Efficient Bandwidth Regulation at Memory Controller for Mixed Criticality Applications

George Tsamis, Stamatis Kavvadias,
Antonis Papagrigoriou, Miltos D. Grammatikakis,
Kyprianos Papadimitriou
Technological Educational Institute (TEI) of Crete
Heraklion, Crete, GR 71004, Greece
Email: gtsamis, kavadias, apapa, mdgramma,kpapadim@cs.teicrete.gr

Abstract—**We design a bandwidth regulation module, by adapting and extending the algorithm of MemGuard Linux kernel module for hardware implementation. Our extensions differentiate among NoC sources with rate-constrained and best-effort traffic provisions, support a violation free-guaranteed operating mode for rate-constrained flows, and support dynamic adaptivity through EWMA prediction. Our strategies enhance support for mixed criticality applications on MPSoCs. C++-based statistical simulation shows improvements over hardware adaptation of the original MemGuard algorithm without our extensions. Using SystemC, we further evaluate MemGuard at the memory controller of a NoC-based SoC model using an MPEG4 traffic model and compare its hardware cost using synthesis from Xilinx Vivado HLS and Vivado, with ARM AMBA AXI4 and a 4x4 STNoC instance.**

I. Introduction

Multi-Processor Systems on Chip (MPSoC) offer tremendous potential in embedded systems, due to combining computational capacity and energy efficiency. However, current MPSoC architectures have significant limitations in supporting safety-critical applications in space, avionics and transportation industry [1]. Thus, modern shared memory-based MPSoCs must efficiently integrate mission-critical and non-critical subsystems on the same platform to support Quality of Service (QoS) features that provide latency and bandwidth guarantees in an automated way [2].

Although scheduling mechanisms at the memory controller of an MPSoC translate memory requests into sequences of SDRAM commands [3] and manage memory traffic requests to achieve minimum latency and optimized bandwidth utilization, most memory controllers usually suffer from high unpredictability of dense memory accesses. Therefore, current memory controller technology experiences frequent guaranteed bandwidth violations, which destabilize system critical tasks.

Some previous attempts towards real-time guarantees, with soft or hard real-time constraints, have relied on a hardware approach at the router level, which stores packets in different queues based on their criticality level and serves them (usually) in weighted order based on a restrictive resource reservation model [4] [5] [6]; this model is usually non-adaptive, as it is unable to support look ahead functions. Moreover, current memory controllers which support mixed criticality applications focus on the worst-case execution time (WCET) scenario, usually ignoring overall system efficiency (e.g. bandwidth utilization) of soft real-time and non-critical tasks that share the same resources [7].

In this context, modern SDRAM controllers are classified as statically or dynamically scheduled [8]. The first class is highly predictable in terms of memory performance (bounding the execution time), since they operate based on pre-computed SDRAM patterns at design time (static schedules). In general, these controllers suffer from poor performance and limited flexibility which restricts them to a small set of applications. Conversely, the second class uses smart on demand scheduling, based on the currently running tasks. Therefore, these systems are much less predictable, but more flexible in organizing concurrent accesses to multiple addresses.

Successful management of guaranteed bandwidth to achieve Quality of Service (QoS) is based on advanced traffic sharing and reservation techniques. Different approaches have been developed to successfully manage shared resources using hardware or software techniques, e.g. at CPU [9] [10], GPU [11], router [4] [5] [10], memory [6] [7] [8] [12], or network interface level [13]. However, these approaches involve specialized hardware, and are not so generic, autonomous and configurable, when they are deployed in software.

Commercial-Off-The-Shelf (COTS) systems may exploit efficient, generic and easily manageable software algorithms that can be directly installed (in user or kernel-space) as regulators of bandwidth utilization, adjusting overall memory system scheduling. The Linux Resource Kernel approach supports real-time resource management and reservation, while serving tasks with real-time constraints [14]. It receives information about resource requirements from recently activated threads during their initialization cycle, which means that all user- and kernel-specific tasks must be aware a priori of their maximum resource needs (during simultaneous access to multiple resources), which is not always feasible. It is therefore not practical for a single resource kernel to deal with the individual real time deadlines and resource needs of all tasks.

II. Background and Motivation

Unlike traditional state-of-the-art real-time scheduling algorithms, which deploy weighted round robin (WRR), deficit round robin (DRR), or aging protocols, the MemGuard algorithm by Heechul Yun et al. [12] [15], distributes the minimum guaranteed memory bandwidth based on per-core reservations and a reclaiming algorithm. Reclaiming, is based on past traffic demand (history) and residual guaranteed bandwidth. Its self-adaptive mechanism is similar to an extended self-adaptive Dynamic Weighted Round-Robin (DWRR) [16], and is especially important for efficient management of real-time traffic.

A. *Genuine MemGuard Algorithm*

The original MemGuard access control algorithm (called Genuine MemGuard) is focused on per-core allocation of the minimum guaranteed memory bandwidth (denoted r_{min} in the algorithm), i.e., the bandwidth that can be guaranteed even for the worst-case memory

This research has been co-financed through the European FP7 DREAMS project under the Grant Agreement No. 610640.

access patterns. This metric intends to capture the effects of worst-case DRAM traffic patterns, which consist of repeated accesses to the same memory bank, on different bank rows each. It is essential to note that guaranteed bandwidth (r_{min}) is significantly less than the maximum attainable memory bandwidth (e.g., usually close to 20%), thus, it is important to favour, as much as possible, best effort traffic (BE), i.e., traffic in excess of r_{min}.

With MemGuard, any core can transmit critical, rate-constrained (RC) traffic up to a reservation of a portion of the minimum guaranteed bandwidth. MemGuard policy employs reservation and reclaiming. To achieve this, the algorithm enables a global repository (called G) mechanism. In each regulation period, bandwidth consuming components (e.g. cores) get an initial allocation of some portion of their reservation quantum (according to history-based prediction) and donate the rest of their reservation to G, from which dynamic reclaiming will occur. Allocation of guaranteed bandwidth from G, during the regulation period, is done on-demand, when a source exhausts its previous allocation. For correct operation, the aggregate bandwidth reservation to traffic sources must be less than the minimum guaranteed bandwidth of the system.

Once all guaranteed bandwidth has been exhausted, in a regulation period, MemGuard supports two approaches to distribute best effort (BE) bandwidth. In one, it allows all cores to freely compete for bandwidth, by posing regulation until the end of the period. In the other, it applies sharing of BE bandwidth proportionally to reservations, by immediately starting a new regulation period. There is no explicit provision for best effort traffic sources in MemGuard algorithm. As long as r_{min} is not exhausted, genuine MemGuard allows sources with a zero reservation (or sources that have otherwise exceeded their reservation), to repeatedly extract guaranteed bandwidth from G, up to the configurable minimum allocation (Q_{min}) each time[1].

B. Genuine MemGuard Weaknesses

In order to discuss some weaknesses in genuine MemGuard, in the following, we refer to *RC traffic sources*, or simply *RC sources* as the sources that have a non-zero reservation. With the MemGuard implementation discussed above, the following side-effects may occur during certain regulation periods:

(a) RC sources may *steal* guaranteed bandwidth from each other; in fact, one RC source may exhaust the global repository, while other RC sources have not yet demanded their full reservations;

(b) RC sources may *overbook* guaranteed bandwidth in excess of their needs and, thus, unnecessarily stall best effort traffic; and

(c) Traffic *sources with zero reservation* will acquire part of guaranteed bandwidth, before non-zero reservations of RC sources are satisfied, potentially leading to guarantee violations.

Therefore, in our view, MemGuard implementation choices target *average* instead of peak bandwidth reservation for RC sources to exploit bandwidth reclaiming. In fact, MemGuard provides a mode, called reservation-only (RO), that avoids side-effects (a) and (c) above, by removing prediction and reclaiming and allocating to RC traffic sources their full reservation in each regulation period. However, this mode performs poorly, in terms of executed instructions per cycle.

Finally, Genuine MemGuard predicts memory bandwidth requirements of each core using two extreme cases of Exponentially Weighted Moving Average (EWMA) that provide limited adaptivity. More specifically,

- ALL_PERIODS algorithm, uses average memory bandwidth demands during all previous periods (does not adapt to short bandwidth fluctuations);
- ONE_PERIOD algorithm, examines only the memory bandwidth demands of the previous period (adapts to abrupt changes).

III. Hardware MemGuard Extensions

Our MemGuard extension, intended for a hardware component placed in front of the memory controller, aims awareness of regulation decisions about concurrent RC and BE traffic sources, to prevent stealing of RC bandwidth, without removing reclaiming, including in violation-free (VF) mode. Furthermore, we detect overbooking by RC sources, to allow more BE-traffic in its place, and target more gradual bandwidth reclaiming, to allow adaptive targeting of *maximum* bandwidth requirements of RC sources. Finally, we extend traffic prediction, by exploiting more general EWMA, which computes a weighted average of all past periods based on a parameter lamda (λ) which determines the impact of history. For each CPU core, when a new MemGuard period starts, EWMA is calculated using the following formula:

$$S_t = \lambda \cdot Y_t + (1 - \lambda) \cdot S_{t-1}, for\ t > 1, 0 \leq \lambda \leq 1\ and\ S_1 = Y_1$$

where S_t is the consumed bandwidth moving average and Y_t is the actually consumed bandwidth, for some core, in the t^{th} regulation period. We use S_t as the prediction for period $t + 1$.

The following subsections explain the rationale in our implementation of hardware MemGuard focusing on its input interface (at receive-side), as well as our proposed new MemGuard extensions.

A. Blocking Excessive BE Traffic

A bandwidth regulating component implemented at the receive-side of a target device, such as a memory controller, needs a way to block excessive traffic from some sources in order to prioritize other sources. That is because, if a source is requesting excessively more than its reserved memory bandwidth, its packets will appear with a disproportionately high frequency at the on-chip network input interface of the hardware component (i.e. the memory controller). MemGuard would try to postpone the majority of these packets, but it would also needs to dequeue packets from the NoC, in order to advance RC traffic of other sources. Note that such misbehaving sources would require large amounts of buffering within our bandwidth regulating component, or could, otherwise, result in unbounded delay for other RC traffic.

To resolve this issue, we assume that all traffic sources in our system have a maximum number of outstanding accesses and, correspondingly, our bandwidth regulating module provides per source buffering of that amount. This allows postponing packets of a traffic source that exceeds its reservation, for as long as necessary. Notice that, such, per-source buffering is required, for a hardware MemGuard implementation, at the receive-side of the bandwidth-regulated system, regardless of whether we use the genuine, or our extended algorithm.

Usually, core caches employ a small number of Miss Status Holding Registers (MSHRs –e.g, see [17]), which limits the maximum number of outstanding accesses. This implies an access protocol that requires either a response or an acknowledgement, to release initiator resources. On the flip side, core uncached accesses and device DMA (usually, to uncached memory) are seldomly implemented with a maximum number of outstanding accesses, because initiator resources

[1] Q_{min} should not be set to zero, in the Linux version of MemGuard, because a core's bandwidth request in excess of reservation would lead to an interrupt on every cache miss, untill r_{min} is exhausted.

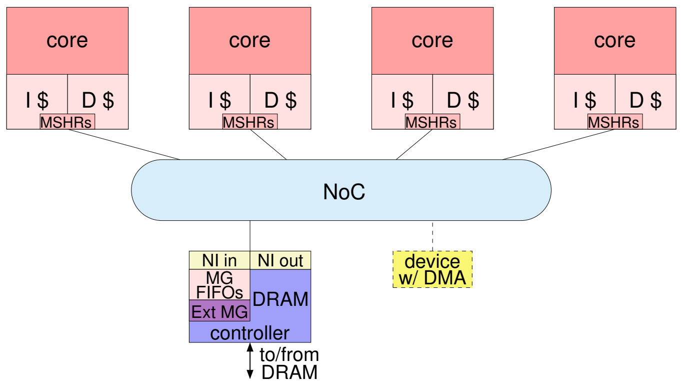

Fig. 1. Hardware MemGuard component system placement.

can be released immediately after the access is issued. Thus, our proposed hardware bandwidth regulating design would not work correctly in systems that support such memory traffic sources, without an outstanding access upper bound (see also subsection III-B). Nevertheless, our design supports an independent, relatively long queue for traffic sources of zero reservation, which can tolerate a number of accesses from sources without a maximum outstanding access limit, without compromising bandwidth regulation.

B. Placement and Limitations of a Hardware MemGuard Component

Figure 1 shows the intended placement of a hardware MemGuard component in a simple 4-core system. MemGuard is placed after the input network interface of the DRAM controller, to manage the order in which packets are delivered to the controller. Although such a component could also be used to regulate access to other types of memory (e.g., in section VI, we use it also for SRAMs), it is designed for DRAM bandwidth management. Per-source buffering FIFOs are presented as an independent part of our extentions, since they are not associated specifically to the MemGuard algorithm extentions, which we discuss in subsection III-C. The figure also shows a potential DMA-capable device without an outstanding access upper bound, which our design can tolerate up to a degree.

Other than such DMA devices, there are some other, potentially limiting, factors for the effectiveness of a hardware MemGuard component. Per-regulated-source buffering depends on the regulated sources having an independent NoC access point, so that they can be identified in the request packet header (to allow for an acknowledgement) and be classified separately. The case of uncached core accesses sharing the NoC access point of cached ones, would require special provisioning in the per-source buffering FIFOs component, to identify their protocol. More importantly, NoC access over shared caches, or coherence directories with placement or protocols that mask, modify, or completely remove the regulated source information from DRAM accesses, would complicate and could even make unattainable the design of a hardware MemGuard module.

C. Algorithm Extentions

Based on the discussion in subsections II-A, III-A and III-B, Figure 2 shows the block diagram of the intended hardware module for extended MemGuard. There is one FIFO per regulated source (serving what we call an RC flow), plus a BE-traffic FIFO (8 and $8 \cdot number\_of\_sources$ packet-entries respectively, in the implementation and evaluations of following sections). In each cycle of a regulation period, the algorithm accumulates some per FIFO state data (in Prepare Make Traffic block) and computes the conditions to pass a packet from each of the FIFOs (in Make RC/BE Traffic

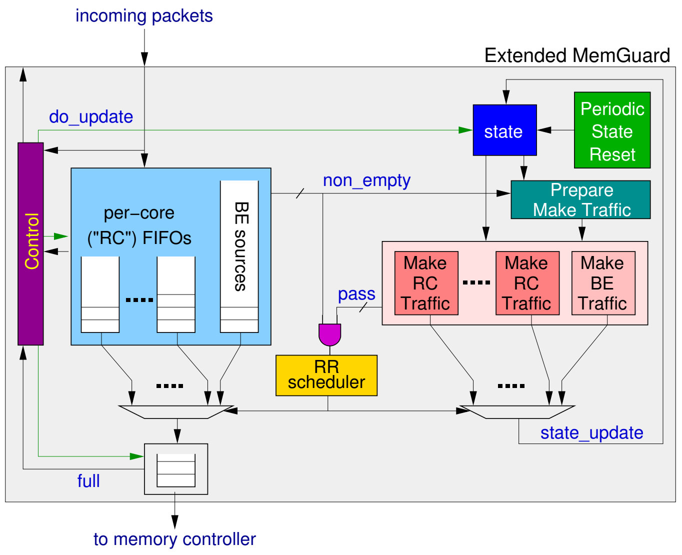

Fig. 2. Block diagram of Extended MemGuard.

blocks). A round robin (RR) scheduler iterates over Make Traffic decisions, to pass or postpone a FIFO's top packet, but only for non-empty FIFOs, and selects the first positive one. State affected by the Make Traffic decision is then updated. At the beginning of a new regulation period, per FIFO state is reset and EWMA state is updated (by Periodic State Reset block).

Figure 3 shows the implementation of Make Traffic, for regulated sources (make_RC_traffic, lines 1-20) and for BE traffic (make_BE_traffic, lines 21-30). A source's packets passed in the regulation period up to the current cycle is represented by u_i. The total reservation for the source is Q_i and the temporary allocation for the source q_i. For accounting in the hardware MemGuard component, everything is measured in time-slots (cycles). Thus, bandwidth (packets per time slot) is represented as *a number of time-slots in a period*. During each period, the remaining unused guaranteed bandwidth is monitored in guaranteed_window, reserved bandwith that is not yet granted to RC sources is tracked by reserved_slots and the number of remaining time-slots, past the current, is provided by residual_window. In make_BE_traffic, we allow a BE packet (from the FIFO of BE sources) to pass if:

(a) Reserved RC traffic is fully served for this period (line 23); or

(b.1) Remaining guaranteed slots exceed residual time slots in period (line 24); and either

(b.2.1) No more than all residual time slots in period (except current) are reserved (line 25); or

(b.2.2) There are no packets pending this cycle, in per-core FIFOs, for RC sources that have residual reserved bandwidth slots this period (line 26).

For the per-core FIFOs (make_RC_traffic), the case that u_i is more than q_i is handled first (line 3). If the global repository is empty (line 4), we may decide pass only if we would pass a BE packet (same condition as in make_BE_traffic, lines 5-8). If the repository is not empty, we try to reclaim bandwidth (line 14), i.e., an update of q_i, which we discuss below. Then, if u_i is less than q_i (for the original or updated value of q_i), we decide pass (lines 15-18).

Finally, bandwidth reclaiming (reclaim_bandwidth), shown in Figure 4, may extend q_i by a minimum amount, based on configuration parameter Q_{min}. In case u_i is less than the total reservation for the source (Q_i), q_i is extended by a portion of the residual reservation

```
1  function make_RC_traffic
2  begin
3    if ( ui $\geq$ qi ) then
4      if ( G = 0 ) then
5        if ( reserved_slots = 0 OR
6            ( guaranteed_window > residual_window AND
7              ( reserved_slots $\leq$ residual_window OR
8                no_critical_exists ) ) ) then
9          update next EWMA prediction;
10         ui $\longleftarrow$ ui + 1;
11         return true; // Let it pass
12       else
13         return false; // Don't let it pass
14     reclaim_bandwidth();
15   if ( ui < qi ) then
16     update next EWMA prediction;
17     ui $\longleftarrow$ ui + 1;
18     return true;
19   return false;
20 end
21 function make_BE_traffic
22 begin
23   if ( reserved_slots = 0 OR
24       ( guaranteed_window > residual_window AND
25         ( reserved_slots $\leq$ residual_window OR
26           no_critical_exists ) ) ) then
27     $ui_{BE\_FIFO}$ $\longleftarrow$ $ui_{BE\_FIFO}$ + 1
28     return true; // Let it pass
29   return false; // Don't let it pass
30 end
```

Fig. 3. Extended MemGuard algorithm.

for the source (line 6). When the source already exceeds its total reservation, if not in violation-free mode, we allow a minimum allocation from G (line 9), which may result in side-effects (a) and (c) of section II-B; in violation-free mode we only extend q_i if, at the beginning of the regulation period, there was in G guaranteed bandwidth not allocated in Q_i's, in lines 11-13 (G_{excess} = r_{min}-$\sum Q_i$).

IV. C++ to Synthesis Results

We have used a high-level synthesis tool (Xilinx Vivado HLS) on our component, described in C++ and bit-accurate SystemC. In addition, we have optimized the hardware, initially using Vivado HLS directives associated with code structures and, subsequently in an iterative process, modifying the C++ code and adding or changing directives (data pack for the data structures, horizontal array map for the FIFOs, unroll for loops, and inlining for most function calls). This process came across four reasons for modification of the original code:

1. Introduction of additional C++ methods (operator() was required for MemGuard class inclusion in a top-level function) and classes (to replicate per-source structures);

```
1  function reclaim_bandwidth
2  begin
3    minAlloc $\longleftarrow$ min( $Q_{min}$, G );
4    prev_qi $\longleftarrow$ qi;
5    if ( ui < Qi ) then
6      qi $\longleftarrow$ qi + min( (Qi - ui), minAlloc );
7    else
8      if ( VF = false ) then
9        qi $\longleftarrow$ qi + minAlloc;
10     else
11       minAlloc $\longleftarrow$ min( $Q_{min}$, $G_{excess}$ );
12       $G_{excess}$ $\longleftarrow$ $G_{excess}$ - minAlloc;
13       qi $\longleftarrow$ qi + minAlloc;
14   G $\longleftarrow$ G - (qi - prev_qi);
15 end
```

Fig. 4. Extended MemGuard algorithm (continued).

TABLE I
Hardware Cost (4 regulated sources).

	Vivado HLS		Vivado	
Version	**Before Co-Sim**	**Co-Sim**	**Before Co-Sim**	**Co-Sim**
FF (106400)	703 (0.66%)	776 (0.73%)	543 (0.51%)	599 (0.56%)
LUT (53200)	2453 (4.61%)	2504 (4.71%)	802 (1.50%)	868 (1.63%)
BRAM_18K (280)	2 (0.71%)	3 (1.07%)	2 (0.71%)	3 (1.07%)
DSP48 (220)	4 (1.81%)	4 (1.81%)	0 (0%)	0 (0%)

2. Changing method call order (to represent parallelism, available in hardware);
3. To adhere to Vivado HLS coding style for loop boundaries (so they can be optimized); and
4. To introduce a Round Robin (RR) scheduler, similar to the way it is coded in Verilog (to represent parallelism available in hardware, which requires logical operations on wide integers to overcome usual software sequential semantics).

In addition, we had to modify our directives for minimizing logic of interface ports (from using an *ap_none* interface, to an *ap_hs* interface), so that Vivado HLS could co-simulate our final C++ code with the resulting HDL code with the same synthetic traffic testbench, allowing us to verify results.

After creating the IP using Vivado HLS tool, we imported it to Vivado for synthesis and obtained the hardware cost for a Zedboard Z7020 FPGA. Table I presents FPGA resource utilization, for an extended MemGuard design with 4 bandwidth-regulated sources, comparing Vivado HLS and Vivado, before and after modifying the interface optimizations on top-level ports, for co-simulation. DSP48 modules have been used to implement integer multiplications, for the EWMA calculations in Vivado HLS and have been replaced with LUT-based logic in Vivavo.

Table II shows hardware cost scalability with the number of regulated sources (power of two in our design), for the final Vivado Co-Sim version. Our bandwidth regulation module scales well and is always small, requiring less than 3531 FPGA LUTs (6.63% of the

TABLE II
HARDWARE COST SCALING (2, 4, 8, 16 SOURCES).

Sources (#)	**2**	**4**	**8**	**16**
FF (106400)	360 (0.33%)	599 (0.56%)	1213 (1.14%)	2250 (2.11%)
LUT (53200)	471 (0.88%)	868 (1.63%)	1622 (3.04%)	3531 (6.63%)
BRAM_18K (280)	2 (0.71%)	3 (1.07%)	2 (0.71%)	2 (0.71%)
F7 Muxes (26600)	0 (0%)	0 (0%)	17 (0.06%)	0 (0%)
F8 Muxes (13300)	0 (0%)	0 (0%)	95 (0.35%)	5 (0.03%)

TABLE III
LATENCY SCALING IN VIVADO HLS (2, 4, 8, 16 SOURCES).

Sources (#)	**2**	**4**	**8**	**16**
Latency (cycles)	1-9	1-12	1-20	1-32
Cycle-time (Syn/IP)	8.72ns/ 7.299ns	8.67ns/ 6.783ns	13.82ns/ 6.876ns	8.22ns/ 8.914ns

TABLE IV
EXTENDED MEMGUARD (4 SOURCES) COMPARED TO AXI BRIDGE AND STNOC INSTANCE.

Component	**LUTs**	**FFs**	**BRAMs**
Extended MemGuard	868	599	3
AXI Bridge	723	971	
STNoC (12-nodes, 4x4 routers)	24939	17983	

low-end XC7Z020 FPGA) for 16 regulated sources. Flip-flop count is small and also scales well with the number of sources. Table II shows that Vivado can exchange a number of LUTs for F7/F8 MUX primitives (about 6 LUTs each), in the larger designs.

Although we have exploited concurrency of the independent Make RC/BE Traffic blocks, we have not attempted to also pipeline our design. Nevertheless, timing results, shown in Table III, are encouraging. Our design requires 9 to 31 clock cycles, for different regulated source counts, with cycle times that Vivado HLS synthesis estimates to be worse than 10ns only for 8 sources. Even in that case, when extracting the RTL as intelectual property (IP) for Vivado, the estimate improves to meet our timing constraint, for a 100 MHz clock.

Finally, Table IV further shows the small cost of the Extended MemGuard hardware module, compared to AMBA AXI4, and an instance of STMicroelectronics STNoC on the same FPGA fabric.

V. SYNTHETIC TRAFFIC EVALUATION

To evaluate the relative performance of Genuine vs. Extended MemGuard, we run experiments with annotated SystemC models of our bandwidth regulating device, and different synthetic traffic configurations (uniformly random, variable bit rate (VBR) and bursty) arriving from two or more "hypothetical" cores. We consider both violation free (VF) and non violation free (non-VF) mode. Each core reserves $r_{min}/number\_of\_sources$, except for one, which has a zero reservation.

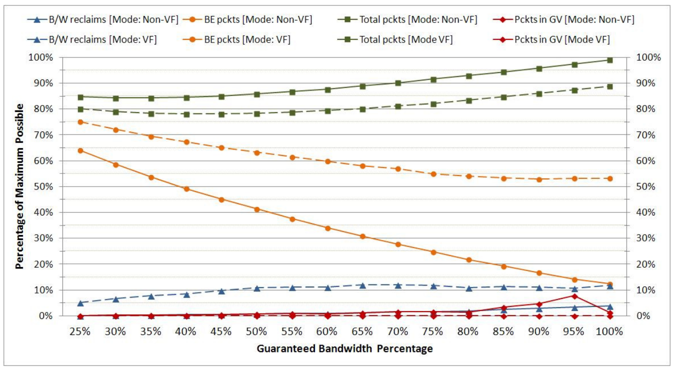

Fig. 5. Extended MemGuard with VBR traffic (4-cores).

For improved accuracy in our simulation experiments, we assume n = 10240 periods and repeat each experiment for m = 100 times. The regulation period duration is period = 20 units (i.e. up to 20 accesses can be forwarded to the memory controller during a period). Hence, the total number of simulated memory accesses by all cores is: $(number\_of\_sources \cdot period \cdot n \cdot m)$. All MemGuard performance metrics are normalized in a range from 0 to 100%, by taking into account:

- The maximum number of bandwidth reclaims: $period \cdot number\_of\_sources \cdot n$;
- The maximum total number of accesses for all traffic sources can be computed as: $period \cdot n$;
- Since a guarantee violation may also lead to best effort, the maximum number is: $(period - r_{min}) \cdot n$ for best effort and $(period - r_{min}) \cdot number\_of\_sources \cdot n$ for guarantee violations.

Figure 5 shows average performance from 100 simulations of a 4-core platform, using a VBR traffic scenario for the extended MemGuard. The total used bandwidth (U_i) reaches up to 99.02% (for Non-VF mode) and 88.87% (for VF mode) of the maximum possible, and bears an almost linear relationship to the increase in guaranteed bandwidth. In addition, our simulation experiments reveal that for NonVF mode, guarantee violations increase with the guaranteed bandwidth reaching maximum when r_{min} is equal to 95%, and are always at low rates, less than 10% of the maximum possible. For the VF mode, there are zero guarantee violations.

Similarly, for Genuine MemGuard (graph omitted due to space limitations) we observe that: (a) used bandwidth is almost similar in Non-VF mode, but much smaller on average in VF mode; (b) bandwidth reclaim requests are much higher (e.g. 35% vs. 12.03% in VF mode); (c) best effort packets are less by 10% in Non-VF and 20% or more in VF mode; and (d) the number of guarantee violations is similar.

Figure 6 considers bursty traffic and compares performance of Extended vs Genuine MemGuard. Results indicate that Extended MemGuard continues to improve compared to Genuine algorithm for a larger number of cores, as it manages a smaller number of bandwidth reclaims (15 to 35 times less), has higher best effort traffic rate (up to one order of magnitude), utilizes almost 100% of the provided bandwidth, and generates a comparable number of violations for non-VF mode; notice that for the sake of simplicity only minimum and maximum bounds are shown for the same r_{min} range as in Figure 5. Simulations based on random and VBR traffic have drawn similar conclusions.

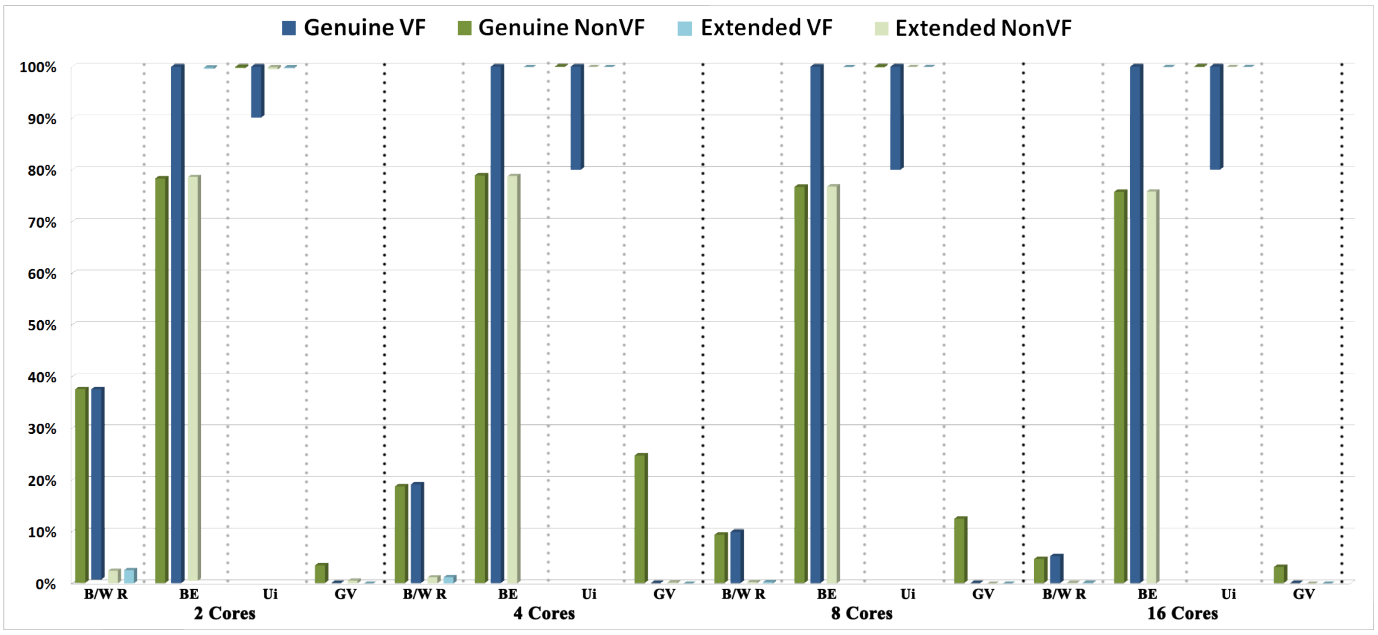

Fig. 6. Max/Min ranges for Bandwidth Reclaim (B/W R), Best Effort (BE), Used Bandwidth (Ui) and Guarantee Violations (GV) in VF and NonVF mode for Genuine (left pair of bars) and Extended (right pair of bars) MemGuard for 2, 4, 8 and 16 cores.

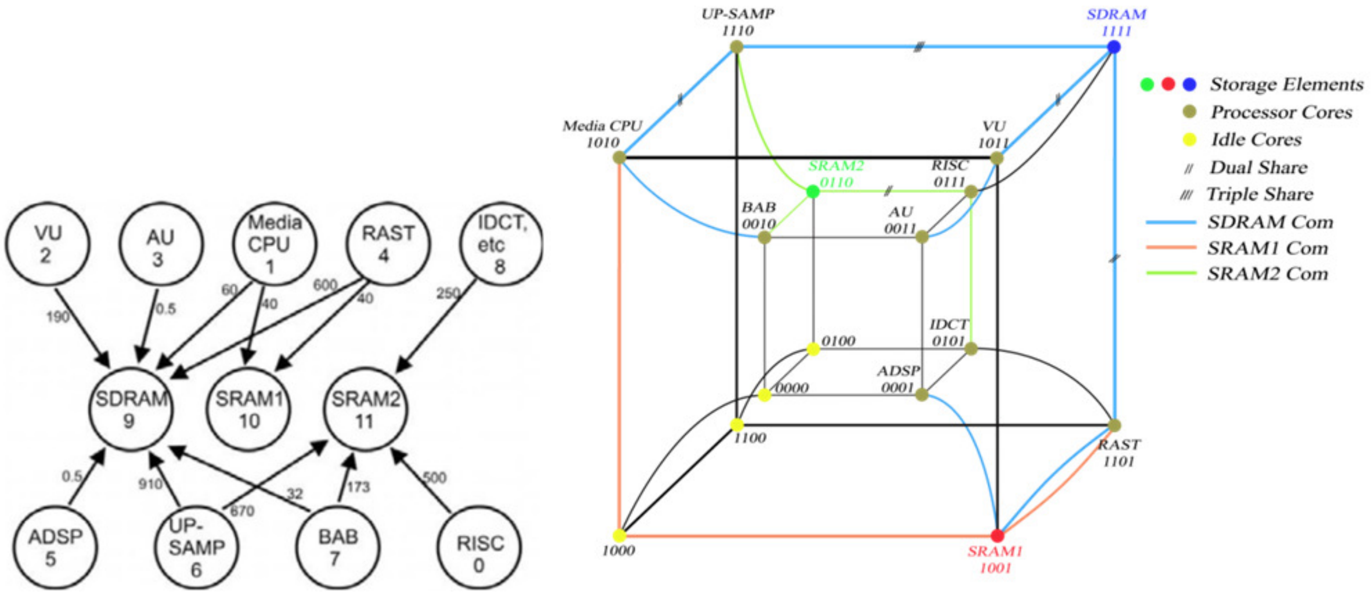

Fig. 7. Mapping the MPEG4 task graph (shown on the left) on the custom 16-node hypercube NoC, where a separate MemGuard instance is located at each memory component (SRAM1, SRAM2, SDRAM).

VI. NoC-based Evaluation

MemGuard was originally proposed for Linux, thus, arbitrating traffic consumption by different cores. However, the core part of MemGuard algorithm is generic, and can be *adapted for custom on-chip network traffic management*. In this case, RC and BE traffic can be distinguished by NoC virtual channels, or by a special tag in the packet, enabling finer granularity traffic classification, e.g., to applications.

This section provides experiments towards this generalization of the algorithm and evaluate relative performance of Genuine vs. Extended MemGuard at multiple target devices (memory controllers) in a cycle-approximate custom binary hypercube NoC topology, cf. open source HSoC [18]. This NoC topology provides a rich set of edge-disjoint paths which can be appropriately utilized for providing static partitioning of the IPs in our proposed application scenario, and, as we will discuss later, predictability of the traffic flows arriving at each memory controller.

As a traffic pattern, we assume an MPEG4 decoding speed transfer test [19]. In the MPEG4 graph, nine initiators (active cores) generate and transmit packets at highly different rates, e.g. compare 1580 MB/s bandwidth for UP-SAMP (core 6) to 0.5 MB/s for ADSP blocks (core 5), while three targets (SDRAM, SRAM1 and SRAM2) are passive storage elements which are only receiving data.

As shown in Figure 7, we have used a best map of the MPEG4 task graph resources (IPs) onto the 16-node NoC topology using offline partitioning via an efficient high-level partitioning tool called

TABLE V
GENERIC MEMGUARD INITIALIZATION FOR ALL THREE MEMORY CONTROLLERS (SRAM1, SRAM2, SDRAM).

Memory	**SRAM1**			**SRAM2**			**SDRAM**		
Initialize	**period=20**			**period=20**			**period=20**		
	Packets	**Relative Percentage**	**Initial Q_i**	**Packets**	**Relative Percentage**	**Initial Q_i**	**Packets**	**Relative Percentage**	**Initial Q_i**
Packet Distribution	40	50%	$Q_1 = 2$	250	15.69%	$Q_1 = 1$	600	33.46%	3
	40	50%	$Q_2 = 2$	500	31.39%	$Q_2 = 3$	60	3.35%	1
				173	10.86%	$Q_3 = 1$	32	1.78%	1
				670	42.06%	$Q_4 = 3$	910	50.75%	6
							0.5	0.03%	1
							190	10.6%	1
							0.5	0.03%	1
Total	80	100%	$\sum Q_i = 4$ **guaranteed**	1593	100%	$\sum Q_i = 8$ **guaranteed**	1793	100%	$\sum Q_i = 14$ **guaranteed**

Scotch [16]. Idle cores are marked in yellow, active cores in brown, and each of the three memory controllers in a different color (green, red, blue). Moreover, the color of a hypercube link identifies the destination of the packets that are transferred through that link, i.e. if the color of a link is blue (e.g. the connection between Media CPU and UP-SAMP), then all packets travelling through this link are sent to the SDRAM memory device (i.e. the one with blue color). Moreover, some links are marked with a Dual (or Triple) Share sign. This symbol identifies that the link is actually shared by packets originating from two (resp. three) different initiator cores and destined to the same memory controller.

In this optimized mapping based on the underlying e-cube shortest-path routing algorithm each memory controller receives packets through edge-disjoint paths. More specifically, paths directed to any memory controller are distinct from the paths to any other memory controller. This aspect provides static partitioning at NoC-level and allows for two important properties:

- Non-interaction of the traffic flows directed to different memory controllers.
- Consequently, based on the underlying MPEG4 traffic model, predictability of the traffic flow arriving at each memory controller. In this case, certain links are actually shared by packets originating from different initiator cores which are destined to the same memory controller (cf. dual or triple shares in Figure 7). However, this buffering issue does not actually affect MemGuard performance.

Due to static partitioning which provides edge-disjoint paths to each memory controller, the MemGuard instance on each memory controller can be configured independently based on the number of sources and guaranteed bandwidth.

More specifically, assuming an EWMA parameter $\lambda = 0.2$ (to emphasize history) and a period of 20 quanta (time slots) for each of the memory controllers in MPEG4, we use reservations of

- 4 time quanta for SRAM1 IP (receives from 2 sources);
- 8 quanta for SRAM2 IP (receives from 4 sources); and
- 14 quanta for SDRAM IP (receives from 7 sources).

Notice that the number of sources corresponds to the number of make_RC_traffic blocks in each memory controller. Initial reservation of the quanta to different traffic flows at each memory controller are based on the MPEG4 rates as shown in Table V. Thus, the initial reservations for the traffic sources of SDRAM IP are (3, 1, 1, 6, 1, 1, 1), with 6 corresponding to UP-SAMP and 3 to RAST (the two maximum rates). Notice that, since EWMA is in place, algorithm

TABLE VI
PERFORMANCE AT DIFFERENT MEMORY CONTROLLERS (MPEG4).

NonVF (Genuine, Extended)				
Memory	**B/W Reclaim**	**BE**	**GV**	**Delay (MG periods)**
SRAM1	(136, 62)	(21, 69)	(2, 4)	(300, 205)
SRAM2	(3661, 1167)	(1082, 2309)	(192, 153)	
SDRAM	(2691, 2490)	(210, 2444)	(66, 56)	
Total	(6488, 3719)	(1313, 4822)	(260, 213)	
VF (Genuine, Extended)				
Memory	**B/W Reclaim**	**BE**	**GV**	**Delay (MG periods)**
SRAM1	(441, 55)	(15, 31)	(0, 0)	(1698, 1675)
SRAM2	(22302, 2685)	(16, 28)	(0, 0)	
SDRAM	(14410, 3016)	(0, 8)	(0, 0)	
Total	(37153, 5756)	(31, 67)	(0, 0)	

bandwidth reservations quickly adapt to the actual rates, especially since a small period (20 quanta) is used for all cases.

Our simulation is based on MPEG4 traffic with more than 60K packets; there is very small difference (1-2%) from large sets of experiments with up to 998K packets. Results shown in Table VI indicate that when using MPEG4 task graph to generate traffic, extended MemGuard generates a smaller number of bandwidth reclaims (socalled interrupts) and more best effort traffic (in both NonVF and VF mode) resulting in an improved bandwidth rate (routing all packets with a smaller delay), while also generating less guarantee violations in NonVF mode than the Genuine version. Finally, we note that the actual value of λ (set to 0.2 in all experiments) does not affect performance since MemGuard is able to adjust very quickly to MPEG4 traffic rates.

We have also performed experiments involving distributed database transactions based on parallel equi-join operations. Equi-joins are implemented using parallel hashing of keys which results to an all-to-all NoC communication pattern, whereas the length of each vector depends on the distribution of the keys and the hash function. In our model, 8 initiators make parallel accesses to 8 different target memories via the hypercube NoC to access 2x5K records, resulting in a total of 80.000 RC packets being transmitted to all memory controller (with ideally 10.000 packets per memory controller). In this scenario, we use a period of 20 time quanta, for all MemGuard configurations and for guaranteed bandwidth all possible values correspond to a minimum 8 to the maximum of 20 quanta. Our simulation results (graphs omitted due to space limitations) show similar behavior characteristics as MPEG4, with Extended MemGuard producing more Best Effort traffic and less Bandwidth Reclaims in NonVF mode than the Genuine version, while the number of guarantee violations is larger. Moreover, both Genuine and Extended MemGuard take almost the same simulation time to successfully schedule all packets.

VII. Conclusions

We have proposed a new efficient, low-cost hardware component (called extended MemGuard) for bandwidth regulation at target devices. Compared to the original (genuine) MemGuard, our extended MemGuard component differentiates between rate-constrained and best effort messages, supports a violation free operating mode for rate-constrained flows, and provides dynamic adaptivity through EWMA prediction. Considering both NoC-independent and NoC-dependent traffic scenarios, we have shown that our extended hardware MemGuard compares favorably with an equivalent implementation of genuine MemGuard.

In future work, we are interested in examining the precise interplay between the proposed mechanisms and existing QoS techniques at the network interface and router layer in NoC-based multicore SoC. Moreover, we plan to experiment with alternative hardware MemGuard implementations at network interface, router, and as a Linux kernel scheduler extension for managing mixed criticality traffic by different cores, processes or VMs.

References

[1] R. Obermaisser, C. E. Salloum, B. Huber, and H. Kopetz, "The time-triggered system-on-a-chip architecture," in *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, June 2008, pp. 1941–1947.

[2] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (mpsoc) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, Oct 2008.

[3] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 208–222. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.24

[4] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Des. Test*, vol. 22, no. 5, pp. 414–421, Sep. 2005. [Online]. Available: http://dx.doi.org/10.1109/MDT.2005.99

[5] K. Goossens and A. Hansson, "The æthereal network on chip after ten years: Goals, evolution, lessons, and future," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 306–311. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837353

[6] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, Dec 2009.

[7] Y. Li, B. Akesson, and K. Goossens, "Dynamic command scheduling for real-time memory controllers," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014, pp. 3–14.

[8] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[9] M. Caccamo, G. C. Buttazzo, and D. C. Thomas, "Efficient reclaiming in reservation-based real-time systems with variable execution times," *IEEE Transactions on Computers*, vol. 54, no. 2, pp. 198–213, Feb 2005.

[10] E. Rijpkema, K. G. W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 350–355.

[11] T. Reichl, J. Passenger, O. Acosta, and O. Salvado, "Ultrasound goes gpu: real-time simulation using cuda," pp. 726 116–726 116–10, 2009. [Online]. Available: http://dx.doi.org/10.1117/12.812486

[12] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, April 2013, pp. 55–64.

[13] A. Radulescu, J. Dielissen, S. G. Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens, "An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 4–17, Jan 2005.

[14] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Readings in multimedia computing and networking," K. Jeffay and H. Zhang, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, ch. Resource Kernels: A Resource-centric Approach to Real-time and Multimedia Systems, pp. 476–490. [Online]. Available: http://dl.acm.org/citation.cfm?id=383802.383915

[15] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in multi-core platforms," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 562–576, 2016.

[16] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, ser. HPCN Europe 1996. London, UK, UK: Springer-Verlag, 1996, pp. 493–498. [Online]. Available: http://dl.acm.org/citation.cfm?id=645560.658570

[17] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 133–143. [Online]. Available: http://doi.acm.org/10.1145/264107.264156

[18] "Hsoc systemc virtual platform," 2013. [Online]. Available: https://sourceforge.net/projects/hsoc/

[19] E. B. van der Tol and E. G. Jaspers, "Mapping of mpeg-4 decoding on a flexible architecture platform," pp. 1–13, 2001. [Online]. Available: http://dx.doi.org/10.1117/12.451067