MemorEDF

Authors omitted for review.

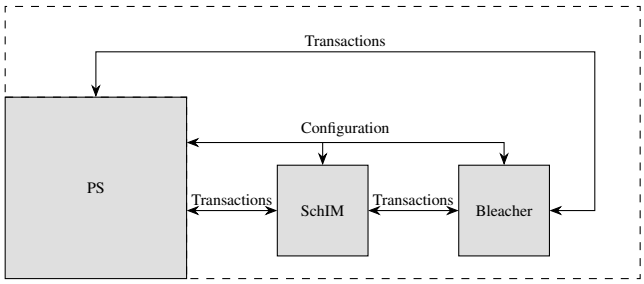


Fig. 1: Caption

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

This paper makes the following contributions: 1) We demonstrate that without any modification to the SoC circuitry, a configurable module could be interposed between the core and memory controller to perform traffic shaping. The proposed module operates at the granularity of transactions based on their interarrival time. 2) We perform transaction-level memory scheduling in configurable hardware. Various scheduling policies implemented in the module and could be configured at run-time.

3) A groundbreaking view on memory access scheduling, namely MemorEDF, has been proposed. In this method, the memory is viewed as a single resource .....

4) We provide a run-time profiling interface to log the transaction-level behavior of an application. Recorded data exploited to make and enforce efficient scheduling decisions for EDF and LLF policies.

5) This work implement a sandbox framework to test scheduling platforms on real hardware. Newly proposed scheduling techniques can be evaluated against the actual device, without rewiring the platform.

6) We implement and evaluate a full-stack design that includes the memory scheduler hardware module. Using this implementation, we analyze several well-known scheduling algorithms, namely, Earliest Deadline First (EDF), Least Laxity First (LLF), Time-division multipleaccess (TDMA).

II. Overview

III. Related Work

Ask Tomasz and Gero for their ECRTS article

We should look at this paper "BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors"

Citation list:

- PREM
- MemGuard
- PLIM
- Three Phase Model

IV. System Background

A. Cache and DRAM temporal partitioning

B. Programmable Logic in the Middle(PLIM)

Here we detail the necessary background to understand how to interpose a module between a traditional multi-core processor and main memory.

Commercially available SoCs that integrate a traditional embedded multi-core processor system (PS) and a block of programmable logic (PL) with high-performance PS-PL communication interfaces. There are high-performance masters (HPM) and high-performance slaves (HPS) to send and receive transactions to and from the PL, respectively.

The underlying mechanism is the ability to intercept memory transactions originated from the processors inside the PS, at the PL. Transactions are then forwarded from the PL again toward the memory controller inside the PS. The primary mechanism of PS-PL and PL-PS redirection of a transaction is called the Memory Loop-Back. Loop-Back is done through address bit manipulation of the transaction such that it falls in the range of the target HPM(HPS) for the PS-PL(PL-PS) interception. In this way, the main memory content is accessed, but through a programmable environment. It is possible to act on the characteristics of the traffic that now traverses the PL. For example, in the PL, it is possible to direct the transaction to arbitrary modules before, eventually, redirecting it back to PS and the memory controller, ultimately.

This provides a unique capability of manipulating individual memory transactions. Hence, by sitting between CPUs and main memory, PLIM is exploited to perform memory scheduling. A configurable memory scheduler in the middle, namely SchIM, is designed to implement several elected scheduling policies of Fixed Priority, TDMA, and Memguard. With SchIM, now we can enforce policy at the level of the transaction altogether by the hardware.

1) PS-PL SoCs:

2) Memory-Loopback:

3) Cache Bleaching:

4) Transaction-level Inspection:

5) Transaction-level Profiling:

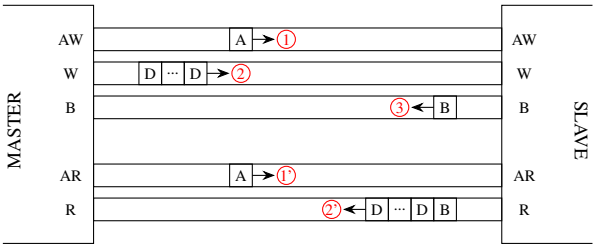


Fig. 2: Caption

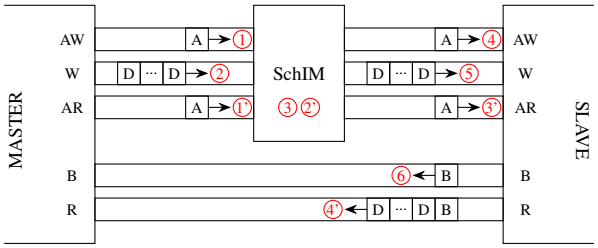


Fig. 3: Caption

C. Jailhouse, the partitioning Hypervisor

1) Partitioning:

2) Run-time Zero-Copy Recoloring support:

D. Advanced eXtensible Interface (AXI)

The Advanced eXtensible Interface (or AXI for short) is a widely spread open specification bus protocol proposed by ARM [1] and exploited by Xilinx to interface both the PL side and the PS side.

The AXI protocol is based on the master-slave duality. Typically, the former is in charge of instantiating transactions directed to any of the slaves composing the system. On the other hand, the latter, does not emit any transaction but receive the requests emitted by a master and answer them. The masters and the slaves communicate between each other through five different channels named AW, W, B, AR and R as illustrated in figure 2. A write transaction will start first by its address phase (1). That is, the transmission through the channel AW of meta-data regarding the transaction such as the destination address, the transaction ID, the amount of bursts and so on. Upon the completion of this phase, follows the data phase (2) which, as its name suggests it, consists in the transmission of the payload itself through the W channel. Thereafter, comes the response phase (3) performed on the B channel. This phase, the only one being initiated by the slave, is used to inform the master whether the transaction has been completed successfully. The transmission of a read transaction is carried out in a similar way. In fact, as for the writing, the address phase (1') is transmitted through the equivalent channel called AR and is directly followed by the data phase (2'). However, unlike the writing scheme, the data being fetched, the data phase is instantiated by the slave. The reading response phase is performed simultaneously and is thus merge within the R channel.

The protocol is said to be asynchronous as transaction behaves similarly to packets each having a dedicated ID. Hence, multiple outstanding transactions can be emitted successively by a single master which can managed them in an out-of-order manner.

V. SchIM Implementation

As previously mentioned, SchIM is a PLIM module that performs a memory loop-back through the PL side similarly to [3]. The objective of the module is to arbitrate the access of the bus to the main memory between the different cores of the PS side at the transaction level by enforcing a given policy. Roughly, the module receives transactions from the PS side, acknowledges them and finally repeat them with the main memory as destination. The exact order in which inter-core transactions are being repeated is decided by an embedded on-chip hardware scheduler.

The present section exposes how SchIM interacts with the remaining of the systems in V-A, how its internal logic enforces transaction scheduling policies in V-B and finally, an example of the transaction life cycle within the SchIM module is provided in V-D.

A. Altered communication scheme

In order to achieve the objective of re-ordering transactions, one must alter the standard AXI communication scheme explained in the subsection IV-D. To this end, we instantiate a in-between the master and the slave a pass-through module named SchIM as depicted in figure 3. This module is the one in charge of the re-ordering of the transactions and does so by intercepting on the fly the transactions emitted by the masters before they reach the desired slaves. As shown in the figure 3, only the phases instantiated by the masters (i.e. address phase on AW and AR and the data phase on W) are intercepted for re-ordering by SchIM. The introduction of SchIM has a direct consequence on the overall communication scheme. Indeed, while the response phases on channels R and B remain unchanged, the address and data phases are duplicated. Consequently, a write transaction will start exactly as in the standard AXI scheme with its address phase (1) and data phase (2). These two transactions are buffered within the SchIm module in (3) and only repeated when decided by the latter's internal logic. This release of the transaction leads to the initialisation of two new address and data phase (4) and (5). Finally, the response phase (6) goes directly from the slave to the master without being intercepted. The same modifications apply to the transmissions of read transactions as the address phase (1') is being buffered in (2') for some time before being re-emitted in (3'). As for the writing, the response phase (4') is not intercepted by SchIM.

B. Micro-architecture

The SchIM module is composed of many sub-modules that can themselves be grouped into three different domains. In fact, as illustrated in figure 4, one can distinguish the

scheduling domain, the *queuing domain* and the *interfacing domain*.

The scheduling domain encompasses all the sub-modules that enable arbitration of the bus between the transactions issued by the different cores of the PS side. Hence, this domain boasts several transaction schedulers implemented at the hardware level. The scheduling policies offered by SchIM include Fixed Priority (FP), Time Division Multiple Access (TDMA), Earliest Deadline First (EDF), Least Laxity First (LLF) and MemGuard (MG). Each of the parameters required by the aforementioned algorithms such as the priorities, the periods, the deadlines and the budgets are re-configurable at the run-time thanks to the inclusion of a configuration port. Finally, the scheduling domain is also the one in charge of the control of the remaining the SchIM module, driving and selecting the adequate signals and ensuring the coherence and integrity of the data.

The queuing domain is in charge of the storing the incoming transactions emitted by the PS side. The motivation behind the use of queues is implied by the fact that all the masters located on the PS side share a common AXI bus (namely HPM0 as shown in figure 1). Therefore, in order to cancel the Round Robin arbitration policy applied in the PS side and in order to avoid that one high priority core is stalled by a lower priority one, each core is granted a queue within the SchIM module. Not only the queues act as containers and buffers for transactions, they also embed logic and provide information to the scheduling domain regarding their current state in order to avoid the queues to overflow or underflow similarly to the producer-consumer problem. As suggested by figure 4, transactions are inserted to the adequate queues on the basis of the emitters identifier via the dispatcher module. Similarly, transactions are evicted from their queue, routed by the selector module and sent directly to the output of the module upon the action of the scheduling domain.

The interfacing domain encompasses the sub-modules in charge of interfacing both the scheduling domain and the queuing domain with the remaining of the system using the AXI protocol. More accurately, three sub-modules compose this domain, the configuration port previously mentioned, the packetizer and the serializer. While the packetizer and the serializer serve the purpose of slave and master ports, they are also in charge of respectively transforming the AXI transactions into an equivalent packet and to transform these packets back to a AXI compliant transactions. The need for packetizing (i.e. flattening) the AXI transactions is driven by the necessity of storing transactions that are by nature serial within the queuing domain. For instance, a standard AXI transaction is composed of one address phase followed by a data phase which itself composed of multiple successive bursts.

C. Embedded Schedulers

1) Fixed Priority: The Fixed Priority scheduling module aims at enforcing prioritization of the traffic of the cores. The priority ordering is explicitly defined by the user through

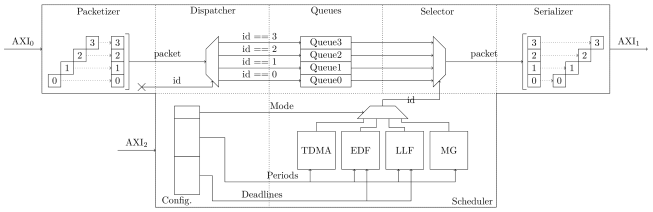


Fig. 4: Caption

the configuration port. While SchIM only has four queues, 16 different levels of priority are offered. Note that, as it stands, the ordering is strict, meaning that two cores cannot be assigned with the same priority.

The FP scheduling module only needs two information: the priority associated with each queue and whether a given queue contains at least one buffered transaction. Intuitively, the module logic consists in always considering the highest priority hence, a argmax function is implemented. However, lower priority queues must also be considered when higher priority queue do not have transactions. Consequently, the user defined priority is temporarily considered as 0 when its corresponding queue does not contain transaction.

2) Time Division Multiple Access: The Time Division Multiple Access (or TDMA) scheduling module provided by SchIM is a non-work conserving policy that splits the bus usage between the cores for given periods (also referred to as slots). The proposed embedded TDMA module enables the user to specify a different TDMA slot size for each core. The periods are expressed in clock cycles, enabling a fine grained granularity. The configuration port enables the end-user to specify and change the periods at runtime.

The implementation of the modules relies on a single register used as a counter. In fact, we use this register to count the time elapsed in the current TDMA hyper-period (i.e., the sum of all the cores period) and is reset to 0 once this hyper-period reached. Alongside this register, a logic is there to determine in which core slot the counter actually is and to forward the information to the queue selector. The logic is able to determine the core to schedule by summing the period of each previous cores. Provided that the current value of the counting register is contain between the sum of the previous periods and the sum of the previous periods and the current one, the information forwarded to the remaining of the system will be the core id corresponding to the interval.

DH: @DH TODO improve description!

3) MemGuard: The proposed MemGuard transaction scheduling policy is inspired by the Software-based bus regulation technique ??. The latter has been proved to ensure memory isolation for all the cores involved. Unlike the original MemGuard, the proposed version does not rely on memory budgets and replenishment periods. Instead, it enforces inter-arrival time (i.e. a guaranteed minimal period) between two consecutive transactions emitted by a given core. This distinguishes our approach from [2].

This scheduling module is implemented as follows. For each of the queues to schedule, the module has a register counting the time elapsed since the start of the period. Once this counter has reached the period set by the user (through the configuration port), the module checks if the queue corresponding to the core contains any transaction. In the case where a transaction is available in the corresponding queue, the latter is forwarded to the output of SchIM (i.e., the serializer) and the counting register is reset to 0. Otherwise, the counting register is blocked to the desired period until a transaction is available for scheduling in the corresponding queue, leading to the counting register to be reset to 0. Any tie between two cores is solved using a fixed priority arbitration defined by the user thanks to the configuration port.

D. Transactions Life Cycle

Let us consider a system with four cores (noted $C = \{c_0, c_1, c_2, c_3\}$) sending transactions $T = \{t_0, t_1, ..., t_n\}$ to the SchIM module. Consequently, the latter boasts four queues (noted $Q = \{q_0, q_1, q_2, q_3\}$) buffering the transactions under the form of packets $P = \{p_0, p_1, ..., p_n\}$ where $p_i = Packetizer(t_i)\ \forall i \in [0 : n]$.

In the present example, we will assume t_1 as being the transaction under analysis. The latter is emitted by c_2 in direction of the SchIM module. The packetizer receives this transaction and, once the AXI protocol completed, transform it into an equivalent packet $p_1 = Packetizer(t_1)$. Following this transformation, the newly created packet is forwarded to the dispatcher which, thanks to the emitter's id embedded within the transaction, is re-routed to the corresponding queue q_2 (since emitted by c_2). After the insertion of p_1 in q_2, the state of the queuing domain is as follows: q_0 has two packets p_0 and p_k and q_2 only has p_1. At this point, q_0 is considered for scheduling by the scheduling domain. In consequence, p_0 is forwarded to the serializer through the selector. Simultaneously to the reception of the packet by the serializer, the latter receives an activation signal from the scheduling domain informing the serializer that the packet is valid and that a transaction can be started. Similarly to the packetizer, the serializer will transform the packet p_0 back to its initial AXI transaction form $t_0 = Serializer(p_0)$. Thereafter, once the t_0 has been sent, the serializer will inform the scheduling domain via a signal, that he is ready to accept the next packet as input. Upon the reception of this signal, the scheduling domain will both re-direct the latter to the queue of the previous packet to indicate that it has been consumed and change the selected queue according to the scheduling policy so that the first packet of this queue can be forwarded to the serializer through the selector module. In the present example, the "consumed" signal forwarded by the scheduler is sent to q_0 which is then empty. At this instant, two scenarios are possible:

1) q_0 is still considered for scheduling following the selected scheduling policy. Therefore, as q_0 is empty, it outputs an "empty" signal received by the scheduling domain. The latter then decides to not send any activation signal to the serializer because there is nothing left to transmit in the selected queue. In other words, the access to the main memory is being stalled on purpose by the scheduling policy i.e. the scheduling policy is not work conserving. For instance, such a scenario could happen in the case of TDMA or if all the queues are empty. The logic will resume as soon as the selected queue is filled.
2) q_2 is now considered instead of q_0 for scheduling. In this case, the "consumed" signal is repeated to q_0 while the queue ID changes in order to select q_2. This results in the packet contained inside q_1 to be forwarded to the selector.

VI. Evaluation

We have performed a full system implementation on a Xilinx ZCU102 development board, featuring a Xilinx Zynq UlraScale+ XCZU9EG SoC. For this work, we utilize a combination of synthetic and real benchmarks. Our synthetic benchmark.

Talk about IO intensive and memory intensive benchmarks to be designed.... We also include a study of the behavior of real applications from the San Diego Vision Benchmark Suite (SD-VBS) [4], which comes with multiple input sizes....

DH: Maybe we could use AES from MiBench ?

As mentioned earlier, the high-performance master ports, namely HPMs, serve as a gateway from the PS to the PL. We implemented our scheduler IP, SchIM, responding under the physical addresses of HPM0 right after the PS. By doing so, any transaction toward the PL has to go through the SchIM. Hence, all memory requests yield to the scheduling policy being enforced by the scheduler. Design features of SchIM implementation and configuration interface are detailed in section ??.

At the preprocessing stages, an AXI Performance Monitor (APM) unit connected to the bus interface between the HPM0 and the scheduler. It is crucial to affirm that this medium does not affect the transaction flow toward the PL. APMs are powerful tools available in the PS and instantiable in the PL capable of measuring primary performance metrics (for AXI4, AXI4-Lite, or AXI4-Stream-based systems) such as bus latency for specific master/slave or amount of memory traffic for the particular duration. Moreover, APM offers the functionality of logging the necessary information about data transfers between any master and slave communicating in the AXI protocol. In this work, we program APMs to profile the behavior of the benchmarks to analyze the task's deadline. SchIM employs profiled information on task deadlines to make scheduling decisions (e.g., enforce EDF policy) for hard real-time jobs.

A. Performance degradation

B. Platform Capabilities

Here, discuss Fig.5

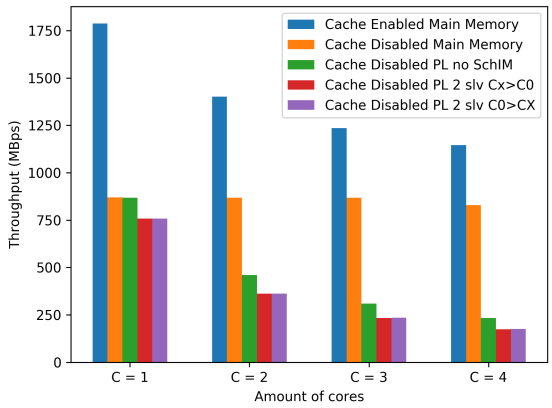


Fig. 5: Board bandwidth

TODO: outdated version. Has to be repeated for cached transactions only with the latest version of SchIM.

Broad evaluation of the bandwidth offer by the ZCU102 depending on the route considered.

C. Internal Behaviour of SchIM

Here, show and emphasize on the behaviour of SchIM with Fig.6

Here, we use the trace snapshots

D. Memory Isolation

- FP: The benchmark running alone with the highest priority VS. the same setup with mem-bombs
- TDMA: The benchmark running alone with all the cores having a slot of the same size VS. the same setup with mem-bombs
- MG: The benchmark running alone with all the cores having a given periodicity VS. the same setup with mem-bombs

VII. Overhead

DH: Is this section still relevant ?

A. Discussion

VIII. Conclusion

DH: As stressed by @Renato, we must mention that SchIM is the building brick for profile-based Bus scheduling/regulation!

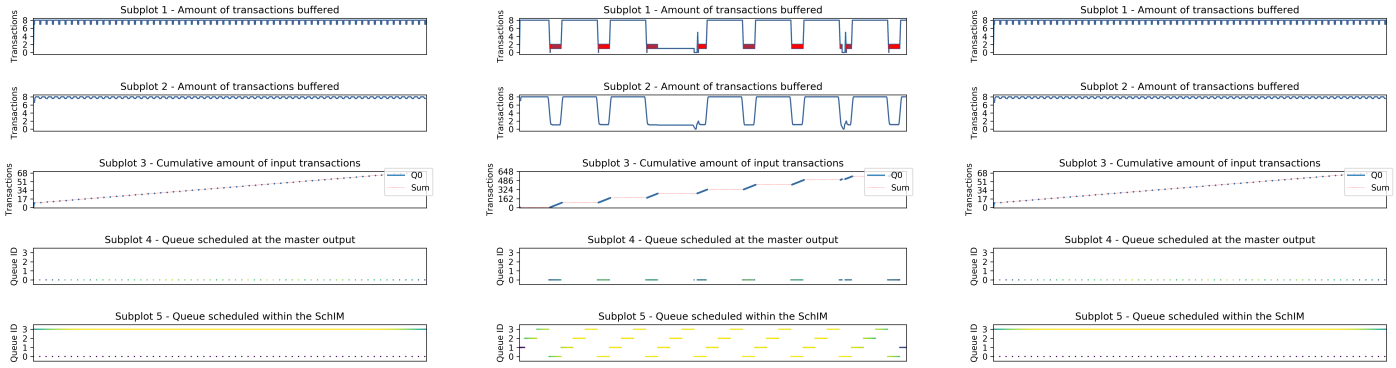
References

[1] ARM, "Amba axi and ace protocol specification," Tech. Rep., 2019. [Online]. Available: https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf

[2] F. Farshchi, Q. Huang, and H. Yun, "Bru: Bandwidth regulation unit for real-time multicore processors," *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 364–375, 2020.

[3] R. M. S. Roozkhosh, "The potential of programmable logic in the middle: Cache bleaching," in *26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)*, Sydney, Australia, April 2020, conference.

[4] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 55–64.



(a) Put your sub-caption here

(b) Put your sub-caption here

(c) Put your sub-caption here

Fig. 6: Put your caption here

TODO: outdated version. Has to be repeated for cached transactions only with the latest version of SchIM and the plots have to be reworked.