero and one with an aggregation of upstream user traffic. Packets are afterwards transmitted to two gateway links (ports 2 and 3). Symmetrically, these two ports are fed with downstream data, switched then to the subscriber's lines. Results are not as even as in the previous experiment. In case of contention in ports zero and one, the memory stores basically downstream packets while on the contrary upstream traffic is the main queue contributor whenever ports two or three get saturated. As up/downstream packet patterns are totally different, alternative segment pools are exhausted depending on the contented port.

Assigning different shares of memory to the pools does not help. First we tried the average values mentioned above obtaining very poor results due to the fact that the 64-Byte pool was clearly undersized. So we calculated new shares using the last six files in Table 5 which only contain downstream and upstream user traffic, to proceed then simulating with pool sizes of 45%, 22%, 5% and 28% of the space for the 1024, 512, 128 and 64-Byte respectively. Though we get an improvement, the charts depicted in Figure 53 still show alternative pools running out of segments while simultaneously others are nearly empty. Altogether, the memory usage never surpasses 69% what implies wasting several MB of memory. This inefficiency comes in addition to unused space in busy data segments (internal fragmentation), thus such low memory usage is unacceptable even considering the low cost of DRAM.

In conclusion, only with dynamic segment pool allocation can the packet buffer administration effectively respond to the variations in the traffic pattern and so efficiently store packets with multiple segment sizes in all studied scenarios.

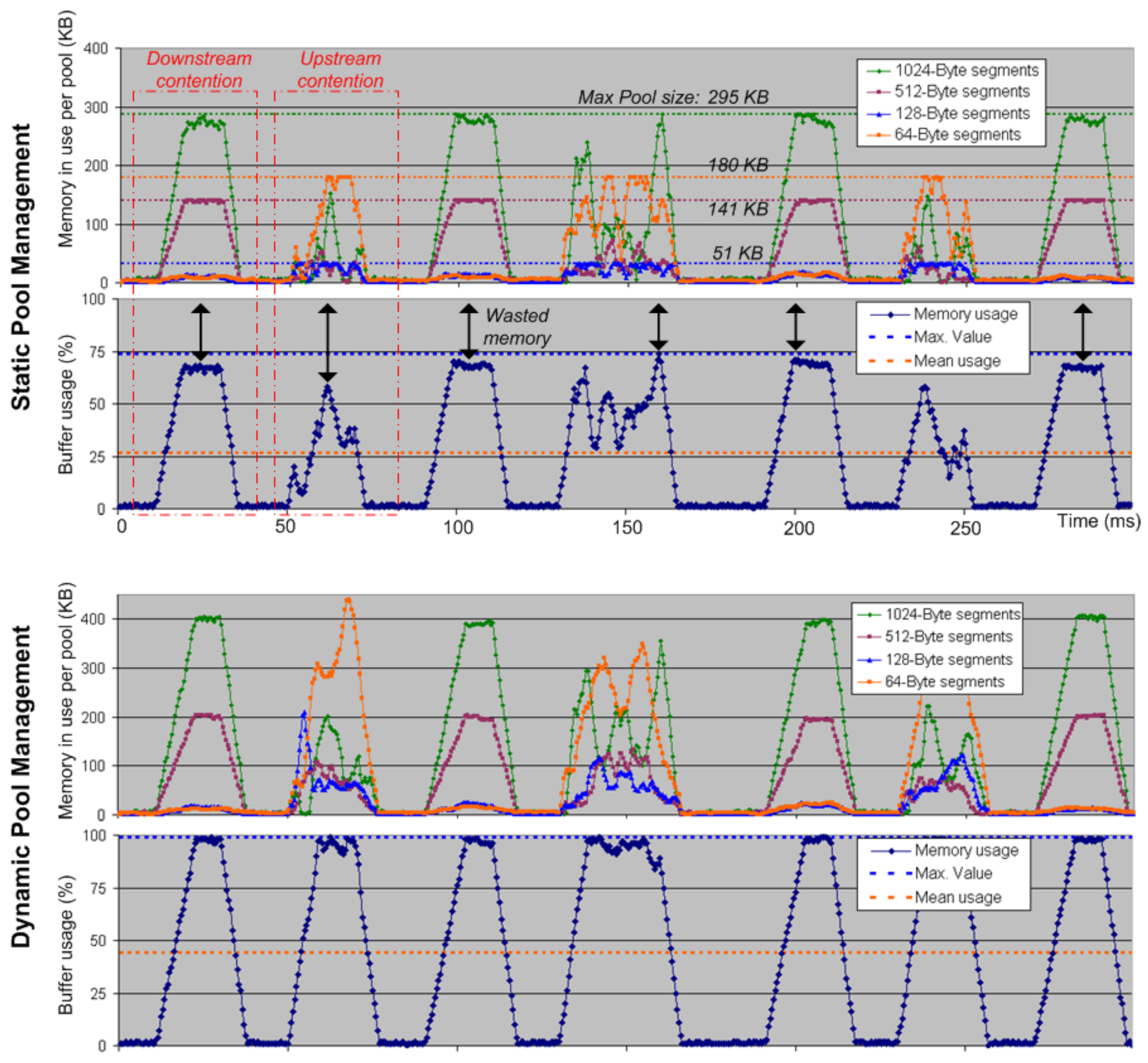


Figure 53: Packet Buffer usage during episodes of memory saturation in an access router with static and dynamic segment pool management

4.3 Memory allocation for dynamic buffer organization

Once established the convenience of dynamic resizing segment pools upon traffic requirement, the discussion focuses on the allocation policy for the SLA algorithm with four segment sizes. Most of the methods presented in the following have been extensively studied in the context of GPP processing but their conclusions are not directly applicable to the case we are concerned with. A Packet Buffer containing four different segment sizes presents a memory-accessing pattern quite different to that of GPP. For instance, data is stored and retrieved much faster hence triggering more update operations to the Control Buffer. Moreover, the packet departing order may be substantially different to that in the reception side due to complex QoS policies and other internal reordering effects. These considerations need to be taken into account when analyzing GPP memory administration methods for its adoption in SmartMem.

The Memory Allocator is the executor of the allocation policy selected. It is the mechanism that the Memory Manager uses to on the RX side allocate free memory blocks while on the TX discarding segments for reuse. If the function of the NP could be abstracted as a processing pipeline, the memory could be managed as a large FIFO queue. Such supposition cannot be assumed as complex queuing is fundamental for a large array of protocols, thus substantially modify the departing order of the packets. For the same reason, the Memory Allocator cannot rely on any particular "departure pattern" as different traffic shaping policies shuffle the transmission order or even randomly drop some of them. To add in uncertainty, less-than-best-effort protocols as the Scavenger class (e.g. P2P) may experience very long delays. Fragmentation is highest when a mixture of short and long-lived packets coexists in the same memory space.

In this section, different memory allocation methods are considered for networking applications with the SLA algorithm. We review first the concept of segmenting and coalescence of memory blocks, already present in previous computer applications. Then, the work contributes introducing the method of Gravity Centers (GC) as a possible criterion for allocating free blocks to any of the four segment pools. This is done in the context of the discussion about how to a achieve buffer homogeneity, which is desirable to allocate segment with the same size as close as possible and thus increase the probability of coalescence.

A second and more suitable allocation method is suggested considering not only its ability to forestall fragmentation, but also in terms of effort to update the corresponding control structures. We will see how this method achieves a huge reduction in the size of the Control Buffer, which is only possible because the maximum number of segments per packet is 2. In fact, we will arrive to the conclusion that slightly increasing the complexity in the control structures leads to a great reduction in the size of the Control Buffer as well as the effort invested in control data updating.

4.3.1 Segmentation and coalescence of memory blocks

Segmentation-and-coalescence methods (Figure 54 and Figure 55) pursue different ways to organize on-the-fly a unique memory space for variable-length packets. Essentially it consists in splitting the memory in very small atomic units from which larger memory blocks, the segments in our case, are assembled when a packet is allocated in memory space. When a packet arrives, the Memory Manager searches for a big-enough contiguous memory space where to allocate the packet. Then, it reserves the corresponding atomic elements, for example marking them as busy in a bitmap memory. The discard operation is also quite simple, as the allocated segments must simply be reversed to the free status. Actually, it is a flexible method that allows any object size provided its length is multiple of the atomic size.

The biggest drawback with the coalescing method is that searching for free space is a high-effort non-deterministic operation. Several methods exist to improve this search, like classifying similar-sized objects or simply ordering them per size. Anyhow, maintaining such control structures demands too much effort, only assumable when memory updating happens at low pace, which is hardly the case of high-speed packet switching applications. Moreover, high fragmentation is to be expected as even a handful of long-lived packets may jeopardize coalescence of a much larger memory area.

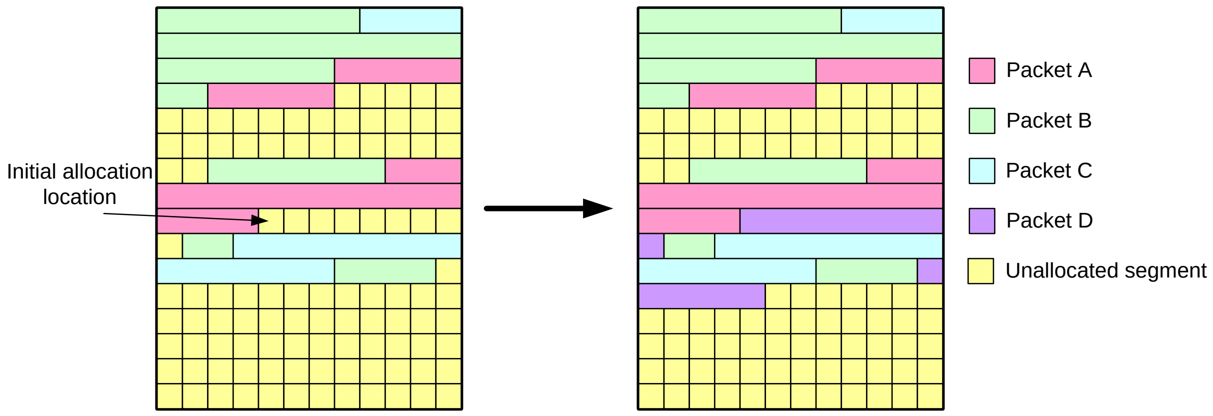


Figure 54: Memory allocation in the coalescence method, packet D is divided in two segments.

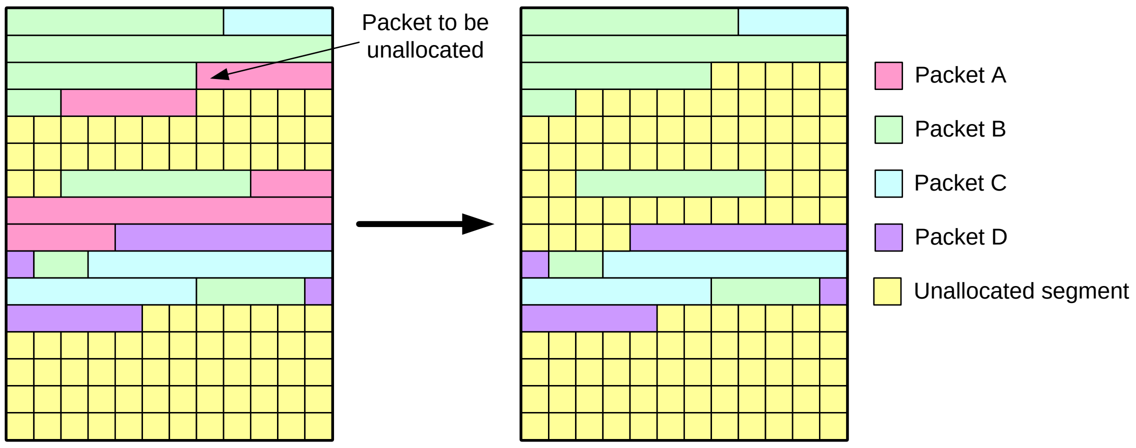


Figure 55: Memory discarding in the coalescence method, packet A is retrieved from two separate memory locations.

4.3.2 Gravity Centers

Here we introduce an alternative approach to the problematic of how to dynamically allocate memory to the four segment pools while allowing for reallocation of memory avoiding memory fragmentation. This method proposes to concentrate same-sized segments within a loosely-defined memory area. Thus it pursues storing homogeneity, which is achieved when same-size segments are stored as close as possible to a given memory position, known as Gravity Centre (GC). With four segment pools, four GCs are required and each new segment will be allocated in the area of influence of the corresponding GC. This automatically creates variable-size areas where segments have the same size, so no fragmentation occurs. Only in frontier portions of memory, packets of different sizes will share memory resources.

The goal is to maximize homogeneity in segment allocation to thus maximize the chances of coalescence taking place. In the example depicted in Figure 56, we observe that even though both memories are equally loaded, the one on the right is more homogenously distributed and as a consequence several additional 128-byte segments can still be allocated. Storing homogeneity is achieved by placing segments close to the Gravity Centre. There is no guarantee that this allocation policy achieves optimal results in all cases, as discards are random, especially if pools are resized quite often and several long-live packets are present.

Now, two parameters are of interests: where to place the GCs and whether they remain fixed or if on the contrary are allowed to move freely according to the variable size requirements of the segment pools.

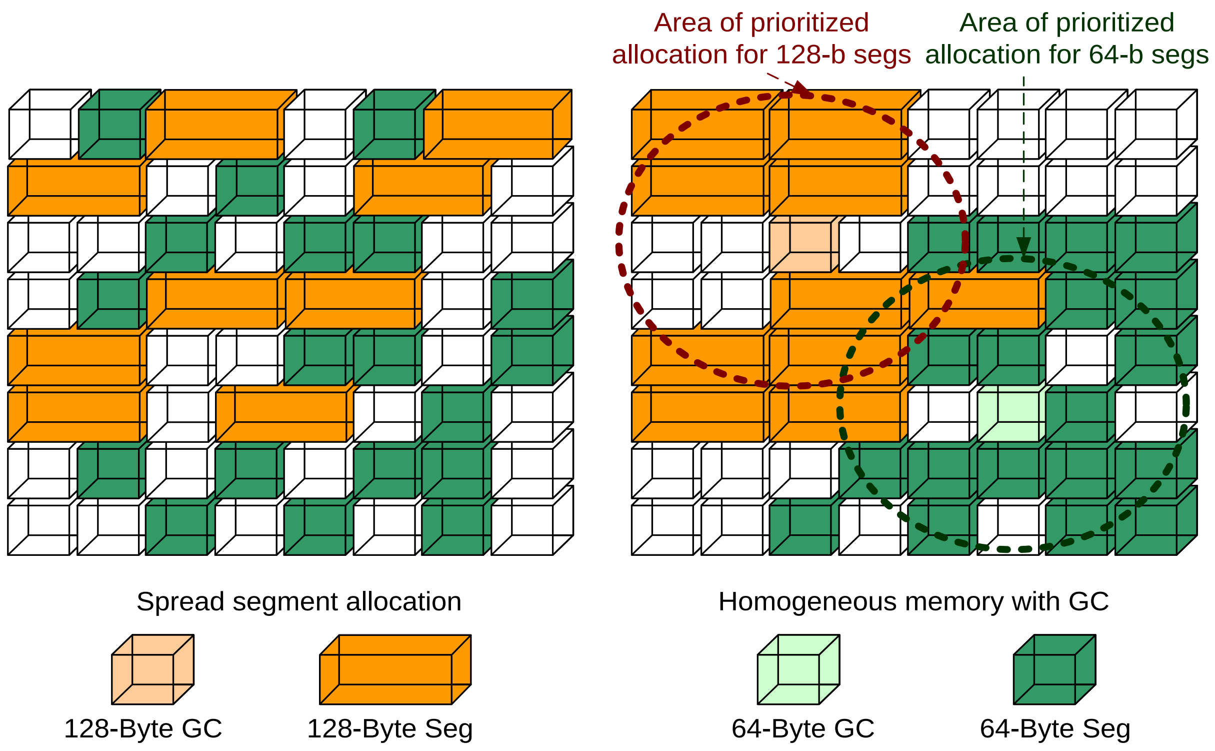

Area of prioritized allocation for 128-b segs
Area of prioritized allocation for 64-b segs
Spread segment allocation
Homogeneous memory with GC
128-Byte GC
128-Byte Seg
64-Byte GC
64-Byte Seg

Figure 56: Homogeneous allocation of segments (right) compared to random allocation (left)

The default position of the GCs (Figure 51) can be the one that maximizes the distance between them, thus allowing for equally big “free-from-interferences” oscillation areas (i.e. without any other different-sized segment there allocated) for each one of the four segment pools. Alternatively, we could set the GC taking into account the average pool size, for example according to packet statistic-aware distributions as for instances the ones described previously in.

In a GC-static approach, the initial position of GCs remains constant. They are then used as references by the Memory Manager to store segments in the corresponding GC’s influence areas. Although static GC is the simplest method (and the easiest to implement), its capacity to cope with strong traffic pattern variations that in its turn produce different segment requirements is questionable. In fact, in order to maximize the oscillation areas the GCs must be allowed to fluctuate according to changing pool size requirements.

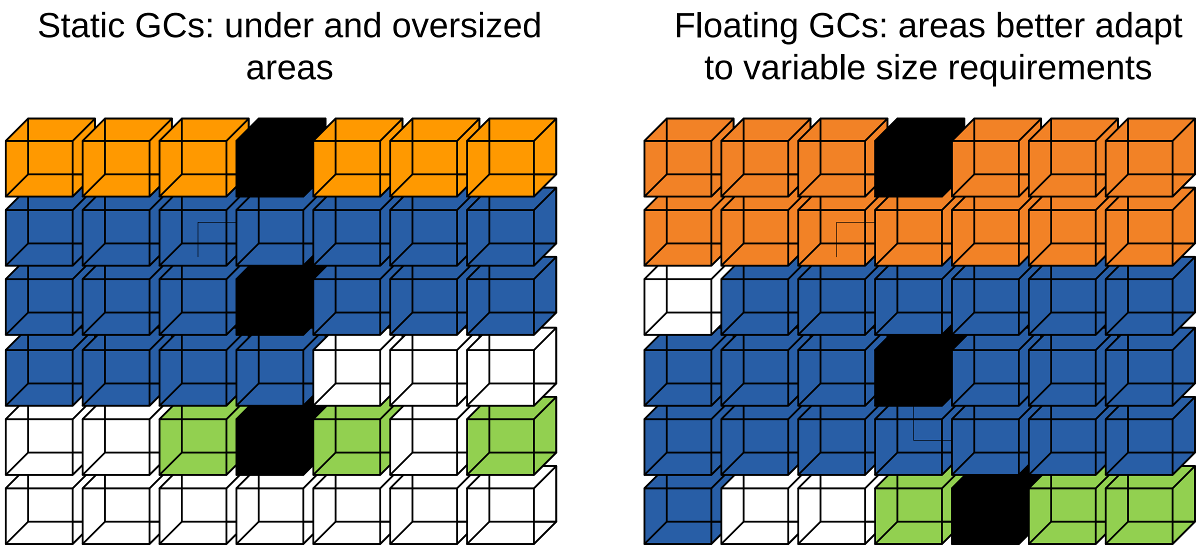

Static GCs: under and oversized areas
Floating GCs: areas better adapt to variable size requirements

Figure 57: Floating Gravity Centers allow for a better adaptation to changing segment requirements. The different colors represent areas allocated to different segment-size areas.

Now, free memory spaces need to be tracked in the influential area of GC. In order to search for them, some control data are required to:

- Control data to mark free segments as such, for example with a bitmap or a segment descriptor. As the granularity of the memory is equal to the minimum segment size, 64 bytes with SLA-M2-N4, the corresponding amount of bits or descriptor addresses is required.

- A searching structure like an indexed tree or a linked list that guaranties a search hit within a deterministic time, other than for example parsing the complete buffer until a free space is found.

- Afterwards, a reordering algorithm that calculates the distance between the GC and free segments to give allocation priority to those segments closer to the GC. A possible example of such a method consists in inserting segment at the appropriate position in the ordered list once discarded. Alternatively, a hierarchical tree would achieve similar results, though requiring additional efforts to recursively parse the tree and maintain its nodes.

$$L_p = \left[\sum_{i}^{P} \left(x_i - y_i \right)^{p} \right]^{1/p}$$

(7)

The criterion for selecting the most suitable free segment to allocate an incoming packet is proximity to the GC. The Euclidean distance is the preferred definition as it enables multiple dimensions when shaping the GC's influence area (see *(7)*, where *L* stands for the Euclidean distance between two points *x* and *y*, *p* is the number of dimensions). Having a two- or even a three-dimension area defined around a GC instead of a linear definition increases the homogeneity of the memory, without actually increasing the complexity for the searching mechanism. The coordinates of the block are logically implemented splitting the block address in as many parts as dimensions. Then, separate searches are independently executed in the three coordinates. The separate results are assembled back to obtain the final address.

The most important drawback of the GC method is that it poses formidable difficulties to the hardware unit in charge of searching for the closest segment to the mobile GC. It also requires of a large and very-fast control buffer to contain and access control data. For that reason, GCs are not considered for its implementation within SmartMem.

4.3.3 Block-based memory organization

A third option divides the memory into large (larger than the largest possible segment size) blocks, which can be subdivided into segments of different sizes as needed (see Figure 58). The assigning process is done dynamically, so an empty block may be allocated to different segment sizes if the memory management algorithm deems it appropriate.

Allocation using the segmentation method is easy in the sense that each block is assigned only to one segment size (which can be any value) and then segmented into the appropriate number of segments of the selected size. Inconveniently, to reallocate a block to a new segment pool the block must be completely free. For that reason, it is convenient to reuse allocated blocks rather than allocating new free memory, maintaining in that way the biggest amount of free blocks possible to adequately respond in case of a busy period.

The **block size** is the parameter that needs most accurate consideration. It requires balancing the size of the block, which needs to be large enough to accommodate any of the four segment sizes, but at the same time small enough to have sufficient number of blocks available for granular pool resizing. If the memory is organized in big blocks while the segmentation algorithm requires much smaller segments, a handful of long-lived allocated segments may prevent the reallocation of entire sections of memory. This can be especially dangerous in case a burst of traffic with a different segmentation pattern arrives while the memory is nearly full. In that case, the allocator cannot respond resizing the segment pools and thus meet the new segmentation requirements. Consequently, we find ourselves with a typical case of external fragmentation whenever a poor administration prevents the reallocation of these free resources.

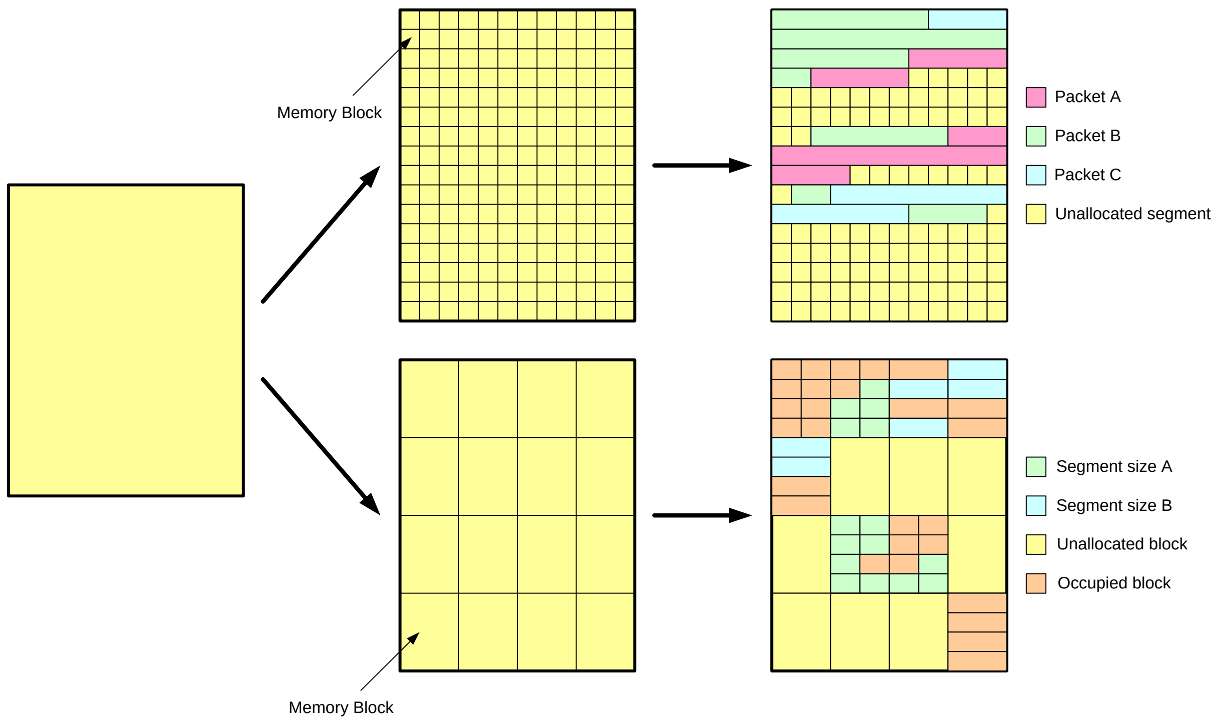


Figure 58: Comparison between segmentation-and-coalescence (up) and block-based methods (down)

Although reusing semi-used blocks asks for a more complex management, especially for a large number of object sizes, it successfully maximizes the number of free blocks available. Advantageously, the reduced number of segment sizes of the SLA algorithm (four) facilitates the mentioned tracking of allocated free segments, which poses a new requirement for the control structures (tackled in the next section).

All in all, provided that external fragmentation can be avoided by organizing the Packet Buffer in blocks while using small block sizes, it suits the requirements of the new packet segmentation algorithms. It is also a simple method to implement.

4.3.4 Conclusions

In this section, three different methods that dynamically allocate memory to multiple segment partitions are reviewed.

1. The segment-and-coalescence method delivers the functionality required but cannot prevent high fragmentation loss. In addition, the time required for allocating a new segment is non-deterministic and quite effort-demanding.

2. The newly-introduced method of the Gravity Centers prevents fragmentation by concentrating segments of the same size close to a static or floating point of the memory. Inconveniently, the method requires of a demanding effort in reordering the free segments according to their distance to the GC and although promising, requires of further investigations. Moreover, the searching mechanism for allocating new segments has been considered too complex for this version of the SmartMem Buffer Manager.

3. The block-based administration together with the proposed data structures (see next section) can search for free memory within a deterministic time as well as requiring simple implementation. Moreover, it generates no external fragmentation loss.

Due to the better suiting characteristics of the block structure, the Packet Buffer in SmartMem uses the block organization to the detriment of coalescence and Gravity Pointers. However this conclusion, there are still two parameters that need careful consideration regarding the Block method: the block size and block allocation policy that better avoids external fragmentation, which can only be discussed in conjunction with the control structures.

4.4 Control Structures for administering multiple segment pools

Now that we have determined that the segment pools must be dynamically resized and that memory will be organized into big blocks, it is time to focus our attention on the control structures necessary to implement these concepts. Control data is context information generated by the Packet Buffer to track free memory as well as packet data. This chapter studies which control structures better suit SmartMem.

The control structures present in current NPs are suitable for administering multiple segment pools, provided they are static. We can see such an administration in Figure 59. Segment Descriptors contain one pointer to build the different linked lists. Each descriptor is tracked by one list so many lists are necessary depending on the segment size and whether the segment is free or assigned to a packet. One free list per segment size is necessary when searching for free addresses, as this is more efficient than parsing a shared free list with mixed segment sizes. Besides, multiple lists allow deterministically obtaining a hit within one cycle. In addition, each segment descriptor must include an enlarged number of bits in the valid field, since obviously more data can be stored in the big segments.

When some free segments are assigned to a packet, a new linked list needs to be built and the first and last element kept in the Packet Descriptor. As segments of different sizes are originally tracked by different free lists, the **allocation of segments requires accessing all descriptors to rebuild the linked list**. Previously with only on segment size, that was not the case, as segments were assigned in block to a packet and only the free list was modified.

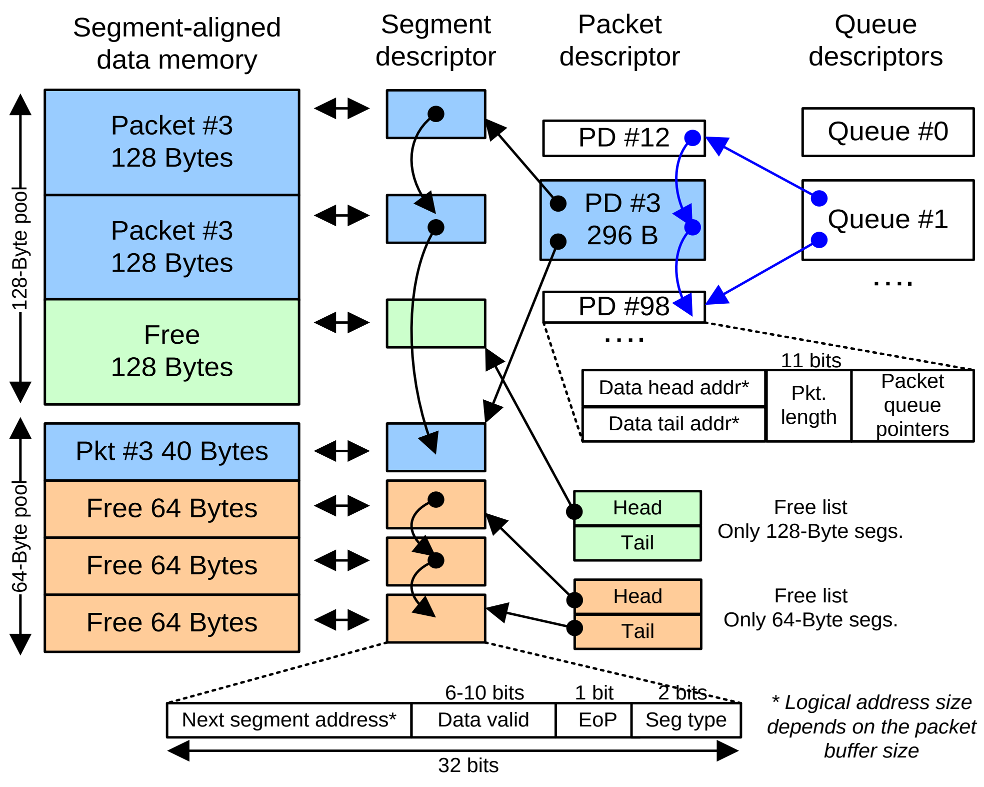


Figure 59: Control structures based on linked lists with multiple pools.

In that way, the main difficulty when using linked lists and multiple segment sizes regards tracking packet segments, which now includes segments previously being linked within independent lists. The number of control operations increases, as now every single pointer must be rewritten. The memory manager must access the Control Buffer whenever two different-size segments are consecutively assigned to the same packet, which increases the throughput required from the Control Buffer.

A similar operation is required in the transmission side to discard unused segments, in this case to attach the discarded addresses to the corresponding free linked lists. In the case of the SLA algorithm (max. two segments per packet), seven control accesses per packet are needed conjunctly between RX and TX in case of requiring two different segments (worst case).

The size of the Control Buffer is also critical as it is mapped on costly SRAM. In all buffer organizations, the Packet Buffer size is the ultimate parameter that defines the Control Buffer size. The use of longer data segments reduces control data as fewer descriptors are necessary for the same memory space, but it is in any case still in the order of a few Mbytes, depending on the final memory shares assigned to each one of the segment pools.

Inconveniently with linked lists, a larger Packet Buffer leads to a logarithmical increase in control data requirements, a consequence of increasing the address length stored in the descriptors, which is the result of calculating the base-2 logarithm of the number of segments. Sure enough, the number of descriptors increases as well with the packet size. With typical Packet Buffer sizes already in the order of GBs and growing linearly with the line rate, this is a trend hardly sustainable in future NPs.

4.4.1 Bitmaps and linked lists of blocks

In the following, a totally different concept is proposed to achieve a huge Control Buffer reduction and decrease the number of accesses to the Control Buffer. By taking advantage of the bounded number of segments per packet, it is possible to **get rid of linked lists** altogether. Since the packet descriptor (PD) contains two pointers to hold the packet's head and tail addresses of the linked list, with the introduction of the SLA algorithm and the hard bound of two segments per packet, there is no intermediate segment to be tracked. Not only the next segment address is unnecessary in the segment descriptor, neither is the valid data field in the segment descriptor. It becomes redundant provided that the packet length is stored in the packet descriptor. When packet length is smaller than the first segment size, we know that the first segment contains all packet data. Otherwise, the first segment is completely full and the rest of the packet is stored in a second segment. Finally, we need to reserve four additional bits in the PD to indicate the length of segments the addresses point at (Figure 60).

Therefore the only remaining function of the linked list is keeping track of free segments, which can be done by other means. A single bit per segment is now enough to mark its status as free or busy, however a fast method to search for free segments is still required, other than parsing the bits. As mentioned in the state-of-the-art, a hierarchical tree with three 32-bit levels was recently patented [80] precisely for the purpose of tracking free segments in the context of Network processors targeting OC-192 rates. This buffer management is software-oriented and only aimed to state-of-the-art FSS segmentations. But a CPU-based buffer management is suboptimal because it consumes several processing resources, critically needed for packet forwarding or protocol processing. This is indeed a heavy operation for the CPU. According to that patent, four Find-First-Bit instructions are executed for each free address lookup, which comes in addition to the sequential fetching of at least four tree-branches from the Control Buffer. To discard a segment, the tree has to be rebuilt, still adding more load to the processing core. This operation may violate the maximum search time, which is a function of the line speed of the NP.

Instead of that method, we propose a much simpler bitmap-based administration. It consists in grouping the status bits (free/busy) in blocks of 32 bits and associating each group with a 20-bit block descriptor. Notice that two bits suffice as control data for each 64-B of memory whereas the ratio with segment descriptors is 4-8 Bytes of control data for each 64 bytes of Packet Buffer. As with SLA the minimum segment size is 64 Bytes, a block contains 2 KB of memory. The block size could be 1 Kb, thus increasing pool size granularity, or bigger, which reduces the number of block descriptors along with the effort to update control data. 2-Kb is considered a satisfactory trade-off between both aspects, as we will see below in this section.

At this point we have two alternatives for tracking semi-used and free blocks. The simplest alternative consists in using the block descriptor for building linked lists, a well – known mechanism. Otherwise the linked lists can be completely avoided by implemented an indexed tree. This second mechanism is somehow more complex along with requiring more accesses to the Control Buffer but interestingly yields more accurate block allocation as well as deterministic operation time. However, the simplicity of a linked list is here preferred due to the quite small number of memory blocks to be tracked, even with large Packet Buffers.

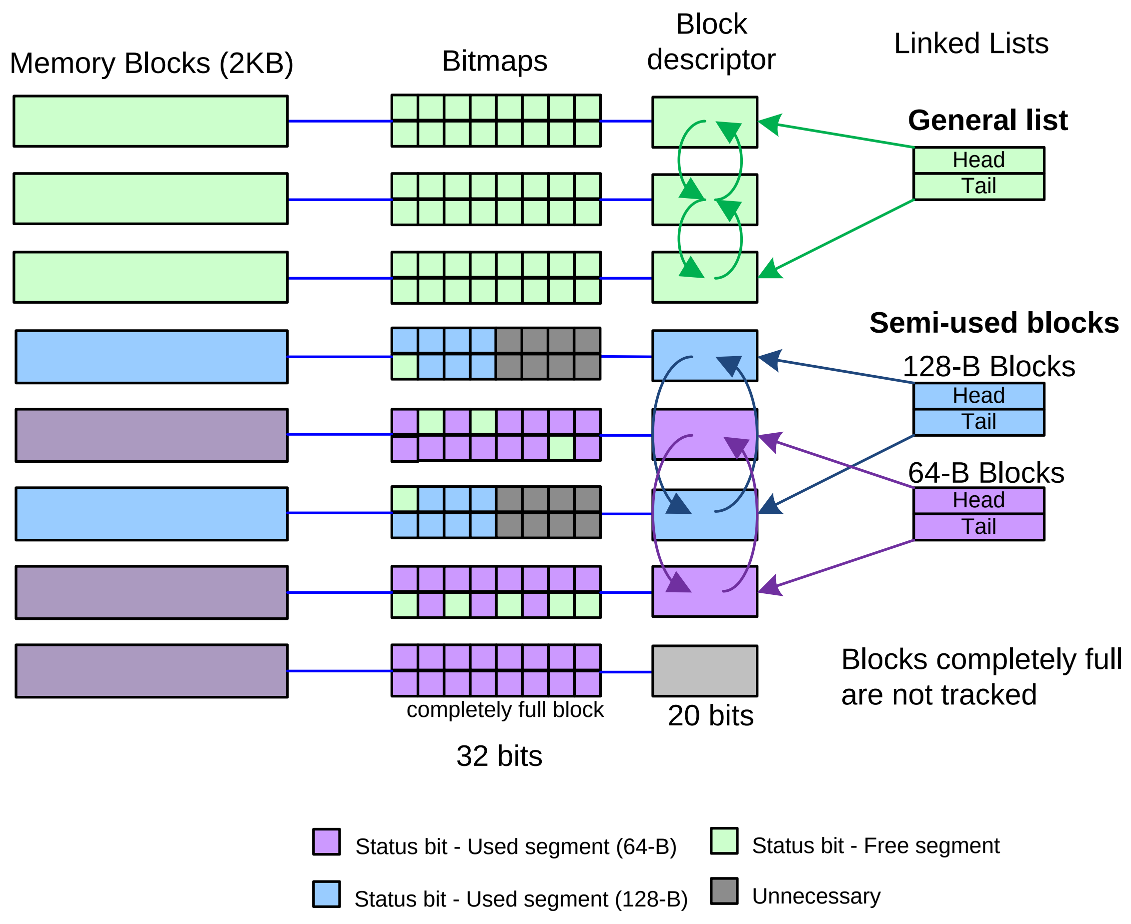


Figure 60: Bitmap-based memory organization

One linked list is required per segment pool, in addition to another for the free blocks. The concept of an array of free lists (or *quick lists*) each keeping track of a *sub-pool* of different memory-block sizes was first described for GPP in this quite venerable source [81], but with several differences regarding implementation and requirements.

In our concept, when allocating a new block, the corresponding quick list is consulted and if it is not empty, the first address in the list is picked. Otherwise, the manager looks at a general list that contains unused memory blocks. Its top element is segmented into multiple smaller sub-blocks that are then appended to the previously empty *quick list*, being available for new memory requests. Previously allocated blocks only return to the general list when all its segments are freed, so they can be reassigned again to the free list. Hence, it is highly recommended to reuse blocks already segmented to avoid excessive fragmentation.

As depicted, the block descriptor stores one pointer to a next block containing at least one free segment while belonging to the same type. Another list must be implemented to chain all free blocks and thus without assigned segmentation. If a block has no free segments, it is not pointed by any list.

4.4.2 Operations with bitmap-based control structures

Figure 61 depicts a block allocation to a segment pool and packet discarding for a 64-Byte segment. Only one access to the head of the block-list is required to fetch the first 32-bit vector, for sure containing at least one free segment (but possibly up to 31). Once the block is extracted, all its free segment addresses are made ready for storing incoming packets. Then, the head of the block list is updated with the second element in the list that is stored in the block descriptor. Last, the 32-bit vector is completely set to busy. All in all, in three control accesses up to 32 free segment addresses can be retrieved.

Likewise simple is the discard of unused addresses, as only the segment bit must be switched from busy to free. Exceptionally, when the corresponding block does not contain any free segment, it must be reinserted at the end of the free list and the previous tail pointer updated. By appending blocks at the bottom of the semi-used list, the tendency is to accumulate those blocks with more free segments at the top, hence increasing the average number of free addresses obtained in a single list look-up.

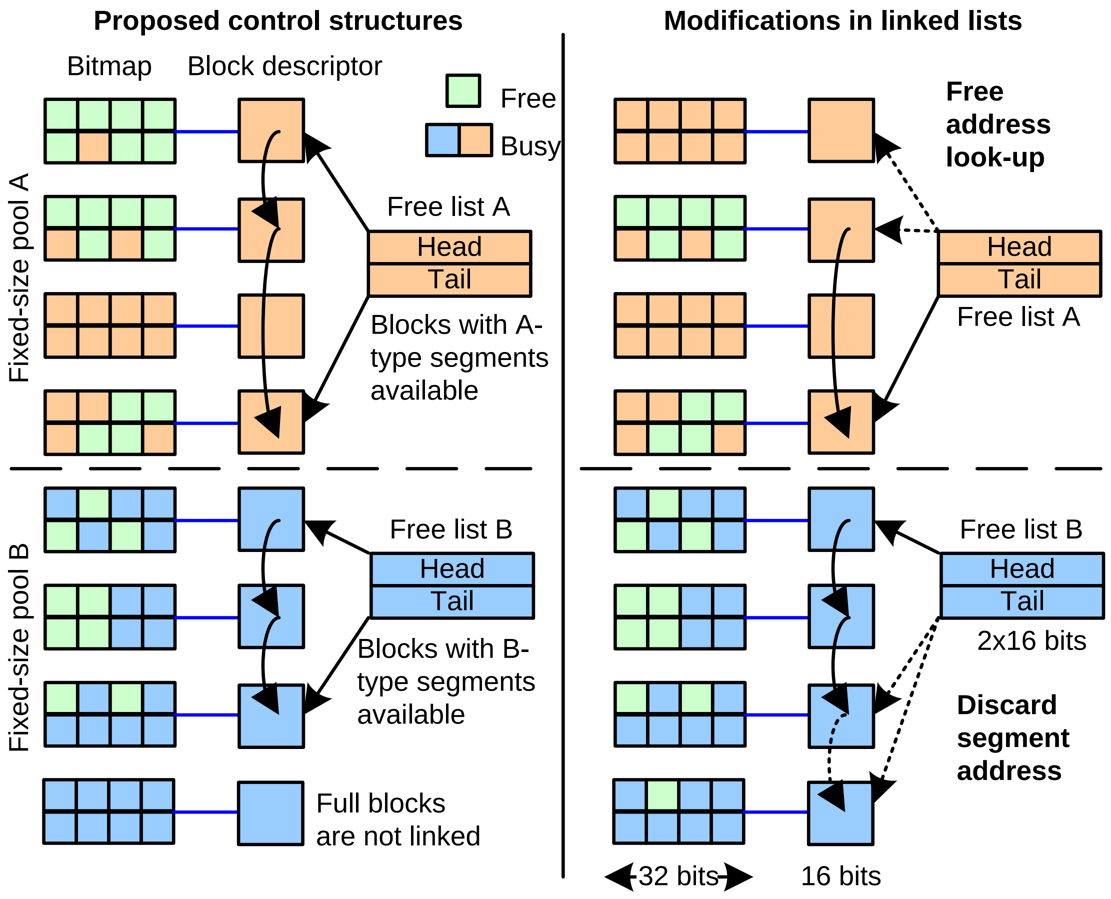


Figure 61: Control structures based on bitmaps and blocks with multiple pools

The control size requirement is quite small and depends on the smallest packet size, in our case 64 Bytes. For a standard buffer size of 256 MB with 64-Byte granularity, the bitmap needs less than 100 KB of memory while the block descriptors call for 50 KB more if using 2 kB-blocks. As a reference, the Control Buffer in state-of-the-art NPs using 64-bit segment descriptors and 64-Byte segments requires 32 MB, 218 times more than our method. Moreover, the number of accesses to the Control Buffer is again lower, six per packet in the seldom worse case when a new block need to be allocated but much lower in average.

A further adaptation of these structures is mandatory for allowing reallocation of memory blocks to different segment pools (see, Figure 62 left). As in the static case, each segment has a status bit indicating whether it is in use or not. Free address lookup and address discard operations are similar as well. The difference is that all blocks may theoretically be assigned to the minimum segment-size pool (64 Bytes), so the bitmap must be dimensioned to support such a contingency. Several bits are unused in those blocks with segments larger than 64 Bytes, which cannot be avoided as theoretically all blocks can be assigned to the smallest size.

Another difference with the static management is the double-linked list necessary to avoid an arbitrary number of accesses to extract an element from any free list. (Figure 62 on the right side) depicts a situation where double pointing is mandatory: in order to return block 3 to the general list and maintain the free list where this block was before, it is necessary to know which the previous element in free list B is, as block 2 must now point at block 4. Afterwards, block 3 has to be appended to the general list, so both block 0 and the lists' tail address are redirected to 3.

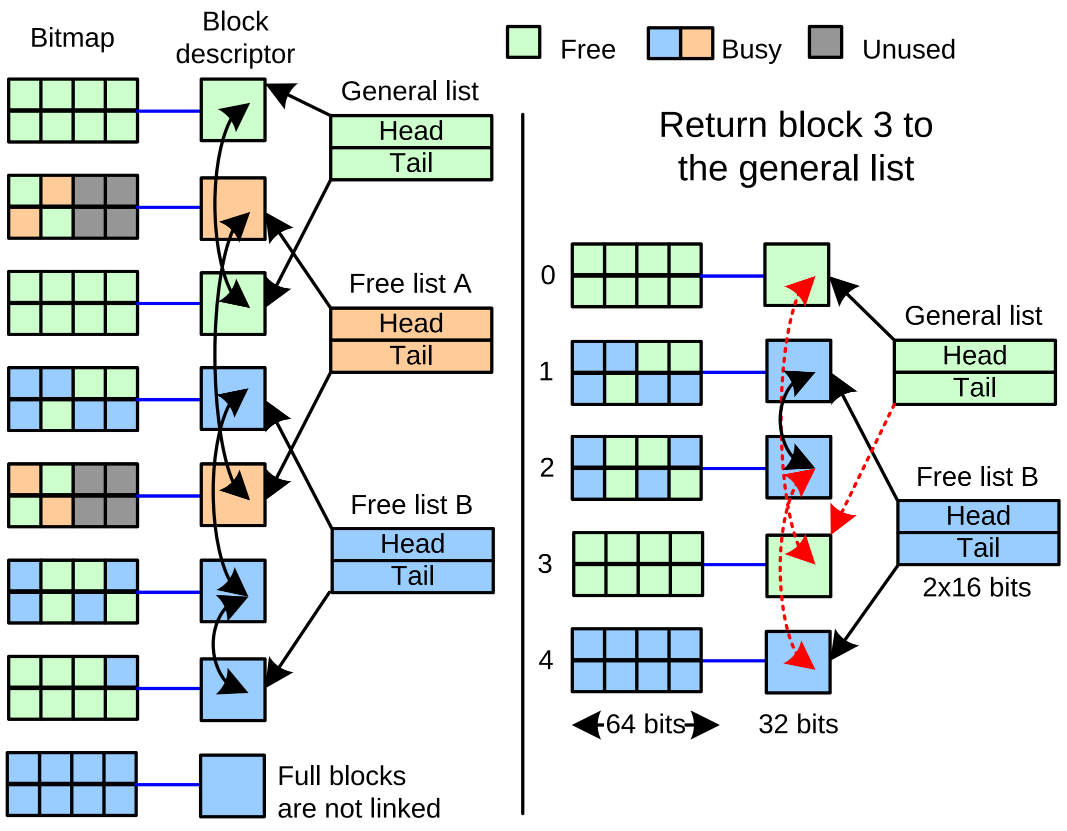


Figure 62: Bitmap-based structures supporting dynamic resizing of segment pools

However complex this may seem, the observed average number of accesses to the Control Buffer per packet was lower than with state-of-the-art linked lists. Little updating takes place as most control operations only require flipping one status bit. This devises a strategy where the bitmap is stored in on-chip SRAM while the linked list of blocks can be optionally mapped on external memory, even DRAM, to in this way further reducing SRAM requirements by taking advantage of the little BW needed for control updating.

With the SLA algorithm, four segment sizes are allowed: 64, 128, 512 and 1024 bytes. In accordance, the lowest constraint for the block size is 1Kb while the maximum value could theoretically be the total Packet Buffer size divided by four. Any size in-between is thus in principle possible, though we shall consider that using very-large sizes reduces a good deal of flexibility. On the other hand, small blocks means also a huge number of blocks and correspondingly, more control structures (and updating effort). In fact, the block size is a trade-off between pool resizing flexibility and Control Buffer requirements, in terms of memory size, consumption and BW.

When trying to find a sweet spot on the block size, it is mandatory to evaluate the total control memory requirements. For efficiency reasons in digital systems, the descriptor size can only be allowed to take values that are powers of two. Table 6 shows the control memory requirements when using four different block sizes: 1, 2, 4 and 8 Kb. Notice that the total bitmap size does not depend on the block size selected as it is a directly calculated from the Packet Buffer size (most left column). Opposite, the length of the block descriptor is the base-2 logarithm of the number of segments and presents those bit increments consequence of the round-up effect that produces a jump in the linked list size.

Larger Packet Buffers require 5 Bytes for each descriptor hence 4-Kb blocks are the right choice. The alternative of using 16-Kb blocks or larger must be counter-balanced against the increasing probability that blocks containing the smallest segments, may not be easily reassigned. A situation like that occurs whenever a long-lived object blocks a memory block therefore making difficult the resizing of segment pools when the buffer is nearly full.

Pkt. Buffer	Num. blocks	Bitmap size (KB)	Bytes for LL	LL size (KB)	Total (KB)	BRAMs
Block size: 1 KBytes						
4 MB	4096	8	3	12	20	10
64 MB	65536	128	4	256	384	192
256 MB	262144	512	5	1280	1792	896
1 GB	1048576	2048	5	5120	7168	3584
4 GB	4194304	8192	6	24576	32768	16384
Block size: 2 KBytes						
4 MB	2048	8	3	6	14	7
64 MB	32768	128	4	128	256	128
256 MB	131072	512	5	640	1152	576
1 GB	524288	2048	5	2560	4608	2304
4 GB	2097152	8192	6	12288	20480	10240
Block size: 4 KBytes						
4 MB	1024	8	3	3	11	5,5
64 MB	16384	128	4	64	192	96
256 MB	65536	512	4	256	768	384
1 GB	262144	2048	5	1280	3328	1664
4 GB	1048576	8192	5	5120	13312	6656
Block size: 8 KBytes						
4 MB	512	8	3	1,5	9,5	4,75
64 MB	8192	128	4	32	160	80
256 MB	32768	512	4	128	640	320
1 GB	131072	2048	5	640	2688	1344
4 GB	524288	8192	5	2560	10752	5376

Table 6: Control Buffer size with different block sizes

The Control Buffer necessary for a 256 MB-Packet Buffer is 256 KB for the block descriptors together with 512 KB for the bitmap, summing up a total of 768 KB. Figure 63 puts these values into perspective comparing them with the requirement of state-of-the-art linked lists, for two typical segment descriptors sizes (32 and 64 bits). The reduction is significant, especially after noticing the logarithmical scale on the bar diagram, whatever the Packet Buffer size even when the Bitmaps are compared to a linked list system using the optimistic 4-byte descriptor. A Control Buffer of hundreds of KBs, comfortably allows for an SRAM implementation. In order to grasp the importance of reducing the amount of SRAM in high-end NPs, we may need to get back to 2008 when Netronome overtook Intel's IXP franchise. On their very first press release, they announced the replacement of the QDR-SRAM memories of the IXP-2800 by slower DDR ones, in an effort to curve chip costs.

To summarize the results here presented, the bitmap-based control structures lead to a huge reduction of the Control Buffer size without increasing the BW requirement. The introduction of the Bitmaps is only feasible in combination with packet segmentation that uses multiple segment sizes, in particular the SLA-M2-N4 as it enforces a hard bound of two segments per packet. The improvement of the Control Buffer comes without lessening the other benefits of the new packet segmentation, including fewer control accesses and less segments to handle.

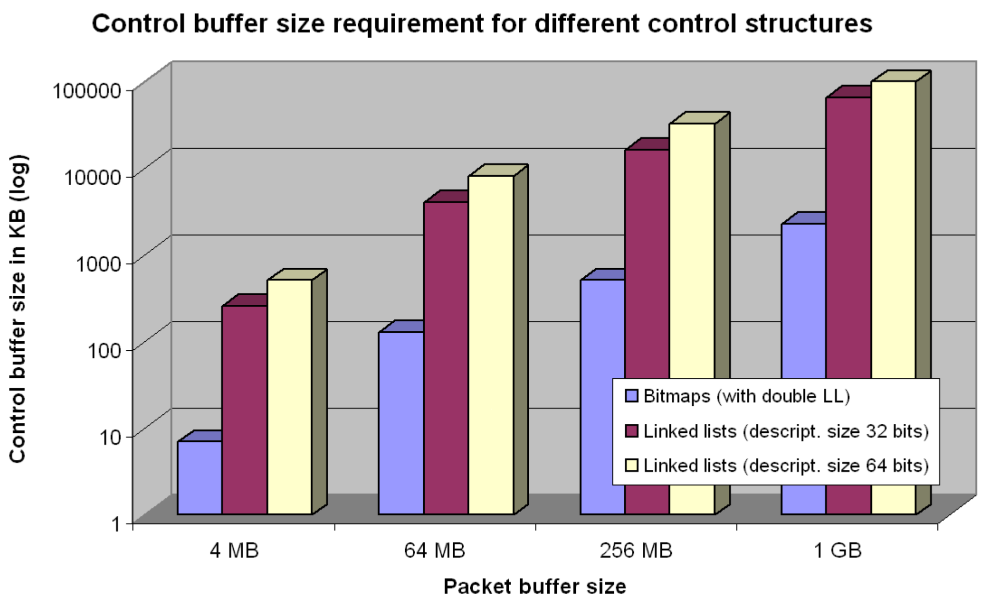


Figure 63: Control Buffer requirements. Comparison between state-of-the-art linked list and the proposed bitmap method. Notice the logarithmical scales used in the graphic.

4.5 Conclusions

This chapter examined the allocation of multiple segment sizes in the Packet Buffer of Network Processor Units. The starting point was the previously described SLA algorithm that slices variable-length packets using four different segment sizes. Although we observed a high similarity in the average segment-pool size required by most traffic patterns, that is not always the case. Therefore, static pool sizing does not meet high buffer efficiency. Regardless of the initial memory allocation, different pools run out of space alternatively.

A system-level analysis confirmed that in a typical router forwarding internet traffic, a static pool resizing management does not suffice to administrate multiple segment pools efficiently. That is also the case in access routers processing simultaneously up and downstream user traffic, where remarkable segment requirement variations have been observed. In consequence, only a strategy where segment pools can be dynamically resized is deemed appropriate for the administration of the new Packet Buffer with multiple segment sizes.

The chapter investigates the best method to organize the memory while enabling dynamic segment pool resizing. Three methods were investigated, segmentation-and-coalescence, Gravity Centers and a Block-based organization. The first two were discarded due to problems with memory fragmentation as well as non-deterministic time during the free memory look-up.

The chapter ended introducing new control structures suitable for managing a Packet Buffer with multiple segment pools, allowing for dynamic segment pool resizing upon traffic requirement. In addition to this new feature, the proposed structures largely reduce the Control Buffer size requirement in comparison to state-of-the-art management techniques based on linked lists. Moreover, both in average as well as in the worse case, they require fewer accesses to the control memory.

Here concludes the study of multiple segment sizes for packet segmentation. But using smart packet segmentation enables further enhancements for the NP architecture. Packets can be segmented using other criteria than storing efficiency or throughput maximization. For example, a small segment can be used to keep the packet’s header while a large one can contain the payload. Interestingly, two separate buffers could contain header and payload data. The characteristic of these two buffers open new promising lines of investigation to further improve the architecture of the memory sub-system in NPs.

5 The Packet Processor’s Local Buffer

The Memory Subsystem is a major factor limiting the performance of Network Processors (NPs). Memory latency is usually hidden by HW multithreaded processing engines or by caches if standard RISC cores are used. Traditional data caches in NPs, however, have only limited effect due to low locality of packet processing applications. Therefore, this chapter proposes and discusses alternatives for **injecting packet headers** preemptively into the core's local memory (or cache) instead of waiting until a new packet has to be fetched from the Packet Buffer into the CPU local memory for processing, and so avoid the additional delay associated with this method.

Current NPs experience two main system bottlenecks: first, the short processing time available in case of maximum packet rate when the NP deals with a continuous stream of 64-Byte packets. At 100 Gbps, each CPU in a core cluster of 64 PPEs must forward one packet every 328 ns, which strongly constraints the processing effort per packet. A second challenge is the lack of memory bandwidth (BW) when interfacing the Packet Buffer. When servicing a line rate at R Gbps, the system must be able to store and retrieve packets at least once to/from the buffer hence the required BW from the memory device is at least *2R*.

Moreover, a portion or the complete packet is retrieved during processing and then written back, which shifts the final memory interface's requirement to somewhere between *2R* and *4R*. A final effect that pushes up the BW requirement is the so-called 65-Byte effect. In digital systems, packet lengths not rounded-up to the next power of two generate overloads when transferred over wide data buses. In the case of a DRAM-based buffer organized in 64-Byte segments, the second segment for a 65-Byte packet must be padded to fulfill the access granularity in DRAM. In the event that a continuous stream of 65-Byte packets arrives requiring payload processing, load may go up to *6R* on the memory interface if the access granularity is 32 Bytes or even *8R* if the minimum access is 64-Byte long.

Besides high-speed packet forwarding and deep packet processing, NPs are also required to perform complex Quality-of-Service (QoS) and link load balancing functions. Both concepts required of complex queue management, which requires of high processing effort from the CPU. To sum up, lack of memory bandwidth and scarce processing resources are the main bottlenecks in a NP, as we have seen in previous chapters. The introduction of the Packet Processor’s Local Buffer addresses simultaneously these two problems.

The PPLB is an intelligent unit that conveys the incoming packets to a CPU’s local memory (or cache). The packet destination is determined by a packet classification unit that considers the load status of the multiple cores and the protocol processing requirements of the incoming packet. Therefore, it is possible to know in advance with a high degree of certainty which part of the packet will be later needed in the CPU.

The required packet data is thus made available in advance so when packet processing starts, the processor fetches data from the local buffer without any additional time penalty. This approach also significantly reduces the load on the memory interface and especially avoids short (and hence inefficient) accesses to DRAM from the processor cores.

For the new scheme to be feasible, some assumptions need to be made regarding the data flow in the NP. First, the CPU queues need to be small. This is required to enable an on-chip realization of the Processor Buffer and so accelerate data access from the CPU. If we pay attention to the systems depicted in Figure 64, where packets are stored immediately after arrival and Figure 65, where the main buffering is done after packet processing, we may appreciate better understand the reasoning behind this requirement.

In the first case, packet data is stored in the Packet Buffer after its arrival and then a small Packet Descriptor (PD) is pipelined through multiple stages, thus requiring only small control buffering to keep the packet tokens. On the contrary, in the proposed architecture a small on-chip CPU Buffer contains the portion of the packet required for processing, which accelerates CPU's packet data accessing. It is thus of the highest importance the minimization of queuing taking place before the CPU cluster to constrain inasmuch as possible the SRAM requirement. In our simulations, small input queuing is taking place as long as the CPUs can cope with the incoming traffic. Otherwise, the input buffer overflows regardless of the memory provided. A final remark, the scheme proposed allows for compatibility with the previous scheme as packets can be dynamically selected to skip the PPLB altogether. To demonstrate the feasibility of this concept, this chapter introduces the Packet Processor's Local Buffer in the framework of the SmartMem project.

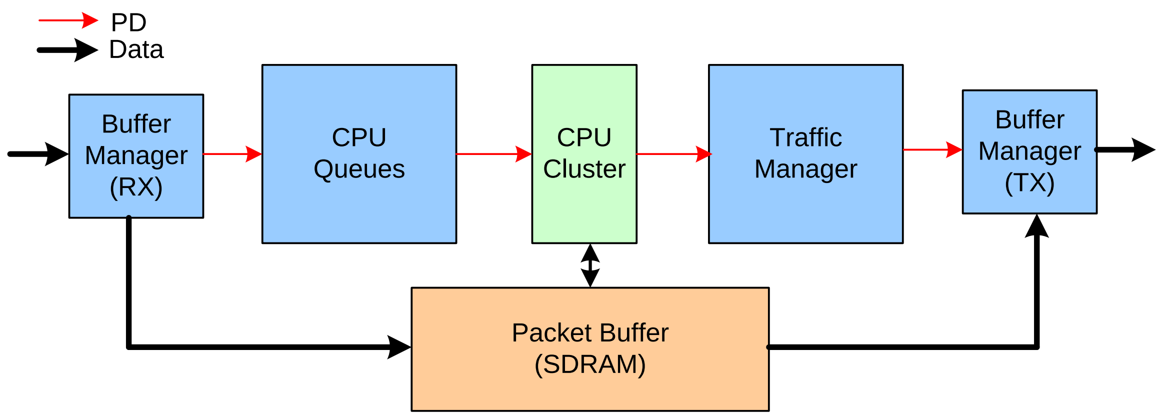


Figure 64: Packet buffering immediately after packet arrival

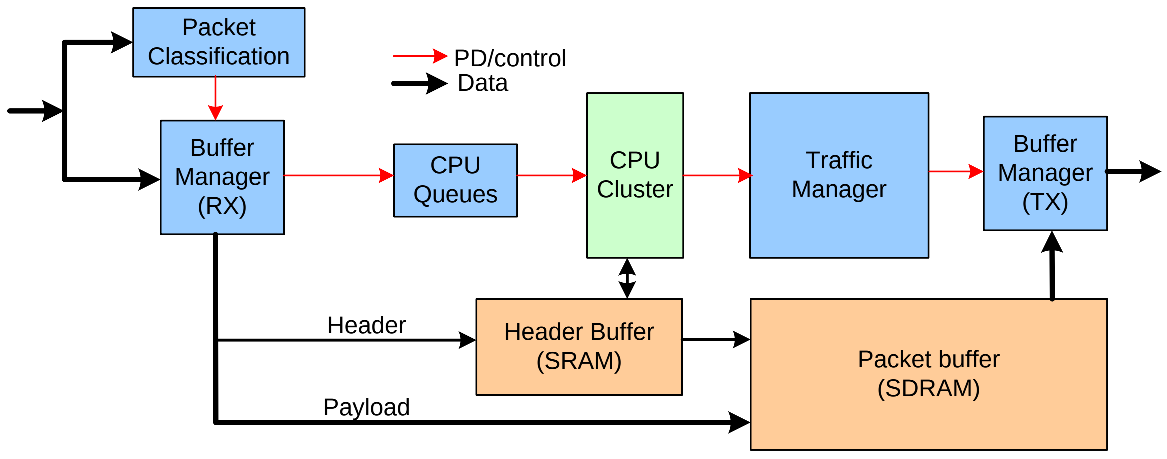


Figure 65: Processor Buffer Packet Buffer

5.1 Packet injection in the local buffer of the CPU

In most commercial NPs, once a packet arrives it is stored immediately in the main Packet Buffer, an operation generally executed by a DMA engine under the supervision of a CPU (Figure 66 left). In some cases, a dedicated hardware accelerator as the Buffer Manager performs storing and packet segmentation autonomously together with allocating the appropriate memory space. Once storing is accomplished, the CPU is interrupted, which triggers the packet processing application. The CPU receives a packet descriptor containing pointers to the data location in the Packet Buffer. Data is then accessed, and after long latency, modified and written-back in the Packet Buffer. The CPU passes then the token to the Output Queue Manager, which schedules packets and indicates the BM to transmit them in the proper order (Figure 66, centre).

In the proposed architecture (Figure 66, right), packets are pushed into a processor local buffer autonomously of the CPU. There the CPU accesses packet data and stores the results of its processing, making the packet ready for transmission. The processed packed is then transferred to the larger Packet Buffer where output queuing is implemented.

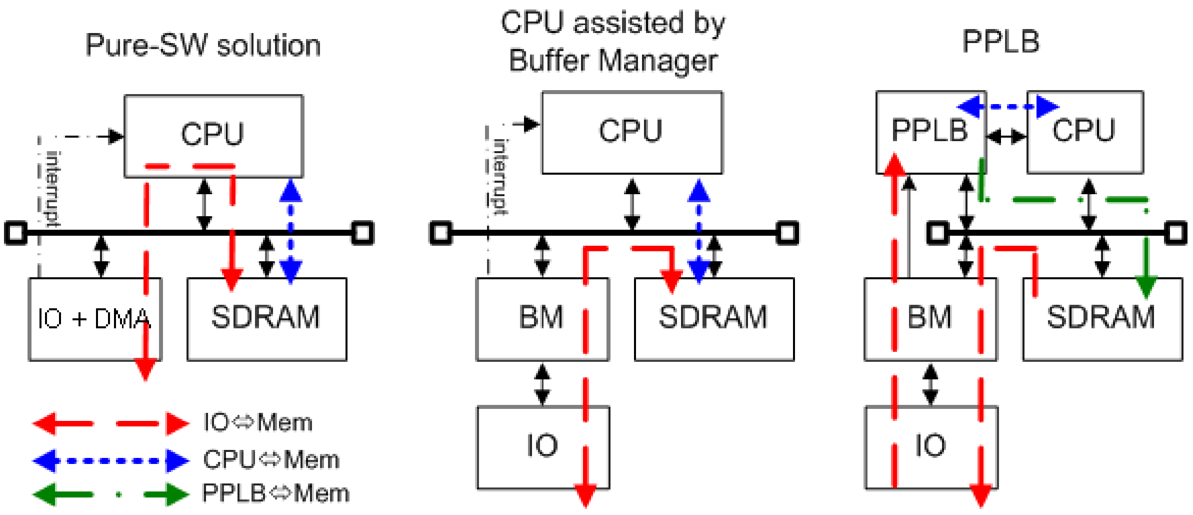


Figure 66: Data flow comparison between three Memory Subsystem architectures

Having a processor's local memory like the PPLB lessens the burden on the Packet Buffer because the CPU-cluster does not generate short (and so inefficient) accesses to this high-latency buffer hence liberating bandwidth for the Buffer Manager operation (see comparison between Figure 67 and Figure 68. Now SDRAM access pattern is more efficient while the overall memory throughput required is substantially lower, especially when handling short packets. Now, only processed packets are sent to the Packet Buffer.

This opens the door to the introduction of multiple new mechanisms to accelerate DRAM accessing or at least to optimize existing ones. This chapter discusses which these mechanisms are, and why the PPLB can indirectly improve the memory throughput. Several academic works have proposed multiple methods to overcome the shortage of DRAM throughput with different degrees of success, as discussed in the prior-art section. Among them, per-flow buffering has emerged as a major contending approach that, beyond its large SRAM-requirement problem, offers promising features. It relies, though, on the fact that packets must be already processed before they can be assigned to a queue in the Packet Buffer. This applies for the PPLB and thus enables the introduction of per-flow-administered Packet Buffer in high-performance NPs.

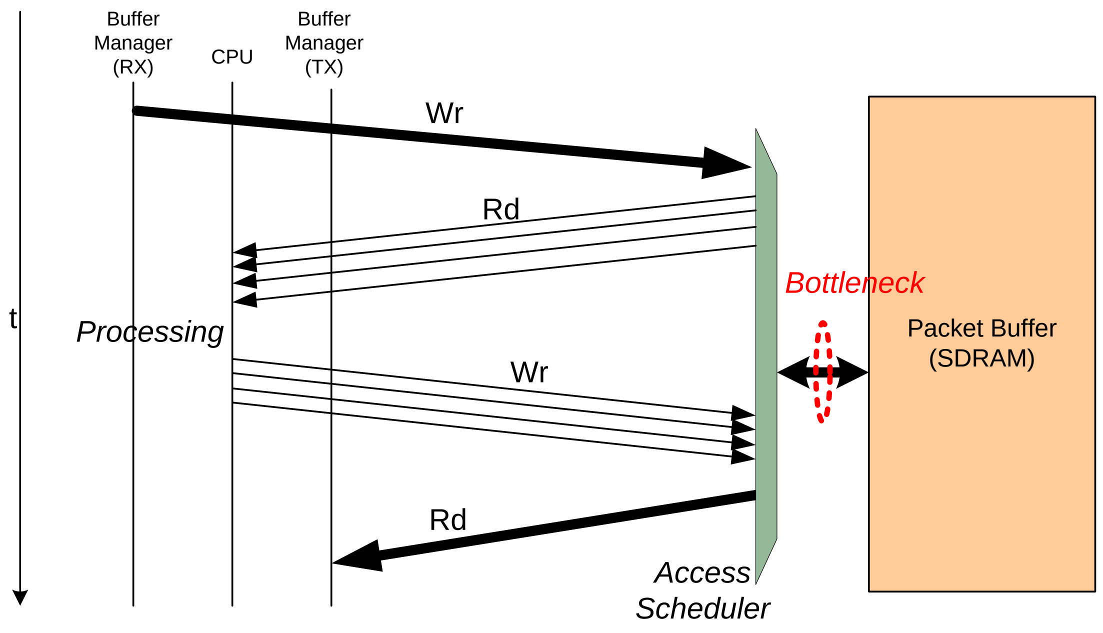


Figure 67: access requests to the Packet Buffer in a state-of-the-art architecture

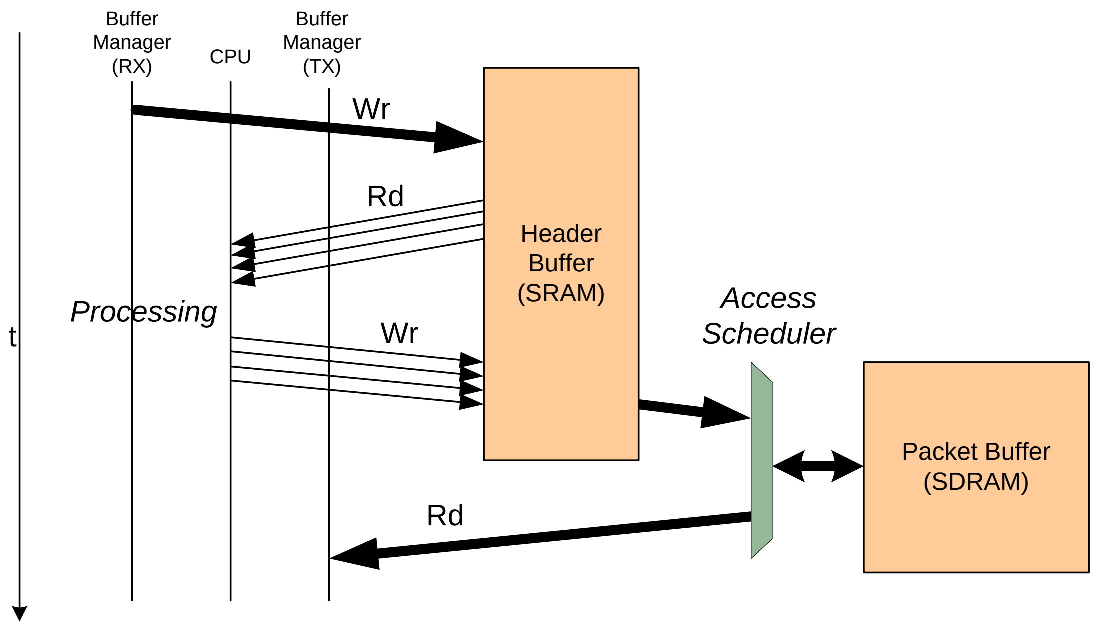


Figure 68: Access pattern to memory with the PPLB, notice the simplification in the Packet Buffer interface

One feature is how to proceed when storing the packet (Figure 69). The Buffer Manager in the RX side receives a packet and according to some classification results can store the packet header and the payload separately in memory, the full packet only in the Packet Buffer or the full packet in the PPLB. It can even make a copy of the packet header in the PPLB and still store the full packet in the Packet Buffer. Although the Buffer Manager can be capable of working the packet in all these modes, it is preferable if the full packet is first stored in the PPLB and only after processing stored in DRAM. This presents several advantages, mainly an increase of access efficiency to the main Packet Buffer produced by longer memory accesses, as header and payload are stored in a single access to the DRAM-based buffer.

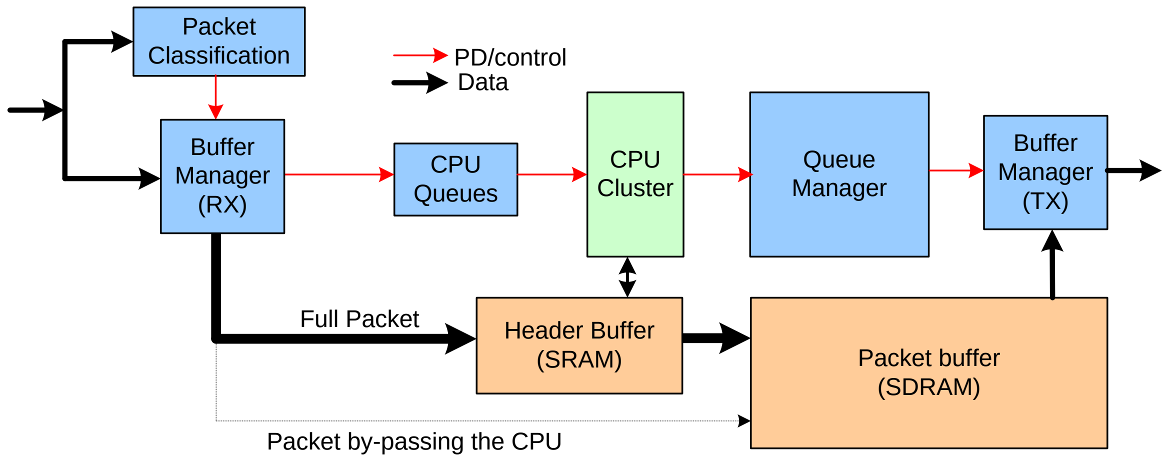


Figure 69: Full packets first stored in the Header Buffer and once processed in the Packet Buffer

A very important advantage in case of packet resizing is avoiding memory reallocation, which may severely impact the performance of the Packet Buffer. Another benefit consists in storing the processed packet in the right output queue during packet processing, for example after detecting the flow the packet belongs to, it can be consecutively stored in the same memory bank. As a drawback, some extra delay is introduced when transferring the packet first to the PPLB and then to the Packet Buffer. In the worst case for a 1500 Byte-packet, 940 ns (bus: 128b@100MHz). All other packet storing options either generate an inconvenient access patter or overload the interface with the Packet Buffer. In conclusion, it is recommended not to store any part of the packet in DRAM before processing is done.

Several academic works have proposed multiple methods to overcome the shortage of memory throughput with different degrees of success. The PPLB introduces a new range of options because of the more deterministic access pattern to the Packet Buffer obtained when the CPU does need to access it. In section 5.4, we will re-evaluate some of these mechanisms reasoning about the convenience of their implementation in combination with the PPLB’s.

To sum up the benefits of a system with PPLB, it accelerates packet processing while avoiding an excessive on-chip memory requirement. Moreover it allows for less complex microengines due to the fact that multithreading is no longer needed as the pipeline never stalls waiting for memory transactions to be completed. This leads to a reduction of chip area and of heat dissipation as the CPU data-cache can be reduced or utterly removed.

5.2 Internal data flow

This section proposes a system architecture that accommodates a PPLB. Four main modules are required: the PPLB itself, the Buffer Manager, the Packet Buffer and the CPU cluster Figure 70. Let us see in detail how data flows internally:

1. After arrival, the packet is stored locally in the BM, where the appropriate segmentation is calculated. In parallel, a classification engine recognizes the packet protocol and indicates to the BM which type of traffic the packet belongs to. The Control Plane can setup rules to assign each type of traffic to different CPUs. The CPU queue scheduler must ensure that high priority packets are processed with the lowest delay while simultaneously best-effort traffic does not starve in the queues.

2. The packet is then forwarded partially (only the header) or completely to the corresponding CPU queue. There are two queuing steps, in the first step the header and its PD are queued according to the type of traffic, for example following a high and low priority criterion, but not yet assigned to a particular CPU (shared queuing). Payloads can be directly stored in the Packet Buffer thus bypassing the PPLB.

3. A CPU-load balancer or the CPUs themselves indicate to the PPLB which queues will be serviced by which CPU or group of CPUs. For instances, one CPU can be exclusively dedicated to time-critical traffic while the rest are doing best-effort processing. Whenever a slot in a CPU’s local buffer is detected, the PPLB injects the next packet header in the assigned queue into the corresponding CPU’s local buffer that will process this packet. This mechanism is explained in greater detail later in Figure 71.

4. Once processing is fully accomplished, the CPU commands the PPLB to flush the packet header to the main Packet Buffer. Data is stored in a segment previously-allocated by the BM and in accordance with the SLA-M2-N4 packet segmentation algorithm. An upgrade of the method consist in performing packet segmentation after packet processing to in that way consider possible packet resizing, as in the case of a packet entering/leaving an MPLS network (Label must be attached/swapped) or ingoing/outgoing into/from an IPSec tunnel. Packet segmentation after processing has not been included in the architecture here introduced, but it is a promising pointer for further investigation.

5. Once the processed packet is in the Packet Buffer, the PD is sent to the Queue Manager, where QoS takes place. In that way, the Packet Buffer contains the bulk of memory space for traffic policing and port contention buffering, two functions that become thus decoupled from the CPU-buffer's hence enabling for a small-sized PPLB implementation.

Finally, the transmission side of the BM fetches the packets from the Packet Buffer, reassemblies it and transmits data to the corresponding output port, always in accordance to the packet order designated by the Queue Manager.

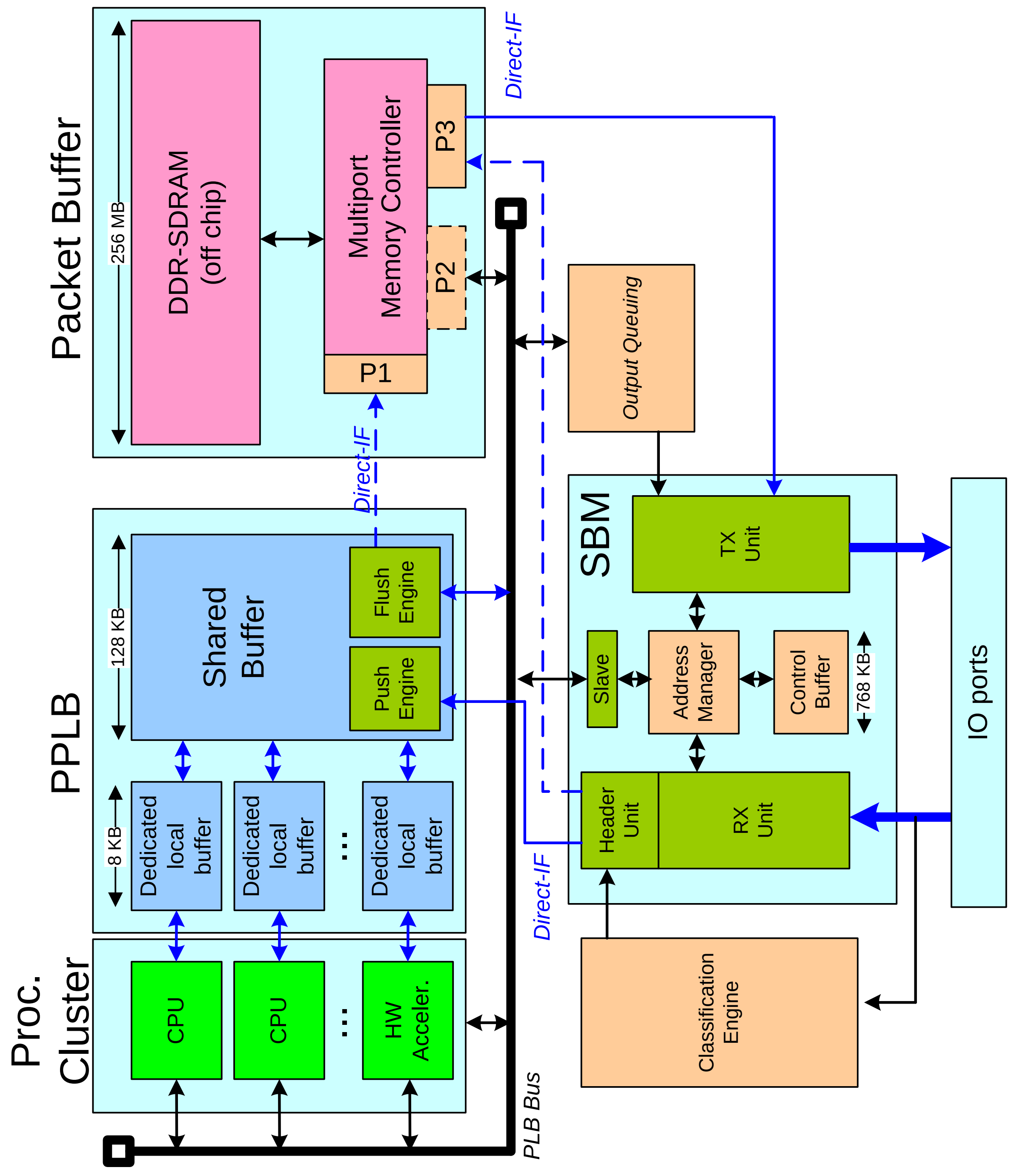


Figure 70: Memory Subsystem architecture with the Packet Processor’s Local Buffer

5.3 Packet Processor's Local Buffer architecture

5.3.1 Functionality

The PPLB is a small fast buffer where a limited number of packets are kept during processing time. A crucial condition for a successful PPLB implementation is scalability with the number of cores in the system. When several devices interface with a single memory, the communication infrastructure must provide a mechanism to arbitrate conflicts and guaranty fair access to the resource, let alone provide enough BW. Although a system bus or an interconnection matrix meet the access requirement, they do not properly scale as either access latency grows exponentially in the first case or the amount of resources in the matrix increases, again exponentially, with every new element to be connected. In addition to scalability, three additional difficulties are confronted during the PPLB designing:

- In advance it is unknown which CPU will process a particular packet as multiple cores may access the same queue to achieve better load balancing.

- Packets have different priorities and arrive at variable intervals, so some input queuing is needed for prioritize traffic and feed the processor without interruptions.

- A critical question is the dimension of these queues and whether dynamic management can be avoided. The length of the CPU queues is thoroughly studied in 5.4.

- If more than one CPU or HW-accelerator is required for the processing of a single packet the buffer must provide the means for inter-CPU communication.

In all NPs, there is a queue of processing requests before the CPUs, sometimes referred to as input queues. The main goal is to achieve optimal load balancing among several cores while simultaneously reserving some capacity to process high priority traffic. A single best-effort queue for all CPUs would achieve optimal load while a set of input queues dedicated per CPU would ensure low latency for high priority packets albeit at the price of unbalancing some cores. Input queuing must carefully avoid head-of-line blocking, which occurs when a slow packet early arrived blocks the stream of subsequent incoming packets.

The proposed solution consists of including a mix of CPU-dedicated queues with higher priority together with a best-effort queue containing low-priority packets from which all CPUs may be feed once their dedicated queues are emptied or after a service quota has been met. The best-effort queue thus helps to better balance the processing effort among the cores as traffic from different flows can be assigned to multiple CPUs [24]. This queue is sometimes called spray, as it services several CPUs. An additional queue for less-than-best effort packets (Scavenger class) can be applied. Whenever this queue overflows, Scavenger packets can be easily discarded without processing and memory de-allocation.

One possible approach to allow all CPUs accessing all queues consists in implementing a cross-connection scheme capable of connecting all CPUs with all queues in the PPLB. However, all-to-all connectivity cannot scale when the number of CPUs or of queues in the system increases. Too many resources would be required only for the intercommunication infrastructure.

To solve the problem, a two-level memory scheme for the PPLB is proposed (see
Figure 71). The first level consists of multiple small buffers one for each CPU, where four packets can be simultaneously stored: one under processing, two waiting its turn and another been flushed to the main buffer. CPUs access their dedicated buffer through a direct channel, in our case a 32-bit On-Chip Memory interface. Opposite, the second level is a shared buffer composed of several small FIFO queues, with a configurable number of queues that can be flexibly assigned to one or more CPUs.

- ➢ An incoming packet is conveyed to the appropriate input queue, according to the processing priority established by the Classification Engine in accordance with the setup of the Control Plane. Or still simpler, one queue can be assigned per CPU. Packets in the fast path can completely by-pass the PPLB.

- ➢ Each CPU sets its own priority for getting packets from multiple shared queues. In accordance to that configuration, the Control Unit feeds the local memories scheduling first high priority queues and then the rest. This is done until the CPU's buffer is full or the selected queues run empty.

- ➢ During processing, the CPU accesses its local buffer at the processor's frequency without neither additional latency nor cache missing.

- ➢ In some cases the processing of a single packet may be split among different CPUs (pipelined processing or HW accelerator-aided processing). In that eventuality, packets may be looped back to the corresponding Level 2 queue, waiting for the another CPU/accelerator.

- ➢ After processing is done, packets are autonomously flushed to the Packet Buffer either through the system bus or using a direct interface to the main packet memory. The PD is transferred to the Queue Manager.

The number of FIFO's and the scheduling policy are configurable individually for each CPU, which can access a control register that sets its priority for receiving packets from every queue. One push and one flush/forward operation can be executed simultaneously, provided they do not have as destination/source the same local buffer. When such a conflict occurs, the control unit reschedules the command queue thus preventing performance loss.

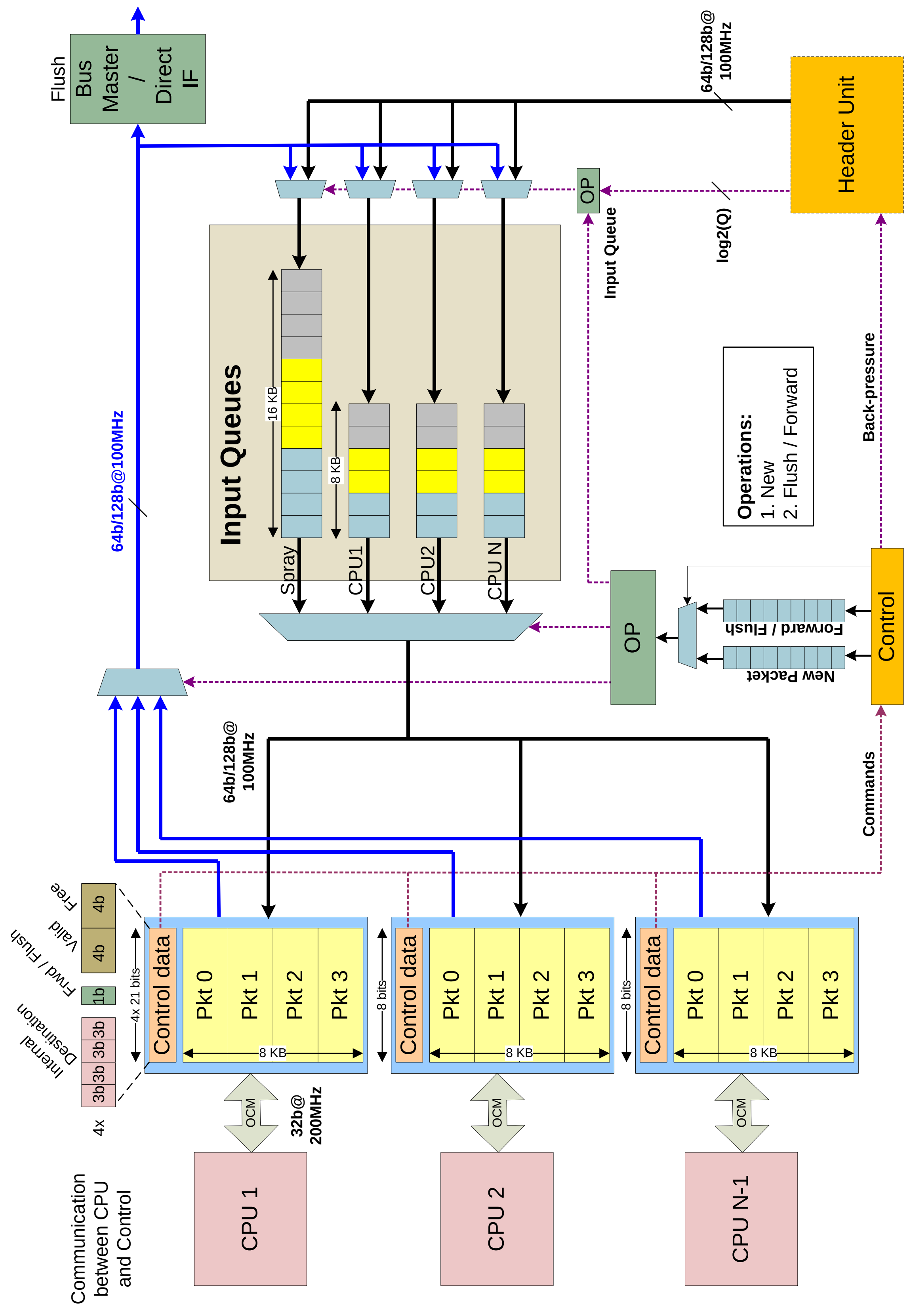


Figure 71: Block diagram of the Packet Processor Local Buffer

5.3.2 Buffer Manager Unit

The PPLB cannot work standalone without a specialized Buffer Manager capable of separating the packet header from the payload and then transferring data separately to the PPLB and the Packet Buffer. This operation is not identical for all packet types. The BM operation must be adapted to process the information from the classification engine. Dynamically modifying how the system reacts to a particular protocol can be necessary, so the Control Plane must be able to modify the instructions of the BM for each type of traffic.

The CPU requires only the IP header for straight-forward IPv4 forwarding whereas for encrypted packets, the full packet data is necessary in the processing core. In the first case, the BM extracts a fixed number of bytes from the first part of the packet and stores these data in the PPLB, while the payload is directly transferred to the Packet Buffer. In the second case, the full packet must be stored in the PPLB whereas no payload goes to the Packet Buffer. The architecture details are further discussed in the next chapter. This specific functionality is implemented in a new unit inside the Buffer Manager, the Header Unit. To better understand how the Header Unit works, it has been considered more appropriate again to include its description within chapter 6, which tackles the complete implementation discussion.

5.3.3 Classification Engine

The Classification Engine identifies the packet protocol extracting certain fields from the layer three and four. Typically this includes source and destination IP addresses, source and destination ports and Layer-4 protocol type (the five-tuple). This produces a unique identification per traffic flow, which is used to group traffic into similar classes according to their processing requirements.

For our design, we rely on a the FlexPath classification engine [85] that extracts the IP-relevant fields and assigns different paths according to a programmable set of rules dynamically configurable by the control plane. For example, packets carrying ICMP or packets belonging to route protocols (recognizable for their specific TCP field) can be directly send to the control plane while IPSec traffic can be conveyed to a Crypto-core to be decrypted before processing.

As example, a NP supporting QoS with DiffServ has a classification unit that assigns higher processing priorities according to the DSCP field. Traffic marked as Expedite Forwarding (EF) receives highest priority for processing. This includes critical routing information or even voice. Video conferencing and other time-sensitive traffic would receive the second best classification. Finally, most traffic like video streaming, HTTP, POP3/SMTP, etc requires simple IP forwarding so this traffic is conveyed to the best-effort queue. Less than best effort traffic (peer-to-peer for instances) is usually assigned with a constrained bandwidth and needs to be discarded even before consuming processing resources.

5.4 Memory size requirement of the CPU queues

In order to better evaluate the level of queuing taking place in front of the CPUs, this has been modeled at transaction level using SystemC. The amount of memory is critical because the PPLB can only be implemented on on-chip SRAM, which is only feasible if queuing in front of the CPU is small enough. The PPLB buffer must have low access latency allowing the CPU to retrieve data as fast as possible, as this memory replaces data cache and so must deliver the same level of performance.

We observe the occupation of the shared queue (level 2) while servicing traffic rates between 1 and 10 Gbps. The stimulus is real traffic recorded on an OC-48 link, presenting a typical internet packet size distribution mix with variable arrival time. Over the inter-arrival time a factor is applied in order to regulate the line rate from 1 up to 10 Gbps and thus incrementally observe the average and peak load on the queues. The processing power of the cluster depends on the number of CPUs selected, in the experiments values of 2 and 3 cores are chosen for two independent tests. A second parameter tuned during the experiment is the time it takes to process a single packet. With the processing load it is possible to modify the effort of the CPU per packet. Finally, the size of the CPU local buffer (level 1 queue or dedicated queues) has been defined first as 4 KB and then as 8 KB, which is equivalent to say a maximum of 2 or 4 packets per local buffer (one packet slot is 2 KB). This is preferred to allow storing the full packet in the slots, rather than provisioning enough space for the header.

The observed maximum fill level of the CPU queues in all scenarios (Figure 72 and Figure 73) is low as long as the processing cluster keeps pace with the line rate. In these conditions, packets can be processed on-the-fly as virtually no queuing takes place in front of the processors as there is always enough processing capacity. When CPU capacity is clearly surpassed by the arrival rate, input queues simply quickly overflow independently of the memory resources available. But most interestingly, both L1 and L2 buffers experience a remarkably low level of maximum queuing when the CPUs work close to their maximum capacity.

The simulations were repeated again, this time configuring the cluster with a four-CPU core (Figure 74 and Figure 75). The effect of increasing the packet processing capacity is clearly observed in the lower queuing taking place in the input queues as well as in the fact that the system only saturates at high speed with long processing time. Again, implementing 4 KB of additional memory in the level 1 buffer (e.g. 8 KB per CPU) leads to a clear reduction in the queue levels of the shared queue. The overall amount of data stored in Level 2 is substantially lower than in the previous experiment as more CPU capacity leads to less CPU-queuing for the same traffic rate. This confirms that either the system is well-balanced and thus a small PPLB is required or the CPUs cannot deliver enough processing capacity hence saturating the input queues whatever the memory size available.

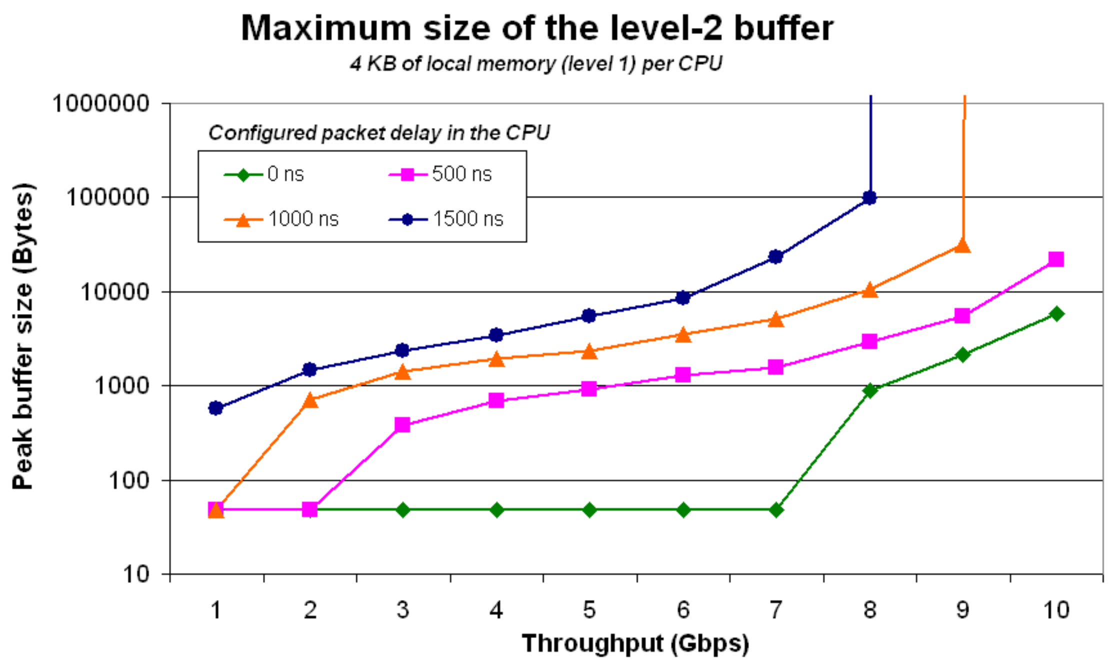


Figure 72: Maximum fill level with 2 CPUs – 4 Kb of Level-1 local buffer

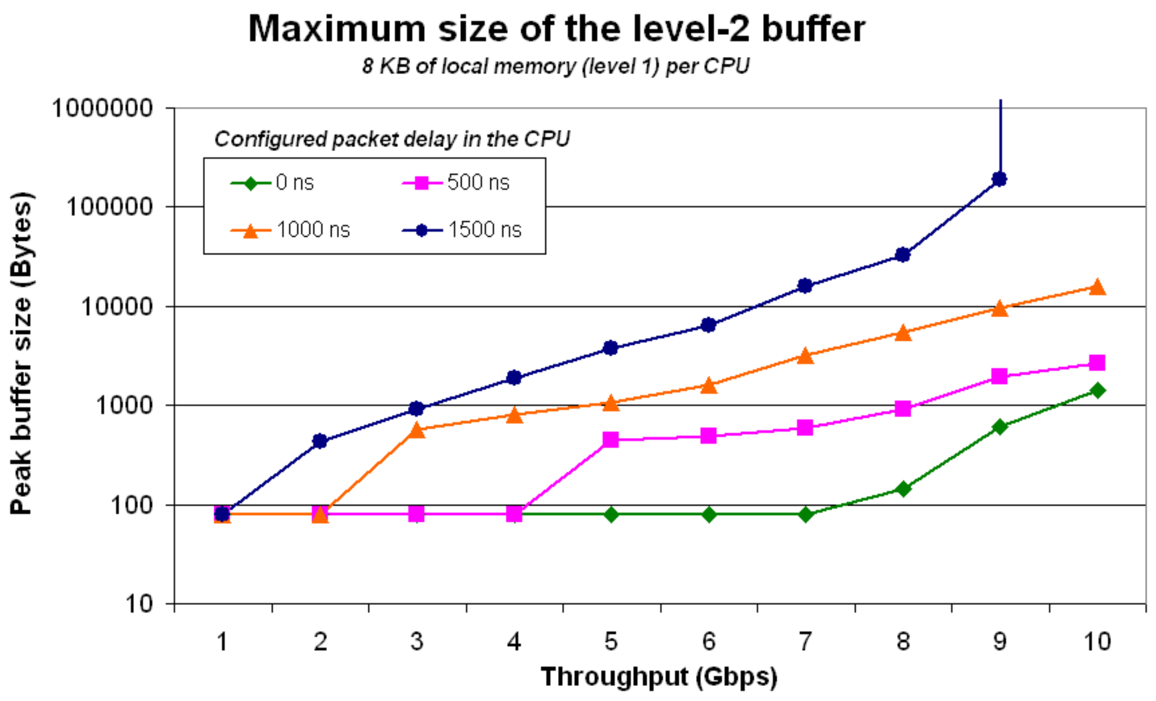


Figure 73: Maximum fill level with 2 CPUs – 8 Kb of Level-1 local buffer

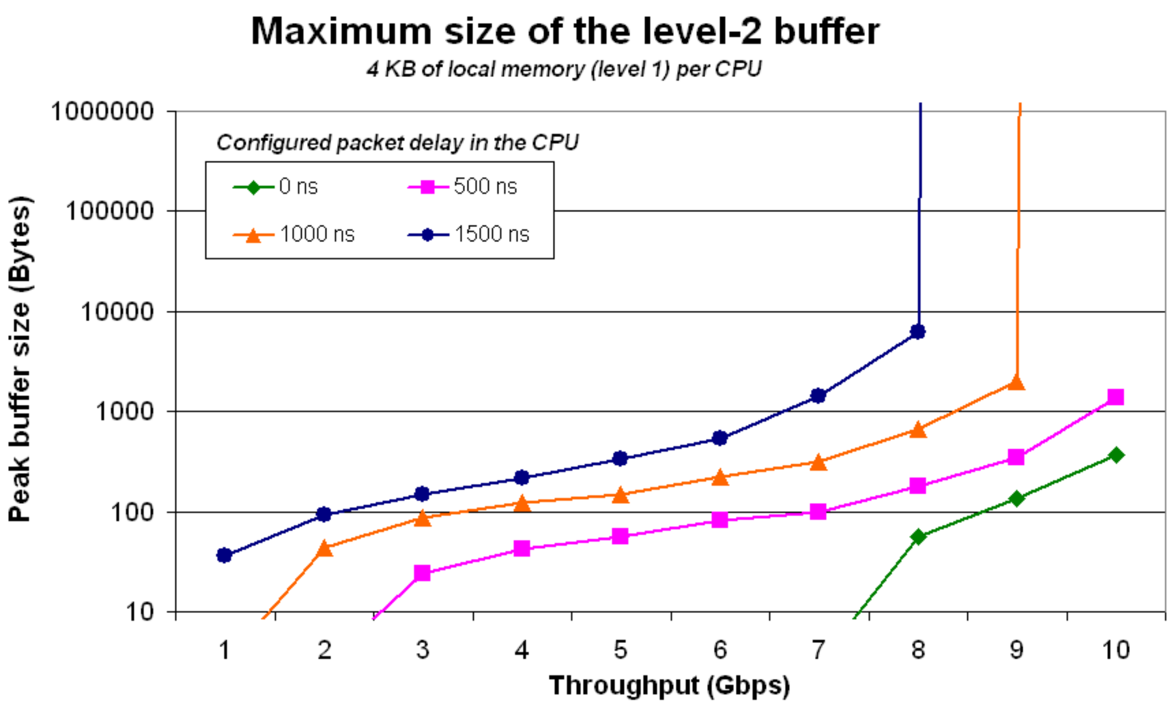


Figure 74: Maximum fill level with 4 CPUs – 4 Kb of Level-1 local buffer

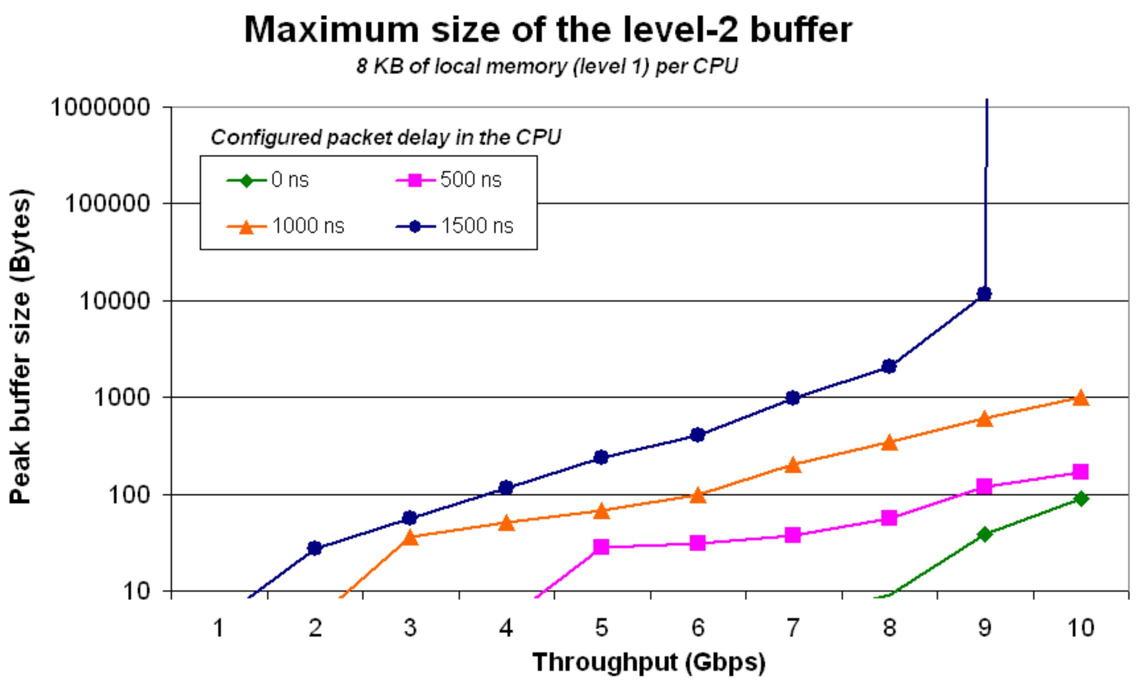


Figure 75: Maximum fill level with 4 CPUs – 8 Kb of Level-1 local buffer

In that situation, we observe that provisioning additional memory in the local buffer (Level 1) enables for smoother transferring of packets from L2 to L1, which reduces between 6% and 225% the memory requirement in L2 in all cases studied. However **enlarging L1 beyond four Packet Buffers does not bring any further peak reduction in L2**. This is because four packets suffice for an optimal operation of the CPU: one packet being processed, another waiting, a third being transferred from L2 and finally one being flushed to the output.

The PPLB model sheds light on the required dimensions of the L2 buffer as well. The maximum queue level observed in L2 increases with the line rate without surpassing a 128-KB requirement in any of the studied scenarios (only if L1 is big enough to keep at least four packets). This is the maximum load observed before CPU saturation, at which point the L2 buffer is overflowed regardless of the queue size. The small size required for the CPU queues observed in these simulations supports the feasibility of the PPLB concept.

Another conclusion of the simulation was that the validation of the arbitration scheme chosen for the control unit of PPLB. The arbiter selects which packets go from L2 to L1 in accordance to the setup information. Each CPU indicates to the Queue Scheduler from which queue it wants to receive packets. In case of blocking , there is the option of dynamically rescheduling the orders for the unit that pushes data into L1 queues, so in that way collisions are prevented and full CPU utilization is achieved (for line rates up to 10 Gbps for the selected bus width of 128 bits). Actually, an efficient packet injection is the key for the small queuing observed in front of the CPUs as well as for optimal feeding of the CPUs.

5.5 Benefits of the Packet Processor Local Buffer on DRAM accessing

This section discusses how to better organize the packet buffer to accelerate the access to the DRAM. It reviews three alternatives that pursue reducing the access latency to maximize memory throughput focusing on how they are affected by the introduction of the PPLB. These three methods are access reordering, virtual row addressing and access rescheduling.

5.5.1 Access reordering to avoid bank conflicts on DRAM

The probability of bank conflicts when accessing DRAM is **always** there, whatever the method implemented. No matter the complexity in the access scheme, a combination of unlucky coincidences can invariably end up in a slowdown of the Packet Buffer. This cannot be avoided because the traffic manager defines the packet transmission order regardless of the optimal access to DRAM, hence making the read accessing unpredictable. Even though the worst-case scenario may rarely appear, it must be considered in the design. In bank conflict cases, the input buffering placed before the Packet Buffer absorbs small DRAM performance oscillation. Not so irrelevant is the fact that provisioning for extreme worst-case scenarios to prevent improbable situations lead to over-costs, which arguably is not the best of the design practices.

To sustain this statement, here let us review the probability of bank conflict occurrence on a DRAM memory with *B* banks. A bank conflict occurs whenever two consecutive accesses are addressed to the same memory bank, an event with a probability inverse to the number of banks in the memory, supposing that accesses are random and no dependency is built among them *(8)*.

$$P_{bank\_conflict} = \frac{1}{B}$$

(8)

The first conclusion is that a memory controller capable of activating multiple banks simultaneously already delivers a great improvement in access efficiency. Proper bank interleaving is probably the best BW-improver strategy. The implementation of such a controller is not straightforward because of the complexity of the control logic capable of ensuring optimal utilization, however most advanced controllers already include this feature.

If interleaving is allowed, access reordering can additionally reduce the probability of bank conflicts still further. This technique consists in re-scheduling memory requests to achieve a conflict-free access pattern and has been already suggested for segment-based [41] as well as for per-flow memory organizations [88]. With access reordering, the probability of a bank conflict is determined not only by the number of banks, but also by the deepness of the access queue where to shuffle. In these conditions, a bank conflict only occurs if all elements in the request queue are directed to the same bank, which is highly improbable if the queue is big enough. Intuitively, the larger the selection range *P*, the higher the chances to find two non-consecutive accesses to the same bank. Probability in this case is defined as in ***(9).***

$$P_{bank\_conflict} = \prod_{i=1}^{P} \frac{1}{B} = \frac{1}{B^P}$$

(9)

This is indeed a very low probability of bank conflict. For the sake of clarity, Figure 76 depicts it for different *P* sizes and variable number of DRAM banks. Notice that for instances, in a memory with 8 banks and 16 positions in the request queue bank conflicts will occur with a $3{,}55^{-15}$ % probability or to give a numerical example, in a NP servicing 10 Gbps and a 64-byte segmented memory, once every 166,51 days.

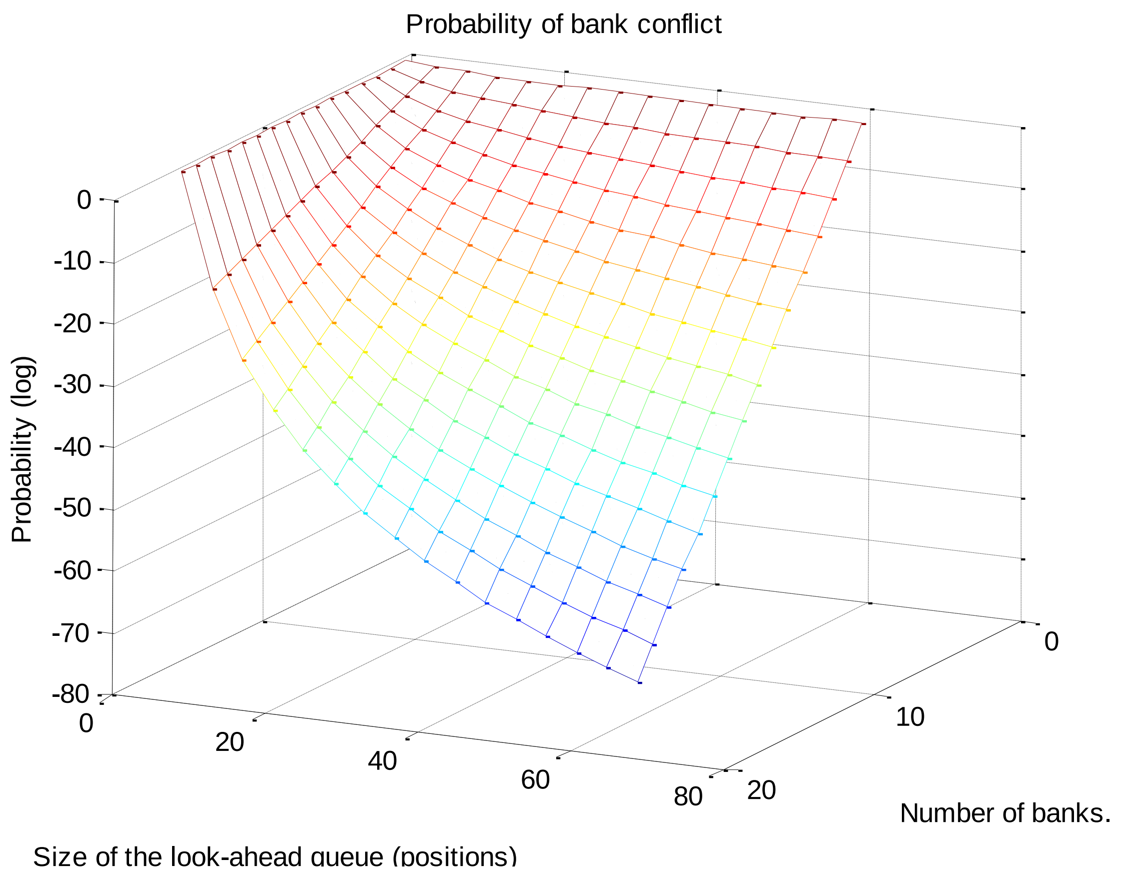


Figure 76: Probability of a bank conflict with access reordering

This analysis is not so bright if we consider the effects of latency. The delay that a request to DRAM may experience is proportional to the length of the reordering queue (*P*). If the only reordering option consists in rescheduling the last element in the queue, then the latency for all requests in the queue increases proportionally to the service time of the last access. In the worst case, all accesses go to the same bank (A) except the last one. These 15 accesses will require of other 15 accesses interleaved to completely preventing bank conflicts. Now if we consider an example where all accesses in the queue are 1-Kbyte long, in a 64-bit memory such accesses require 128 clock cycles each. In consequence, if *P* is equal to 16 (elements in queue), the worse-case delay is a non-negligible 4.096 cycles. This translates into 10,24 µs, for a Packet Buffer with reordering mapped on a standard DDR2-400 memory.

The advantage of the PPLB in this scenario is that access delays are not as critical for the PPLB as they are for a CPU directly accessing the memory. A CPU needs to implement advanced mechanism as multithreading to hide this latency, which is costly, while in comparison the impact for the PPLB is limited to the introduction of a small additional delay in the packet when crossing the system. In that sense, the considerably delay introduced by access reordering in exchange for extra throughput becomes less of a problem if such a mechanism is implemented together with the PPLB.

5.5.2 Optimized address mapping

Smart address mapping techniques can achieve an average reduction of bank conflicts. Their approach is usually probabilistic rather than providing a hard bound on worst-case conditions. For that reason, these methods have little support as their performance is linked to the access pattern and thus variable depending on traffic characteristics. They are however of easy implementation as no additional effort is required beyond multiplexing some bits of the physical address bus of the memory controller. Moreover, they yield not-inconsiderable BW improvement with little effort along with improving the operation of access reordering by reducing the probability of consecutive same-bank accesses.

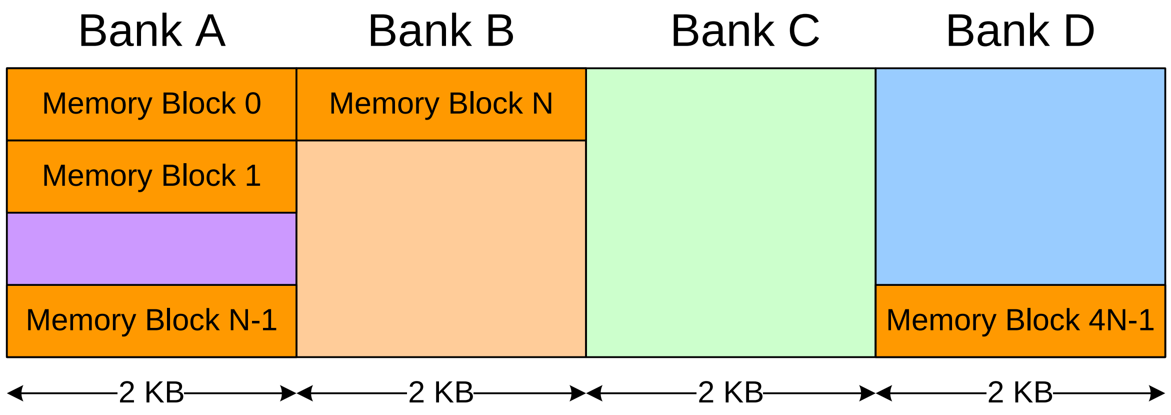


Figure 77: Direct memory mapping on DRAM

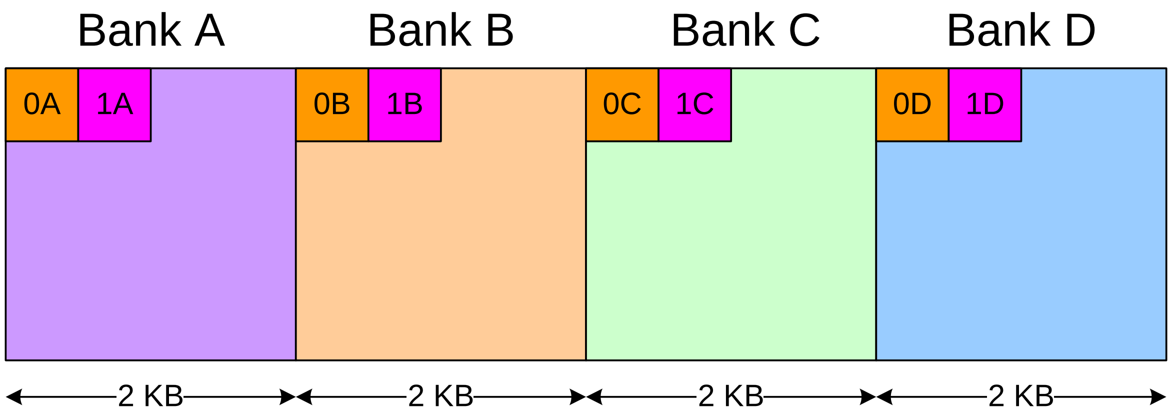


Figure 78: Virtual Row addressing method with embedded bank interleaving in the memory block

As a piece of example, this thesis introduces the Virtual Row method that considers mapping one memory block across several banks. According to the memory management scheme proposed in chapter 4, the Packet Buffer is organized in single-sized blocks of 2 Kbytes. If these addresses are mapped physically continuous, their mapping on the memory looks as depicted in Figure 77. Intuitively, the probability of storing a burst of incoming packets in the same bank after allocating a new memory block is quite high. This effect is however not observed in the egress side due to the multiple packet order modifications taking place internally in the packet processor. Anyhow, the equations presented previously to describe the bank conflict probability are based on the assumption that accesses are random and independent, which does not hold true in case of consecutive incoming packets stored in the same bank. Virtual Row improves the operation of access reordering by preventing a burst of requests to the same bank (Figure 78).

The method has been investigated by developing a SystemC-based model of standard 64-bit DDR memory controllers and feeding it with different accesses patterns. The model can accommodate multiple memory devices as well as interleaving requests to different banks when such feature is activated. As stimulus, the simulations utilize the IMIX packet pattern with 64-byte segments.

The benefits of Virtual Row as observed with the model are depicted in Figure 79, for different memory devices. As reference continuous 8-word requests are issued without interleaving thus permanently generating conflicts, which not unexpectedly delivers low bandwidth efficiency (+ *dark blue*). A clear improvement is obtained with memory controllers capable of activating simultaneously several banks, although yet without any reordering capability (+ *light blue*). In some cases, the request pattern allows the controller to hide bank activation latency behind a previous access hence enabling additional memory bandwidth efficiency. The benefits of bank interleaving deteriorate for faster memory devices, a consequence of relative higher bank activation latencies when using faster clock frequencies. As a reference, near-perfect efficiencies can be achieved by issuing long requests, 2-Kb long for example (+ *green*). In that case and even without access interleaving, efficiencies are close to 100%, an effect hindering the influence of latency with longer data bursts. This pattern is actually never observed as the internet packet mix contains a large amount of medium and small packets.

Virtual Row also accelerates the memory interface by facilitating the interleaving of bank accesses (+ *red*). Regretfully, the improvements quickly deteriorate for faster memories devices, again due to longer bank cycles. However, the results achieved with the mapping of memory blocks across multiple banks can be clearly observed for all memory devices tested. Virtual Row can be implemented regardless of whether packets have been already processed or not. Its biggest drawback concerns unpredictability as an adverse reordering of packets in the CPU or in the Queue Manager may generate bank conflicts again, though with a lower probability. In this sense, the PPLB introduces again higher predictability in the access pattern to the Packet Buffer due to the elimination of the short accesses from the CPU and so can also positively impact the operation of Virtual Row.

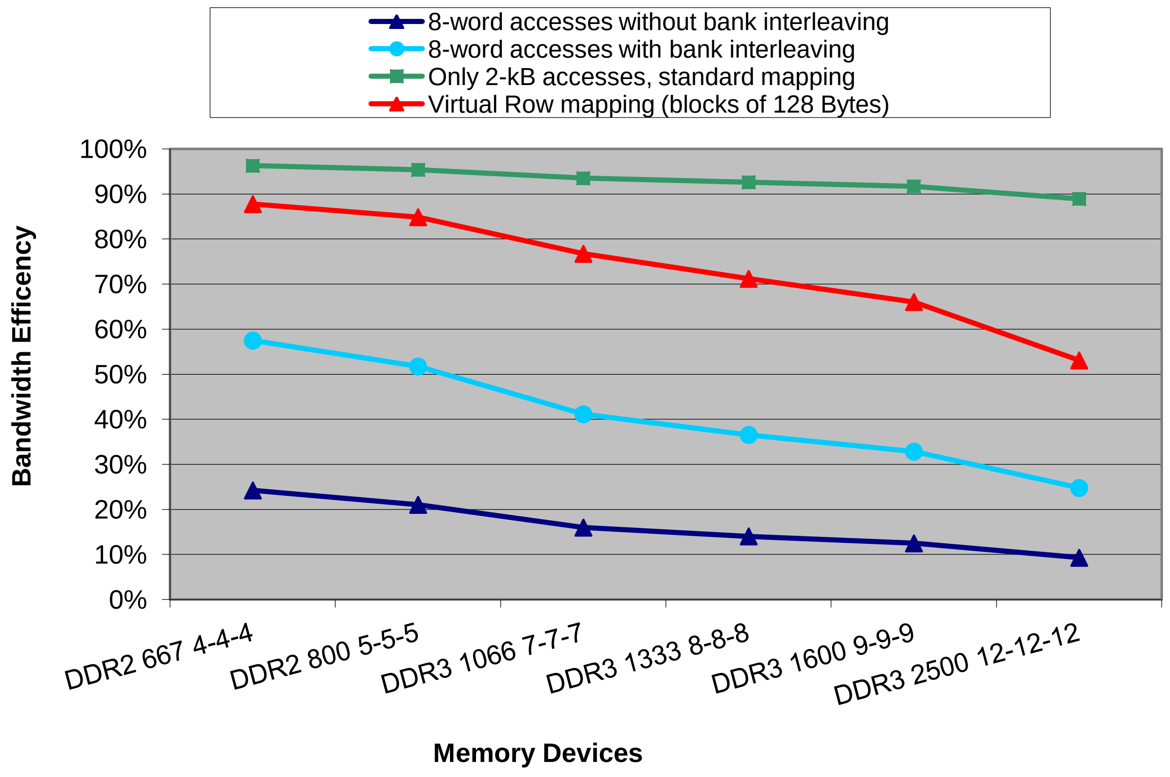


Figure 79: Access efficiency comparison with Virtual Row address mapping

5.5.3 Bank conflict-aware segment allocation

One of the most interesting throughput acceleration methods for DRAM in NPs consists in allocating free memory to actively prevent bank conflicts. This enhancement is only enabled by the introduction of the PPLB and for that reason, a new contribution in this work. We consider a system where memory accesses can be actively allocated to generate bank-interleaved access patterns. The method can even completely eliminate bank conflicts without introducing additional packet delay as is the case with access reordering.

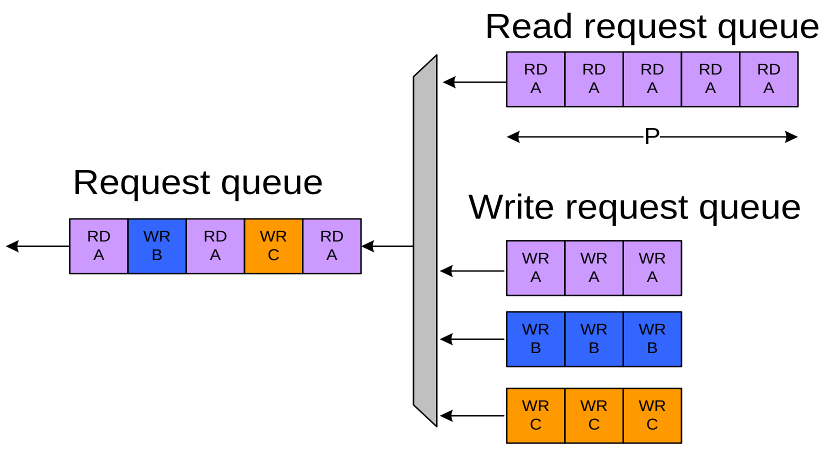


Figure 80: Rescheduling of accesses to the Packet Buffer

In such a system (Figure 80), free segment addresses are classified according to their bank destination and made available for packet storing after CPU processing. Remember that the packet order in the read queue cannot be altered, as it is established by the Queue Manager and reordering could lead to out-of-order packet transmissions.

Once the memory allocator in the ingress side has access to the status information of the read queue, new packet segments can be allocated in addresses not conflicting with read requests (active memory allocation). Then, Read and Write accesses can be interleaved avoiding two consecutive activations of the same bank. The novelty with the PPLB is that no accesses are issued by the CPU to the Packet Buffer, only the PPLB and the TX side of the Buffer Manager interact with the Packet Buffer. This allows actively allocating memory space for writes to prevent conflicts with the given read access pattern. This is **fully compatible with the segmentation algorithms in chapter 0**, with the only difference that physical addresses are assigned to packet segments after packet processing, while segmentation can be calculated as before in the Buffer Manager.

This concept is very innovative in the approach, as most proposed mechanisms to improve DRAM access focus on improving a **given** access pattern, while the method here proposed can **actively generate** it. To the best of our knowledge, the realization of this concept has not been proposed for networking applications or even in other computer science fields. Only when all engines in the system susceptible of writing in the Packet Buffer are prevented to do so, can the active access pattern generation take place. CPU accesses to the Packet Buffer are avoided with the PPLB buffer, so we can conclude that the PPLB enables this method to accelerated DRAM accessing.

It is theoretically possible to achieve conflict-free accessing to the Packet Buffer, without delay and with a relatively straightforward implementation. Importantly, active memory allocation works correctly in the assumption that there is free space available in at least one alternative bank than the conflicting one. Such is usually the case attending the low average usage of the Packet Buffer, where most memory resources remain unused during normal operation.

For the implementation of the mechanism (Figure 81), two principal modifications in the system architecture are required. First, it is necessary to integrate the memory allocator together with the memory controller. The same unit must receive all memory requests, allocate memory for Write requests avoiding conflicts with Read ones and finally issuing them interleaved to the memory controller. Besides, the new allocated addresses need to be placed in the Packet Descriptor. There are several approaches for doing this.

One way (Figure 81) consists in implementing a unit with a small address cache containing free addresses for the different memory banks. Once addresses are assigned to segments, they are written in the Packet Descriptor, which is queued in the PPLB while waiting for data to be fully stored. This operation is done in the output of the PPLB, which is aware of the status of the read requests queue to the Packet Buffer (direct interface between memory controller and the PPLB). Inconveniently, the memory controller must be design-specific to be compatible with PPLB and so some degree of modularity is lost. Besides, a Buffer Manager capable of classifying the available memory addresses according to the number of banks available requires insight into the memory characteristics, in particular the size and number of banks available.

This is precisely the main drawback of the rescheduling method. The Memory Subsystem architecture strongly deviates from state-of-the-art architectures present in current NPs. However, the intelligent memory assignation described yields promising advantages in terms of performance as well as memory access optimization and so makes a strong case for further investigating its feasibility.

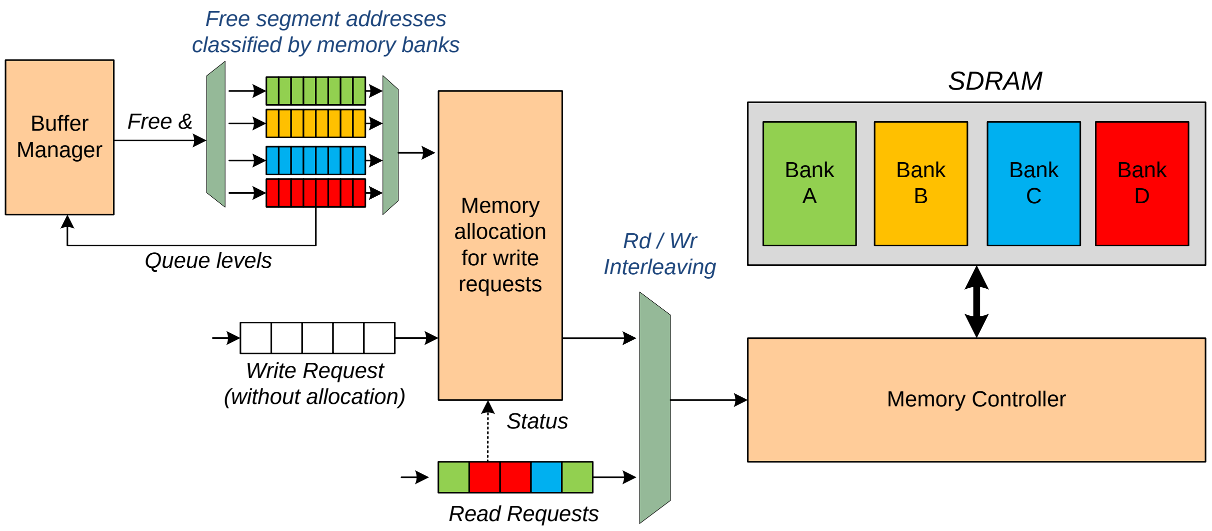

Free segment addresses classified by memory banks
Buffer Manager
Free &
Queue levels
Memory allocation for write requests
Write Request (without allocation)
Status
Read Requests
Rd / Wr Interleaving
SDRAM
Bank A
Bank B
Bank C
Bank D
Memory Controller

Figure 81: Block diagram for the implementation of Active Memory Allocation

5.6 Summary

This chapter discussed a new packet-injection architecture that eliminates cache misses in the CPU while improving the access to the Packet Buffer. In current architectures, packets are stored first in the main buffer and then *pulled* by the CPU after a cache miss, which slows down the packet processing capacity. Injecting the header or the complete packet in the CPU's cache or in a local buffer accessed through a direct interface is considered an excellent way to address two key issues with NPs: DRAM throughput and CPU performance.

Following this idea, the Packet Processor Local Buffer (PPLB) is conceptually introduced along with a discussion of its architectural trade-offs. The PPLB allows pushing each packet to the corresponding CPU's local buffer and once processing has been accomplished, the packet is stored in the main Packet Buffer. By avoiding all direct interaction between the CPU and Packet Buffer, it is possible to reduce the load on the DRAM, a typical bottleneck in NPs. Moreover, the benefits of access-reordering techniques can be enhanced when implemented in combination with the PPLB, essentially because the access pattern becomes more deterministic. This can be used to implement an active allocation of memory addresses that can completely avoid bank conflicts in DRAM-based Packet Buffers.

6 The SmartMem Buffer Manager

The investigations presented in the previous chapters make a strong case for the introduction of multiple segment sizes for packet segmentation in NPs along with the bitmap-based control structures. These statements are here backed with measurements from a Hardware Platform and therefore its claims further reinforced. In order to consistently proof the benefits predicted with the SystemC-based model, a hardware prototype has been implemented and tested on a Xilinx's FPGA device. Making use of it, we can measure the throughput acceleration delivered by the new Memory Subsystem in realistic scenarios in comparison to state-of-the-art buffer management.

But the proof-of-concept is not the only valuable outcome of a hardware implementation. An idea or new method might seem really promising until the hardware resources required for its implementation prove to be too high or even the elevate complexity of the module architecture questions the feasibility of its implementation. In that sense, the discussion of the architectural decisions taken during the system's design as well as the number of slices needed together with clock frequency achieved gives insight into the requirements of the HW implemented. Besides, the analysis of the platform bottlenecks reports about the limitations in the hardware implementation.

All improvements achieved with the new packet segmentation must be referenced to state-of-the-art schemes, which raises the question of how to fairly compare the new concept against the previous. Resorting to previously published figures is not acceptable because differences may come as a product of factors external to the algorithms under discussion. To prevent such influences, the experiments have been done following a gradual strategy, first presenting the throughput obtained with a reference system to then compare it with the newly proposed schemes. Such an experimentation strategy is more demanding in terms of effort due to the fact that it requires not only the implementation of our ideas but also the old ones. For instances, the accelerator here presented supports all possible packet segmentation strategies, provided that their segment sizes are the powers of two between 64 and 2048. This approach enables us measuring performance exactly under the same conditions for both the reference and the proposed improvement, with in my view is the fairest way to demonstrate the superiority of packet segmentation with multiple segment sizes, even considering the additional effort over a specific implementation.

Strictly, there is no need of developing a full hardware accelerator for the implementation of the proposed segmentation concept instead of a SW-based memory manager. There is however compelling evidence in scientific literature (see 0) that a CPU cluster aided by a Buffer Manager Unit as the one this section introduces is a good design practice in terms of performance. For that reason and in order to better evaluate the buffer administration free from application interferences, the SmartMem Buffer Manager (SBM) was developed and implemented in VHDL, tested and extensively debugged. The measurements obtained reinforce the central claim of this doctoral thesis, as we are going to see following suit.

6.1 SmartMem Buffer Manager in standalone mode

6.1.1 Functionality

The SmartMem Buffer Manager is a programmable engine capable of storing and fetching simultaneously variable-length packets in the context of Network Processing Units. The operation is done autonomously without CPU supervision, thus freeing processing resources what accelerates packet processing. The device is capable of working in a full-duplex mode simultaneously dealing with incoming and outgoing flows of data, while administering at the same time the Packet Buffer and its associated control structures.

The unit's architecture consists mainly of three functional modules, the RX and TX units and the Address Manager (AM). The RX unit receives and stores temporarily the incoming packets in local SRAM. Then the segmentation module splits the packet in as many segments as required, according to the algorithm configuration. The segment sizes allowed are all powers of two between 64 Bytes and 2 KB with a maximum of 24 segments per packet, which is the maximum value required for a 1518-Byte packet stored using only 64-B segments. Once the packet structure has been calculated, the segmentation unit requests memory space accordingly from the Address Manager.

In order to accelerate data storing, a memory cache contains free segment addresses thus effectively decoupling control operations from the RX data pipeline. Once the free address is available, the storing unit transfers segment data to the external Packet Buffer through the Peripheral Local Bus (PLB) or a direct interface, which is configurable. We use an optimized bus interface that allows burst transfers of up to 2 KB, the LIS-IPIF, so as to increase the bus usage by reducing arbitration and transfer times. As the PLB Bus supports simultaneous read and write transfers, two master interfaces have been included so as to optimize bus utilization, following the recommendations extracted from the SmaSyMo model.

A Packet Descriptor (PD) containing the segment pointers and context data (input port, packet length, etc) is generated once the packet data has been completely stored. This structure is of variable size depending on the maximum number of segments per packet *M*, as configured for meeting the segmentation algorithm requirements and being its minimum size 128 bits. This minimum PD size corresponds to configurations with *M* equal to one or two segments per packet. Every additional 64-bit word allows for three extra 16-bit pointers with their corresponding segment type field. Keeping all segment addresses in the PD has the advantage of reducing data fetching latency, as no linked list needs to be parsed, and additionally prevents control updating to interfere in the measurements. Inconveniently as the *M* parameter grows, additional memory space has to be reserved for the implementation of the queue system, typically built up out of linked PDs. However this overhead does not play a role when using the algorithm proposed (SLA-N4-M2) as only two segment addresses are stored. This is the same size that would be needed when using linked lists to keep track of the head and tail pointers.

An interrupt to the CPU is triggered when a new packet has been stored and is available for processing. After packet processing has been done, the TX unit receives the PD of a packet ready for transmission. The Fetching Unit immediately starts to retrieve the corresponding data segments from the Packet Buffer into its local memory. Data is stored in the TX local buffer continuously hence accomplishing packet reassembly. Afterwards segment addresses are returned to the AM, where they are returned to the pool of segments and so made available for reuse. The transmission concludes once the packet has been completely sent to the corresponding output port.

The SBM requires control structures to manage a Packet Buffer typically with a size in the order of hundreds of MB or even more. In our case, the Address Manager uses the advanced method presented in the previous chapter, which allocates memory dynamically using a combination of bitmaps and double linked lists that track 2-Kb memory blocks.

6.1.2 SmartMem Buffer Manager architecture

The block diagram of the SBM has been depicted in Figure 82. Blue arrows depict the data flow, while blacks ones stand for internal control communications. The SBM communicates with the CPU and the Packet Buffer over a PLB bus while external traffic generators feed the system.

The design of the SBM confronts several challenges. It needs to process input and output traffic simultaneously (full-duplex), while keeping control structures updated at any time. Such degree of parallelization can only be achieved by deploying **three independent units** while allowing for a flexible communication between them that does not stall a unit when waiting for acknowledges. As example, the RX Unit storing of packets should not stall after requiring a free address from the AM as well as the TX Unit should not put data fetching on hold until the previous packet has been discarded. This design principle aims to deliver maximum performance to meet inasmuch as possible high line rates.

The main three units communicate in-between taking advantage of address caches and message queues, which prevents the stalling of a unit while waiting for response from another. Internally, both the RX and the TX unit follow the same communication principle, thus implementing flexibility in the internal data flow as well. Such internal queuing allows for smoother performance oscillations due for example to variable accessing delay to the bus, both in the ingress and egress sides. Continuous data storing/fetching to/from the Packet Buffer cannot be guaranteed as it depends on a shared data bus. In case of temporary bus congestion, local data buffering allows the Transmission Unit to keep feeding the output ports with a certain degree of independence of the bus load and thus maximize TX performance. A symmetrical situation can be experienced in the ingress side, where the operation of the Storing Unit depends on bus availability too, so again a small internal buffer enables continuous data reception even when the bus experiences short busy periods.

Several parameters allow for a flexible operation configuration. This is necessary to explore performance under different conditions and enables for instances selecting the segmentation algorithm, the maximum burst length on the bus, the activation of a direct interface to the DDR-SDRAM memory controller among others.

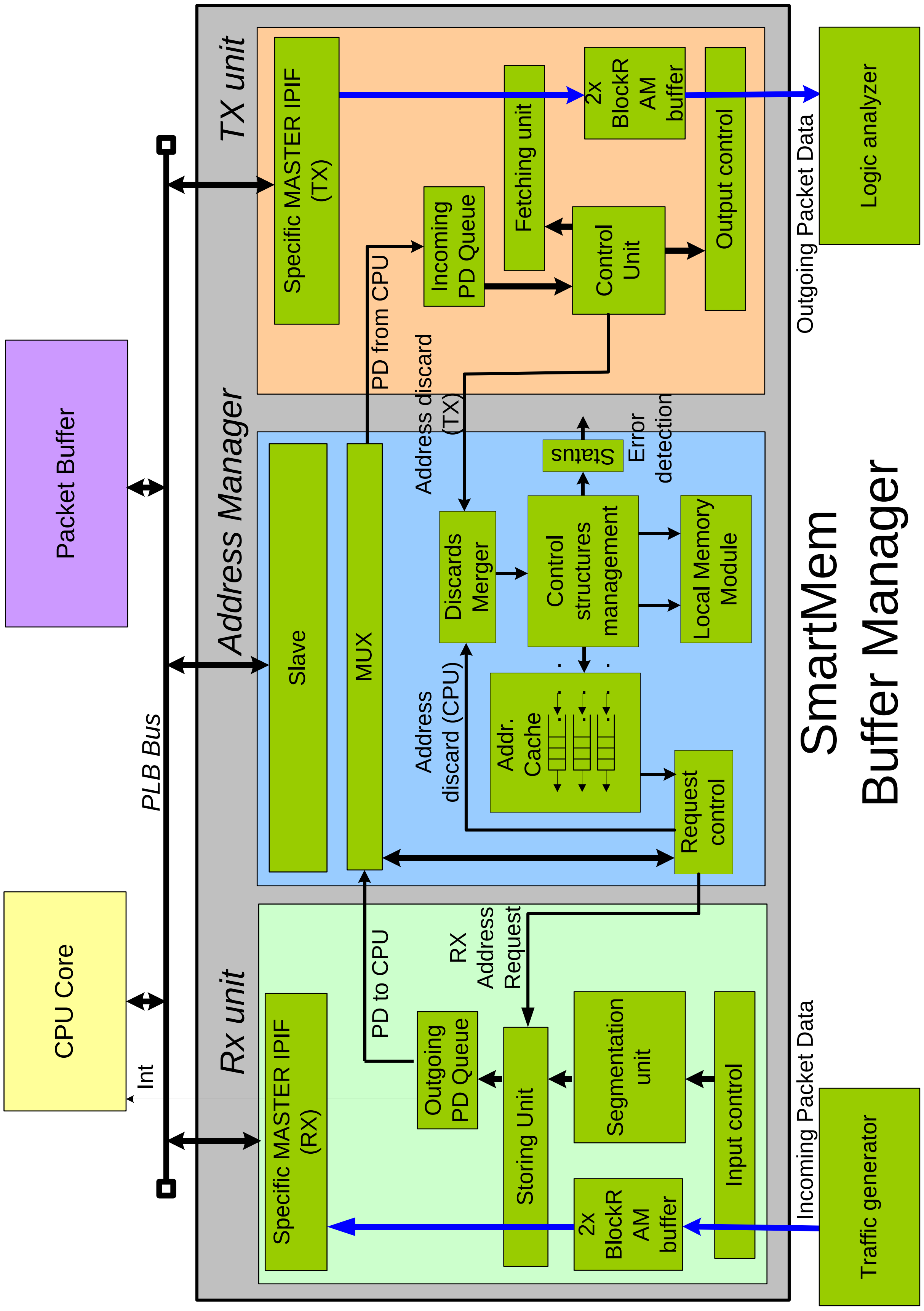


Figure 82: SmartMem Buffer Manager internal architecture

Finally, mentioning that the SBM is capable of interacting with other units in the system, such as the CPU or other processing units requiring access memory allocations or data discarding. For that purpose, a slave bus interface is included in the AM unit capable of receiving commands through the bus. This interface may receive petitions for new memory space allocation (malloc), to discard a corrupted packet without triggering transmission as well as allowing for packet resizing, which is a helpful function for example when a new header/trailer must be introduced in the packet due to de/encapsulation. Let us now review with greater detail the BM's units.

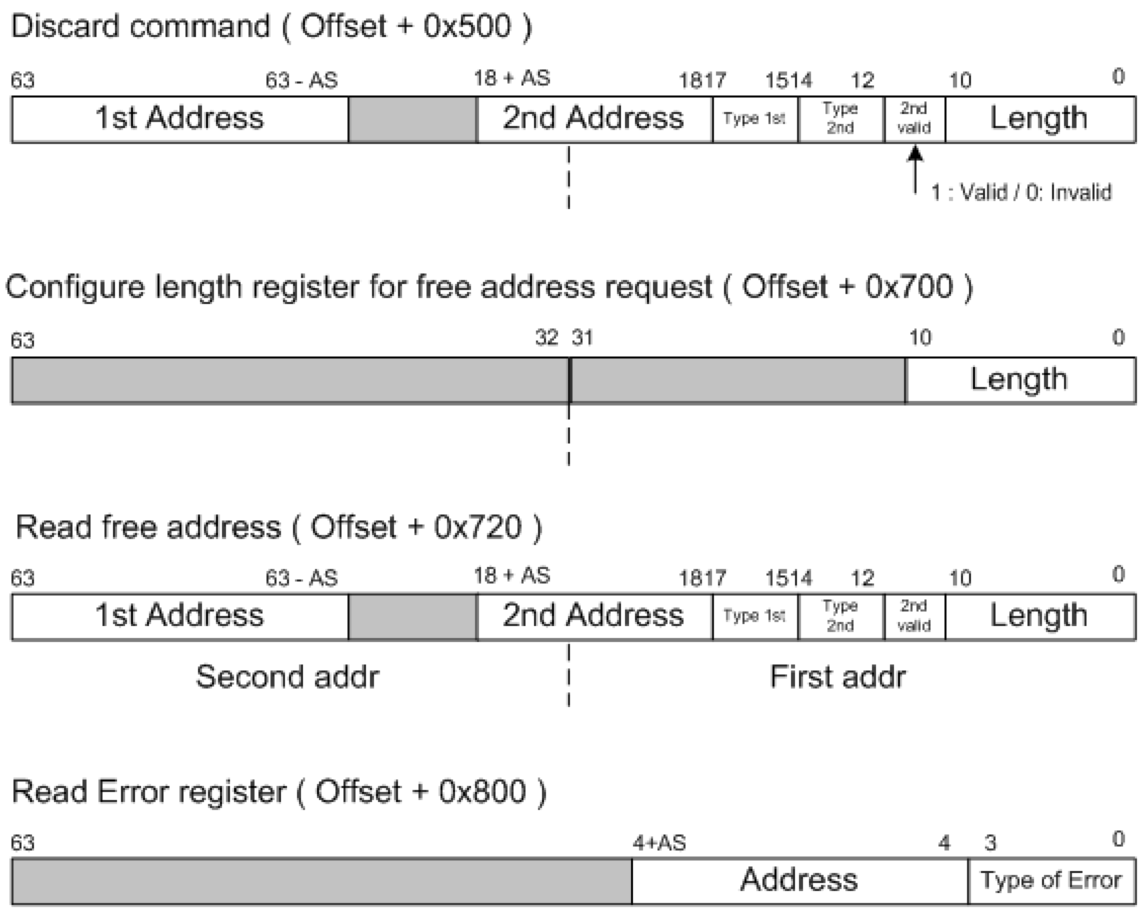
6.1.2.1 **Address Manager Unit**

According to the investigations in 4.2, only a memory management that supports dynamic assignment of memory to different segment buffers suits multiple size segmentation. In consequence, the BM supports dynamic buffer administration by using the combination of bitmaps and linked lists presented section 4.4. These functions are implemented in the Address Manager (AM), which performs control updating, manages the Packet Buffer and provides local storage for the Control Buffer. The AM is a modular unit. It allows for a standalone configuration or for the integration into the SmartMem Buffer Manager.

The AM must service petitions for free segment addresses (up to six sizes), address discards and even allocating memory for new packets with the appropriate segmentation. This is done taking advantage of a segmentation unit implemented within the Request Control unit, independent of the RX Unit to allow for standalone implementations. Free memory requests may originate in other units inside the BM (through a direct interface) or triggered by a command from any other unit in the system (processing cores, co-processors...) via a slave interface attached to a PLB bus (Figure 83 depicts the syntax of these commands). An error register, very useful for debugging errors in the operation of the AM, can be accessed as well.

Control data is kept in a local buffer. This solution accelerates the updating of control structures as it removes the delay introduced when accessing the Control Buffer over the system bus. Both the bitmap and the linked lists share the same SRAM memory, with different address offsets. This Control Buffer is resized according a parameter that defines the Packet Buffer size. Then, the required number of dual-port BlockRAMs is generated during the HW synthesis to adequately fit the size of the Packet Buffer.

The internal architecture consists mainly of an address unit in charge of performing all necessary modifications in control structures. Actually the control intelligence in the AM relies on that unit, the Control Structures Manager. There, two parallel state machines are implemented. One refills the Address Cache whenever its level drops below a certain threshold. The second state machine performs all operations required for address discards. Whenever the threshold of a segment pool goes under a defined level, the Refill state machine automatically fetches new memory blocks, first using those that are partially used and if that is not possible, assigning free ones (+ for discussion of why this is done, see 4.3.3).



NOTES:

- Logical addresses have a maximum size of 23 bits, in NP3 this value is 16
- AS = Logical address size
- Address type indicates the segment length (Bold values are the valid ones in NP3)
  - ***000: 64 Bytes***
  - ***001: 128 Bytes***
  - 010: 256 Bytes
  - ***011: 512 Bytes***
  - ***100: 1 KB***
  - 101: 2 KB
  - 111: Invalid address
- Error register values
  - 0001: Error during discard. Segment already marked as free.
  - 0010: Error during discard. Wrong type of block, bitmap does not match pattern.
  - 0011: Memory full, waiting for segment discards.
  - 0100: Unassigned block detected with some segments set to in use.

Figure 83: format of commands accepted by the AM

The discard process is equally fast. After a command for address discard is received, regardless from the TX Unit or from the CPU, the unit checks if the address is valid. Three sorts of data integrity checks are executed to recognize wrong update operations and thus alert the system of an error in control data:

- First, the segmentation in the PD is tested. The packet length stored in the least significant bits of the PD must match the information stored in segment sizes fields, which gives a high certainty about the PD integrity.

- Second, the bitmap vector of the memory block is fetched and the busy bit corresponding to the address to be discarded checked. If the address bit is already set to ‘1’ (free) data is corrupted so the error register is updated.

- Third and last, the memory block must be assigned to the same segment pool as the segment size to be discarded. To check whether this is the case, the bitmap vector pattern cannot contain ‘1’s (free segments) at certain positions. A vector of a 2-Kb block has 32 bits, each standing for a 64-Byte space. If the block is assigned to the 256-byte pool for instances, its bitmap vector can only hold ‘1’s in the first bit of a 4-bit group (e.g. “1xxx1xxx1xxx1xxx1xxx1xxx1xxx1xxx”). See previous Figure 61, for more details on the control structures.

For new memory allocations, the Request Control unit receives the requests for free addresses both from the RX Unit (in a segment/length format) or from the CPU (indicates the size of a new packet). In the first case, a small internal cache contains free addresses classified per segment types (up to six), which hides free-address lookup latency to the RX Unit. This is again the case for a malloc request from the CPU, with the significant difference that the packet segmentation must be first calculated. For that purpose, a look-up table contains the segmentation data and returns the appropriate number of segments the packet requires, which afterwards are fetched from to the address cache.

The implementation of the unit on a Xilinx’s Virtex-4 FPGA requires of 1.650 slices in with the system’s clock constrained to 100 MHz. The local memory may work at twice that frequency if a speed-up of control data updating is desirable, thus provisioning for future upgrades of the SBM.

6.1.2.2 **RX Unit**

The RX Unit is essentially an autonomous Direct Memory Access (DMA) unit that conveys input data from multiple input ports to the Packet Buffer. In addition to the storing process, the unit also performs packet segmentation according to the algorithms exposed in chapter 3.3.5, request free memory space from the AM and finally generates a Packet Descriptor that the CPU can fetch taking advantage of the BM’s bus slave interface.

Internally, three main sections can be distinguished. A configurable number of input data queues interface with the MAC ports and store in an internal queue the incoming packet data. These queues adapt the incoming bus width of 32 bits to the system bus width, which is 64 bits. As the whole system is clocked at 100 MHz, the BW offered to the bus side is twice that of the input IF. Theoretically, if maximum traffic is offered, the RX Unit can achieve throughputs close to 6,4 Gbps, though this figure is constrained by the load in the system's bus as well as in the Packet Buffer.

Once the packet is fully stored in the RX's local buffer, the control Finite State Machine (FSM) passes a message to the Segmentation Unit, where the packet segmentation is calculated. The HW implementation of the packet segmentation algorithm is straight-forward: it consists of a small table containing all possible combinations of segment. The engine returns the first combination equal/bigger to/than the packet length. In order to try a different segmentation algorithm, this table need simply to be actualized with the corresponding values (VHDL parameter).

The BM supports all algorithms working with six segment sizes: 64, 128, 256, 512, 1024 and 2048 Bytes. Though this implies a little implementation overhead, this drawback is overcome by the flexibility it offers for the validation of the algorithm selected. The algorithm is implemented with a small look-up table where packet sizes are compared to all available combination of segment sizes.

Once segmentation has been calculated (it takes only one cycle), the control FSM requests the corresponding segment sizes from the AM and stores these values in one FIFO queue along with the packet length and some control data. Afterwards, the segmentation unit immediately switches to the next packet, without waiting for acknowledgement from the AM. Only when any of the FIFOs before the Storing Unit is full, it back-pressures the input pipeline. In parallel, the AM fetches the corresponding number of free addresses from its internal cache and pushes them to the corresponding address FIFO.

Only when all three control FIFOs have received valid data (segment sizes, segment size, source port), the operation of the Storing Unit is triggered. Its functionality is actually the most complex in the RX Unit, as it requires:

- ➢ Writing in the Packet Buffer the segments with the appropriate length.

    a. Requesting the corresponding burst length to the Master IPIF and accessing the PLB Bus. The transfer length can be either split in 128 bytes or unconstrained, always respecting the segment structure calculated before.

    b. Once the access to the bus is granted, it monitors transfers of data from the internal input queues to the Packet Buffer (control signals are generated by the bus IPIF).

- ➢ Once the packet has been completely transferred, the unit generates a Packet Descriptor containing relevant control data such as segment addresses, packet length and input port.

- ➢ Checking status of control queues together with ensuring that the PD Queue contains enough space for the new PD.

The last unit in the ingress pipeline is the already mentioned PD Queue, where packet descriptors wait until the CPU fetches them through the slave bus interface. The BM dos not implement a CPU load balancing system, which is an extensive topic usually out of the scope of the BM’s functionality. The BM requires though a mechanism to inform the CPU that packets are ready to be processed if working in standalone modus. In that case, the BM activates interrupts to the available CPUs (one per CPU), the number of which is configurable with a parameter. The generation of interrupts do not consider balancing the CPU core. Interrupts are activated according to number of PDs waiting in the PD Queue. In order to prevent multiple CPUs reacting to the same interrupt, only one interrupt is triggered per packet.

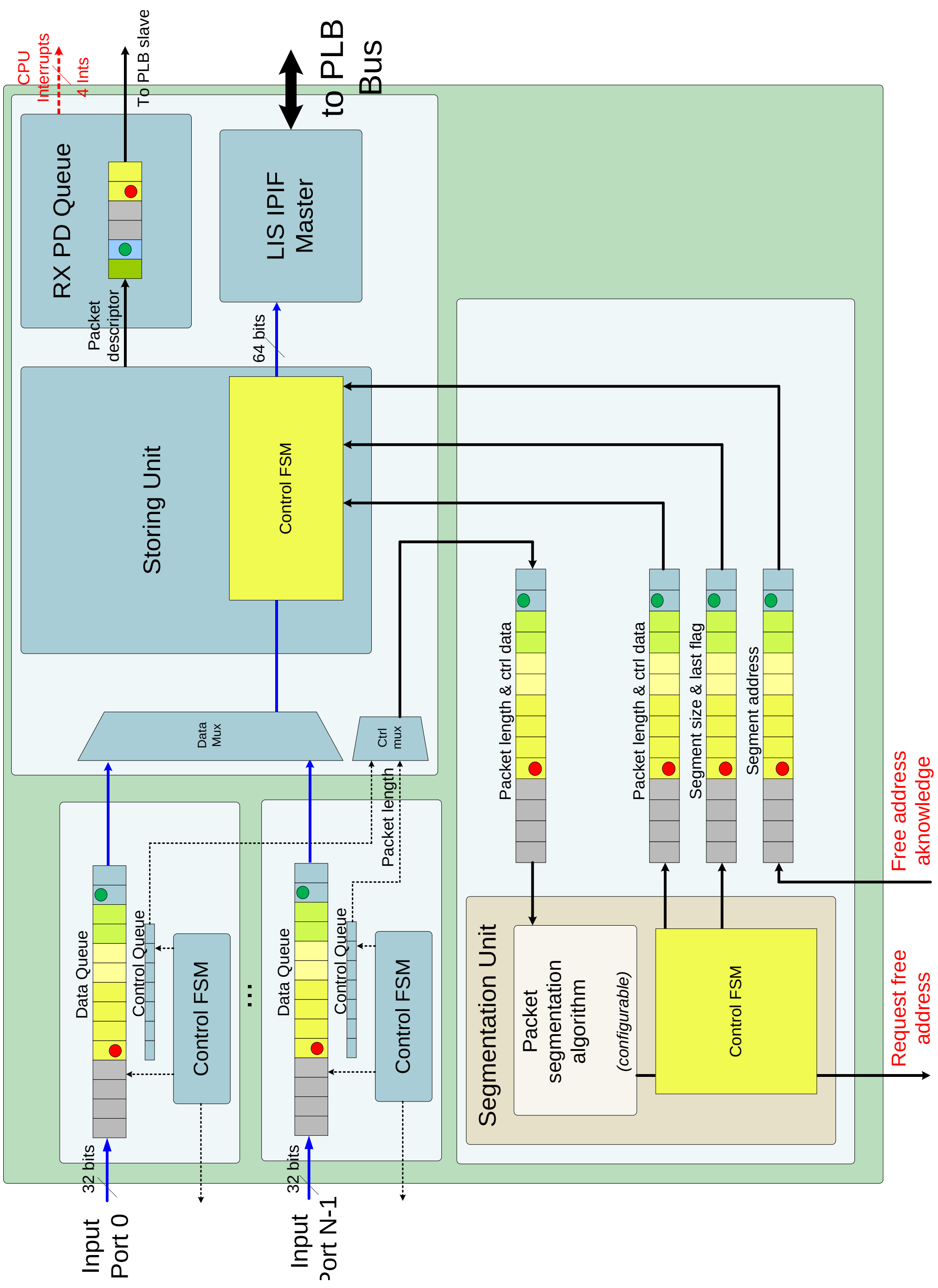


Figure 84: Block diagram of the RX Unit

6.1.2.3 **TX Unit**

The TX Unit implements the functionality required for fetching packet data from the Packet Buffer, re-assembling the packet and transmitting it to the corresponding output port. First, the Fetching Unit begins data retrieving from the Packet Buffer after receiving a valid PD from the PD Queue (if the PD is not valid, it is ignored). Fetching data from memory requires calculating the segments of the packet with valid data, how many bytes are stored in each of them and the right address offset to request from the Packet Buffer. When the packet contains context data, this must be fetched as well and put before the packet data. All information is then stored locally in two buffers, waiting for transmission to the corresponding output port. Once the packet has been completely fetched according to its segmentation structure, the segment addresses are sent to the AM for discarding.

Received packet data are then stored internally in the corresponding internal data queue, having one for every output interface. The TX Unit cannot modify packet fetching order to avoid packet reordering problems, so a packet addressed to a full queue stalls the pipeline. For that reason, the Queue Manager needs to have information about the levels in the output MAC units, a functionality external to the SBM. Having a queue per interface allows to keep several interfaces transmitting simultaneously hence increasing the total throughput. This buffer is very important for improved performance because the output interfaces are 32 bit-width and slower than the PLB bus (double channel, 100 MHz, 64-bit wide). Each SBM interface delivers 3.2 Gbps and so capable of feeding three/four Gigabit Ethernet interfaces.

The number of the SBM interfaces is configurable to a maximum of four. This is provisioned to allow for a maximum speed in the Tx Unit of 12.8 Gbps, which can only be achieved when the TX Unit is directly connected to the memory controller. When the TX unit is attached to the PLB Bus, throughput is constrained to 6,4 Gbps. The throughput discussion is better understood in the following section, where the fast direct interface to the Multi-port Memory Controller is discussed.

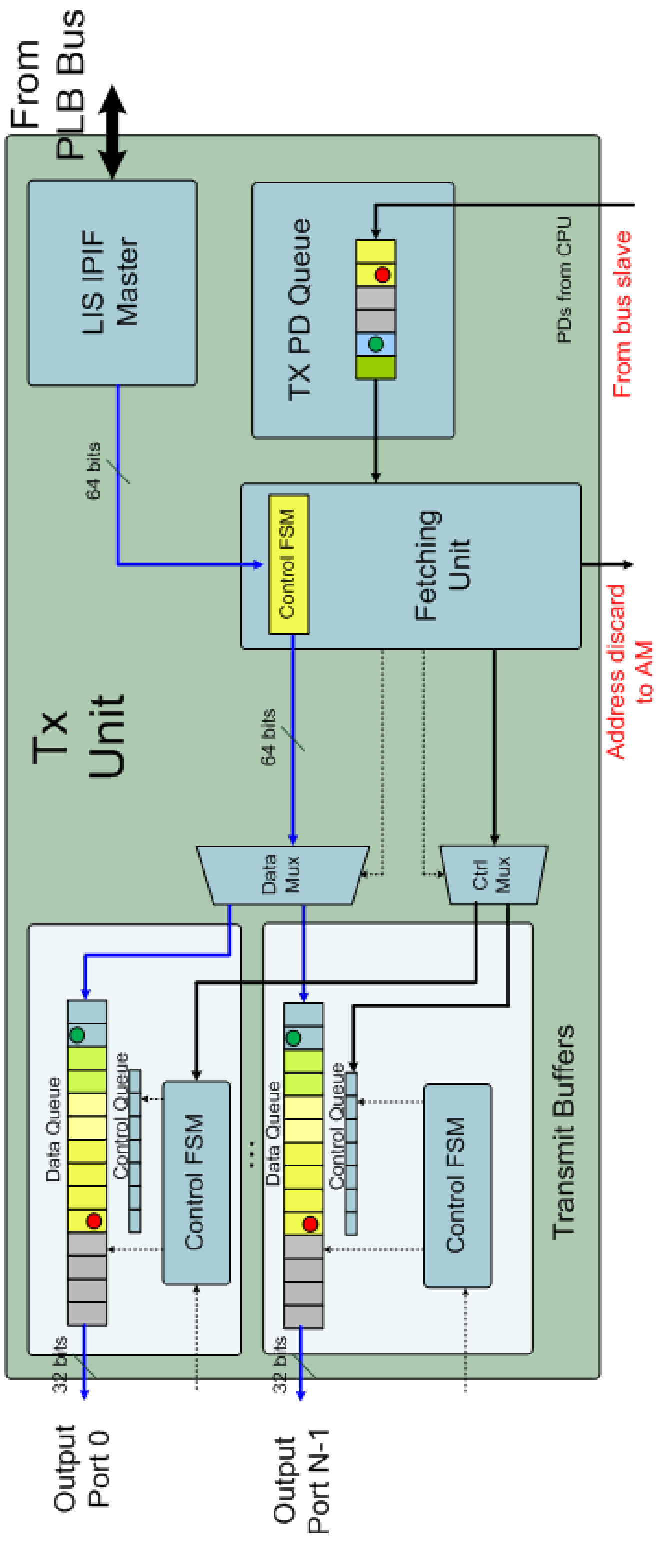


Figure 85: Block diagram of the TX Unit.

6.2 ***SmartMem Buffer Manager with FlexPath***

6.2.1 **Motivation the integration of the Buffer Manager in the FlexPath platform**

In NPs, the Buffer Manager main task consists in storing/fetching packets at high speed along with administrating the Packet Buffer. However, there are important additional features that the BM can provide by taking advantage of its central position in the data pipeline, thus helping to improve the general operation of the system.

Most of these tasks cannot be discussed independently of the rest of the NP architecture. For their study and specification, it is necessary to place the BM in the context of a complete NP architecture. Within this general framework, the SBM can assume the functions of administering **context data** in the NP. Context data are intermediate processing results that the units in the system use to communicate in-between. This data can be stored together or separately from packet data, but in any case require of memory administration and data storing/fetching. This creates a synergy with the packet data functions and so, the BM can efficiently manage both types of data. Context is generated by multiple co-processors and the amount of information per packet can be quite high thus further straining the memory subsystem.

To study such questions, the SBM has been integrated within the FlexPath architecture. FlexPath is an advanced packet forwarding system that combines the flexibility of programmable CPUs with the performance of a pipeline of co-processors that can forward most of the typical IP packets without requiring CPU intervention. The concept aims at bridging the gap between application-centric architectures and pure-HW weakly-programmable processors.

One of the characteristics making FlexPath very interesting for the BM is the so-called *AutoRoute* data path, where packets flow through the system bypassing the CPU-cluster. By taking advantage of several HW accelerators, this feature allows packet classification and automatic forwarding without even requiring CPU processing. It is of the highest interest to measure the combined performance of the SmartMem Buffer Manager and FlexPath, which can greatly help to offload the CPU cluster, one of the typical NP bottlenecks.

6.2.2 **FlexPath description**

In state-of-the-art NPs, all packets follow the same internal processing path so they are not differentiated according to protocol requirement or maximum latency. In essence, the FlexPath architecture allows multiple internal paths to which packets can be assigned in accordance to a set of rules that may be dynamically modified by the control plane. After the packet arrival, the five-tuple in the header is used for packet classification and assigned to one of the existing internal paths. Packets with very simple processing requirements may bypass the CPU cluster and can be handled in hardware alone (AutoRoute). Alternatively, packets requiring header or payload processing can be for instances directed to a dedicated hardware accelerator for en/decryption and only afterwards sent to the CPU, which offloads IPSec-related operations from the CPU.

Some parts of the packet processing task are relocated from the central processor cluster to specific processing units in the ingress and egress data pipelines. In short, the new architecture offloads many packet forwarding tasks from the CPU. Actually, the FlexPath NP implements a new architecture for network processors based on an open SoC architecture, consisting in a combination of standard RISC-processor cores, off-the-shelf components like buses and memory controllers along with a set of specifically-developed hardware accelerators.

The architectural design pursues to demonstrate the applicability and performance of dynamically reconfigurable methods in high-end networking applications. In order to accomplish that, it proposes dynamically reconfigurable hardware that enables an application-optimized packet routing through various functional entities in the NP that shall guarantee a better overall utilization of the existing processing resources and hence increase the system throughput. This platform pursues the reusability of its components in other platforms, together or independently. For that sake, the Intellectual Property (IP) cores developed are highly modular, enabling multiple configurations to better suit the requirements of the Network Processor in terms of hardware resources, features to be implemented and performance.

Following a selection of the investigations conducted in the context of the FlexPath project together with pointers for further reading:

- The main concept of FlexPath, which is the assignment of incoming traffic to different internal dynamically-reconfigurable paths according to its processing requirements, was here introduced [23].

- In order to optimize the use of CPU resources it is paramount to feed the CPUs continuously with packets. An optimized load balancing between cores can be achieved by monitoring the input queue of the processors and actively assigning new packets to those cores less busy, while reserving some CPUs for high-priority traffics in QoS-capable NPs [84].

- One of the problems that multi-core NPs present is the existence of a non-negligible probability that packets belonging to the same data flow depart in a different order than they arrived. In order to fulfill its service level agreement, the NP must avoid such out-of-order packets. The FlexPath project allows the introduction of a re-sequencer unit that by taking advantage of per-flow circular buffers prevents reordering to occur [25].

The architecture of FlexPath is depicted in Figure 86. We observe two pipelined structures in the ingress and egress sides, with the memory manager placed between both and working in parallel. An interconnection infrastructure communicates the Processor Cluster running the Data Plane, the Control Processor and the system memory where the Packet Buffer is mapped on. Any other accelerator required, as the mentioned Crypto-core, may be connected to the system through the common bus.

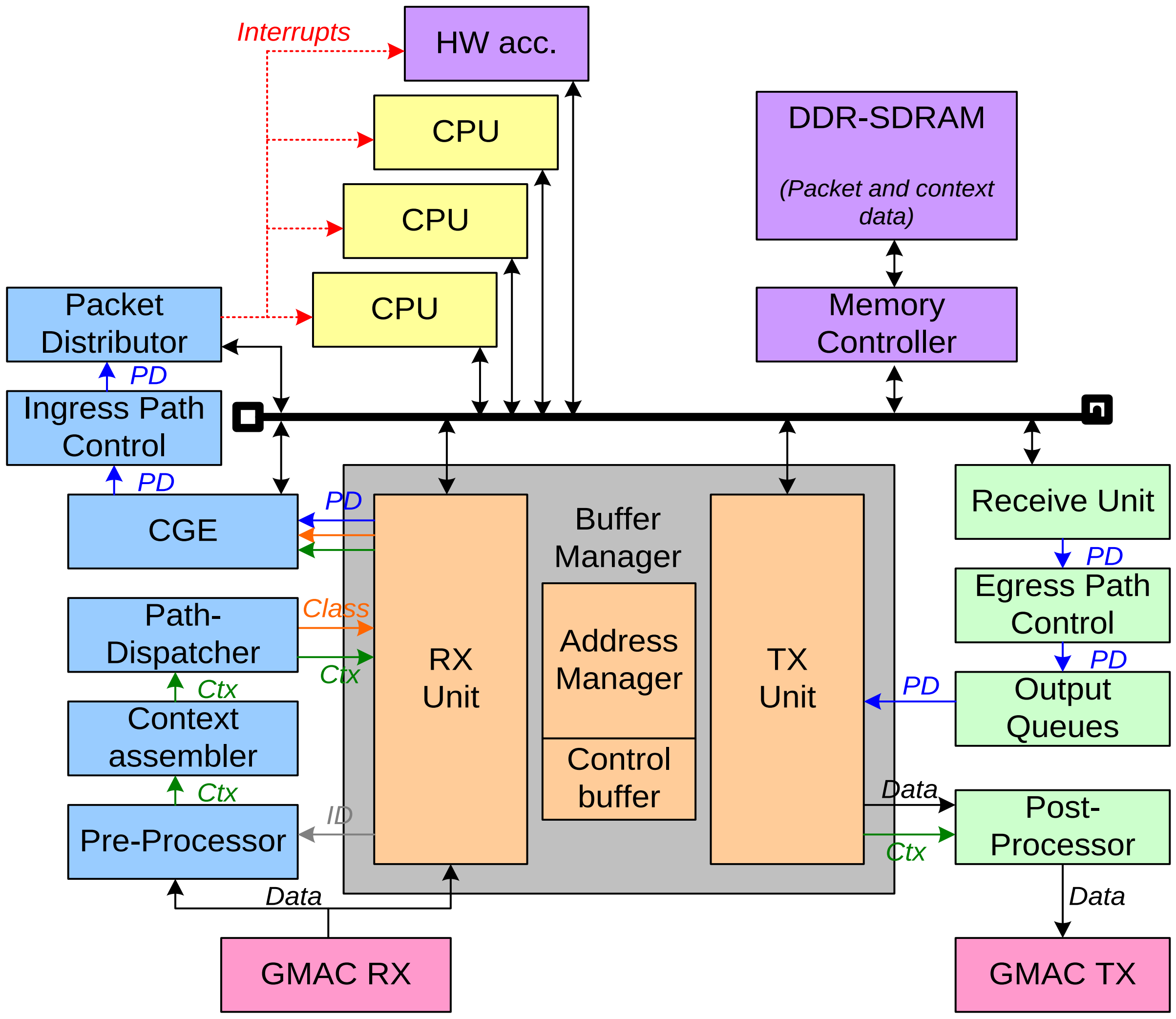


Figure 86: Overview diagram of the FlexPath NP

6.2.2.1 Internal data flow

When a packet is received in the input side, it is simultaneously transferred to the Buffer Manager and the Pre-Processor. While the BM performs data storing, the Pre-processor inspects the packet header and transfers relevant fields to the Path-Dispatcher. The Pre-Processor recognizes packets with Ethernet II, DynaCORE, IPv4, TCP, UDP and ESP protocols and indicates that to the Path-Dispatcher.

Next, the Context Assembler performs synchronization of raw context from the Pre-Processor and looks up for the destination address using its lookup engine. The context data is then sent to the following Path-Dispatcher unit. In case the packet is not recognized or is malformed, a default context indicating the packet malformation is generated and passed on to next unit in the ingress pipeline.

The Path-Dispatcher implements the HDGA packet classification algorithm in order to determine the further processing path of every incoming packet [85]. It communicates with the Context Assembler (which performs context synchronization) and then with the

SmartMem Buffer Manager, to which it transmits the packet classification along with the results from the Pre-Processor (the IP 5-tuple).

Next in the pipeline, the BM receives context data from the Path Dispatcher and the packet data from the input, allocates memory in the Packet Buffer, stores the packet in the main memory and generates a Packet Descriptor. The PD is passed on to the Context Generation Engine (CGE) together with the results generated by the Pre-Processor, the CGE and the Path Dispatcher.

When the CGE receives the token from the BM indicating that the packet has been fully stored in memory and is ready for processing, the CGE stores the input context in a memory space reserved by the BM in the packet structure for that purpose. It also consolidates the Raw Context and Packet Descriptor into their final, packet-specific form. Depending on the results determined by the Path Dispatcher, a uniform ingress context (CII) or packet-specific output context (CIO) has to be created and stored in the Packet Buffer. The CGE extracts the addresses of the memory structure from the Packet Descriptor. After storing context, the fields in the Packet Distributor are updated and forwarded to the next unit along with a Queue Number for the Packet Distributor.

The Ingress Path Control stamps each incoming packet with an increasing sequence number on a per flow basis. It receives a Packet Descriptor from the Context Generation Engine while using the Hash Value (obtained out of the IP 5-tuple), traffic class and priority and uses it as flow ID. Also, an 8-bit counter for each flow ID is maintained to generate the sequence number. Control packets and own packets (i.e. with destination to the Control Plane) will not be stamped, since these packets (probably) will end at the NP and thus not forwarded.

Next in the pipeline, the Packet Distributor is responsible for distributing incoming Packet Descriptors to the connected processing elements. It contains 64 queues and can serve up to 16 CPUs. The integrated Multi-Processor Interrupt Controller (MPIntC) decides which PE(s) is(are) interrupted, which triggers the fetching of the Packet Descriptor by the CPU. The MPIntC contains several configuration registers to determine the association between the queue and PE. The association may be 1-to-1 (one queue is served by one PE), or 1-to-n (one queue is served by n PEs). The PE will be informed by interrupt. Packet Descriptors can be read out by a bus connection (PLB). If multiple associated queues contain Packet Descriptors when reading out, the PE automatically gets the Packet Descriptor with highest priority. Packet Descriptors may either be written to the Packet Distributor by the proprietary Packet Descriptor Interface (LIS-IPIF) or by PLB.

When a CPU in the data planes finally receives a PD, it processes the IP-stack and makes the corresponding modifications in the packet header, which are then written back to the Packet Buffer. Once the operation is done, the CPU writes the processed PD through the PLB bus to the Receive Unit in the egress path of the FlexPath.

In the transmission side, the Egress Path Control monitors flow order and reorders it if an out-of-order packet is detected. In-order PDs continue to the Output Buffer (or Output Queues), where a set of queues contain traffic with different priorities. The BM finally receives the packet token. It calculates the packet structure, fetches packet data and output context and forwards it to the Post-Processor. This last unit implements some simple packet operations like Time-To-Life counter decrementing or Check Sequence updating, as instructed in the context data. CIO is in fact a list of instructions for the Post-Processor regarding the modifications to be done in the outgoing packet.

6.2.3 SmartMem Buffer Manager design for compatibility with FlexPath

The integration between FlexPath and the SmartMem Buffer Manager arises several design challenges for the BM. It requires the enhancement of existing capabilities along with a new set of features:

- Context data management

The principal new feature of the FlexPath SBM is the administration of memory for context data along with fetching in the egress side and subsequent transmission to the Post-Processor. In NPs, meta-data can be stored separately in a dedicated buffer or as header/trailer together with the packet data. For the SBM, this later option is preferred to avoid an add-on effort of separate administration of a Control Buffer and the corresponding control structures. This is done is spite of the fact that preemptively allocating memory for context data reduces storage efficiency, a drawback compensated by the fact that the amount of memory dedicated to context can be parameterized.

By default, 128 Bytes per packet are always allocated at the beginning of the first data-packet segment. This figure is for sure somewhat conservative as 64-Byte context headers should suffice for most applications. Anyhow, for the operation of the SBM, the amount of reserved memory for context is less of a matter than the packet structure required, and thus the value is open to the designer's need (a parameter allows modifying the context area).

- Packet Identifier

A 5-bit identification number is used throughout the ingress pipeline for queue synchronization as well as debugging purposes. It is also used by the CPU. The BM generates this packet identification and then the ID is transmitted to the pre-processor instead of the other way round. If the ID generation were external to the BM, this would create a dependency of the BM on that external unit.

- Address discarding

Another point of discussion is **discarding** and how should it be executed. Several cases may occur where packet discarding is commanded to the BM, for example when the Pre-Processor marks a packet as bad frame and thus triggering its discard. An option is to immediately indicate such event to the SBM, so as to avoid storing the packet data in memory and hence saving in BW. For such operation, an additional interface between the Pre-Processor or the Path-Dispatcher and the AM would be required. As data flows in parallel, there is no guaranty that the discard request arrives before the packet has departed the RX Unit, or even precisely in the middle of the storing operation. To avoid contending with multiple scenarios that are prone to errors, all packets are stored and then discarded after the packet has been stored. Discards can only be commanded over the bus or on the TX side.

- Semi-pipelined architecture

In a pure pipelined architecture, the BM would only begin data storing when the Pre-Processor and Path-Dispatcher were ready. This is undesired because packet data must be buffered locally during pre-processing and additionally creates unnecessary delays, consequently increasing packet latency. It seems a reasonable compromise to place the BM in parallel to the ingress units, then beginning packet storing immediately and thus minimizing the internal buffer requirement. Only when packet data has been completely transferred to the Packet Buffer, the CGE begins to store context data.

In a similar way in the egress side, the BM can only fetch packet data after PD reordering and QoS policing has taken place. This data is then forwarded to the Post-Processor, which is placed after the BM. That order in the pipeline offers little alternative if we consider that the SBM fetches not only the packet data to be modified in the Post-Processor but also the commands (output context) required for the Post-Processor operation. In that sense, the SBM performs adaptation data widths between the bus (64 bits) and the Post-Processor (a 32-bit unit). Figure 87 depicts a timing diagram of the signals on this interface.

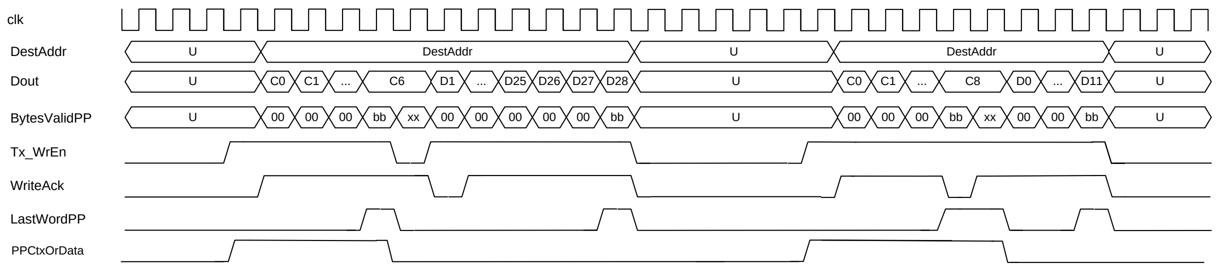


Figure 87: timing diagram of the interface between the SBM's TX unit and the Post-Processor

- ➢ Packet Descriptor fetching and receiving

Packet Descriptors are sent to the CGE after packet storing, instead of been fetched by the CPU. On the egress side, PDs are received from the Queue Manager. In this interface, the BM activates a back-pressure signal to stop further PD transmissions when its queue is full.

- ➢ New logical address format

Another important issue is the definition of the segment address format. The SBM stores the segment addresses in the PD (two per packet). Logical addresses are preferred rather than physical addresses for bit-saving reasons. Logical addresses are 64-Byte-aligned hence in order to calculate the memory offset, the logical address is shifted six times to the left and padded with zeros, a quite straightforward operation *(10)*.

$$Physical\ Address = Base\ Address + (64Bytes \cdot Logic\ Address)$$

(10)

- ➢ Packet Descriptor format

Some fields need to be reserved in the Packet Descriptor for storing FlexPath-specific data, for example the flow-specific sequence number or the length of valid context. On the ingress, the RX Unit sets the logical addresses, packet length, the ingress port and the Packet ID. On the egress side, the packet and CIO length are considered for adequately fetching data from the Packet Buffer, along with the segment addresses and the destination port. In Figure 88, the new PD structure is depicted.

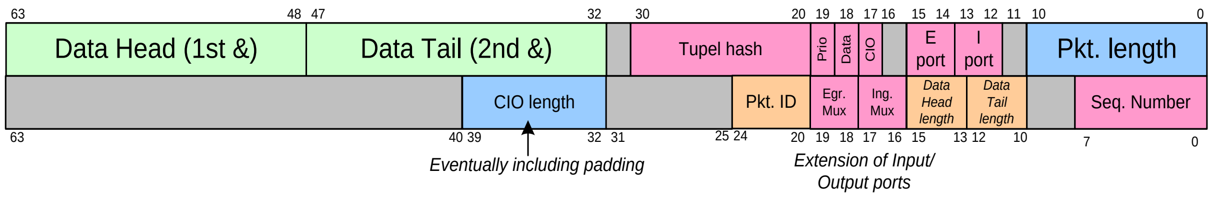


Figure 88: Packet Descriptor structure

6.3 Compatibility with the Packet Processor Local Buffer and the MultiPort Memory Controller

6.3.1 Adaptation of the architecture to the Packet Processor Local Buffer

This section describes the SBM upgrades required to make it compatible with the the PPLB architecture previously described in Figure 70. The SBM has been designed to incorporate PPLB-specific features while still being able to work in standalone mode. Modularity was one of the main goals considered for the SBM design.

There are many functional differences between the standalone and the PPLB-compatible modes, mainly in the RX Unit. Incoming packets need to be stored according to the rules set by the Control Plane and classification results provided by the Classification Engine. Packets can be partially stored in the Packet Buffer (payload) and the PPLB (header), only stored in the PPLB or completely stored only in the Packet Buffer. What is more, the SBM must be programmed dynamically to react in different ways depending on the packet classification results. This programming is set by the control plane, by accessing the Configuration Table in the RX Unit and over the PLB Bus. Each traffic class is assigned with different instructions (stored in a look-up table), which the RX Unit consults for every packet before proceeding with packet storing.

The information in the Configuration Table includes different fields; not only information regarding PPLB storing (Header length and storing mode) but also additional information to calculate an advanced packet structure (Figure 89). This information includes:

Parameter for PPLB compatibility	
HeaderLength_s	Portion of packet to be stored in PPLB (default: 64 Bytes)
HeaderStoringMode_s	Store header in PPLB / Store full packet in PPLB / Bypass PPLB
SegmentationMode_s	Packet segmentation depends on traffic class (Not implemented)
Parameters for the advanced packet structure	
DataOffset_s	Initial offset to the beginning of valid packet data
ExtraHeaderLength	Extra space allocated between packet data and context.
ExtraTrailerLength	Extra space allocated at the end of the packet for enlargement.

Table 7: Configuration Table parameters

As mentioned before, packet resizing can be really inefficient if it implies the reallocation of data in the Packet Buffer. An early detection of such traffics enables the allocation of extra space for the additional header and/or payload. Moreover, an additional space can be allocated in front of the packet data for context data, which can be sized differently to adapt memory allocation to the specific requirements of different traffic types. The complete structure of the packet is shown below and helps to shed light into the meaning of the values in the Configuration Table.

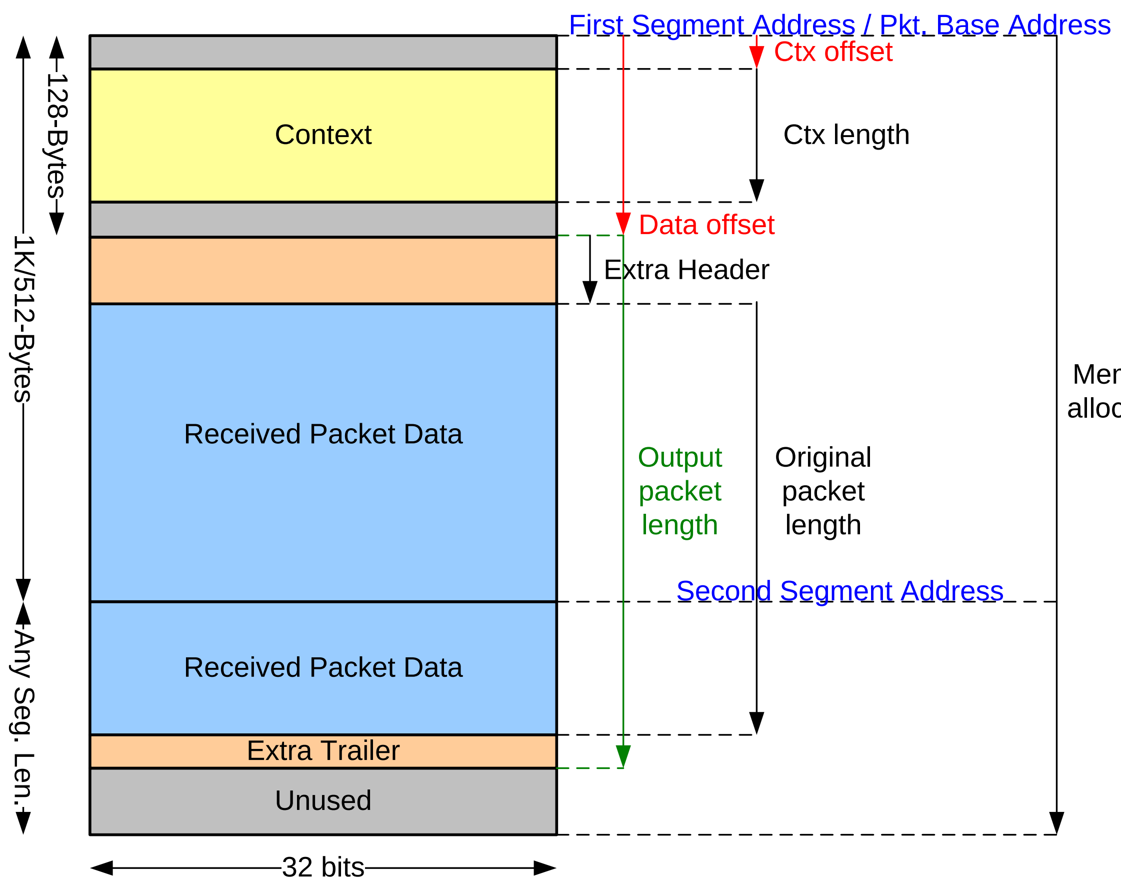


Figure 89: Packet Structure with the PPLB

In order to implement PPLB compatibility and the advanced packet structure in Figure 89, the functionality of the RX Unit needs to be upgraded. The new features require several architectural modifications in the RX Unit and for that reason they are implemented in a different RX Unit, which is instantiated during the HW synthesis (selected with a parameter).

For every single packet, an 8-bit word is received from the Path Dispatcher indicating the traffic class of the packet (Figure 90). The relation between this tag and the different packet protocols is known by the control plane, so it can instruct the SBM to operate differently according to the class. For example, if a packet which is entering an IPSec tunnel is detected, the SBM can provision extra space in the Packet Buffer for the extra IPSec encapsulation. The tag is used as an index in the Configuration Table to extract the storing instructions for the SBM and then these results are pushed forward towards the segmentation algorithm as well as the Storing Unit.

After packet segmentation, the Address Manager provides the logical address for the segment sizes calculated. Only when the segment addresses have been received, the PPLB is ready to receive data and the control data (packet length, segment sizes and source port) has been generated, packet data can be transferred to the PPLB or the Packet Buffer, according to the commands from the Configuration Table. If the packet is stored in the Packet Buffer, an interrupt to the CPU is triggered and this fetches the PD. If the packet is sent to the PPLB, the PD is stored together with packet data in the PPLB to thus avoid the bus delay when accessing the PD Queue in the SBM over the system bus.

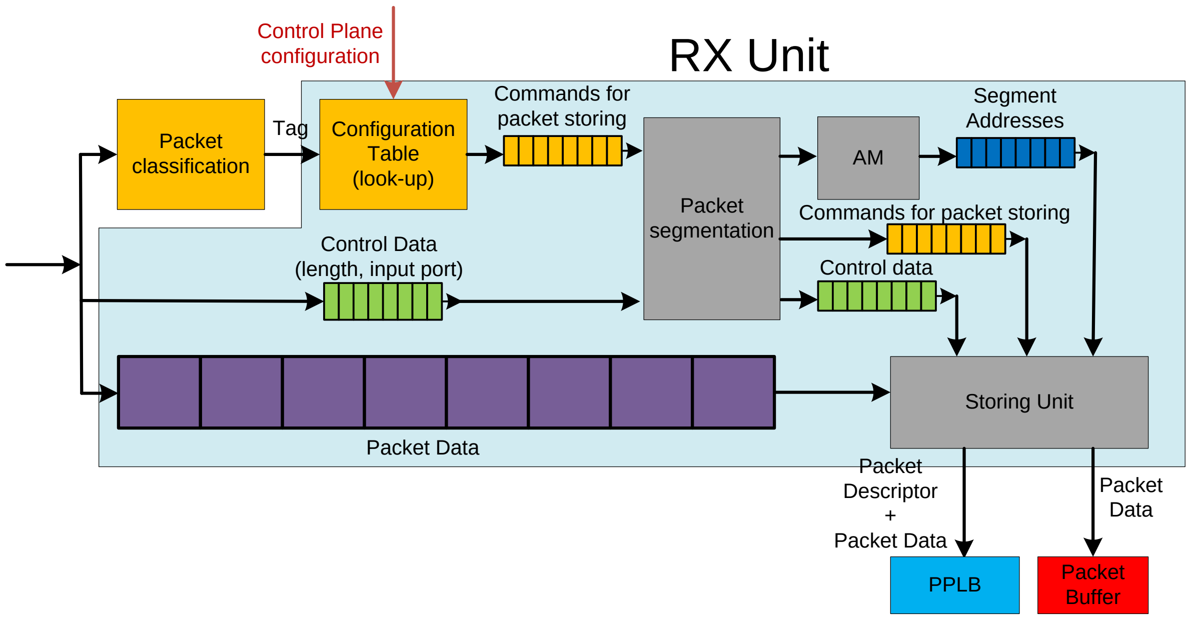

Control Plane configuration
RX Unit
Packet classification
Tag
Configuration Table (look-up)
Commands for packet storing
Packet segmentation
AM
Segment Addresses
Commands for packet storing
Control Data (length, input port)
Control data
Storing Unit
Packet Data
Packet Descriptor + Packet Data
Packet Data
PPLB
Packet Buffer

Figure 90: Data flow in the RX Unit in compatibility mode with the PPLB

In the TX Site, there is no need to implement any new functionality for the PPLB. However, the Fetching Unit in the TX Unit must understand the advanced packet structure with the variable offsets to properly calculate the physical address in the Packet Buffer.

6.3.2 Compatibility with the Multi-Port Memory Controller

A DRAM memory controller is a device that generates the right control signals to read/write data and refresh the memory banks of a DRAM memory. It is usually a complex device with very strict timing constrains and with up to 6 different clock signals in case of DDR devices. The SBM was originally implemented to work with any PLB-compatible memory controller to in that way keep compatibility with any embedded system that uses this popular bus.

However, when using the PLB bus for high-speed data transferring between the SBM and the memory controller there is a performance bottleneck in the bus. Early in the system-level exploration presented in 3.4, the throughput of the bus and the access delays to it were identified as limiting factors for performance. That is highly undesirable for the purpose of measuring the throughput of SBM and to fully exploit the benefits of the new segmentation algorithms.

For that reason, the SBM has been adapted to be compatible with the Multi-Port Memory Controller (MPMC) from Xilinx, which delivers much higher performance and specially allows for the implementation of a direct interface to the SBM, thus reducing access latencies to the Packet Buffer. The only drawback of this solution, besides the additional implementation effort, is that compatibility to Xilinx’s proprietary Private Interface Module (PIM) has to be included in the RX and TX Units. The PIM interface operates a 64-bit data bus at 200 MHz, which allows up to 12.8 Gbps throughput full-duplex. This introduces a second clock domain in the design of the SBM and so extra design complexity, as we are going to see following suit.

6.3.3 RX Architecture with the Multi-Port Memory Controller

In Figure 91 we can observe in greater detail the architecture of the RX Unit including the new PPLB functionality and compatibility with Xilinx's MPMC. Packets are still received from a configurable number of interfaces, each capable of up to 3,2 Gps, and then stored in the local buffer. Packet segmentation cannot be done until the total length of the packet is known, a value that now depends on packet length and the specifics of the traffic class (to allocate extra header/payload space). Once classification results have been received and storing instructions from the Configuration Table are available, packet segmentation is calculated and a request is sent to the AM to obtain the corresponding segment addresses.

The SegUnit_Wrapper Unit in front of the Storing Unit synchronizes the arrival of three types of control information: control data (segment sizes, length and source port), the storing instructions from the Classification Table and the logical addresses from the AM. Only when these three types of data are ready, can the operation of the packet storing begin. The Storing Unit does not supervise anymore the transmission of data to the Packet Buffer, as was the case in the standalone mode. Now it only calculates the number of accesses and their length, because data must be fetched from the local buffers at 200 MHz to satisfy the MPMC requirement, which cannot be accomplish as the Storing Unit is placed in the 100 MHz clock domain. A different unit operating at 200 MHz must handle data storing.

The DirectIF_write_Multiport unit is now responsible for interfacing with the Packet Buffer and the PPLB, as well as of generating the Packet Descriptor. It has two interfaces: a direct interface to the memory controller (RX-PIM) and a direct interface to the PPLB. This presents a challenging design problem as both interfaces must transmit data at the same rate while operating at different clock speeds. To make it more complicate, a packet can be transferred partially to the PPLB (the header) and the rest to the MPMC (payload). While the PPLB is designed to work at 100 MHz, the Xilinx's Multi-Port Memory Controller (MPMC) used in the implementation requires data to be transferred at 200 MHz.

The solution selected consists in splitting the hardware in two different clock domains and widening the data bus towards the PPLB to 128 bits while using a 64-bit bus at 200 MHz to the MPMC. This ensures equal data rates on both data interfaces. This solution is very performing and, as we will see later in this chapter, the throughput results obtained with the direct interface are much higher than when using the bus interface.

Problematically, the logical functions in the two clock domains need to be isolated so the synthesis tool can properly calculate the different timing constrains without any violation. This is achieved using specialized FIFO queues for transferring data between domains, which enable write and read at different clock frequencies. For control signals, a three-register adaptation strategy (first register operates at 100 MHz, while the other two at 200 MHz) has been followed to prevent any clock violation.

A final function of the DirectIF_write_Multiport unit is the injection of the Packet Descriptor into the PPLB along with packet data. In the standalone mode, the PD is written in a queue a different interrupts to the CPU are then triggered. However, maintaining such a design with the PPLB seems inappropriate as the CPU must fetch the Packet Descriptor using the bus, precisely generating the bus-access latencies we want to avoid with the PPLB. Consequently, the PD is stored together with the Packet Data in the PPLB so the CPU can efficiently reach the Packet Descriptor in its local buffer. The PD is always located in front of packet data, so no additional control data must be transmitted to the CPU.

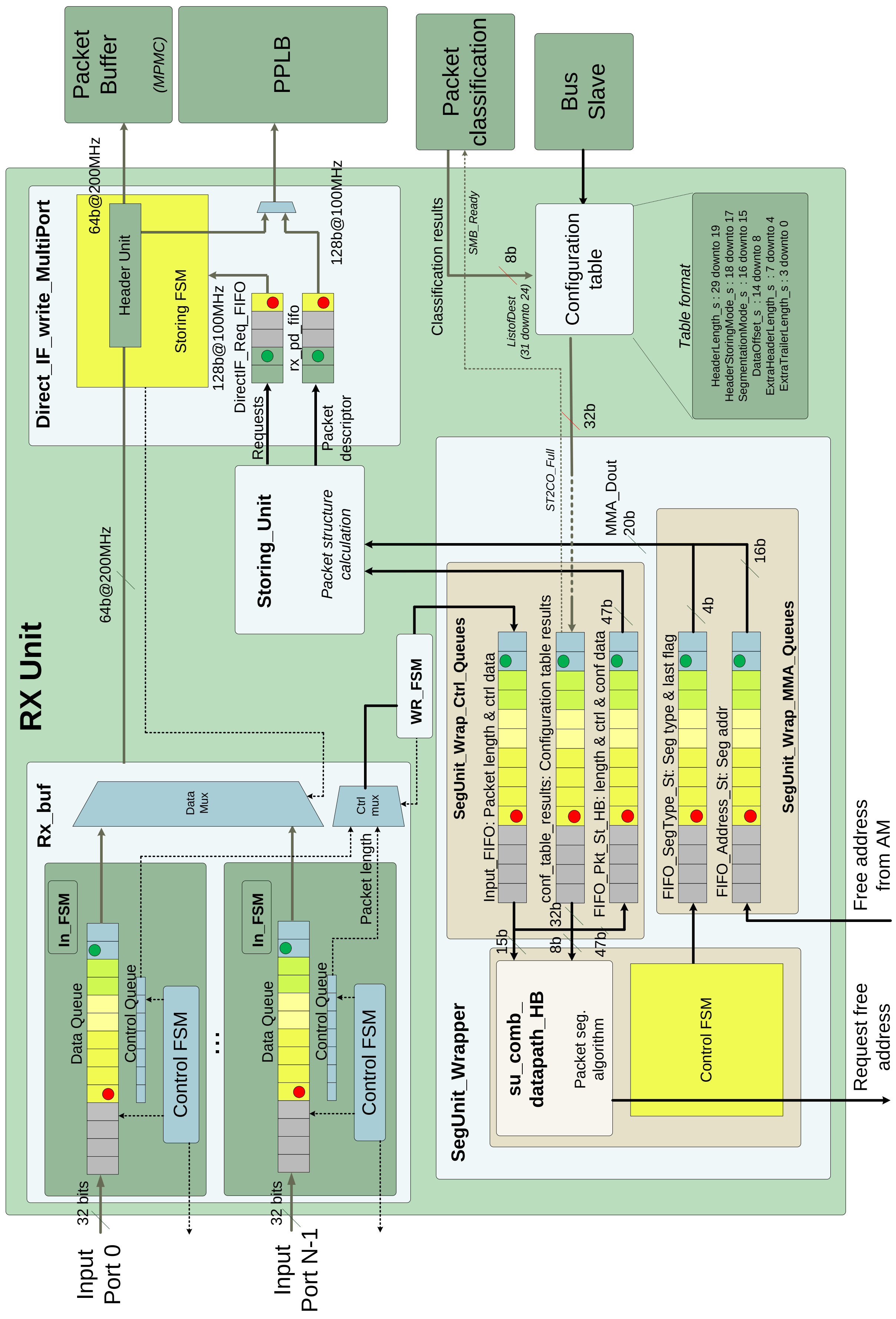


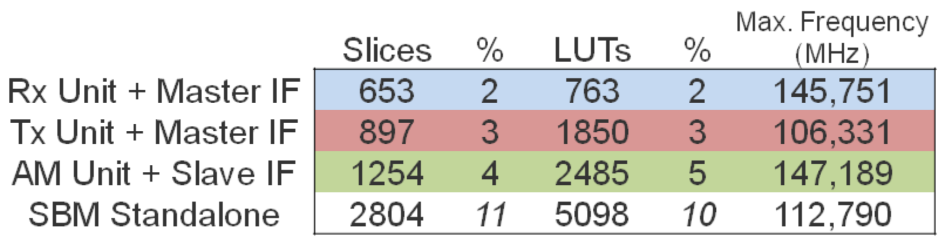
Figure 91: RX Unit in compatibility mode with the PPLB

Finally, mentioning that all new hardware modifications yet discussed regard the modifications done in the architecture of the RX Unit. However, the new packet structure presented in Figure 89 must also be understood in the TX Unit so it can correctly fetch the packet data. The TX Unit obtains all information required to calculate the structure of the packet from the Packet Descriptor. Therefore offsets must be correctly set when the PD is created in the RX Unit or by the CPU if packet length is modified during processing. Besides, the Fetching Unit must be compatible with the MPMC PIM interface and so operate at 200 MHz. This has been implemented in the same way that in the RX unit, using FIFOs for bridging clock domains. Notice that the TX Unit does not have an interface to the PPLB, only to the Packet Buffer.

Due to the many differences this introduces, two different TX units are instantiated for MPMC compatibility and for PLB access.

6.4 *Synthesis results - resources and clock frequency*

The SmartMem Buffer Manager has been synthesized and routed using the default tools provided by Xilinx, specifically their development environment software tools ISE, for IP-core design, and EDK, for the building a complex system-on-chip with buses, memory controllers, several IP-cores and GMAC IO ports. Behavioral simulations were undertaken at RTL-level using ModelSim and finally verified with simulation after post-place-and-route. For measuring performance, a FPGA-based hardware prototyping platform was implemented. The field-programmable device is a Xilinx XC4VFX60 Virtex-4 with a fabric array consisting of 25.280 slices and two IBM's PowerPC-405 processors embedded on-chip as hard-macros.



	Slices	%	LUTs	%	Max. Frequency (MHz)
Rx Unit + Master IF	653	2	763	2	145,751
Tx Unit + Master IF	897	3	1850	3	106,331
AM Unit + Slave IF	1254	4	2485	5	147,189
SBM Standalone	2804	*11*	5098	*10*	112,790

Table 8: Total HW resources and maximum possible clock frequency required for the BM implementation on a Virtex IV FPGA

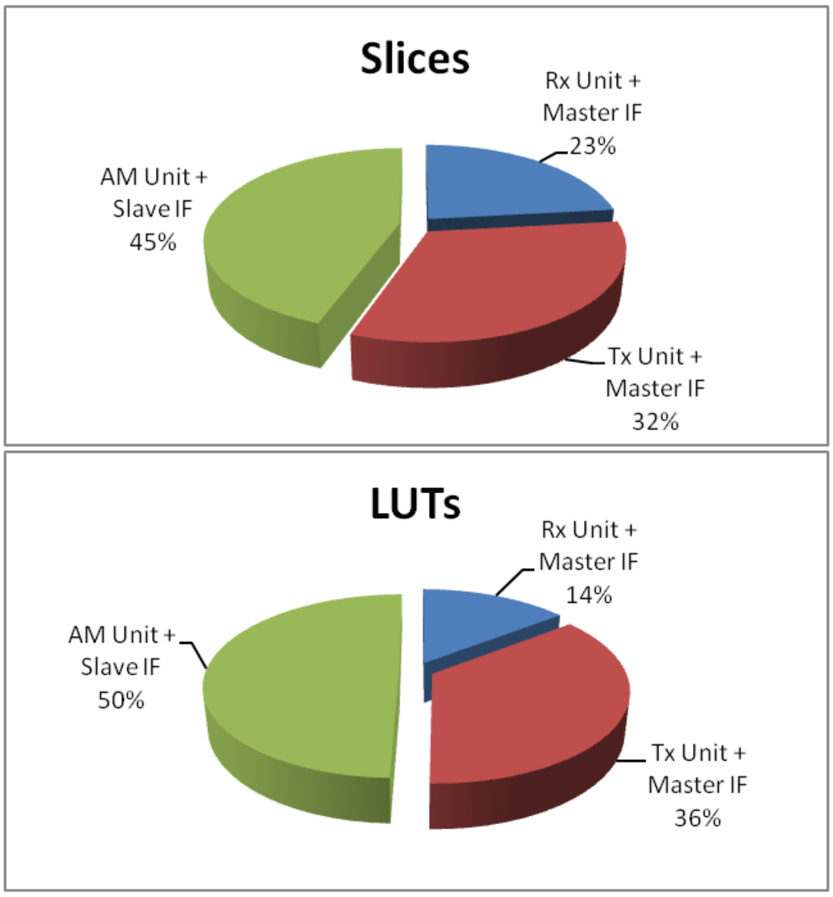
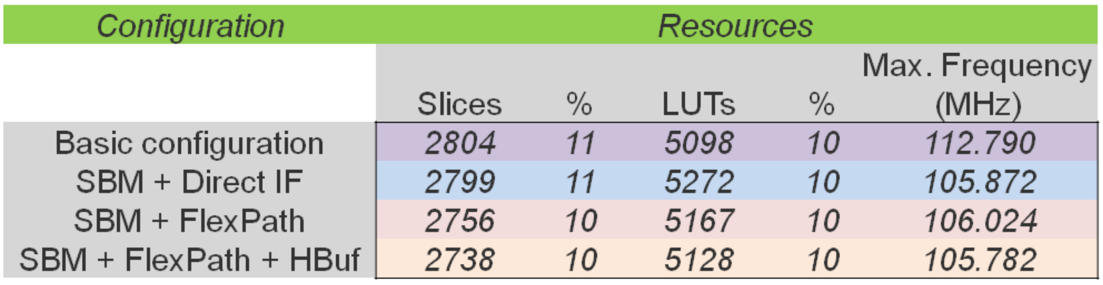


Figure 92: Contribution of the three main BM units to the total resource requirement of the BM (The figures are in terms of FPGA slices and Look-up Table (LUTs))

The FPGA is implemented on a ML410 hardware development board, with a 32-bit component DDR memory (64 MB) and a 64-bit DDR2 DIMM (256 MB). The two on-board 10/100/1000 Ethernet PHYs (MII/RGMII and SGMII) with RJ-45 connectors are of the highest importance for testing a networking device as the Buffer Manager. Several other important hardware devices as serial/parallel ports, PCI-express interfaces as well as a myriad of LEDs, switches and displays are also available, though not required for the present design.

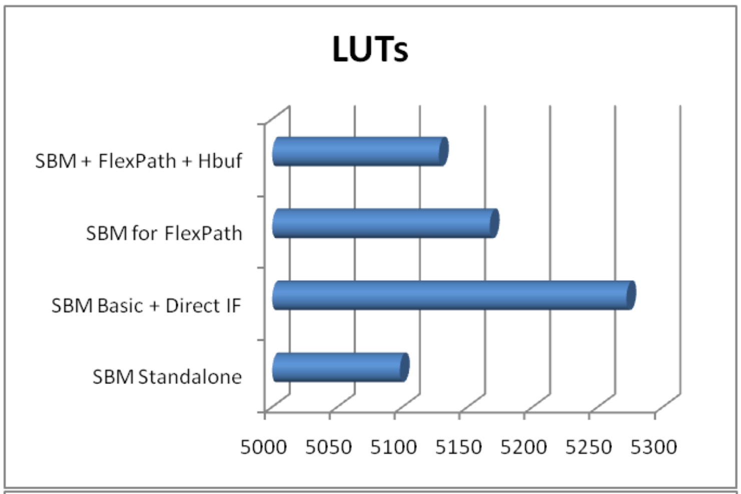
The implementation of the SBM on the Virtex-IV XC4VFX60 FPGA requires 2.837 Slices out of a total of 25.280 (~10%) running at 100 MHz. The figures below include the three bus interfaces, two masters and one slave. As observed in Figure 92, the main contributor is the Address Manager with 45% of the slices, followed by the RX Unit with 23%. The TX Unit is the smallest of the main units. In all three cases, frequency constrainers are those entities that interface with the system bus. The values are for the SBM with PLB-access to the Packet Buffer.

The SBM does not experience large changes in the number of required HW resources depending on the configuration selected for the synthesis (Table 9). The basic or standalone configuration requires the highest resources as it implements more functionality, for example the logic for bus interfacing (which is not present with the direct interface to the MPMC) or the PD queues, which are no required for the FlexPath configuration. All in all, little differences are observed regardless whether the parameter is LUTs, slices or clock frequency.



Configuration	Resources				
	Slices	%	LUTs	%	Max. Frequency (MHz)
Basic configuration	2804	11	5098	10	112.790
SBM + Direct IF	2799	11	5272	10	105.872
SBM + FlexPath	2756	10	5167	10	106.024
SBM + FlexPath + HBuf	2738	10	5128	10	105.782

Table 9: SBM resource requirement for different system configurations



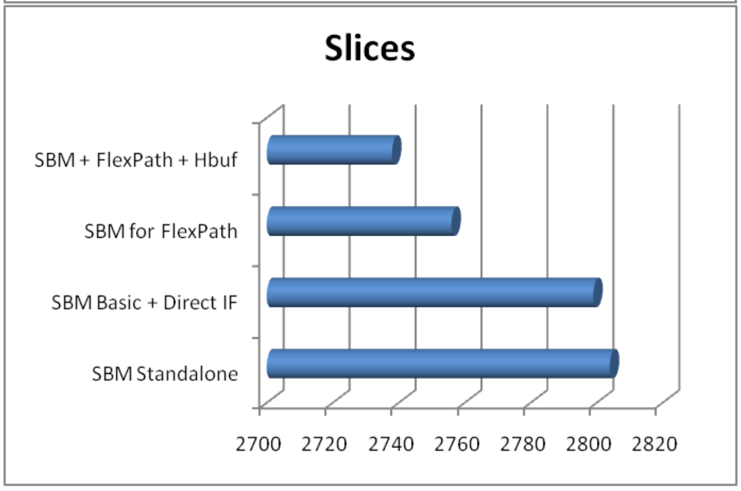


Figure 93: Comparison of Look-up Tables and FPGA slices required by the SBM in different system configurations.

6.5 *Measurements with the prototype*

6.5.1 Prototyping platform description

The system design is verified taking advantage of state-of-the-art laboratory facilities, with the Spirent SPT-2000A as **traffic generator**, which also helps to analyze traffic losses and system-crossing delay. For the on-chip system testing along with some measurements as performance, the Tektronix TLA7016 **logic analyzer** proved to be a very value resource during the debugging process.

An LIS-developed extension card, connect to the ML410 board through a RocketIO interface, allows for monitoring up to 40 internal signals, conveniently selected internally and routed to the output pins within the FPGA. Feeding at high-speed the described platform requires more traffic than what two GMACs are capable of providing. In comparison, the throughput of the system is in several configurations well above 1 Gbps. For that reason, traffic needs to be increased in some way in order to fully exploit the Buffer Manager capabilities. To overcome this throughput wall, a traffic replicator has been implemented between the GMAC buffer and the input of the Buffer Manager in the RX side. This unit replicates each packet received 16 times depending on the configuration and offers a satisfactory solution when the traffic pattern consists of only one packet size, as is our case in several simulations for reasons exposed later on.

The ML410 board offers two external memories, the 32-bit DDR-200 SDRAM memory and the 64-bit DDR-400 SDRAM. Though the theoretical performance expected from those devices can be quite high, in practice the selected memory controller is determinant in the final performance obtained from DDR-DRAM. The off-the-shelf Xilinx controller does not implement advanced features like address pipelining or bank interleaving. Only when this controller is replaced by the more-advanced MPMC (see 6.3.2), is the memory bottleneck finally removed and the BM delivers its best figures. In some cases, the Packet Buffer is alternatively mapped on on-chip SRAM. For each test, the selected memory device-controller is mentioned together with the reasons for its choice.

Figure 94 in the next side depicts the testing platform as used in all the experiments.

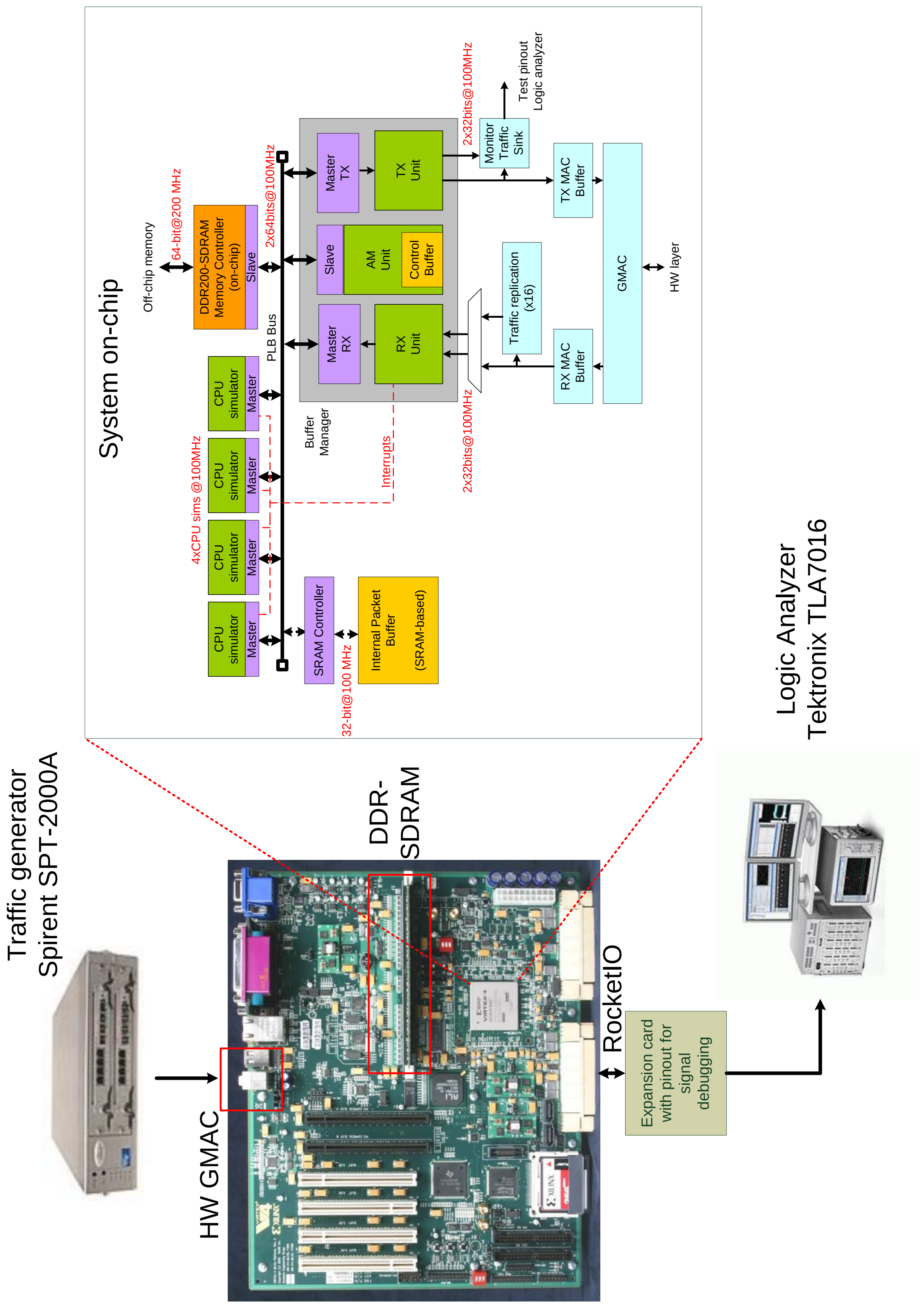


Figure 94: Test prototyping platform

6.5.2 Organization of the experiments

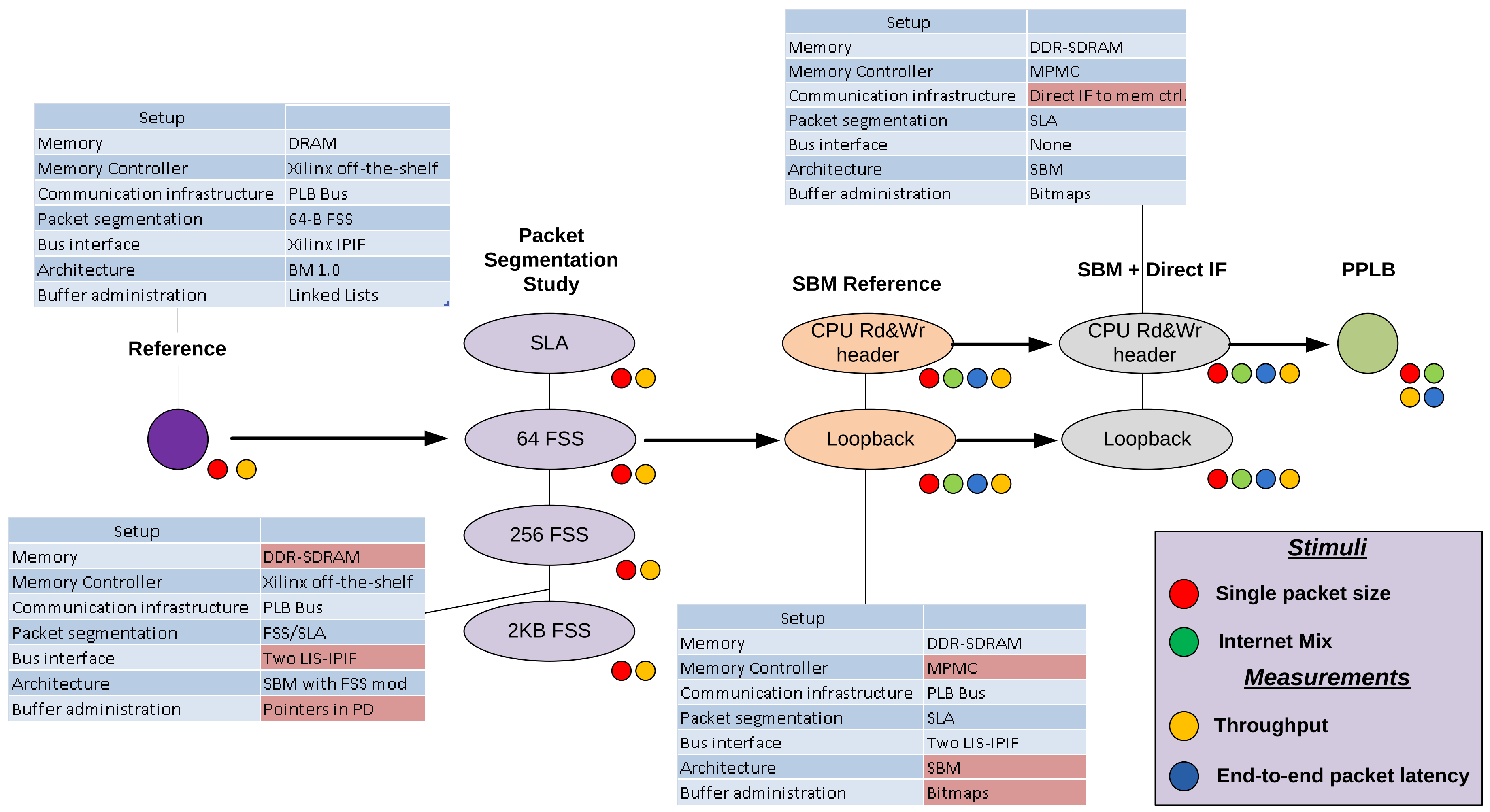
Several studies and measurements can be performed using the SmartMem prototyping platform. Through the following studies, we will mainly focus on the throughput of the Memory Subsystem which is the critical parameter to characterize the Memory Subsystem in an NP. Additionally, end-to-end packet delay is measured for the final SBM configurations while the Address Manager capacity to update the control data is tested as well in a standalone mode. This should demonstrate that the new features do not introduce unacceptable delays in the packet path across the NP. In all cases, the only CPU function consists in fetching the Packet Descriptor from the BM and sending it back for transmission (PD loopback). No forwarding application is run during the experiments.

Figure 95 shows how the following measurements are organized while depicting the setup of the system for each scenario and the main platform modifications between them. In that way, it is clearly depicted the contribution to the system's acceleration achieved after tuning step-by-step different parameters in the platform.

The system reference is a state-of-the-art platform with a Buffer Manager using 64-Bytes Fixed Segment Size (FSS) packet segmentation while taking advantage of linked lists of pointers for administrating the Packet Buffer (BM v1.0). With the measurements of that device, we will study some general behavior of packet segmentation, as for example the relation between the segments per packet and how does it affect performance.

The main contribution in this thesis is the introduction of multiple segment sizes for packet segmentation. Hence, we are after a scenario where FSS and SLA segmentations can be fairly compared in terms of performance but this arises many challenges posed by their different characteristics. While FSS uses linked list that due to its size can only be stored in DRAM, SLA takes advantage of bitmaps which comfortably fit in a small SRAM-based buffer. It seems unfair to compare a FSS system which must access a slow control buffer over the bus to SLA with fast access to control data. It could be argued that the benefits of multiple segment size segmentation are only due to the improved access to the Control Buffer. One solution is managing SLA with linked lists, with a mechanism similar to Figure 59. The drawback with that and the reasoning to discard Linked Lists for SLA stays the same: tracking multiple segment pools with lists of pointers is complex and requires many accesses to the Control Buffer. This would not be fair for SLA, as its benefits would be shadowed by unnecessary administration overload.

A mixed solution has thus been adopted. The SBM has been upgraded to support FSS segmentation. Linked Lists have been replaced by an administrative system where all segment pointers are stored in the Packet Descriptor. The size of the Packet Descriptor is adapted to the maximum number of segments an algorithm may generate. For instances, FSS-256 (the main competitor to SLA) requires 6 segments to store a 1518-Byte packet, and so 6 pointers need to be stored in the PD (2 in case of SLA). This administration method allows storing all control information on SRAM for both algorithms and so their results are only affected by their ability to access the Packet Buffer. After thorough consideration, this is in my opinion the fairest scenario for comparing throughput between FSS and SLA segmentation algorithms.



Setup	
Memory	DRAM
Memory Controller	Xilinx off-the-shelf
Communication infrastructure	PLB Bus
Packet segmentation	64-B FSS
Bus interface	Xilinx IPIF
Architecture	BM 1.0
Buffer administration	Linked Lists

Setup	
Memory	DDR-SDRAM
Memory Controller	Xilinx off-the-shelf
Communication infrastructure	PLB Bus
Packet segmentation	FSS/SLA
Bus interface	Two LIS-IPIF
Architecture	SBM with FSS mod
Buffer administration	Pointers in PD

Setup	
Memory	DDR-SDRAM
Memory Controller	MPMC
Communication infrastructure	PLB Bus
Packet segmentation	SLA
Bus interface	Two LIS-IPIF
Architecture	SBM
Buffer administration	Bitmaps

Setup	
Memory	DDR-SDRAM
Memory Controller	MPMC
Communication infrastructure	Direct IF to mem ctrl.
Packet segmentation	SLA
Bus interface	None
Architecture	SBM
Buffer administration	Bitmaps

Figure 95: Measurements with SmartMem

Once the superiority of the segmentation algorithm chosen has been evaluated, the SBM is scraped of the FSS capability and adapted to work with FlexPath and the PPLB as explained in the previous section. With the SBM, first we take as a reference a system configuration where packet data is transferred over the PLB bus (with the PD loopback). The measurements are repeated to evaluate the additional load introduced when the CPU simulator fetches and writes-back the packet header (40 Bytes) from/to the Packet Buffer. Both scenarios (loopback and header fetching) are repeated after replacing the PLB bus by a direct interface to the Multi-Port Memory Controller (MPMC).

Finally, the Packet Processor's Local Buffer (PPLB) is introduced in the system and directly connected to the CPUs, 40 Bytes per packet there injected and then this information is flushed to the Packet Buffer. The final measurements with the PPLB demonstrate the acceleration of the multiple improvements introduced within the SmartMem project. In most of the experiments, the stimulus selected is a continuous stream of same-sized back-to-back packets generated by a traffic generator. The packet size is tuned to cover the full range of Ethernet packet sizes (64 up to 1518 Bytes) to thus better understand the behavior of the system for different packet sizes, which is of special interest for packet segmentation algorithms. In the final configurations, new stimuli are injected in the system to measure the response to the SBM with different internet mixes. End-to-end average packet latency is reported as well.

6.5.3 Measurements

6.5.3.1 System reference: Fixed-Size Segmentation 64-Bytes and Linked Lists

The investigations in this chapter begin with the throughput measurements of the Buffer Manager segmenting packets using only 64-Byte segments (FSS-64), a method that has been common in many NPs. In addition, the Packet Buffer administration is done with Linked Lists of pointers that chain all segments belonging to the packet, as well as using an additional linked list for tracking free segments. The architecture of this state-of-the-art Buffer Manager has been already presented in [40]. We can use these results as a base to build upon introducing step-by-step enhancements and thus easily capture the contribution of each modification to the improvement of system's performance.

In Figure 96, the Buffer Manager has been stimulated with single-sized packets and its performance measured with the Packet Buffer mapped first on DRAM and then on SRAM. There are several general characteristics of packet segmentation influence over performance that can be observed in most of the experiments in this chapter. The most remarkable is the trend of data throughput to increase with the packet length. Managing data in large packets is more efficient than doing it when data is fragmented in multiple small ones. However, there is a pattern that breaks the linear increasing of throughput with packet size. **Whenever a new segment is required, performance is slightly degraded** in comparison to the previous size because of the effort of managing an additional segment. Additional control overhead and the inefficiency of storing only some bytes in the final segment generate this degradation. This is an important observation that confirms a relation between the number of segments per packet and performance and so confirms that minimizing the number of segments is equivalent to increasing the Memory Subsystem performance.

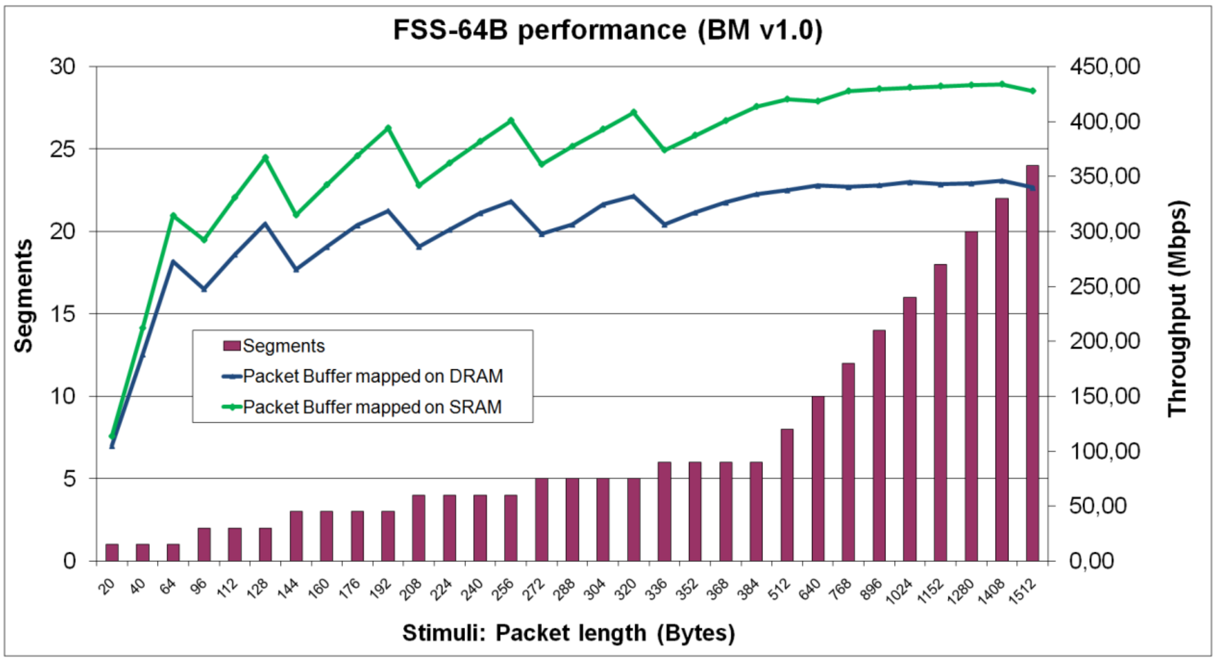


Figure 96: Maximum throughput with FSS-64B and Linked Lists (BM v1.0)

Another interesting conclusion can be yielded when mapping the Packet Buffer alternatively in SRAM and DRAM. A faster buffer improves the operation regardless of packet size and the number of segments, but this increase is bigger for large packets. This is an indicator of the influence of multiple factors other than the performance of the memory device. In this group we can include linked list management, PD manipulation and access delay to the bus. The smaller the packet is, the biggest the importance of administration overload. On the contrary, longer packets require less administration effort and so achieve better throughput.

The most important part in this discussion is realizing that there are several factors constraining the performance of the system and that they are most of the time closely related with the packet size. We will find in the following measurements more and more varied bottlenecks for stimuli with short packets than for stimuli with large packets. And for this reason, in all experiments monitoring performance, measurements have been repeated for a wide range of packet sizes.

6.5.3.2 Study of packet segmentation with Fixed-Size Segmentation and the SLA algorithm

Following the previous conclusion that more segments per packet lead to worse throughput, this experiment pursues to demonstrate that by using multiple segment sizes it is possible to achieve a performance close to that achieved using only one large segment per packet (while simultaneously getting high storing efficiency). For that experiment, the Packet Buffer is mapped on a 64-bit DDR200-SDRAM interfaced through the Xilinx's IP Core library controller. The maximum burst length is not constrained in this experiment hence memory access is optimized for long segments.

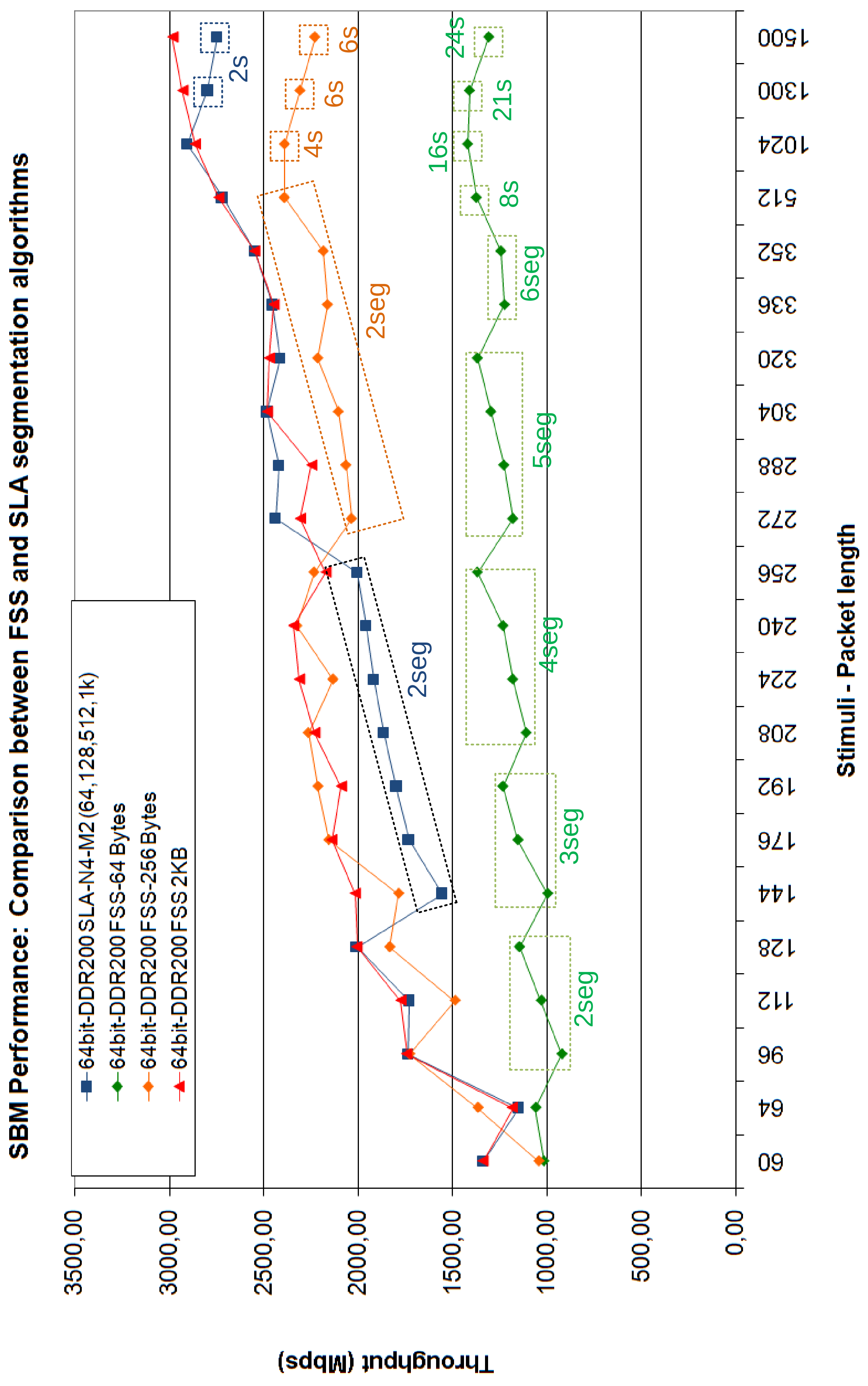


Figure 97: Performance comparison of SLA and FSS with three segment sizes

Figure 97 shows the throughput measured at the system's output with four different segmentation strategies: FSS with segment sizes 64, 256 and 2 KB, and our SLA algorithm with four segment sizes (64, 128, 512 and 1 KB) constrained to a maximum of two segment sizes per packet (SLA-N4-M2). We proceed as in the previous experiment by stimulating the system with a continuous stream of fix-size packets, which helps to better distinguish the effect of different packet sizes on performance. The SLA algorithm achieves better results than state-of-the-art FSS 64 (the only FSS with similar storing efficiency) independently of the packet size chosen. Performance is between 9 and 109% higher, increasing this percentage as packet sizes get larger. A similar observation can be made when comparing SLA and FSS-256 figures, though SLA's performance is only better when SLA takes advantage of the 512 and 1024-Byte segments, which FSS-256 cannot. Again, the number of segments per packet is the decisive factor that drives performance, so the best throughput is invariably obtained when the complete packet fits into one segment.

The reason for this effect is the activation/pre-charge latencies when accessing DDR-SDRAM as well as additional control effort. Every segment requires one memory access, which in turn requires activating the corresponding memory bank as well as writing back data to refresh the transistor value (access is destructive in one-transistor memories). Requesting large chunks of data reduces latency and by extension optimizes the system's performance.

Moreover a small amount of large accesses improve data transmission over the bus due to the reduction of associated arbitration, handshake and acknowledge latencies. Although it must be taken into consideration that large bursts increase the chance of collisions (and thus of higher latency) when requesting bus access. Finally, algorithms with low average number of segments per packet need smaller Packet Descriptors, which reduces the load on the bus. FSS-64/256 algorithms require enlarged PDs to keep up to 24 segment addresses while in comparison, SLA-N4-M2 generates a maximum of two segments per packet.

6.5.3.3 SmartMem Buffer Manager with Bus and Direct-IF to Multi-Port Memory Controller

Once demonstrated the benefits of the multiple-size segmentation also in terms of performance, we proceed to evaluate the SBM with the MPMC and without the FSS capability. In the following experiments, we will inject first a packet pattern composed of only single-sized packets and then packet mixes that closely approximate those to be found in internet. Moreover, end-to-end latency is another parameter of interest to characterize the SBM and how the new methods impact the delay a packet takes when crossing the NP.

We pursue now to measure the total performance of the SBM with two different system configurations: first when the Packet Buffer is accessed over the bus and afterwards when a direct interface between the SBM and the Packet Buffer is implemented. This is done because as results in Figure 98 demonstrate, the bus quickly becomes the bottleneck in the system for all traffics. This new bottleneck is made evident because the improved operation of the SBM together with the introduction of a faster memory controller, which is undesired for exploring the behavior of the memory subsystem. To bypass the bus, the SBM is connected to two direct interfaces of the MPMC, while the slave interface to the bus is maintained for management. The benefits of this are easily observed if we compare results in the upper graphic in Figure 98. Performance with the direct IF is roughly three times that of the system with the PLB bus.

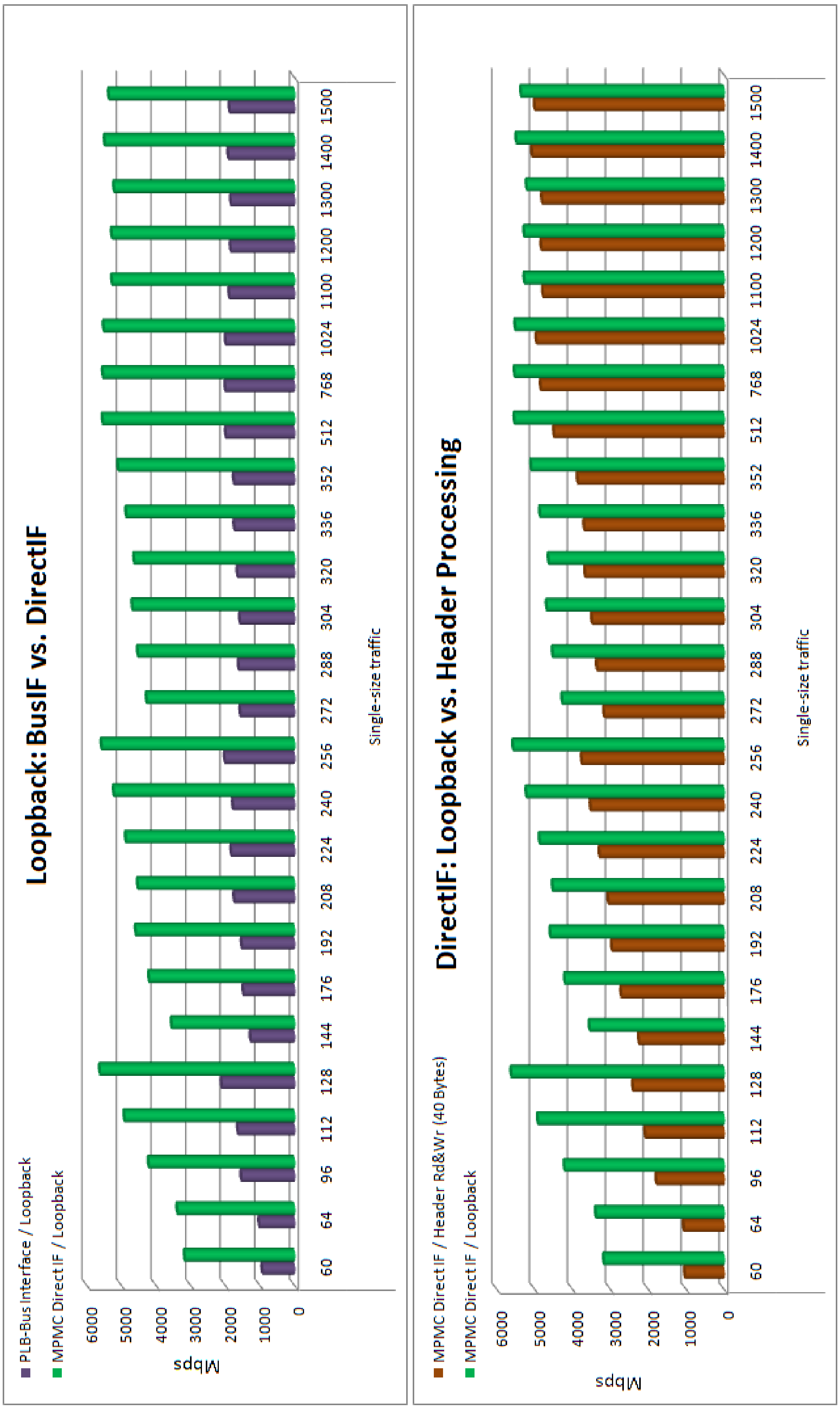


Figure 98: SBM with Direct IF to MPMC. CPU loopback and CPU header processing

We have to annotate poorer performance when comparing the throughput in the upper graph in Figure 98 and the throughput depicted in Figure 97 for the SLA algorithm. This is an effect introduced by the MPMC, which must schedule multiple ports to access DRAM with input FIFOs. These queues introduce longer latency in every data request than the previous controller, which impacts negatively data transferring over the PLB Bus and the overall system throughput.

In all previous experiments, the operation of the CPU simulator has been maintained very simple to focus on the interaction between the BM and the Packet Buffer. There, multiple CPUs received each an interrupt when there was a packet ready for processing after storing in the Packet Buffer had been accomplished. Then the CPU fetched the PD from the BM and immediately sent it to the transmission queue. This correctly reproduces the PD operation in the SBM as well as the additional load on the bus when transferring the descriptor. However, this cannot capture the impact on the Packet Buffer that CPUs generate when fetching packet data for processing. Neither the extra load of the processing results being written back in the Packet Buffer.

For that reason, the results shown in the lower graph in Figure 98 are very interesting as well. There, we study the throughput degradation when the CPU reads 40 Bytes of data (approximate size of the IP and TCP Headers) from the Packet Buffer and then writes 40 Bytes again back to memory (without processing delay). Now the load on the Packet Buffer clearly reduces the total number of packets the system can forward (Figure 99), regardless of the stimuli offered. This statement needs some qualifying, as degradation is worse for patterns with short packets than for large packets. The obvious reason for that is the relative lower impact of manipulating the header when packets are 1500-Bytes long than for each single 64-B packet. The important conclusion we can yield from this experiment is the important contribution to the Packet Buffer load of the CPU accessing to a DRAM-based buffer. And moreover this is in favorable conditions as impact is stronger when packet processing requires the entire packet (payload processing) or when packets are resized after processing.

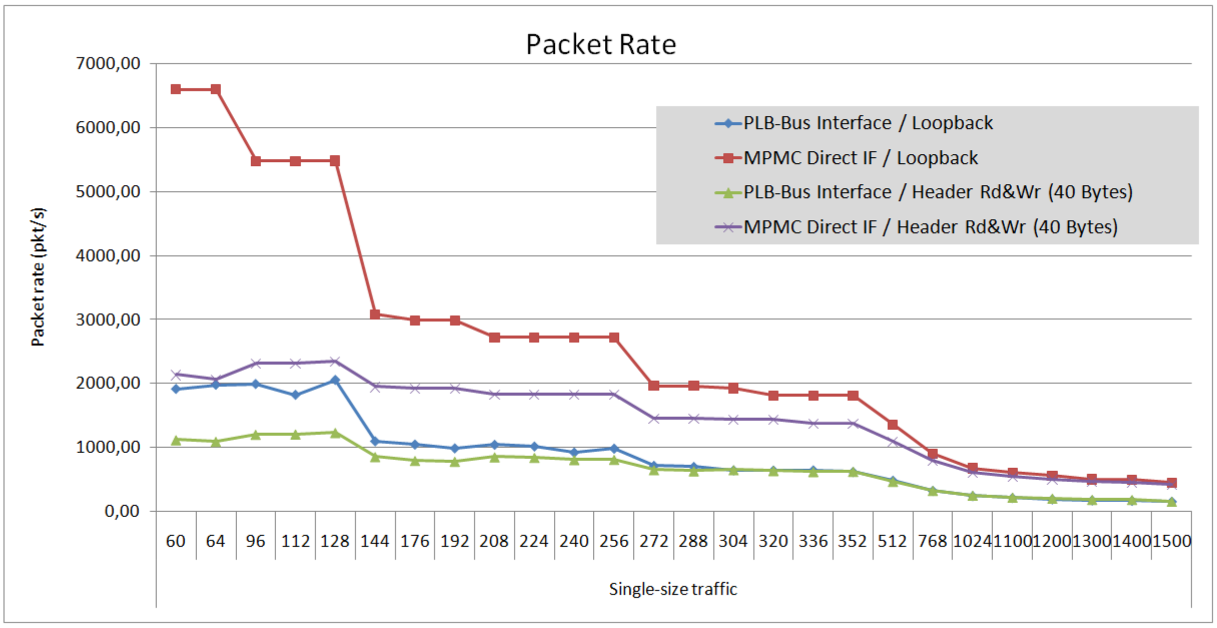
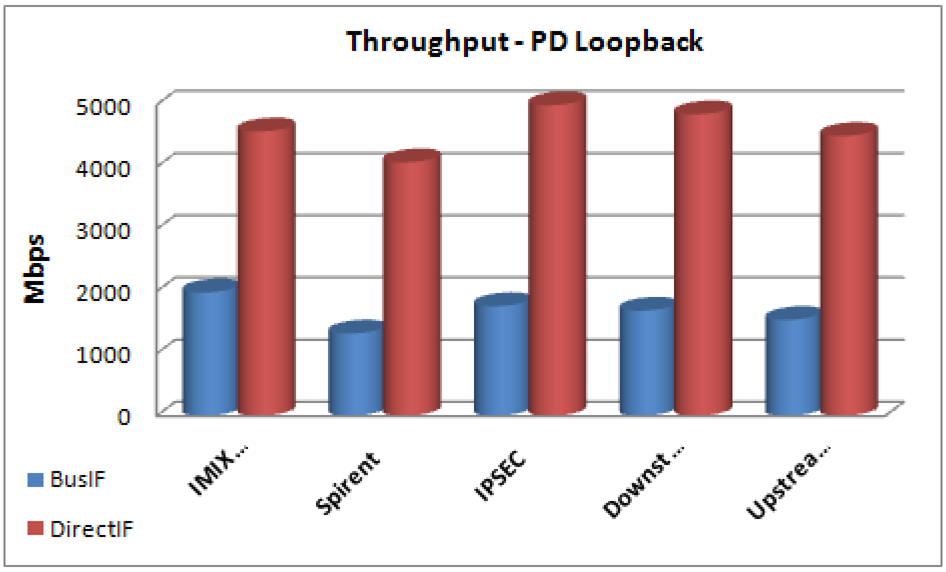


Figure 99: Packet rates measured

6.5.3.4 Prototype performance with different Internet traffic mixes

The same conclusions hold when the stimulus of single-sized packets is replaced by commonly-used packet patterns in network tests (Figure 100). The same platform has been offered with five different packet patterns: Internet IMIX, Spirent internet mix, IPSec mix, downstream mix and upstream mix. These patterns present strong differences in their packet size composition. While the Internet IMIX and the Spirent mix accurately reproduce the average pattern to be found in Internet (with the highest share for small packets and some medium and large packets), others like downstream traffic are mostly composed of large packets. The contrary is true for upstream traffic, usually made of mostly small control packets. This represents a comprehensive-enough range of traffics to generalize the conclusions to all possible traffic scenarios to be found in today's IP networks.



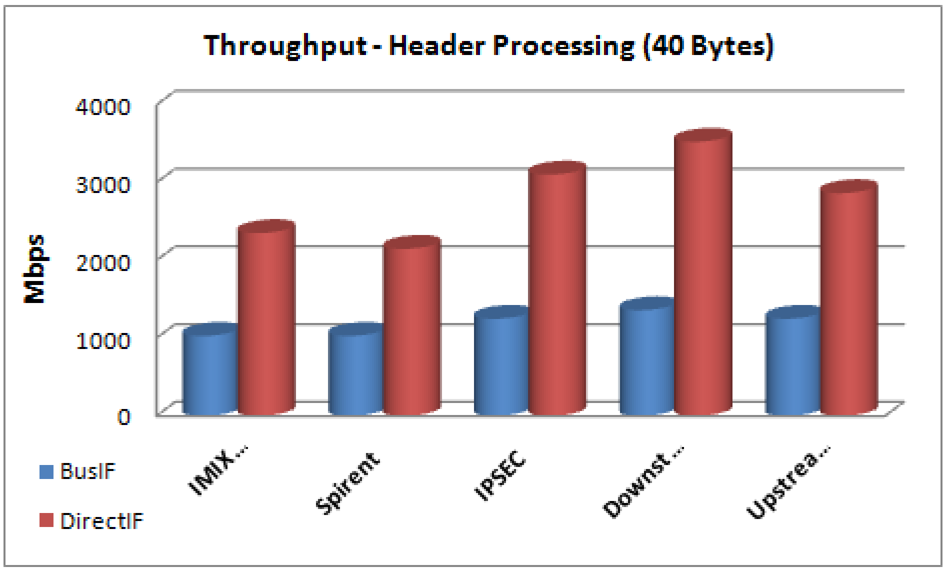
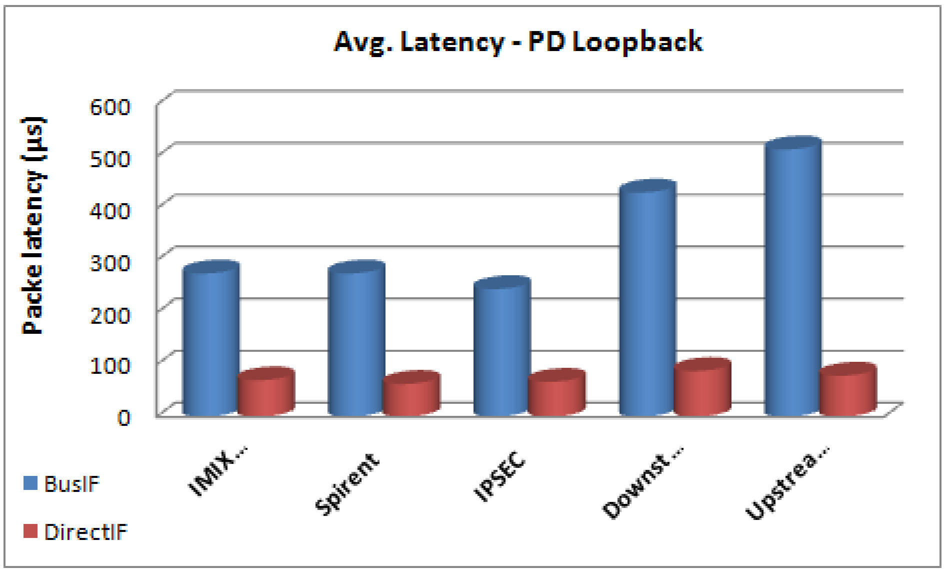


Figure 100: SBM performance with different internet traffic mixes

The experiments with the CPU loopback and the CPU fetching/writing the packet header have been repeated with five different traffic mixes. The system configuration has been modified to access the Packet Buffer both over the PLB bus and the Direct IF to the memory controller. In total, it makes up to four different scenarios which are depicted in Figure 100. Results confirm the conclusions with single-sized packet patterns. Direct IF achieves again more than twice the throughput than the PLB bus, as well as performance is greatly impacted when the CPU accesses the Packet Buffer. Remarkably, all throughputs with the direct interface and the loopback are between 4 and 5 Gbps which demonstrates the performance stability regardless of traffic pattern. In comparison, once the CPU interacts with the Packet Buffer performance degradation can even halve the throughput, as in the Spirent mix, where throughput is reduced to around 2 Gbps, due to the higher percentage of small packets. Other mixes on the contrary are relatively less impacted by header fetching as for instances downstream traffic, which is degraded from 4,8 Gbps down to 3,4 Gbps.



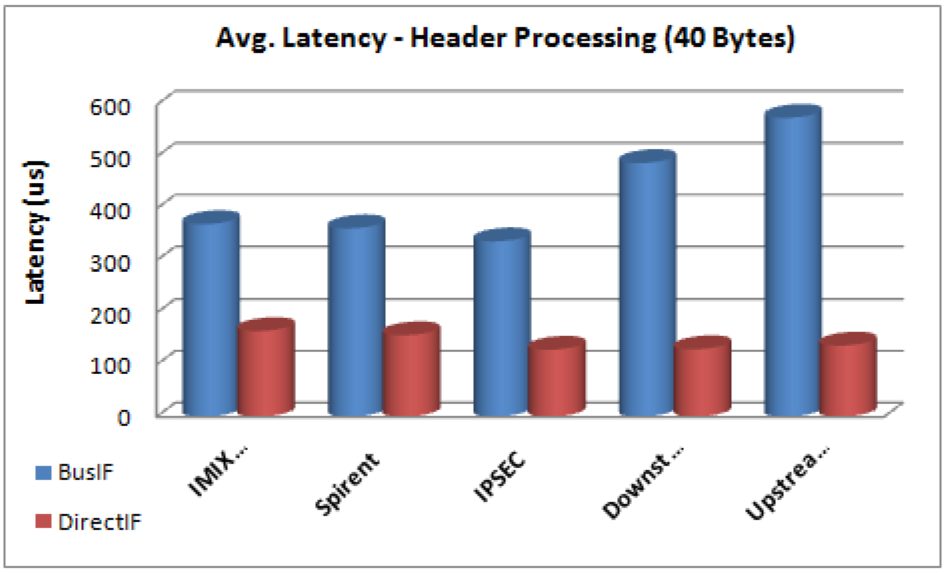


Figure 101: End-to-end packet latency for different traffic mixes

Figure 101 focuses on end-to-end packet delay for again the same traffic mixes. The measurements are automatically done by the Spirent Traffic Generator by marking each packet and then controlling the delay it takes to see the same packet on an incoming interface after crossing the SmartMem platform. Delays are between 250 and 560 µs (bus) while the direct IF greatly reduces this delay, down to 110-140 µs including header fetching/storing (no packet processing delays are generated). This is another good reason to introduce the direct interface in the system, as delays are clearly between 2 and 6 times lower.

6.5.3.5 **Packet Processor Local Buffer performance**

High and variable loss in performance is an undesirable behavior when the CPU accesses the Packet Buffer. Packet delay is also increased, even with the direct interface to the MPMC. To tackle this loss, SmartMem introduces the PPLB as described in Chapter 0 and with the SBM configuration proposed in 6.3. The experiments are repeated now with the SBM injecting 40 Bytes of each packet in the PPLB, where the CPU reads and writes-back exactly the same 40 Bytes as in the previous experiment but this time in its own local buffer (see 5.3). The results are shown in Figure 102, referenced to the throughputs obtained with the SBM with and without CPU accessing the Packet Buffer (only Direct IF).

Results show a very interesting throughput for the PPLB configuration. The data rates are now clearly higher than with the CPUs directly accessing the Packet Buffer. In fact, we observe practically the same performance as with the PD loopback. In other words, the performance loss observed before when the CPU reads and writes the 40-B packet header from the Packet Buffer now all but disappears with the PPLB in the system. This confirms that **the PPLB architecture can successfully eliminate the load on the Packet Buffer produced by the CPU**, to the extent that the system now behaves as if no packet processing were taking place. In Figure 103, we can again yield similar conclusions for the five packet mixes already tested in the previous section. Little if at all performance degradation is observed with the PPLB in comparison to the PD loopback, which proofs the efficiency of the new Header Buffer. But if this is the case in terms of throughput, Figure 104 shows a similar analysis for end-to-end packet latency. Again latencies are only slightly higher than those observed for the PD loopback, while much better than with the CPU directly accessing the Packet Buffer.

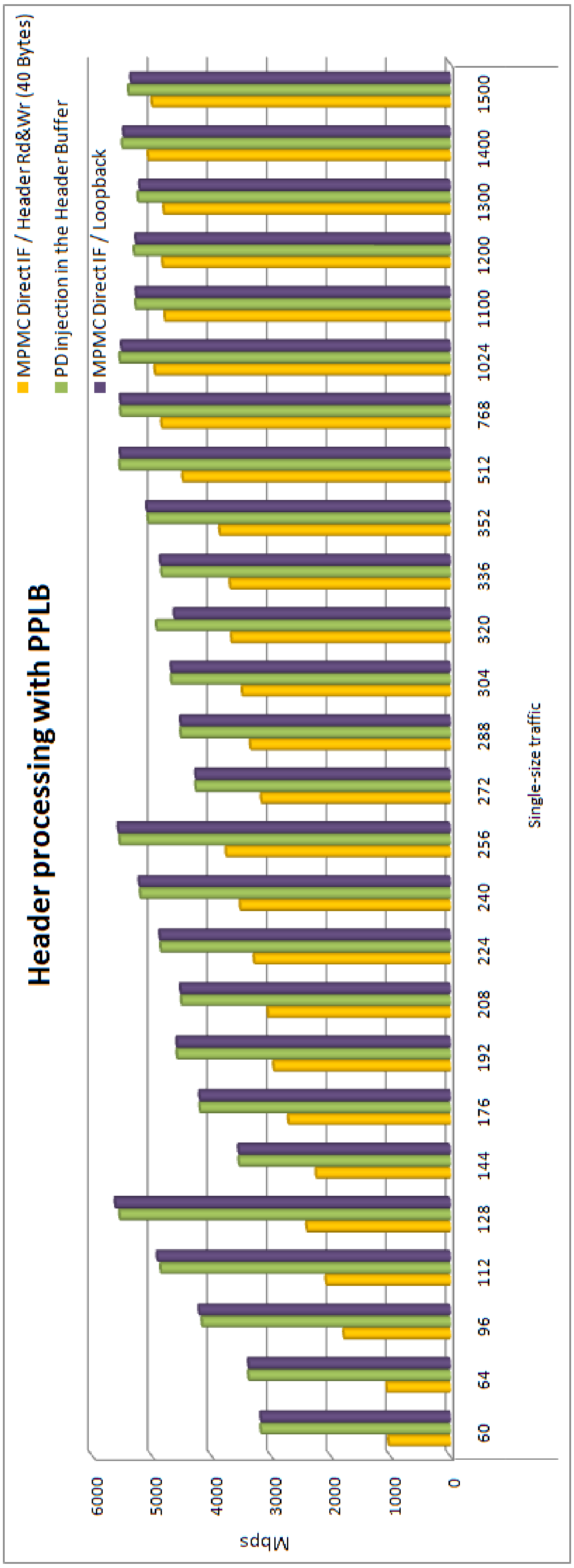


Figure 102: Performance with the PPLB with Direct IF to the memory controller

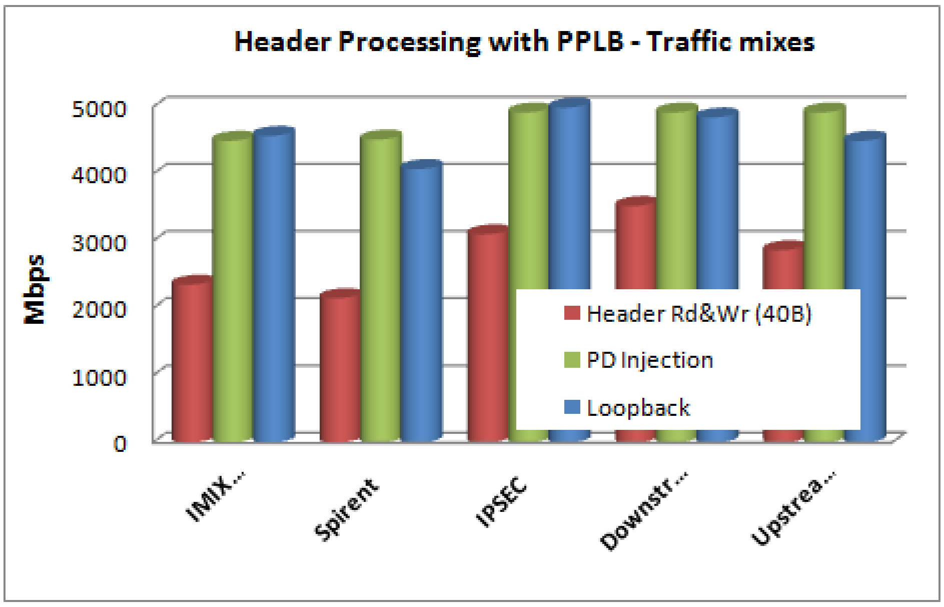


Figure 103: Performance with the PPLB and multiple traffic mixes as stimuli

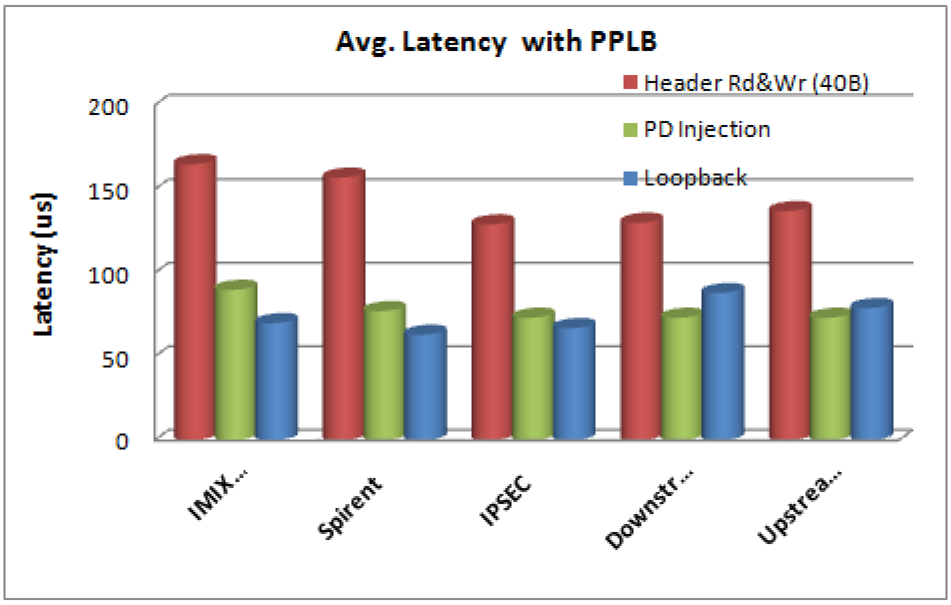


Figure 104: End-to-end packet latency with the PPLB, without processing latency.

6.5.3.6 **The Address Manager in standalone mode**

Until now, all experiments refer to the performance of the system as whole. It is however of particular interest to have some data about the capacity of the Buffer Manager to perform packet administration. This study gives us insight into the maximum capacity of the Address Manager if it were operating stand-alone, for example together with a DMA unit. Therefore, Figure 105 shows the performance of the AM in a standalone mode, as observed in RTL simulations with ModelSim. These observations confirm that the Buffer Manager does not exhaust all the performance the AM is capable of.

The graphic below depicts the response of the AM when receiving continuous and simultaneous memory allocation/discard requests. This is repeated for four segment sizes to measure the capacity of the AM under maximum stress. In these conditions, the AM achieves between 10 and 16 millions of address malloc and discard operations for the largest segment size, 1024 Bytes. Besides, we observe that using 4-kB memory blocks lead to higher performance as a consequence of obtaining more addresses from a single block allocation. In terms of amount of memory allocated, requesting 64-byte segment allocations is the most demanding scenario, even though the number of addresses obtained in one block allocation is the highest. However, the AM is still well capable of servicing in a Buffer Manager targeting line speeds over 10 Gbps, as 20 Mops/s times 64 Bytes allow for 12,8 Gbps.

Bitmaps are not only small but also a very fast memory administration method, which is enabled by the introduction of the SLA algorithm and the hard bound in the number of segments per packet. This reinforces the claim that SLA not only accelerates the access to the Packet Buffer, but even more decisively improves control data administration.

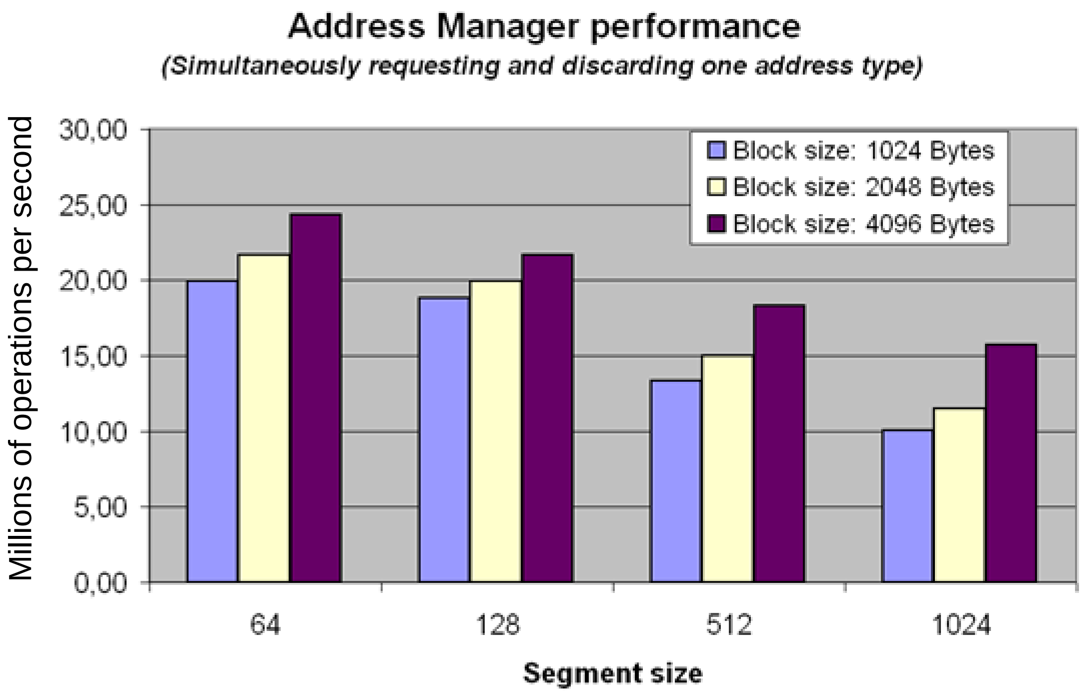


Figure 105: Maximum address discard rate with the Address Manager in standalone mode

6.5.3.7 Performance results of FlexPath and SmartMem

The SBM is compatible with the FlexPath project (see 6.2.3). To test compatibility and evaluate performance, the SBM has been incorporated to the Network Processor Prototyping Platform (NP3) and its functionality successfully tested in this environment. The system was tested in multiple traffic scenarios as well as with multiple FlexPath configurations. The total throughput obtained is not of high scientific value to the SmartMem project as the FlexPath pipeline clearly constrains performance. However, it is very interesting to state that first, the SBM is fully compatible and stable when working with FlexPath. Second, the SBM can deliver more-than-enough performance for the FlexPath requirements even when administering context data as well as packet data. And last but not least, we state that the SBM can work within a platform capable of performing all operations needed for IPv4 packet forwarding, a feature that in previous experiments has not been included.

Figure 106 depicts the throughput obtained in three FlexPath scenarios. The reference system consists of two CPUs aided only by the SBM, with best-effort load balancing between them (called also spraying). A second test is done with the FlexPath co-processors presented in section 6.2, which generate context data to aid the CPU in the IPv4 forwarding operation (CII&CIO packet spraying in the figure). A third setup monitors the benefits of the AutoRoute feature, whereby packet bypass the CPU completely as all required processing is done by the co-processors.

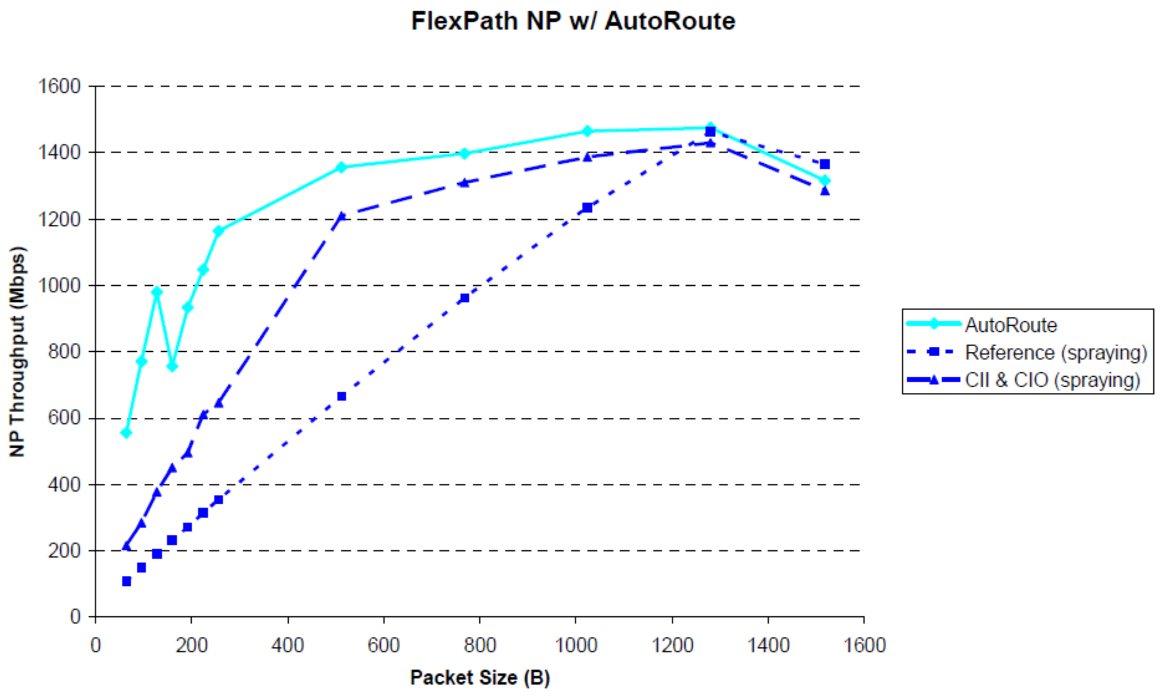


Figure 106: Performance of the FlexPath platform with the SBM.

About the results obtained, first mentioning that throughput is always below 1,5 Gbps due to the constrained forwarding capacity of the two CPUs, much lower than in previous experiments with the SBM. In the initial reference, the maximum packet rate the CPU-cluster is capable of is limited to 170 Kpps. When the HW accelerators of FlexPath are included, packet rate is substantially increased. An additional packet forwarding acceleration can be observed when AutoRoute packets by-pass the CPU-core. It is of special interest to observe the two bottlenecks in such a system: for small and medium packets, the forwarding capability of the CPU/FlexPath units prevents faster throughput. For the largest packets (>1000 Bytes), the PLB Bus becomes saturated so the full packet forwarding capabilities of the CPU/FlexPath are not used. This saturation is partially due to context data as well as extra control data sent over the system bus, which compete with packet data for getting access to the same saturated communication infrastructure.

6.6 Summary

This PhD Thesis is completed with the implementation of multiple segment size packet segmentation in a FPGA-based prototyping platform. The selected packet segmentation algorithms together with the bitmap-based packet administration mechanism have been implemented in the SmartMem Buffer Manager. This chapter discusses the architecture as well as numerous architectural design choices made during the implementation of the SBM. Moreover, it also describes the modularity of the SBM architecture, which enables its implementation in large diversity of systems. This includes compatibility with the PPLB Buffer, the FlexPath project and high-speed MPMC from Xilinx.

The second part of this chapter focuses on the experiments done with the prototyping platform. Mainly performance was tested, using a combination of single-size back-to-back stimuli and multiple traffic mixes that accurately reproduce the traffic pattern to be found in current networks. End-to-end latency has been also measured to proof that the SBM is not only a high-performance accelerator but a low-latency unit, too.

The results verify that throughput with the SLA algorithm achieves rates very close to FSS-2KB and thus validate the simulation results in Chapter 0. SLA algorithm is clearly faster than FSS-64 and FSS-128 with all packet sizes while achieving better results than FSS-256 with most packet sizes, except those in the range between 129 and 256 Bytes. This throughput acceleration is achieved with storing efficiencies in the Packet Buffer above 90%. In addition, SLA allows the introduction of Bitmaps for administering the buffer. The performance observations with the AM unit, which implements that method, demonstrate that the bandwidth required from the Control Buffer is critically reduced, a benefit that adds to the buffer size reduction discussed in chapter 4.

The final contribution in this chapter regards the validation of the SBM architecture as suitable for high-speed NPs, achieving rates over 5 Gbps even considering that clock frequency is only 100 MHz (FPGA-based tests). This was achieved with the introduction of a direct interface to an advanced memory controller (MPMC), thus bypassing the bus and reducing memory access latency.

A final test introduced the PPLB and direct injection of packet headers to the CPU local buffer. The throughput obtained with the PPLB is twice that obtained when the CPU accesses the Packet Buffer, which validates the benefits of our fast local buffer for the CPU cluster.

7 Conclusions and outlook

SmartMem is an advanced Memory Subsystem for Network Processors that implements strong optimizations in two key aspects of the system: performance and management efficiency. Three different areas have been addressed during this research, being the introduction of packet segmentation using multiple segment sizes the central core of the project. Besides, a new memory administration paradigm is enabled by the introduction of these innovative packet segmentation algorithms, which not only satisfactorily meet the additional functionality requirements but also reduce by an order of magnitude the size of the Control Buffer. Finally, the project studies the feasibility of injecting parts of the packet in a processor's local memory. This achieves the elimination of memory access latency when the processor retrieves packet data for processing, thus acceleration both packet forwarding and memory accessing.

Regarding packet segmentation in Network Processors, this work presents two new packet segmentation algorithms that slice the packet using multiple segment sizes instead of only one, thus reducing the number of segments per packet whereas maintaining high storing efficiency (always above 90%). We observe that with longer segments, longer accesses to memory increase performance considerably, even doubling average data throughput for some packet mixes while obtaining high storing efficiencies. This is achieved by selecting the best combination of segment sizes that minimizes the number of segments per packet without reducing storing efficiency in memory. An important conclusion is that those algorithms with one small, one medium and one large segment size systematically obtain the best results. When comparing MEO and SLA, both deliver similarly satisfactory results. However, the SLA algorithm is considered superior because of its ability to provide a hard bound in the number of segment per packet, which is a critical requirement for the control structures.

In order to administer a packet buffer that must contain multiple segment sizes, it is necessary to first characterize the behavior of traffic in Internet. Some state-of-the-art NPs include the option to have several buffers with different segment sizes. The packet size is used as a criterion to select the portion where the packet is stored. Inconveniently, this is a static solution as the portion size is configured upfront and so cannot adapt itself to variable packet patterns, as the study in chapter 4 demonstrates. To tackle the variable requirements of segment sizes, we propose a dynamic memory organization where the buffer is organized in blocks. These large blocks are then assigned dynamically to each portion according to their usage levels, with a mechanism that maximizes the availability of free blocks. In simulations with real internet traffics we have observed an excellent memory allocation response to dynamic memory requirements, even under demanding scenarios.

One of the major contributions of this work is achieved because the SLA algorithm enables constraining the maximum number of segments per packet to two, hence control data management can be drastically simplified. With only two segments, the packet descriptor contains all the information necessary to reassemble the packet on the egress side. Since no linked lists are required, the burden of their management is avoided. Only one bit is necessary per segment to keep their status (free/busy). The introduction of bitmap-based control structures replacing state-of-the-art linked lists leads to a ten-fold Control Buffer size reduction. Such a small buffer can be comfortably implemented on SRAM, thus further accelerating control update performance.

A final contribution of this work consists in the study of a Packet Processor's Local Buffer (PPLB). The PPLB tackles the promising concept of data injection in a processor's local memory. This scheme has two aims: avoiding short and thus inefficient accesses to the Packet Buffer and second preventing CPU-stalling due to cache misses. With the PPLB, cache misses never occur as packet data is preemptively pushed into the CPU's local buffer thus made ready when packet processing begins. In the tests realized, when the CPU accesses the Packet Buffer to retrieve packet data, the overall performance of the system is critically impacted (values depend on the packet pattern). The introduction of the PPLB completely removes this overload. The throughput measured is comparable to other tests without the CPU accessing the Packet Buffer, thus yielding a considerable Packet Buffer acceleration. Besides, CPU processing is also improved as memory latency to access data in memory is no longer slowing the packet forwarding operation.

The previous concepts have been realized in the SmartMem Buffer Manager (SBM) prototype. This unit has been coded in VHDL and tested on an FPGA-based platform to confirm the performance improvement thus validating the benefits of using multiple segment sizes for packet segmentation. Insights from a complete SystemC simulator have been considered in the design of the accelerator's architecture. The prototype is modular and so can be configured differently to be compatible with different system elements as co-processors, memory controllers and bus interfaces. Finally, mentioning that the SBM requires for an standalone configuration 1804 Slices and 5098 LUTs, for a maximum clock frequency of 112 MHz.

This work concludes suggesting several pointers for a further improvement of the Memory Subsystem in NPs. The idea of intelligently pushing only the required data into a local CPU buffer must be deeper investigated. In particular, this should be tested with a real operating system for routers (like IOS-XR) running on the CPU and while dealing with a large variety of protocols. This investigation should be completed linking together the different transport protocols and which parts of the packet are required for packet forwarding. Remember that the PPLB requires advanced packet classification in front of the CPU so it is necessary to establish a recognition method that allows successfully injecting parts of the packet in the PPLB. This probably has similarities with another hot spot of research in networking: Network-based Application Recognition (NBAR) or deep packet inspection. The study of the feasibility of NBAR methods for their application in packet injection will for sure yield fascinating results.

***

The spreading of broadband links, even until last-mile connections, together with the popularization of high-end internet-capable mobile devices is and will keep being, the driving force behind the increasing demand for high-capacity data networks. This huge demand of bandwidth requirement is accomplished by deploying new optical fiber links, an effort that is however constrained by the limited forwarding capacity of the routing nodes. There, Network Processors are implemented on linecards to provide sufficient processing capacity to keep pace with growing line rates, while simultaneously offering programming flexibility. A new generation of packet processor units must emerge to meet the breakneck speed of Internet expansion.

This thesis begins reviewing recent publications in the field of memory subsystems for Network Processor Units. Then SmartMem is introduced, which tackles the major challenges of the Memory Subsystem in a NP: achieving enough data throughput when accessing the memory, while simultaneously attaining high packet storing efficiency, low resource usage as well as keeping system complexity low. The problem is approached by introducing a decisive collection of methods for improving the packet segmentation and its administration, which plays a fundamental role in the overall processor's performance.

In fact, the Memory Subsystem is remarkably similar in a broad spectrum of devices regardless whether we are referring to core/distribution routers, switches, firewalls or DSL access multiplexors (DSLAM). The traffic specifics may differ depending on the network node but the general function of a Buffer Manager (high-speed storing, making data ready for CPUs, queue management and data fetching for transmission) is quite the same in all NPs. Bearing this in mind, SmartMem is applicable to a broad range of environments with little adaptation and so the validity of its claims holds truth for most packet-based network processors.

8 Acknowledgements

This work has been funded by the Bavarian Ministry of Economic Affairs, Infrastructures, Transport and Technology under grant reference IUK 178/001 (SmartFlow) and realized in cooperation with Infineon AG. Thanks to their financial and technical support, this scientific work was made possible.

In addition, there is a long list of people who contributed to this work, I would like to specifically mention some of them. My apologizes to those that for space reasons and although deserving, are not here directly thanked.

Kimon Karras: In most modern research projects, we engineers cooperate closely in the definition of ideas as well as in their implementation. Mr. Karras has been a most resourceful and helpful colleague and friend during most of the SmartMem project. His contributions to the ideas and models presented in chapter 3 are greatly appreciated. In that way, the credit for the segmentation algorithms together with their modeling and evaluation should be equally shared. But more importantly, Kimon has been of invaluable help and always a ready partner for technical (and not so technical) discussions and for that reason I feel a deep feel of gratitude towards him. His insights are spread across most of this work.

Thomas Wild: Dr. Wild has been my direct supervisor for the whole extension of my stay at the *Lehrstuhl für Integrierte Systeme*. Weekly meetings, join discussions that could take for hours and a remarkable capacity for patiently improving the quality of my work make SmartMem as much of his contribution as mine. For these reasons and especially for accepting been the supervisor of my Diploma Thesis, I would like to thank him. And yes, for somebody born next to the Mediterranean Sea, his capacity for detail is worth of admiration. When somebody takes such considerable efforts in reviewing and providing valuable feedback, this is most welcomed and must be positively acknowledged.

Andreas Herkersdorf: Professor Herkersdorf is the current Chair Holder of the *Lehrstuhl für Integrierte Systeme* and thus was the ultimate responsible of the SmartMem project. But his direct implication in the project, much beyond what one would expect considering the obligations that the management of a big research institute like LIS requires, left in me a profound feeling of luckiness for been under his scholar responsibility. His contributions cannot be easily enumerated so I think it makes him better justice if I rather mention his extraordinary capacity to unblock a seemingly impossible situation, suggesting the better way to approach a problem while shedding light even into the most obscure technical discussion. But among the many qualities of his supervision, I would like to remark one: he masters the difficult art of employee-motivation. It is much easier to deliver with a sense of purpose.

Rainer Ohlendorf and Michael Meitinger: The “FlexPath team” have been my neighboring colleagues for all my stay at the institute. We cooperated closely to adapt the SmartMem Buffer Manager to their FlexPath platform and later during the implementation, debugging and final analysis of our joint work. In addition, I feel myself in debt with Mr. Meitinger for his help during the implementation of the prototyping platform as well as with Mr. Ohlendorf for sharing with me his remarkable insights in Network Processors as well as TU-München’s Bureaucracy (by no means least useful).

Walter Stechele: Professor Stechele holds a position as main lecturer at our institute and for that reason was only partially related to SmartMem. However he was ready to introduce his family to me which greatly help to bear the yearning for home while speeding up my integration in the local culture. Now, I hope to be able to carry out this friendship wherever the paths of life head to.

Felix Miller: Mr. Miller was first a Diploma student under my supervision and now a colleague at the institute. He implemented most of the hardware for the PPLB packet injection system. I would like to specifically remark his contribution among all the other students that participated in the project. His skills with ModelSim and other development tools wharf those of my own, so the roles usually switched being me the student and him the teacher.

Johannes Zeppenfeld: another colleague at the institute, he developed the LIS-IPIF, a fast, small and low-latency interface for the PLB bus. It is strange in occasions how such a simple function as connecting an IP core to the bus may so strongly influence the system performance, which was decidedly improved by his bus interface. Moreover, Mr. Zeppenfeld is probably one of the best C++ programmers I have ever met so his readiness to help solving impossible error messages in SmaSyMo was most welcomed.

Holm Rauchfuß: Holm is one of the most striking personalities I have ever met. Though hard in the approach, his intelligence and concept of humanism are unique and awakes in me deep respect. He provided lots of help for the debugging of the BM v1.0 as well as during most of my work at the hardware laboratory. He also programmed the IPStack. However his introduction to the Sushi culture is going to be his longest lasting contribution.

The list of Diploma and Internship students that cooperated in the development of models or implementing specific parts of the Buffer Manager is long. Without them, SmartMem would not have been a reality and thus their contribution must be acknowledged. I was quite lucky to find such skilful and motivated students. In chronological order of graduation, **Alberto Zambrano** cooperated in the SmaSyMo model, **Yuancheng Miao** improved Xilinx's multiport memory controller and **Andreas Rittinger** modeled the address interleaving scheme in SystemC presented in chapter 6. Moreover several internship students were involved in the project. To all them I would like to express my gratitude.

Moreover, I would like to extend my recognition to all members of the institute, especially to those which whom I had a closer relation: **Abdelmajid Bouajila, Christopher Claus, Andreas Laika, Andreas Lankes, Roman Plyaskin and Zhonglei Wang**. In some way or another, a bit of them is also included in this work.

It is common to forget the task of the administration staff as they did not directly contribute to the scholar values a PhD Thesis. However without the resourceful **Verena Draga** I would hardly be here producing these last words of the dissertation. That army that could count with Frau Draga in the position of chief of logistic operations could hardly lose a war. I would also like to extend my thanks to **Frau Zeller** and **Frau Spöhrle** as well as to **Herr Kohtz**, our system administrator, who were always quick and diligent when answering my pleas for help.

And with permission of all above, the most important contribution to this dissertation was that of **Berenice**. This work is hers as much as mines.

To all above mentioned, thank you.

9 References

[1] ITU-T G.694.2 (12/2003), "Spectral grids for WDM applications: CWDM wavelength grid" http://www.itu.int/rec/T-REC-G.694.2-200312-I/en
[2] Robert G. Hunsperger, "Integrated Optics: Theory and Technology", 2009.
[3] http://www.ibm.com/developerworks/power/library/pa-soc1/index.html?S_TACT=105AGX16&S_CMP=EDU
[4] http://www.semiconductor.net/article/CA6512230.html?industryid=47298
[5] Netronome White Papers. "The Evolution to Network Flow Processing, Enabling the Deployment of Intelligent Networks at 10G and beyond" http://www.netronome.com/files/file/The\%20Evolution\%20to\%20Network\%20Flow\%20Processing\%20(5-09).pdf
[6] Bernard Fortz, Jennifer Rexford, Mikkel Thorup, "Traffic Engineering with Traditional IP Routing Protocols" IEEE Communications Magazine, 2002
[7] http://www.cs.uiuc.edu/homes/luddy/PROCESSORS/IXP2850.pdf
[8] Cisco Visual Networking Index: Forecast and Methodology, 2008-2013
[9] IEEE Internet Computing, January/February 2010 (vol. 14 no. 1), ISSN: 1089-7801
[10] Spirent Communication Test Methodology Journal, IMIX (Internet MIX) Journal, March 2006, http://spcprev.spirentcom.com/documents/4079.pdf
[11] Gries, M., Kulkarni, C., Sauer, C., and Keutzer, K. 2003. Comparing Analytical Modeling with Simulation for Network Processors: A Case Study. In *Proceedings of the Conference on Design, Automation and Test in Europe: Designers' Forum - Volume 2* (March 03 - 07, 2003). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 20256.
[12] Intel's IXP 1200. Network Processor. http://www.intel.com/design/network/products/npfamily/ixp1200.htm
[13] "Intelligent Network Applications for 10Gbps and Beyond", Netronome white paper 2008, http://www.netronome.com/files/file/Netronome%20-%20Intelligent%20Network%20Applications%20for%2010Gbps%20and%20Beyond%20(9-07)(1).pdf
[14] "The Evolution to Network Flow Processing", Netronome white paper 2008, http://www.netronome.com/files/file/The%20Evolution%20to%20Network%20Flow%20Processing%20(5-09).pdf
[15] http://www.eetimes.com/story/OEG20000825S0041
[16] http://www.digchip.com/datasheets/download_datasheet.php?id=714080&part-number=NP4GS3
[17] "Network Processors" Pannos C. Lekkas, pg 61.
[18] http://www.eetimes.com/op/showArticle.jhtml?articleID=18302530
[19] http://en.wikipedia.org/wiki/List_of_defunct_network_processor_companies
[20] http://www.linleygroup.com/Reports/npu_guide.html
[21] S. Hauger, A. Mutter, A. Kirstaedter, T. Wild, K. Karras, R. Ohlendorf, F. Feller, and J. Scharf, "Packet processing at 100 gbps and beyond - challenges and perspectives," in 10. ITG-Fachtagung Photonische Netze, 2009.
[22] Carsten Albrecht, Rainer Hagenau and Erik Maehle, "A Comparison of Parallel Programming Models of Network Processors" in ARCS 2004 - Organic and Pervasive Computing, Workshops Proceedings, March 26, 2004, Augsburg, Germany 2004.
[23] Ohlendorf, R., Herkersdorf, A., and Wild, T. 2005. FlexPath NP: a network processor concept with application-driven flexible processing paths. In *Proceedings of the 3rd IEEE/ACM/IFIP international Conference on Hardware/Software Codesign and System Synthesis* (Jersey City, NJ, USA, September 19 - 21, 2005). CODES+ISSS '05. ACM, New York, NY, 279-284
[24] Kumar, V. P., Lakshman, T.V., Stiliadis, D.: "Beyond best effort: Router architectures for the differentiated services of tomorrow's internet", IEEE Communications Magazine, vol. 36, no. 5, pp. 152--164, May 1998
[25] M. Meitinger, R. Ohlendorf, T. Wild, A. Herkersdorf, "A Hardware Packet Resequencer Unit for Network Processors", International Conference on Architecture of Computing Systems (ARCS 2008), Dresden, February 25-28, 2008
[26] http://www.cs.ucr.edu/~bhuyan/cs203A/**IXP2400**.pdf
[27] http://www.digchip.com/datasheets/parts/.../227/**IXP2800**.php
[28] http://www.cs.illinois.edu/homes/luddy/PROCESSORS/**IXP**2850.pdf
[29] http://www.edn.com/article/CA6500111.html
[30] http://www.netronome.com/pages/network-flow-processors
[31] http://www.hifn.com.cn/uploadedFiles/Library/Product_Briefs/5NP4G_pb_v1.pdf
[32]http://www.netronome.com/files/file/The%20Evolution%20to%20Network%20Flow%20 Processing%20(5-09).pdf

[33] http://sysdoc.doors.ch/MOTOROLA/CPORTFAMILY-BR.pdf
[34] “Network Processors” Pannos C. Lekkas, pg 126.
[35] R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, "A Processing Path Dispatcher in Network Processor MPSoCs", IEEE Transactions on VLSI Systems, Vol. 16, Issue 10, October, 2008, pp 1335-1345
[36] http://www.ezchip.com/Images/pdf/NP-2_Short_Brief_online.pdf
[37] http://www.ezchip.com/t_whpapers.htm
[38] http://www.eetimes.com/news/latest/showArticle.jhtml?articleID=206901479
[39] http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.pdf
[40] D. Llorente, K. Karras, M. Meitinger, H. Rauchfuss, T. Wild, A. Herkersdorf , "Accelerating Packet Buffering and Administration in Network Processors", International Symposium on Integrated Circuits 2007, 2007
[41] Kornaros, G.; Papaefstathiou, I.; Nikologiannis, A.; Zervos, N., "A fully programmable memory management system optimizing queue handling at multi gigabit rates," *Design Automation Conference, 2003. Proceedings* , vol., no., pp. 54-59, 2-6 June 2003
[42] “The role of Memory in NPU System Design”, EZChip White Papers, 2008.
[43] http://www.juniper.net/techpubs/en_US/release-independent/junos/topics/concept/m120-description.html
[44] Papagiannaki, K.; Moon, S.; Fraleigh, C.; Thiran, P.; Diot, C., "Measurement and analysis of single-hop delay on an IP backbone network," *Selected Areas in Communications, IEEE Journal on* , vol.21, no.6, pp. 908-921, Aug. 2003
[45] Ashvin Lakshmikantha, R. Srikant, Nandita Dukkipati, Nick McKeown, Carolyn Beck , “Buffer Sizing results for RCP Congestion Control under Connection Arrivals and Departures”
[46] Yan, C., Naiqi, L., and Ke, Z. 2005. Integrated Access Device (IAD) Solution Using Intel IXP2350 Network Processor. In *Proceedings of the Second international Conference on Embedded Software and Systems* (December 16 - 18, 2005)
[47] Kumar, V. P., Lakshman, T.V., Stiliadis, D.: “Beyond best effort: Router architectures or the differentiated services of tomorrow’s internet”, IEEE Communications Magazine, vol. 36, no. 5, pp. 152-164, May 1998
[48] IDT, San Jose, CA, “Taking packet-processing to the next level achieving next-generation classification performance using multiple databases and IP co-processors,” White Paper, 2008
[49] R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, "A Processing Path Dispatcher in Network Processor MPSoCs", IEEE Transactions on VLSI Systems, Vol. 16, Issue 10, October, 2008, pp 1335-1345
[50] S. O'Kane, S. Sezer, L. Lit, “A Study of Shared Buffer Memory Segmentation for Packet Switched Networks”, Proceedings of the Advanced int'L Conference on Telecommunications and int'L Conference on internet and Web Applications and Services, Washington, DC, 55, Feb. 2006
[51] see the Ethernet standard, for example IEEE 802.3 at http://standards.ieee.org/getieee802/802.3.html or in this clearly written article on Wikipedia http://en.wikipedia.org/wiki/Ethernet
[52] http://www.linuxjournal.com/article/1312
[53] D. Llorente, K. Karras, T. Wild, A. Herkersdorf, "Advanced Packet Segmentation and Buffering Algorithms in Network Processors", HiPEAC 2009; In: Transaction on High Performance Embedded Architectures and Compilers, Volume 4, Issue 4, Paphos, Cyprus, January 25-28, 2009.
[54] McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (Apr. 1960), 184-195
[55] C. J. Georgiou, V. Salapura, “Dynamic reallocation of data stored in buffers based on packet size”, US Patent 7003597, Feb 2006.
[56] Jahangir H., Satish C., T. N. Vijaykumar, “Efficient use of memory bandwidth to improve network processor throughput”, Proceedings of the 30th annual international symposium on Computer architecture, Vol. 31, Issue 2, pp. 300-313, May 2003
[57] Iyer, S.; Kompella, R.R.; McKeown, N., "Designing Packet Buffers for Router Linecards", *Networking, IEEE/ACM Transactions on* Networking, vol.16, no.3, pp.705-717, June 2008
[58] M. March and J. Corbal, ‘‘A DRAM/SRAM Memory Scheme for Fast Packet Buffers,’’
IEEE Trans. Computers, vol. 55, no. 5, May 2006, pp. 588-602.
[59] Wang F., Hamdi, M., “Memory Subsystems in High-End Routers”; Micro, IEEE Volume 29, Issue 3, May-June 2009 Page(s):52 – 63
[60] Agrawal, B. and Sherwood, T. 2009. “High-bandwidth network memory system through virtual pipelines”. *IEEE/ACM Trans. Netw.* 17, 4 (Aug. 2009), 1029-1041.
[61] Papaefstathiou, I., Orphanoudakis, T., Kornaros, G., Kachris, C., Mavroidis, I., and Nikologiannis, A. 2005. Queue Management in Network Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3* (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 112-117.
[62] http://www.hynix.com/datasheet/pdf/dram/HY5DU283222F(Rev.1.2).pdf
[63] http://www.eetasia.com/ART_8800569252_499486_NP_50350665.HTM
[64] “Intel shifts network chip to startup”, EE Times, 11/12/2007

[65] Mudigonda, J.; Vin, H.; Yavatkar, R. “Overcoming the memory wall in packet processing: hammers or ladders? “ Proceedings of the 2005 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2005, Princeton, New Jersey, USA, October 16-18, 2005 2006
[66] Lawrence G. Roberts, “A Radical New Router“, IEEE Spectrum issue July 2009.
[67] Zhen Liu; Kai Zheng; Bin Liu, "Hybrid cache architecture for high speed packet processing" High Performance Interconnects, 2005. Proceedings. 13th Symposium on , vol., no., pp. 67-72, 17-19 Aug. 2005
[68] http://download.intel.com/pressroom/kits/32nm/westmere/Intel_32nm_Overview.pdf or also in this article http://www.physorg.com/news109344893.html
[69] P. R. Wilson et al.“Dynamic storage allocation: A survey and critical review. Proc. International Workshop on Memory Management” Kinross, Scotland, UK, Sep. 1995.
[70] The Wall Street Journal, “Iran's Web Spying Aided By Western Technology”, JUNE 22, 2009. http://online.wsj.com/article/SB124562668777335653.html
[71] K. Papagiannaki, D. Veitch, N. Hohn, “Origins of Microcongestion in an Access Router”, Lecture Notes in Computer Science, Volume 3015, 2004, Pages 126-136.
[72] K. Papagiannaki, S. Moon, C. Fraleigh, P. Thiran and C. Diot, "Measurement and Analysis of Single-Hop Delay on an IP Backbone Network." IEEE Journal on Selected Areas in Communications, Spe-cial Issue on Internet and WWW Measurement, Mapping, and Modeling, 3rd quarter, 2003
[73] J. Fu, O. Hagsand, G. Karlsson, “Queueing behavior and packet delays in network processor sys-tems”, Proceddings of the IEEE Workshop on High Performance Switching and Routing, 2006
[74] NLANR PMA: Special Traces Archive: http://pma.nlanr.net/Special/chronIndex.html
[75] CAIDA OC-48 Trace Archive: http://www.caida.org/data/passive/ index.xml#oc48
[76] PCAP library: http://en.wikipedia.org/wiki/Pcap
[77] CoreConnect bus architecture: http://www-306.ibm.com/chips/products/coreconnect/
[78] Website of the SystemC standardization organization: www.systemc.org
[79] T. Wild, A. Herkersdorf, and G.-Y. Lee, "TAPES--trace-based architecture performance evaluation with SystemC," Design Automation for Embedded Systems, vol. 10, no. 2-3, pp. 157-179, 2005.
[80] Uday R. Naik, “Efficient buffer management”, US Patent 20060143334, Jun-2006.
[81] B.H. Margolin, R.P. Parmelee, M Achttzoff, “Analysis of free-storage algorithms”. IBM Systems Journal, 10(4):283-304, 1971.
[82] Milenkovic, A. 2000. “Achieving High Performance in Bus-Based Shared-Memory Multiprocessors”. *IEEE Concurrency* 8, 3 (Jul. 2000), 36-44.
[83] Huggahalli, R., Iyer, R., and Tetrick, S. “Direct Cache Access for High Bandwidth Network I/O”, in Proceedings of the 32nd Annual international Symposium on Computer Architecture (June 04 - 08, 2005). IEEE Computer Society, Washington, DC, 50-59
[84] R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, "An Application-aware Load Balancing Strategy for Network Processors", HiPEAC 2010 International Conference on High-Performance Embedded Architectures and Compilers, Pisa, Italien, January 25-27, 2010, pp 156-170
[85] R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, "A Packet Classification Technique for On-Chip Processing Path Selection", Proceedings of the 5th Workshop on Application Specific Processors (WASP'07), pp. 95-102; Salzburg, October 4-5, 2007
[86] http://en.wikipedia.org/wiki/CAS_latency
[87] Mrinmoy Ghosh and Hsien-Hsin S. Lee. "Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs." In Proceedings of the 40th ACM/IEEE International Symposium on Microarchitecture, pp.134-145, Chicago, IL, December, 2007.
[88] García, J., Corbal, J., Cerdà, L., and Valero, M. 2003. “Design and Implementation of High-Performance Memory Systems for Future Packet Buffers” In *Proceedings of the 36th Annual IEEE/ACM international Symposium on Microarchitecture* (December 03 - 05, 2003). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 373.
[89] Ostler, C. and Chatha, K. S. 2007. “An ILP formulation for system-level application mapping on network processor architectures” In *Proceedings of the Conference on Design, Automation and Test in Europe* (Nice, France, April 16 - 20, 2007). Design, Automation, and Test in Europe. EDA Consortium, San Jose, CA, 99-104.
[90] James Manchester, Jon Anderson, Bharat Doshi, and Subra Dravida. “IP over SONET“, Bell Laboratories.
[91] A. Striegel, G. Manimaran, “Dynamic Class-Based Queue Management for Scalable Media Servers” Journal of Systems and Software, 2003.
[92] Wenjiang Zhou, Chuang Lin, Yin Li, Zhangxi Tan, "Queue Management for QoS Provision Build on Network Processor," ftdcs, pp.219, The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03), 2003
[93] Cristal, A., Santana, O. J., Cazorla, F., Galluzzi, M., Ramirez, T., Pericas, M., and Valero, M. 2005. Kilo-Instruction Processors: Overcoming the Memory Wall. *IEEE Micro* 25, 3 (May. 2005), 48-57.
[94] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080
[95] Wischik, D. and McKeown, N. 2005. Part I: buffer sizes for core routers. *SIGCOMM Comput. Commun. Rev.* 35, 3 (Jul. 2005), 75-78.

[96] G. Appenzeller, “Sizing Router Buffers”, Dissertation for a Ph.D. in Computer Science, Stanford 2004.
[97] http://www.world-nuclear.org/info/inf40.html
[98] Kenneth C. Knowlton. “A Fast storage allocator” Communications of the ACM 8(10):623-625, Oct 1965.
[99] Wikipedia offers a very nice explanation of buddy systems: http://en.wikipedia.org/wiki/Buddy_memory_allocation
[100] http://en.wikipedia.org/wiki/Dynamic_random_access_memory
[101] documentation and software of the stack available here: http://www.sics.se/~adam/lwip/index.html
[102] http://www.cisco.com/en/US/products/ps9402/index.html
[103] http://axmo12.de
[104] http://www.cisco.com
[105] http://www.lsi.com/
[106] http://www.ezchip.com/p_np4.htm