Queue Based Memory Management Unit for Heterogeneous MPSoCs

Robert Wittig, Mattis Hasler, Emil Matus and Gerhard Fettweis (Fellow IEEE)
Vodafone Chair for Mobile Communications Systems
Technical University Dresden
Email: {robert.wittig, mattis.hasler, emil.matus, gerhard.fettweis}@tu-dresden.de

***Abstract*—Sharing tightly coupled memory in a multi-processor system-on-chip is a promising approach to improve the programming flexibility as well as to ease the constraints imposed by area and power. However, it poses a challenge in terms of access latency. In this paper, we present a queue based memory management unit which combines the low latency access of shared tightly coupled memory with the flexibility of a traditional memory management unit. Our passive conflict detection approach significantly reduces the critical path compared to previously proposed methods while preserving the flexibility associated with dynamic memory allocation and heterogeneous data widths.**

***Index Terms*—Tightly Coupled Memory, Memory Management, MPSoC, Embedded System, Interconnect**

I. Introduction

Modern multi-processor systems on chips (MPSoC) deploy a multitude of different processing elements (PE) like RISC, DSP and ASIP cores complemented with hardware accelerators, in order to cope with stringent performance, energy and area constraints [1]. In real-time and data intensive applications, like software defined radios for mobile communications [2], the private tightly coupled memory (TCM) provides PEs with low-latency access to critical data and tasks. However, due to the utilization wall [3] the number of cores that can be active at the same time is limited. Hence, big areas of TCM memory can not be utilized. This results in an over provisioning of on-chip memory, thus limiting the area that could be dedicated to PEs.

To cope with this limitation, modern architectures exploit a system of shared L1 memory, enabling the joint use of data and instructions by all cores [4]–[6]. Unfortunately, these systems still exhibit a number of shortcomings that may cause a degradation of the overall system performance:

- Increase of the critical path between processor and memory due to the arbitration network.
- Coupling heterogeneous cores with various memory data sizes calls for an additional multiplexing network increasing memory access latency even further.
- The unified address space can lead to access conflicts even if private data is addressed.
- The runtime allocation of memory is usually restricted due to the separated instruction and data paths.
- Memory protection is often omitted for regions within the shared memory resulting in security issues.

To cope with the aforementioned design challenges, this paper introduces a queue based memory management unit (Q-MMU). In order to resolve the gap between fast and predictive memory access on the one hand and the flexible nature of memory management on the other, the Q-MMU approach proposes a hierarchical structure of memory banks. Each memory bank can be accessed with the same delay, regardless of specific data widths of the PEs. In contrast to contemporary work, our approach employs an indirect, persistent access arbitration, that decouples the slow-path associated with conflict detection and address translation from the fast access path, that represents the critical path of the system.

Furthermore, the Q-MMU offers a higher flexibility for embedded software designers with regard to dynamic memory allocation and address space virtualization, enabling techniques like instruction preloading in MPSoC environment with dynamic scheduling [2].

The remainder of this paper is structured as follows: Section II presents an overview of related work and background information. Section III introduces the Q-MMU architecture and explains our design decisions in detail. In Section IV implementation results and benchmarks are evaluated. The last section summarizes the results and offers an outlook of future work.

II. Background and Related Work

A memory access system comprises an interconnect (e.g. a bus or switching network) and a control unit dedicated to address translation/decoding and conflict detection/resolving. In recent works [4], [5], [7]–[10] these tasks are performed dynamically within each memory access cycle, thereby increasing the critical path between master (e.g. PE, DMA, etc) and memory banks. In contrast, we introduce a passive conflict detection which is not exhibiting these negative effect.

Moreover, the address translation in contemporary work, if any, is static resulting in an Unified Address Space (UAS). Systems with a Non Unified Address Spase (NUAS) [11], [12] are relying on specialized PEs. In contrast, our Q-MMU provides a NUAS for every master without adaption of exiting IP.

Furthermore, recent approaches [13]–[15] addressing shared TCMs assume mainly homogeneous processors and an address-space-wide random memory access scenario. Consequently, the data path width remains constant in such cases and

978-3-9819263-2-3/DATE19/©2019 EDAA

isolation of memory is not present. In contrast, our architecture can incorporate different data widths without the introduction of bubble cycles and it is possible to enforce runtime memory isolation.

III. System Architecture

A. Overview

The key elements of the Q-MMU are i) a passive reconfigurable low-latency interconnect and ii) a queue based memory controller. The Q-MMU is designed to allow every master to access every memory bank with the same delay. Also, it employs a master-specific NUAS that can be configured at runtime. Furthermore, we employ a hierarchical approach in which the memory banks are addressed with the most significant bits (continuous address mapping). The banks are further divided into sub-banks addressed in an interleaved fashion. The purpose of this design is to allow masters with different interfaces to operate on the same data without wasting processing cycles. For every memory access exactly one bank will be activated. The number of active sub-banks is determined by the data width of the initiating master. Thus, the width of the sub-banks are determined by the masters with the smallest interface.

Fig. 1 provides a more detailed view of the Q-MMU. Parameters like the number of masters, of memory banks and the number of priority levels are determined at design time. In the shown example configuration the system is comprised of four memory banks, four sub banks and two priority levels. For simplicity reasons only one master is shown and the data path is omitted.

B. The Memory Controller

At the center of the controller is a scheduling unit which gets activated during every memory request. It first checks the base address valid table (BVT). The table contains valid flags for every base address of the master to indicate if the interconnect is currently configured for the emitted address. If the flag is set, no further actions are required and the interconnect arbitrates the master to the correct memory bank within the same access cycle. If the flag is not set, the scheduler will raise a busy signal and check the base address. In that case, two further tables are looked up. The base address translation table (BTT) is used to translate the virtual address to a physical address. This is similar to a translation look aside buffer (TLB) in systems with an OS and MMU with two differences. The number of masters and memory banks in embedded systems are usually small enough to hold a complete set of all possible translations in the BTT. In contrast, a TLB cannot hold all possible translations due to the fine grained translations needed by a traditional MMU and memory scalability requirements. Furthermore, the BTT is directly programmable to change the mapping between base addresses and memory banks.

The privilege and priority table (PPT) are used to determine if the master has the correct access rights to issue the transfer. If the check fails, the master is not arbitrated and an exception is issued. Otherwise the request is put into an arbitration queue.

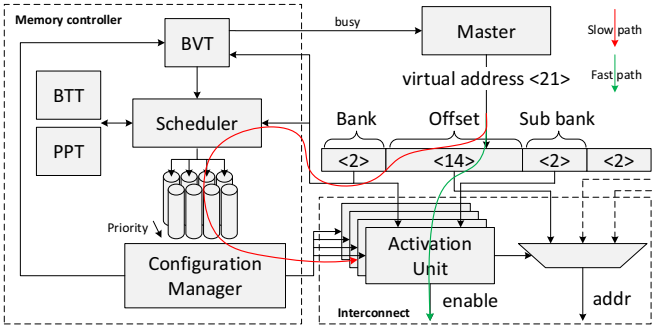


Fig. 1: The Q-MMU architecture

Each memory bank has a separate arbitration queue for each available priority. According to the result of the PPT lookup, the scheduler places the request in the correct queue.

When the configuration manager receives a new request in one of the queues, it follows a given arbitration procedure. If a master of same or higher priority is already requesting the memory bank, the manager waits. Otherwise the manager configures the activation units according to the request and sets the valid flag in the BVT. In the next cycle the busy flag is cleared and the master can continue its operation.

C. The Interconnect

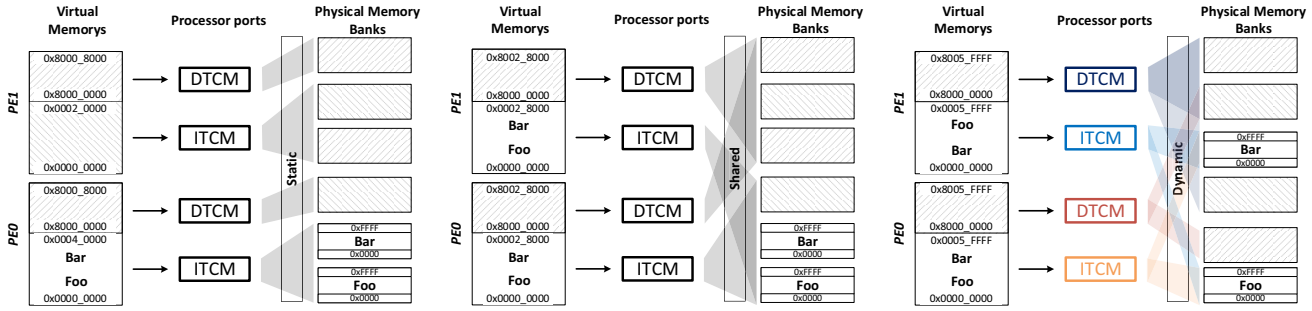
The Q-MMU interconnect mainly consists of the passive activation units for arbitration. As shown by others [7], the arbitration scales logarithmically with the number of master ports and memory banks. In contrast, the Q-MMU interconnect does not perform any conflict detection but gets programmed by the memory controller as described earlier. The result is a fast arbitration path and a slow control path (see Fig. 1).

D. Key Advantages

There are several major advantages in the proposed Q-MMU architecture:

First, the address translation and the conflict detection is pushed completely out of the fast-path between processor and memory. With this passive conflict detection we obtain a shorter critical path compared to the active version applied in contemporary work, which can be used to raise the overall system frequency, see section IV.

The second advantage is the individual, re-programmable address translation for each master which is usually not present in embedded systems and protocols (e.g. AXI). Though not as fine grained as traditional MMUs, it allows the mapping of base addresses to memory banks individually for each master at runtime. Fig. 2 illustrates the difference between our system and a typical static interconnect (2a) and a shared interconnect (2b) like the one presented by [7]. With the static and shared interconnect, the two programs Foo and Bar are directly mapped into the virtual address space of the instruction port (ITCM). In contrast to this, the virtual address space of the PEs with the Q-MMU interconnect greatly increases because each master port can be arbitrated to a flexible number of memory ports at runtime.



(a) Static interconnect (b) Shared interconnect (c) Q-MMU interconnect

Fig. 2: Functional comparison between static, shared and Q-MMU interconnect

Another advantage of the Q-MMU compared to other interconnects like [7] is the runtime configuration of priorities and privileges. This allows a single master to run without interrupts when executing time critical code and provides runtime isolation between different masters.

IV. Results

To evaluate the Q-MMU architecture we first measure the hardware costs of the system. Second, we use software benchmarks to compare the runtime results against traditional round-robin based interconnects.

A. Hardware analysis

We conduct the hardware analysis with an automated design flow which takes a given set of input parameters (e.g. number of masters and memory banks) and which outputs a synthesizeable Verilog description. The results are accomplished with a $22\,\mathrm{nm}$ library for Globalfoundries FDSOI process.

As mentioned earlier, the logic between master and memory bank is often the critical path in embedded systems. Hence we start with an evaluation of the Q-MMU interconnect. Fig. 3(a) compares the critical path of the Q-MMU interconnect both, with an interconnect of private closely coupled memories (static) and the shared interconnect presented by [7]. The results are given for different network cardinalities $mp \times mb$ where mp and mb are the number of master ports and memory banks respectively. All ports have a width of $32\,\mathrm{bit}$ and the banks have an address range of $11\,\mathrm{bit}$.

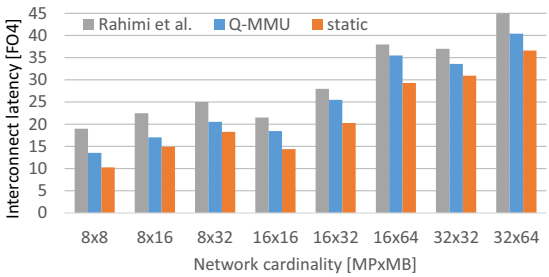
Comparing the static interconnect with the Q-MMU we see an increase in latency of around $25\,\%$ for the smallest configuration, which is due to the higher wiring delay. The interconnect of [7] has an additional delay of around $40\,\%$. These results prove our concept introduced in the previous chapter. By pushing the conflict detection out of the path between processor and memory (passive conflict detection) we obtain a far shorter critical path than interconnects with active conflict detection. Nevertheless for higher cardinalities the difference diminishes because the wiring delay grows faster than the arbitration logic and the conflict detection logic.

Furthermore, Fig. 3(b) compares the area as NAND2 equivalents of the static, shared and Q-MMU interconnect. The static approach has the lowest area consumption. This meets expectations because the fan-out for the static approach is mb/mp times less than a fully connected interconnect. In addition, the area of the shared interconnect with active conflict check increases slightly compared to the Q-MMU interconnect because the active conflict detection needs slightly more logic than the passive one.

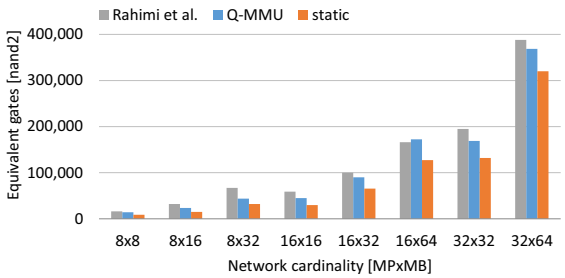
B. Benchmark analysis

To compare the overall system performance we choose three configurations with different memory systems: AXI4, Q-MMU and a shared system like [7]. Please note that the following benchmarks are intended to simulate the worst case performance with a very high workload on a single bank. For a comparison of the performance of the continuously and interleaved address schemes the reader may refer to [16] instead.

To obtain a high workload we choose the Dhrystone benchmark [17] and execute it on an ARC HS processor. This combination leads to an utilization of the instruction port of around $90\,\%$. Fig. 4(a) shows the runtime results for a different number of CPUs concurrently executing the benchmark from the same instruction memory. The dashed lines are showing the earliest finish time (EFT) of a CPU with high priority access. The latest finish time (LFT) shows the result for the CPUs with non priority access. The execution time is normalized with the time obtained from a single core setup with private TCM. Since the system of [7] does not have a priority handling but uses a round robin access scheme the EFT and LFT are the same. For the AXI interface we use the specification from the Xilinx implementation. As it can be seen for one active CPU, the execution time with the Q-MMU and the shared system are over $60\,\%$ faster compared to the AXI interface. This is due to necessary re-arbitration of the AXI system for back-to-back transfers (none burst mode). As expected the shared system performs even better than the Q-MMU because conflict detection is resolved in each cycle whereas the Q-MMU uses a persistent arbitration. However, it can be seen that the advantage decreases for higher workloads and for 4 CPUs the EFT of the Q-MMU systems outperforms the shared system. The reason is, that for high workloads the additional



(a) Latency comparison.



(b) Cell count comparison.

Fig. 3: Latency and cell count comparison between static, shared and Q-MMU interconnect

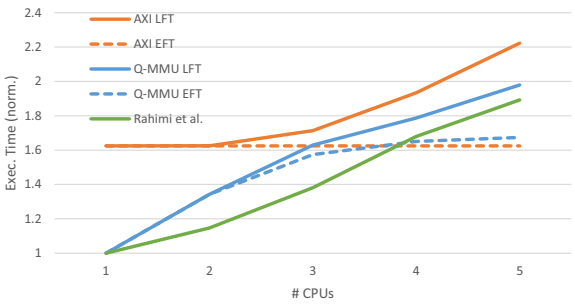


Fig. 4: Dhrystone execution time for a system with AXI, Q-MMU and shared interconnect.

arbitration latency of the Q-MMU can be hidden, since almost every access results in a conflict that needs to be resolved.

However the direct comparison is not completely fair, since we designed the Q-MMU system to have a shorter critical path, thus increasing the overall operating frequency. If we assume that the shown $40\,\%$ decrease in the critical path can be used to increase the frequency by $20\,\%$, the Q-MMU systems outperforms the shared system for every workload.

V. Conclusion and Outlook

In this paper we proposed the Q-MMU: a fully reconfigurable, shared, low-latency memory management unit for heterogeneous, embedded systems. The introduced passive conflict detection results in a shorter critical path compared to contemporary work, which leads to an overall increase in system performance. The hierarchical system design enables masters with different interfaces to operate on the same data without additional delay. Furthermore, it features a non unified address space, resulting in data sharing and isolation capabilities at runtime. The reconfigurable address space can be used to preload instruction and data in environments with dynamic scheduling where position independent code is not available.

In future projects we would like to extend the system to obtain a better performance for high priority users. Also, we would like to decrease the time needed for arbitration to even increase the performance for concurrently working processors.

Acknowledgment

This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 692519 (“PRIME”). This Joint Undertaking receives support from the European Union’s Horizon 2020 research and innovation programme and Belgium, Germany, France, Netherlands, Poland, United Kingdom.

References

[1] TI, “OMAP 4 mobile applications platform,” 2011.
[2] S. Haas *et al.*, “A Heterogeneous SDR MPSoC in 28 nm CMOS for Low-Latency Wireless Applications,” in *DAC*. ACM, 2017.
[3] G. Venkatesh *et al.*, “Conservation cores: reducing the energy of mature computations,” in *ASPLOS XV*. ACM, 2010.
[4] D. Melpignano *et al.*, “Platform 2012, a many-core computing accelerator for embedded SoCs,” in *DAC*. IEEE, 2012.
[5] J. Ax *et al.*, “CoreVA-MPSoC: A Many-Core Architecture with Tightly Coupled Shared and Local Data Memories,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, 2017.
[6] A. Olofsson, “Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip,” *CoRR*, vol. abs/1610.0, 2016.
[7] A. Rahimi *et al.*, “A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters,” in *DATE*. IEEE, 2011.
[8] I. Loi *et al.*, “Exploring multi-banked shared-L1 program cache on ultra-low power, tightly coupled processor clusters,” in *CF*. ACM, 2015.
[9] H. Wong *et al.*, “Demystifying GPU microarchitecture through microbenchmarking,” in *ISPASS*. IEEE, 2010.
[10] M. Gautschi, D. Rossi, and L. Benini, “Customizing an open source processor to fit in an ultra-low power cluster with a shared L1 memory,” in *GLSVLSI*. ACM, 2014.
[11] A. Y. Dogan *et al.*, “Multi-core architecture design for ultra-low-power wearable health monitoring systems,” in *DATE*. IEEE, 2012.
[12] C. F. Fajardo *et al.*, “Buffer-Integrated-Cache: A cost-effective SRAM architecture for handheld and embedded platforms,” in *DAC*. IEEE, 2011.
[13] F. Conti, A. Marongiu, and L. Benini, “Synthesis-friendly techniques for tightly-coupled integration of hardware accelerators into shared-memory multi-core clusters,” in *CODES+ISSS*. IEEE, 2013.
[14] M. R. Kakoee, V. Petrovic, and L. Benini, “A multi-banked shared-l1 cache architecture for tightly coupled processor clusters,” in *International Symposium on System on Chip (SoC)*. IEEE, 2012.
[15] M. Dehyadegari *et al.*, “Architecture Support for Tightly-Coupled Multi-Core Clusters with Shared-Memory HW Accelerators,” *IEEE Transactions on Computers*, vol. 64, no. 8, 2014.
[16] A. Tretter *et al.*, “Minimising Access Conflicts on Shared Multi-Bank Memory,” *TECS*, vol. 16, no. 5, 2017.
[17] R. Longbottom, “Dhrystone Benchmark.”