System-level performance optimization of the data queueing memory management in high-speed network processors

Ch. Ykman-Couvreur
J. Lambrecht, D. Verkest
F. Catthoor*
IMEC, Kapeldreef 75
B3001 Leuven, Belgium
ykman@imec.be

A. Nikologiannis
ELLEMEDIA technologies
Syggrou Avenue 223
EL17121 Athens, Greece
anikol@ellemedia.com

G. Konstantoulakis
Inaccess Networks SA
Sygrou Avenue 230
EL17672 Athens, Greece
gkonst@inaccessnetworks.com

ABSTRACT

In high-speed network processors, data queueing has to allow real-time memory (de)allocation, buffering, retrieving, and forwarding of incoming data packets. Its implementation must be highly optimized to combine high speed, low power, large data storage, and high memory bandwidth. In this paper, such data queueing is used as case study to demonstrate the effectiveness of a new system-level exploration method for optimizing the memory performance in dynamic memory management. Assuming that a multi-bank memory architecture is used for data storage, the method trades off bank conflicts against memory accesses during real-time memory (de)allocation. It has been applied to the data queueing module of the PRO^3 system [8]. Compared with the conventional memory management technique for embedded systems, our exploration method can save up to 90% of the bank conflicts, which allows to improve worst-case memory performance of data queueing operations by 50% too.

Categories and Subject Descriptors

B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids

General Terms

Memory performance

Keywords

System-level exploration, memory management

1. INTRODUCTION

The rapid growth in the dimension and diversity of networks and telecom systems, along with the ever increasing user demand in networking services, has imposed the need for efficient protocol processing and the development of high capacity systems running OC3 (155 Mbps), OC12 (622 Mbps), OC48 (2.488 Gbps) and sometimes OC192 (10 Gbps) speeds. One of the fundamental functionalities, and also one of the most difficult parts in high-speed network processors offering quality of service, is *data queueing*. This has to allow real-time memory (de)allocation, buffering, retrieving, and forwarding of incoming data packets. Its implementation must be highly optimized to combine high speed, low power, large data storage, and high memory bandwidth. It must also be optimized to support worst-case situations as efficiently as possible. First, to minimize large data copying, pointers to these data are better passed as arguments of the data queueing operations. Secondly, to increase the memory bandwidth, a combination of SRAMs (for pointers, flags, ..., which are small and frequently accessed) and DRAMs (for data packets, which are large and not frequently accessed) in the memory architecture becomes a must. Hence the memory performance is affected by two factors: the huge amount of memory accesses and the effectiveness of access to memory. E.g. accesses to non-busy banks in a multi-bank memory architecture can be performed much faster than successive accesses to different pages insisde the same bank.

In this paper, such data queueing is used as case study to demonstrate the effectiveness a new system-level exploration method for optimizing the memory performance in dynamic memory management. Assuming that a multi-bank memory architecture is used for data storage, this method trades off bank conflicts (due to successive accesses to the same bank) against memory accesses during real-time memory (de)allocation. It has been applied to the data queueing module of the PRO^3 system [8]. Compared with the conventional memory management technique for embedded systems, our exploration method can save up to 90% of the bank conflicts, which improves worst-case memory performance of data queueing operations by 50% too. It also fits in Matisse [19], a system-level synthesis approach supporting the memory management of dynamic (de)allocation of data structures in embedded systems.

The remainder of the paper is organized as follows. Section 2 summarizes the design characteristics of existing data queueing implementations. Section 3 presents the abstract specification of the data queueing module in the PRO^3 sys-

*Also prof. at Katholieke Univ. Leuven

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

tem. This module is used to illustrate our method. Section 4 analyzes the system-level parameters that affect the memory performance the data queueing module. These parameters are taken into account in Section 5, that describes our exploration method. Experimental results for the considered data queueing module are discussed in Section 6.

2. RELATED WORK

Data queueing from the *algorithmic* point of view is already well established. To guarantee quality of service, important policies use per-flow queuing, where data packets are organized into queues, one queue per active connection/flow. These policies are described in [6, 12, 18]. In addition to large data storage, to implement the queues, these policies are characterized by real-time (de)allocation and frequent updates of large data structures.

Hence the implementation of the data queueing module (DQM) must also be combined with an appropriate *memory management technique to organize the virtual memory, to allow real-time memory (de)allocation, and to control memory overflow.* Conventionally, as in memory management of embedded systems, the virtual memory is partitioned into blocks to store the data packets and all dynamic data structures implementing the queues [4, 5, 15, 16, 17]. All these blocks that are free in the virtual memory are maintained in a single linked list, called *free list* [14]. An alternative approach is to simulate the DQM with different memory management techniques and then to choose the most convenient one, relatively to power, area, and performance. Another more promising approach is to develop a *cost-driven exploration method* that takes into account the network characteristics and its real-time requirements, and that systematically provides the best memory management technique for that DQM. This approach is followed in [19], where the complete design space is characterized by identifying all its orthogonal decision trees, and where an exploration methodology, driven by the number of memory accesses, is outlined. However nowadays, none of the existing approaches *exploit the features of the used memory architecture.*

Relatively to the selected *memory architecture*, on-chip memories (limited by the storage capacity) are used in [5, 16]. Content addressable memories (limited by both performance and power) are used in [4]. In [7, 15], the memory architecture only consists of off-chip DRAMs. By storing data structures with data packets in the same DRAMs, instead of storing data structures in separate off-chip SRAMs, the number of off-chip memories are economized and the number of chip pins is minimized. However the latency of accessing the data structures is increased, since DRAMs are slower than SRAMs. Additionally storing data structures with data packets in the same DRAMs increases the number of DRAM accesses per segment enqueueing or dequeueing. Hence nowadays to combine high speed, low power, large data storage, and high memory bandwidth, the *use of off-chip* DRAM*s (with multi banks) combined with off-chip* SRAM*s* becomes a must.

To increase the performance of data queueing, not only the memory bandwidth, but also the access to memory must be highly optimized. Research has already been carried out to reduce the number of bank conflicts in a multi-bank memory architecture. [2, 9] proposes data placement schemes suitable for static data dominated applications. [10, 11] propose local optimizations for memory access scheduling.

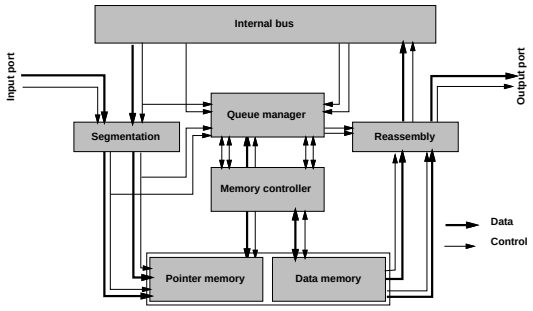

Figure 1: DQM block diagram

However nowadays *minimizing at the system level the number of bank conflicts during real-time memory (de)allocation* has not been given much attention.

What we propose in this paper is to optimize at the system level, where the impact on the memory performance is the most important, the dynamic memory management in data queueing. This is realized as follows:

- Select a memory architecture combining off-chip DRAMs with off-chip SRAMs. This can access data structures in parallel with data packets, and thus to aim at keeping the rate of DRAM accesses to one per segment enqueueing or dequeueing.
- Systematically explore possible memory management techniques, based on network characteristics, and on its real-time requirements, and trading off bank conflicts in DRAMs against memory accesses in SRAMs during real-time memory (de)allocation.

3. DEMONSTRATOR

As previously mentioned, DQMs in network processors are responsible for real-time memory (de)allocation, buffering, retrieving, and forwarding of incoming data packets. These data packets can be of variable length. They are organized into queues, one queue per active connection. The DQM has to perform operations like: create or destroy a queue, enqueue a new packet in a queue, dequeue a packet from a queue and forward it, dequeue a packet from a queue and append it to another queue, delete a packet from a queue. The major bottleneck in the DQM implementation is not in the data path and the controller, but in the communication and the memory organization [3]. In this paper we focus on the worst-case memory performance of the DQM.

The DQM block diagram is shown in Figure 1. The virtual memory is split into the pointer memory and the data memory. All pointers, flags, ..., in the data structures implementing the queues, which are small and frequently accessed, are stored in the *pointer memory.* All data packets, which are large and not frequently accessed, are stored in the *data memory.*

The DQM has to segment the incoming data packets being of variable length into fixed-size segments. Similarly, the outgoing segments from the data memory are reassembled to form the outgoing data packets.

To simplify the functionality and increase the modularity of the DQM, the DQM is divided into two main blocks: the queue manager and the memory controller. The queue manager handles the control information from the network processor for transferring data packets to/from the data memory. It handles the queues and organises the data packets in

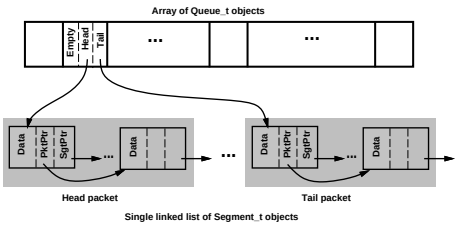

Figure 2: Queue manager abstract specification

the queues. It also manages the memories for dynamically (de)allocating the data structures implementing the queues and for detecting memory overflow.

In the PRO^3 system, the DQM has to agree with the following network characteristics. It must support up to 0.5M connections simultaneously. In congestion situations, it must support a data memory of 256M bytes. It must preserve the OC48 (2.488 Gbps) line rate of the incoming packets. However, taking into account that bandwidth is wasted due to packet segmentation/reassembly, and that the DQM must support 2 input ports, 2 output ports, and 1 internal bi-directional port, the DQM needs to meet a bandwidth of at least 15 Gbps (around $6 * 2.488$ Gbps).

The abstract specification of the queue manager is now detailed. This queue manager is the most critical block of the DQM and an intelligent implementation is needed. It also has to exploit the parallelism not only of the DQM operations, but also of the huge amount of memory accesses in order to meet the strict real-time requirements. In this paper, we assume that an abstract specification has already been selected for the queue manager. This is depicted in Figure 2 and described below. Each queue consists of a single linked list of segments, and it is characterized by a `Queue_t` object consisting of the following main fields: an `Empty` bit indicating whether the queue is empty or not, a `Head` pointer to the first segment of the head packet in the queue, and a `Tail` a pointer to the first segment of the tail packet in the queue. This `Queue_t` object is stored in an array whose index respresents the queue identifier generated by the network processor. *This array is statically allocated.*

Each segment is characterized by a `Segment_t` object consisting of the following main fields: `Data` storing the segment data bytes, a `SgtPtr` pointer to the next segment in the queue, and a `PktPtr` pointer to the last segment of the current packet. *These* `Segment_t` *objects are dynamically (de)allocated.* A memory management technique is then required to organize the virtual memory, to allow real-time memory (de)allocation of these objects, and to control memory overflow. This is selected by our exploration method described in Section 5.

4. MEMORY PERFORMANCE

As mentioned in Section 3, at the system level, all `Data` fields from the `Segment_t` objects are stored in the data memory, which is mapped later during system synthesis to DRAMs. All other fields are stored in the pointer memory, which is mapped later to SRAMs. In the following, we analyze the system-level parameters that affect the performance of both data and pointer memories, in order to take them into account in our exploration method.

4.1 Data memory

To accommodate the requirements of the data memory (both memory storage and bandwidth), a module of double data rate synchronous DRAMs [13] can be used. It provides large memory space and high memory bandwidth, but suffers from bank conflicts. This module has a total capacity of 256 Mbytes and is organized into 8 banks. The memory data bus of this module is 64-bit wide and runs at 133 MHz. Taking into account that both edges of the clock are used to access a double data rate memory, the module offers a bandwidth of 17 Gbps in best case. To effectively utilize the module, this should work as much as possible in the burst mode. The burst has a length of 64 bytes. Hence to optimize memory bandwidth utilization, a segment size of 64 bytes too is selected. In a non-busy bank, a segment can be read/written in 4 clock cycles. In a busy bank, there is a *bank conflict* and the access is delayed (8 clock cycles, i.e. around 60 ns) until the bank becomes available.

Hence at the system level, the data memory performance heavily depends on the number of bank conflicts. A bank conflict happening during a read access cannot be controlled. However a bank conflict happening during a write access can be controlled by storing new data in a non-busy bank.

4.2 Pointer memory

For the pointer memory, a module of zero bus turnaround SRAMs [13] can be used. These memories employ a high-speed and low-power design using an advanced CMOS process, and they eliminate any turnaround clock cycle between read and write memory accesses. Each SRAM has a capacity of 0.5M 32-bit words, and it can run at 166 MHz. To agree with the assumed network characteristics, the DQM requires to support a data memory of 256M bytes, which corresponds to 4M segments of 64 bytes each. Hence 22 bits are needed to specify a `segment_t` pointer. The DQM also requires to support 0.5M connections. Hence within the selected queue manager specification (see Section 3), a pointer memory of 9M `Segment_t` pointers[1] is needed, and it must be mapped to a module of 18 SRAMs. To minimize the number of pins, the number of parallel memories is restricted to 2, and the memory data bus of the module is 64-bit wide. Since data pipelining is possible, a 64-bit word (that is, any `Queue_t` object or the part of any `Segment_t` object stored in the pointer memory) can be accessed every clock cycle. Taking into account that both data and pointer memories can be accessed in parallel, 5 (resp. 15) 64-bit word accesses to the SRAM module can be performed in parallel with a segment reading/writing in the DRAM module without (resp. with) bank conflict.

Hence at the system level, the pointer memory performance can be further optimized by minimizing the number of `Queue_t` and `Segment_t` accesses during `Segment_t` dynamic (de)allocation. These operations are required in the following DQM operations: enqueue new segments in a queue during packet storage, and delete a packet from a queue.

4.3 Conclusion

Our method described in the next section explores the design space of memory management techniques, taking this previous analysis into account, and efficiently trading off: (1) write accesses to non-busy banks in the data memory

[1] 1M pointers for the `Queue_t` objects, 8M pointers for the `Segment_t` objects.

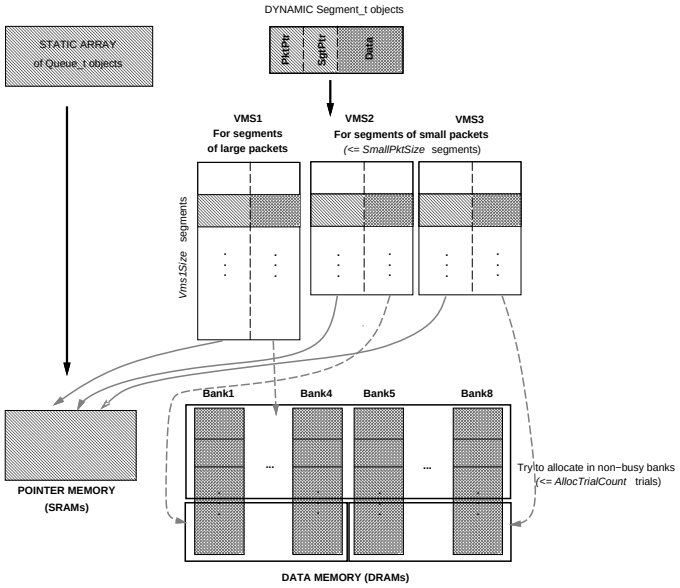

Figure 3: Our DQM memory management, parameterized by $AllocTrialCount$, $SmallPktSize$, and $Vms1Size$

during packet storage, against (2) pointer memory accesses during packet storage and deletion.

5. MEMORY MANAGEMENT

As mentioned in Section 3, the selected queue manager specification uses: (1) the `Queue_t` data structure whose objects are stored in a static array; (2) the `Segment_t` data structure whose objects are dynamically (de)allocated in the virtual memory. Hence a memory management technique is required to organize the virtual memory, to allow real-time memory (de)allocation of the `Segment_t` objects, and to control memory overflow.

Our exploration method to derive an efficient memory management technique for the DQM is depicted in Figure 3 and is described below. It fits in the context of embedded system synthesis, as our Matisse approach [19] based on fast and successive explorations at all levels of the complete design flow, in order to achieve efficicent system designs. Since optimizing the dynamic memory management is one of the first system-level synthesis steps, exploration in this step must be simple and fast.

5.1 Virtual memory organization

The virtual memory is first partitioned into virtual memory segments (VMSs) of fixed-size blocks. In this paper, 4 VMSs are considered. A first VMS stores the static array of `Queue_t` objects, and it is assigned to the pointer memory. Three other VMSs consist of a reserved amount of memory to store `Segment_t` objects. `VMS1` stores segments from large packets (containing $> SmallPktSize$ segments), it has a capacity of $Vms1Size$ segments, and it is mapped later to the 8 banks of the DRAM module (described in Section4.1). `VMS2` and `VMS3` store segments from small packets (containing $\leq SmallPktSize$ segments), and they have a capacity of $(4M - Vms1Size)/2$ segments each[2]. `VMS2` (resp. `VMS3`) is mapped later to the first (resp. last) 4 banks of the DRAM module. To control the bank conflicts at the system level, each block (whose size is `Segment_t` size) in these three VMSs is also initially associated with one of the 8 banks. Also the free blocks of each VMS are maintained in a single linked list by using the `SgtPtr` pointer to specify the next free block in the list. Such lists of free blocks are called *free lists* [14]. As mentioned in Section 4, the pointers in the `Segment_t` objects are assigned to the pointer memory, whereas the `Data` fields are assigned to the data memory.

5.2 (De)allocation implementation

For these three VMSs (`VMS1, VMS2` and `VMS3`), a memory management technique is required, where dynamic (de)allocation is implemented as follows.

5.2.1 Allocation in segment enqueueing

Whenever a `Segment_t` object is created, a free block to store it is allocated as follows, so as to avoid bank conflicts as much as possible. In the DQM, `Segment_t` creation is required in segment enqueueing.

1. If the segment belongs to a large packet (containing $> SmallPktSize$ segments), `VMS1` is selected. If the head block in its free list is associated with a busy bank, it is pushed at the free list tail. This is repeated until a block associated with a non-busy bank is found. We limit the number of trials to $AllocTrialCount$. The current head block is then allocated.

Hence for each trial one pointer memory read access is required to access the next free block in the free list. One pointer memory write access is required to link the rejected head blocks to the free list tail. Another pointer memory read access is required to update the free list head. Hence the total number of pointer memory accesses is less than $AllocTrialCount + 1$.

2. If the segment belongs to a small packet, and if the head block in the free list of `VMS2` is associated with a non-busy bank, this head block is allocated.

Hence only one pointer memory read access is required to update the free list head.

3. Otherwise the head block in the free list of `VMS3` is allocated and only one pointer memory read access is required too.

Hence *worst-case segment enqueueing* either implies a bank conflict in the data memory or requires more than one access in the pointer memory during segment allocation.

5.2.2 Deallocation in packet deletion

Whenever a `Segment_t` object is deleted, the block storing it is pushed to the free list it came from. In the DQM, `Segment_t` deletion is only required in packet deletion.

1. For large packets, all segments are guaranteed to be stored in blocks allocated from the same free list, the one of `VMS1`.

In this case, only one pointer memory write access is required to update the tail of this free list, independently from the number of deleted segments in the packet.

2. For small packets, for each segment to be deleted, the block storing it is pushed to the free list it came from, that is, the one of either `VMS2` or `VMS3`.

One pointer memory read access is required to access each next segment of the packet in the queue. For each segment,

[2]To agree with the network characteristics, the DQM requires to support 4M segments (see Section 4.2).

one pointer memory write access is also required to link the freed block to the free list. Hence the total number of pointer memory accesses per packet deletion is always less than $2 * SmallPktSize - 1$.

Hence *worst-case packet deletion* requires more than one access in the pointer memory.

5.3 Exploration

In our exploration method, the values of the three most important parameters are explored: $AllocTrialCount$, $SmallPktSize$, and $Vms1Size$. Larger $AllocTrialCount$ is, less bank conflicts in the data memory are generated, but more pointer memory accesses are required during allocation of segments belonging to large packets. Larger $SmallPktSize$ is, more segment allocations are performed without bank conflict in the data memory and without extra pointer memory access, but more pointer memory accesses are required during packet deletion. $Vms1Size$ has to trade-off memory overflows during segment allocation between large packets stored in `VMS1` and small packets stored in both `VMS2` and `VMS3`.

Within this design space, the conventional memory management technique, consisting in using one free list, is characterized by: $AllocTrialCount = 1$, $SmallPktSize = 0$, and $Vms1Size = 4M$ (i.e. $2^{22} = 4194304$) segments. In contrast, the best memory management technique is the one that minimizes the memory overflows in the three VMSs of the virtual memory organization, together with the memory performance in worst-case segment enqueueing and packet deletion. However, as illustrated in Section 6, these cost parameters must be traded off against each other. To this end a 5-dimension Pareto curve is generated as follows, from which the designer can select an optimized memory management technique meeting the required trade-off:

• Each possible memory management technique, parameterized by the explored $AllocTrialCount$, $SmallPktSize$, and $Vms1Size$ values, is simulated using some real packet traces. Each simulation profiles worst-case situations and derives the following 3 parameters:

– The percentage of segment allocation failures in order to estimate the number of memory overflows.

– $OverheadMemCycle/WorstCases$ in order to estimate the memory performance overhead in worst-case segment enqueueing, where $OverheadMemCycle$ denotes the total number of overhead memory cycles (in terms of SRAM clock cycles) in all worst-case segment enqueueing executions, and $WorstCases$ denotes the number of these executions.

– The memory performance overhead in worst-case packet deletion is similarly estimated.

• A technique for which all profiled parameters are larger than in another explored alternative is eliminated.

• The set of remaining techniques is called the *Pareto curve* of the considered DQM relatively to the exploration.

Practically, $AllocTrialCount$ which has the larger impact is greedily explored, then $SmallPktSize$, and finally $Vms1Size$. For each parameter, the exploration stops when no better technique can be reached.

6. RESULTS

Results of our exploration method are reported for IP networks receiving large IP packets. These results illustrate that the conventional memory management technique used in embedded systems is far from being well-suited in DQM modules. To this end, simulations are performed on real packet traces found in [1]. First we analyse the impact of each parameter in the exploration. Then we describe the resulting Pareto curve and compare the selected optimized techniques with the conventional one.

Table 1: $AllocTrialCount$ impact

Alloc Trial Count	Mem. acc. Per seg. alloc. Avg/Max	Bank conflict percentage In write ops Wo/with page ctrl	In read ops Wo/with page ctrl
1	1/1	15.8/11.9	12.4/12.4
2	1/3	6.2/1.9	6.7/6.7
3	1/4	5.5/1.9	6.2/6.2
4	1/5	5.3/1.7	6.1/6.1
5	1/6	5.4/1.7	6.1/6.1

First, $AllocTrialCount$ is explored, while $SmallPktSize$ and $Vms1Size$ are kept constant. Results are reported in Table 1 for $SmallPktSize = 0$, i.e. for using only one VMS. From this exploration, we make the following observations. (1) The conventional memory management technique corresponds with $AllocTrialCount = 1$. Its number of pointer memory accesses is constant per segment allocation. However its average number of bank conflicts is the highest. (2) Athough the performance of segment enqueueing depends on $AllocTrialCount$ (since the number of pointer memory accesses per segment allocation is bounded by $AllocTrialCount + 1$), it is not affected by it on average (since this number of accesses remains constant on average). (3) Supporting page control in the DRAM memory controller is a must, since it allows an additional reduction of up to 70% in the number of bank conflicts during DRAM write accesses. (4) Already with $AllocTrialCount = 2$, 84% of the number of bank conflicts are avoided during DRAM write accesses, and this also yields a reduction of 46% in the number of bank conflicts during DRAM read accesses, which are not controlled though. (5) From $AllocTrialCount = 3$, no further reduction in the number of bank conflicts is observed, and the greedy exploration can stop.

Table 2: $SmallPktSize$ impact

Small Pkt Size	Mem. acc. Per pkt deletion Avg/Max	Bank conflict percentage In write ops Wo/with page ctrl	In read ops Wo/with page ctrl
0	1/1	5.5/1.9	6.2/6.2
1	1/1	3.5/1.2	7.3/7.3
2	1/3	3.6/1.5	6.8/6.8
3	1/5	3.4/1.4	6.6/6.6
4	1/7	3.2/1.4	6.6/6.6

Second, $SmallPktSize$ is explored, while $AllocTrialCount$ and $Vms1Size$ are kept constant. Results are reported in Table 2 for $AllocTrialCount = 3$. From this exploration, we make the following observations. (1) Although the performance of packet deletion depends on $SmallPktSize$ (since its number of pointer memory accesses is bounded by $2 * SmallPktSize - 1$), it is also not affected by it on average (since this number of accesses also remains constant on average). (2) Considering three VMSs instead of only one does not allow to further reduce the number of bank conflicts for these considered network characteristics. However as illustrated in Table 4, this allows to trade off the worst-

case performance in segment enqueueing against the ones in packet deletion. (3) The greedy exploration can stop with $SmallPktSize = 4$.

Table 3: $Vms1Size$ impact

$Vms1$ $Size$	Memory Overflow Probability	Bank conflict percentage In write ops Wo/with page ctrl	In read ops Wo/with page ctrl
2097152	1.85	3.6/1.5	6.8/6.8
3145728	2.28	4.2/1.8	5.6/5.6
3495232	2.39	4.0/1.8	5.1/5.1
3670016	2.41	3.8/1.7	4.7/4.7
3774848	2.43	3.8/1.7	4.5/4.5
3984576	2.46	3.5/1.6	3.9/3.9
4054528	2.49	3.4/1.5	3.7/3.7
4089472	2.5	13.5/11.5	12.5/12. 5

Finally,, $Vms1Size$ is explored, while $AllocTrialCount$ and $SmallPktSize$ are kept constant. Results are reported in Table 3 for $AllocTrialCount = 3$ and $SmallPktSize = 2$, and for a particular trace of DQM operations. We observe that this exploration still allows to reduce the number of bank conflicts during DRAM read accesses by 43%, but at the expense of the number of memory overflows. The greedy exploration stops as soon as a degradation in the number of bank conflicts is observed.

Table 4: Optimized techniques compared with the conventional one

$AllocTrCt$/ $SmPkSize$/ $Vms1Size$	Bank conflict percentage With page ctrl In write/read ops	Memory Overflow Percentage	Seg. enq. Overhead Perf/Energy
3/1/2097152	**1.2/7.4**	**1.7**	**4/164**
4/1/3145728	**1.5/5.9**	**2.2**	**4/160**
5/1/3774848	**1.6/4.9**	**2.4**	**4/158**
5/1/4054528	**1.3/4.0**	**2.5**	**4/152**
4/0/4194304	**1.7/6.1**	**2.5**	**3/122**
1/0/4194304	11.9/12.4	2.45	8/337

As mentioned in Section 5, to help the designer to select an optimized memory management technique meeting the required trade-off, a Pareto curve is generated. In this experiment, this consists of 16 techniques out of the 180 explored ones. In these optimized techniques, we observe that: (1) In all packet deletion executions, the memory performance is minimal; (2) Per worst-case segment enqueueing execution, the memory performance overhead is between 3 and 4 SRAM clock cycles; (3) The memory overflow percentage is between 1.7 and 2.5. Some of these optimized techniques are reported in Table 4 and compared with the conventional technique. They can save up to 90% (resp. 68%) of the bank conflicts during DRAM write (resp. read) accesses. This bank conflict reduction allows to improve both worst-case memory performance in segment enqueueing by 50%.

7. CONCLUSION

One of the most difficult part in high-speed network processors is data queueing. In this paper, such data queueing was used as case study to demonstrate the effectiveness of a new system-level exploration method for optimizing the memory performance in dynamic memory management. This method also fits in Matisse [19], a system-level synthesis approach supporting the memory management of dynamic (de)allocation of data structures in embedded systems. In the future we intend to adapt this new method to the memory management of look-up tables, another bottleneck in the design of high-speed network processors.

7.1 Acknowledgements

This paper describes work undertaken in the context of the IST-1999-11419 Protocol Processor Project (PRO^3), partially funded by the Commission of the European Union. The authors would like to acknowledge the contributions of all their collegues involved in the project from Lucent Technologies, Hyperstone electronics, IMEC, National Technical University of Athens and Ellemedia Technologies.

8. REFERENCES

[1] Traces available in the internet traffic archive. http://ita.ee.lbl.gov/html/traces.html.

[2] A.A. Bertossi and M.C. Pinotti. Mappings for conflict-free access of paths in elementary data structures. *COCOON*, p.351-361, July 2000.

[3] F. Catthoor et al.. *VLSI Signal Processing*, volume VII, chapter Global Communication and Memory Optimizing Transformations for Low Power Signal Processing Systems, p.178-187. IEEE Press, New York, 1994.

[4] H.J. Chao et al. Design of a generalized priority queue manager for ATM switches. *IEEE Journal on Selected Areas in Communications*, 15(5):867-880, June 1997.

[5] A.K. Choudhury and E.L. Hahne. New implementation of multi-priority pushout for shared memory ATM switches. *Computer Communications*, 19(3):245-256, March 1996.

[6] V. Firoiu and M. Borden. A study of active queue management for congestion control. *Infocom*, Tel Aviv, Israel, March 2000.

[7] G.Doumenis et al. A parallel VLSI video/communication controller. *The Journal of VLSI Signal Processing, Special Issue on Parallel VLSI Architectures for Image and Video Processing*, July 2000. Kluwer Academic.

[8] C. Georgopoulos et al. A protocol processing architecture backing TCP/IP-based security applications in high speed networks. *Interworking Conference*, Bergen, Norway, 2000.

[9] L.K. John. Data placement schemes to reduce conflicts in interleaved memories. *Computer Journal*, 43(2):138-151, 2000.

[10] M. Kandemir, U. Sezer, and V. Delaluz. Improving memory energy using access pattern classification. *ICCAD*, p.201-206, November 2001.

[11] K.A. Khare, P.R. Panda, N.D. Dutt, and A. Nicolau. High-level synthesis with SDRAMs and RAMBUS DRAMs. *IEICE Trans. on Fundamentals of Electronics, Communications, and Computer Sciences*, 11:2347-2355, 1999.

[12] V. Kumar, T. Lakshman, and D. Stiliadis. Beyond best effort: Router architectures for the differentiated services of tomorrow's internet. *IEEE Communications Magazine*, p.152-164, May 1998.

[13] Micron. http://www.micron.com.

[14] N. Murphy. Safe memory usage with dynamic allocation. *Embedded Systems*, p.49-57, May 2000.

[15] A. Nikologiannis and M. Katevenis. Efficient per-flow queueing in DRAM at OC-192 line rate using out-of-order execution techniques. *Int. Conf. on Communications*, p.2048-2052, 2001.

[16] J.L. Rexford, A.G. Greenberg, and F.G. Bonomi. Hardware-efficient fair queueing architectures for high-speed networks. *Infocom*, p.638-646, March 1996.

[17] D.N. Serpanos and P. Karakonstantis. Efficient memory management for high-speed ATM systems. *Design Automation for Embedded Systems*, 6(2):207-235, April 2001.

[18] B. Suter, T.V. Lakshman, D. Stiliadis, and A.K. Choudhury. Buffer management schemes for supporting TCP in gigabit routers with per-flow queueing. *IEEE Journal on Selected Areas in Communications*, August 1999.

[19] S. Wuytack et al. Memory management for embedded network applications. *IEEE Trans. on CAD*, 18(5):533-544, 1999.