BlueIO: A Scalable Real-Time Hardware I/O Virtualization System for Many-core Embedded Systems

ZHE JIANG, University of York, UK
NEIL AUDSLEY, University of York, UK

In safety-critical systems, time predictability is vital. This extends to I/O operations which require predictability, timing-accuracy, parallel access, scalability, and isolation. Currently, existing approaches cannot achieve all these requirements at the same time. In this paper, we propose a framework of hardware framework for real-time I/O virtualization termed *BlueIO* to meet all these requirements simultaneously.

BlueIO integrates the functionalities of I/O virtualization, low layer I/O drivers and a clock cycle level timing-accurate I/O controller (using the GPIOCP [36]). BlueIO provides this functionality in the hardware layer, supporting abstract virtualized access to I/O from the software domain. The hardware implementation includes I/O virtualization and I/O drivers, provides isolation and parallel (concurrent) access to I/O operations and improves I/O performance. Furthermore, the approach includes the previously proposed GPIOCP to guarantee that I/O operations will occur at a specific clock cycle (i.e. be timing-accurate and predictable).

In this paper, we present a hardware consumption analysis of BlueIO, in order to show that it linearly scales with the number of CPUs and I/O devices, which is evidenced by our implementation in VLSI and FPGA. We also describe the design and implementation of BlueIO, and demonstrate how a BlueIO-based system can be exploited to meet real-time requirements with significant improvements in I/O performance and a low running cost on different OSs.

CCS Concepts: • **Computer systems organization → Embedded hardware**; **Real-time system architecture**; • **Hardware → Reconfigurable logic and FPGAs**;

Additional Key Words and Phrases: Safety-critical System, Real-time System, Predictability, Timing-accuracy, Scalability, Virtualization.

ACM Reference Format:
Zhe Jiang and Neil Audsley. 2019. BlueIO: A Scalable Real-Time Hardware I/O Virtualization System for Many-core Embedded Systems. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2019), 24 pages.
https://doi.org/10.1145/3309765

1 INTRODUCTION

In safety-critical systems, meeting real-time requirements is key. For example, to assure a timely reaction when critical situations occur (e.g. the braking operation of a car always has to be handled within a hard deadline [50]), and to guarantee accurate control I/O devices (e.g. an automotive engine requires I/O timing accuracy to inject fuel at the optimal time [44]). This leads to the following requirements:

(1) Predictability [24] – I/O operations can be always handled within a fixed time duration.
(2) Timing-accuracy [36] – I/O operations occur on an exact clock cycle.

Authors' addresses: Zhe Jiang, University of York, Deramore Lane, York, UK, YO10 5GH, UK; Neil Audsley, University of York, Deramore Lane, York, UK, YO10 5GH, UK.

1539-9087/2019/1-ART1 $15.00
https://doi.org/10.1145/3309765

(3) Parallelism – many I/O operations can occur in parallel to accurately control a different number of I/O devices simultaneously, e.g. the multiple engines on UAVs may require accurate I/O simultaneously in order to achieve stability in flight [39].
(4) Isolation – I/O operations require isolation to prevent interference from other parts of the system that may disrupt timing [35].

Currently, existing approaches do not meet all these requirements, e.g. [9, 22, 23, 34]. In this paper, we propose the *BlueIO* system to address these requirements simultaneously. BlueIO is a scalable hardware-implemented real-time I/O virtualization system, which integrates I/O virtualization and a ready-built timing-accurate I/O controller (GPIOCP [36]). We note that virtualization enables isolation and parallel access for I/O operations [43]. Also, the GPIOCP guarantees the predictability and timing-accuracy for I/O operations [36]. Additionally, the hardware consumption of BlueIO scales linearly as the number of CPUs and I/Os increases, resulting from its modularized design.

1.1 I/O Virtualization

In safety-critical systems, the use of virtualization has been proposed to support isolation of executing software in terms of both CPU and I/O [43] [29] [33] [51]. Specifically, in a virtualized system, the Virtual Machines (VMs) are independent and logically isolated, which means the I/O operations requested from different VMs can never interfere with each other [21, 49**?**]. Meanwhile, these I/O operations are also prevented from being affected by the other VMs [**?**]. I/O virtualization also enables increased resource utilization, reduced volume and cost of hardware [21, 49].

However, I/O virtualization involves complex I/O access paths (i.e. indirection and interposition of privileged instructions) and complicated shared I/O resource management (i.e. scheduling and prioritization) [37, 49], resulting in decreased I/O performance, increased software overhead, and poor scalability etc [37, 49]. That is, predictability and timing accuracy are difficult to achieve with I/O virtualization.

I/O virtualization relies on hardware support, and today's chip manufacturers have included different hardware features in order to mitigate the penalties suffered by traditional I/O virtualization [21, 46, 49]. Intel's Virtualization Technology for Directed I/O (*VT-D*) [34] provides direct I/O access from guest VMs. The IOMMU [22] is used in commercial PCI-based systems to offload memory protection and address translation, in order to provide a fast I/O access from guest VMs. These commonly used hardware-based I/O virtualization approaches simplify the I/O access, but do not support predictable timing-accurate I/O for real-time systems [21, 25, 36, 49].

1.2 Real-time Properties of I/O Operations

Latencies caused by device drivers and application process scheduling make predictable and timing-accurate I/O operations difficult to achieve. One solution is a dedicated CPU for I/O, which has limited scalability. Alternatively, application software handling I/O can be executed at the highest priority by the interrupt handler of a high-resolution timer (e.g. the nanosecond timer provided by an RTOS [17, 18]), although handling multiple parallel I/O operations (for different devices) with sufficient timing accuracy is not easy using this approach. Also the transmission latencies from a CPU to an I/O controller can be substantial and variable due to the communication bottlenecks and contention. For example, in a bus-based many-core system, the arbitration of the bus and the I/O controller may delay the I/O request. For a Network-on-Chip (NoC) architecture, the arbitration of on-chip data flows across the communications mesh will also increase latencies.

With I/O virtualization, these issues are magnified even further, due to complex I/O access paths (i.e. indirection and interposition) [21]. Specifically, if an application invokes an I/O request from a guest Virtual Machine(**VM**), this I/O request will be transmitted through front-end drivers (in guest

OS), back-end drivers (in Virtual Machine Monitor (**VMM**)), and host OS (See Figure 1). Hence, it is difficult for an application from a guest VM to achieve predictable and timing-accurate I/O.

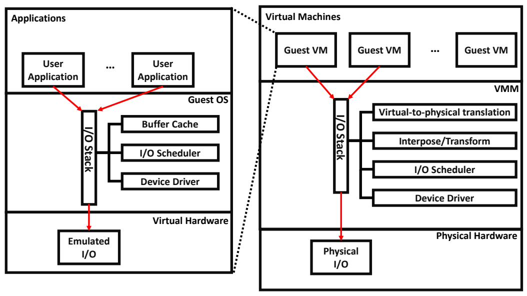


Fig. 1. Flow of I/O Request in Traditional Virtualization System

1.3 Contributions

The contributions of the paper are:

- A scalable hardware-implemented real-time I/O virtualization system, termed BlueIO with the following features:
  (1) BlueIO enables I/O virtualization, so that I/O operations requested from different VMs are isolated (requirement 4), and able to access different I/O devices simultaneously (requirement 3).
  (2) BlueIO integrates the real-time timing-accurate I/O controller GPIOCP, to enable predictable and timing-accurate I/O operations (requirement 1 & 2), whilst maintaining isolation and parallel accesses.
  (3) BlueIO integrates I/O drivers, and provides abstracted high-layer access interfaces to software (Guest VMs), which simplify the I/O access paths and improve I/O performance.
- Experimental results to demonstrate how BlueIO-based virtualization predictable and timing-accurate I/O (requirement 1 and 2), with decreased software overhead, improved I/O performance.
- A hardware consumption analysis of BlueIO, in order to show that hardware costs linearly scale as the number of CPUs and I/O devices increases.

1.4 Organization

This paper is organized as follows. Section 2 describes the system model. Section 3 proposes the design of the BlueIO real-time I/O virtualization system. Section 4 discusses the specific implementation details of the BlueIO, followed by its hardware consumption analysis in Section 5. Section 6 presents the performance evaluation of a BlueIO-based system. Section 7 discusses related work, with conclusions and future work offered in Section 8.

2 SAFETY-CRITICAL REAL-TIME I/O SYSTEM

In this paper, we assume that timing predictability, timing-accuracy, parallel accesses, isolation, and scalability are simultaneously required by applications. We assume that applications are implemented on a multi-core system, specifically an embedded NoC (although BlueIO is architecture agnostic), with a single synchronized timng source to enable the cycle accuracy of multiple I/O devices in parallel.

In standard computer and embedded architectures, an I/O system can be evaluated via multiple metrics, e.g. memory footprint, I/O throughput, etc [36, 37]. Additionally, the safety-critical real-time I/O systems require predictability, timing-accuracy, parallel accesses, isolation and timing scalability. Among these characteristics, predictability and parallel accesses have been addressed, e.g. in [36] and [37] (see Section 6). This paper proposes, within the BlueIO system, support for timing predictability and isolation for I/O (see Section 3).

In this section, we describe the timing-accuracy and scalability models respectively, in order to assess the timing-accuracy and scalability of an I/O system in later sections.

2.1 Timing-Accuracy Model

The error in the timing-accuracy of I/O operations is defined as the absolute time difference between the time at which an I/O operation is required (T_r) and the actual time that the I/O operation (e.g. read) occurs (T_a):

$$E = |T_r - T_a| \tag{1}$$

Thus a smaller E implies a higher timing-accuracy of the I/O operation. If E equals 0, the I/O operation occured at the expected time – i.e. totally timing-accurate. In practice, if E is less than one cycle period, then the I/O operation occurred at the required clock cycle.

The timing-accuracy of existing single-core, multi-core (Bus-based) and many-core (NoC) architectures can be assessed by constructing a system on FPGA and measuring the effect of the latencies between application and I/O device on the timing-accuracy of the I/O. Errors found in 1000 test runs for four systems are given in Table 1 (further experiment design is described in the technical report [5]). It is clear that even in a single-core system, E is not close to a single cycle, with the timing error in multi-core and many-core systems considerably worse due to the communication bottlenecks and contention of the system. With a VMM added, this issue is magnified even further. Note that the experiment merely measures hardware latencies (across buses/NoC meshes) of I/O instructions issued by the application CPU – clearly software effects (control/data flow within

Table 1. The Errors in Timing-accuracy of I/O Operations In Two Typical Systems (unit: ns)

CPU Index	E			
	Minimum	Median	Mean	Maximum
Single-Core Architecture				
	2090.0	2090.0	2012.5	2100.00
Bus-based Multi-Core Architecture (2 CPUs)				
Core 0	2440.0	2480.0	2477.2	2500.0
Core 1	2446.0	2450.0	2470.0	2490.0
NoC-based Many-core Architecture (9 CPUs)				
(0,0)	3140.0	3140.0	3145.8	3160.0
(0,1)	3000.0	3000.0	3005.8	3020.0
(0,2)	2790.0	2790.0	2795.8	2810.0
(1,0)	2720.0	2720.0	2725.8	2740.0
(1,1)	3070.0	3070.0	3075.8	3090.0
(1,2)	2860.0	2880.0	2899.4	2940.0
(2,0)	2580.0	2580.0	2585.8	2600.0
(2,1)	2650.0	2650.0	2655.8	2670.0
(2,2)	2860.0	2930.0	2902.2	2950.0
NoC-based Many-core Architecture (9 CPUs) with VMM (Round-Robin Scheduling Policy)				
(0,0)	4220.0	4220.0	4045.6	4260.0
(0,1)	4000.0	4000.0	4010.2	4080.0
(0,2)	3800.0	3800.0	3890.8	3920.0
(1,0)	3780.0	3780.0	3802.2	3840.0
(1,1)	4070.0	4070.0	4078.8	4100.0
(1,2)	3860.0	3880.0	3920.0	4000.0
(2,0)	3620.0	3620.0	3670.8	3760.0
(2,1)	3710.0	3710.0	3715.2	3770.0
(2,2)	3860.0	3930.0	3940.2	3980.0

code), scheduling (amongst competing software tasks), the Real-Time OS system calls and the implementation of I/O virtualization would add considerably to the overall latencies in Table 1.

2.2 Timing Scalability Model

The scalability of an I/O system in terms of its timing can be considered by evaluating the average response time of an I/O device ($\overline{R}$) in a many-core system with different numbers of CPUs, whilst CPU and I/O are fully loaded. Ideally, the average I/O response time ($\overline{R_N}$) in a n-core system should be n times the average I/O response time as in a single-core system ($\overline{R_1}$). Such a system would be *timing scalable*. The difference between the actual and ideal average I/O response time in a n-core system is termed the *performance loss* of the I/O system, defined as ΔR:

$$\Delta R = \overline{R_N} - n * \overline{R_1} \tag{2}$$

The average I/O performance loss suffers to each CPU is calculated as Δr:

$$\Delta r = \frac{\overline{R_N} - n * \overline{R_1}}{n} \tag{3}$$

In a many-core system, if $\Delta r = 0$, it means no loss of I/O performance occurred compared to a single-core system. Conversely, a larger Δr implies the reduction of I/O performance, and reduced timing scalability of the evaluated I/O system.

The timing scalability of an I/O system can be evaluated in existing single-core, multi-core (Bus-based) and many-core (NoC-based) architectures with different numbers of cores. The average I/O response time of reading one byte data from an SPI NOR-flash and corresponding Δr in different architectures with software implemented VMM can be found in Table 2 (further experiment design is described in [5]). It is clear that in traditional I/O virtualized many-core systems, with the number of CPUs increased, Δr is increased drastically, which implies a significant reduction of I/O performance and limited scalability of the I/O system.

Table 2. Scalability Model in Different Virtualized Many-core Systems (unit: clock cycle)

	Software VMM (Scheduling Policy: RR)		Software VMM (Scheduling Policy: FIFO)	
CPU Index	$\overline{R}$	Δr	$\overline{R}$	Δr
NoC-based Many-core Architecture (1 CPU)				
Single-core Architecture	513	0	408	0
NoC-based Many-core Architecture (4 CPUs)				
(0,0)	9015	1750	2916	284
(0,1)	8995		2875	
(1,0)	9213		2638	
(1,1)	8985		2645	
NoC-based Many-core Architecture (9 CPUs)				
(0,0)	36060	3535.8	9357	496.5
(0,1)	35860		8915	
(0,2)	36049		8415	
(1,0)	36237		8203	
(1,1)	36410		9748	
(1,2)	36576		7476	
(2,0)	36741		7467	
(2,1)	36930		7576	
(2,2)	37102		6121	

3 SAFETY-CRITICAL REAL-TIME I/O SYSTEM DESIGN

BlueIO is an integration of I/O virtualization, low layer I/O drivers and clock cycle level timing-accurate I/O control (the latter is built using GPIOCP [36]) – all within the hardware layer, meanwhile providing abstracted high-layer access to software layers (Guest VMs).

I/O virtualization provides isolation and parallel access to I/O operations. The hardware implementation of I/O virtualization offloads most of the virtualization overhead into hardware, with guest OSs executing in ring 0 with full privilege. Therefore, indirection and interposition of I/O requests are not required. The hardware implemented low layer I/O drivers reduce I/O access paths and improve the I/O performance significantly. The deployment of the GPIOCP guarantees that I/O operations will occur at a specific clock cycle (i.e. are timing-accurate and predictable).

3.1 General Architecture

Figure 2 depicts the proposed general embedded virtualization architecture. The RTOS kernel in each VM can be executed in kernel mode (ring 0) to achieve full functionality. Also, the architecture provides a environment suitable for the execution of real-time applications with deadlines. Finally, the I/O system, running in hardware, is responsible for I/O virtualization, physical isolation between VMs, and providing high layer access interfaces for user applications (in Guest VMs).

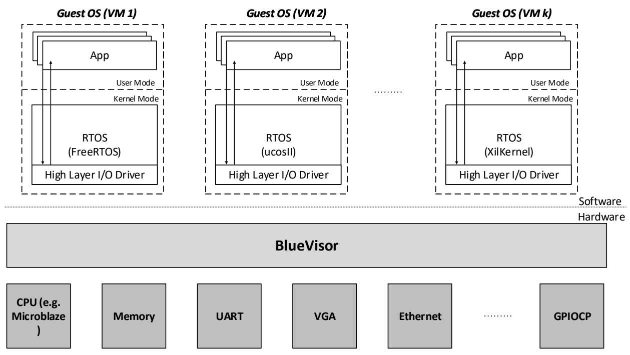


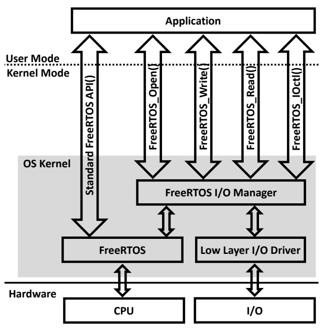
Fig. 2. Embedded Virtualization Architecture

3.2 Virtual Machine (VM) and Guest OS

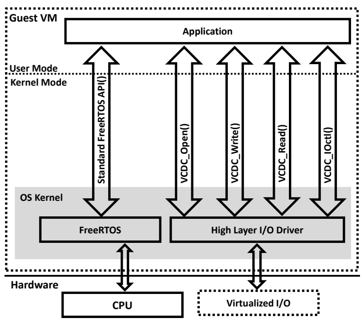
In our proposed approach, each CPU has an individual guest VM. The virtualization in the system has the following features:

- Bare-metal virtualization [49] – a guest OS can be executed on a CPU directly, without host OS. Therefore, a guest OS is able to execute in kernel mode to achieve full functionality;
- Para-virtualization [41] – an I/O management module in each guest OS has to be replaced by high level I/O drivers, which enables smaller OS software footprint and simplified I/O access paths.

For the proposed design, three OS kernels have been modified to support I/O virtualization [20]: FreeRTOS [8], uCosII [3] and Xilkernel [4]. In Figure 3, we use FreeRTOS as an example to demonstrate this modification. Compared with the original FreeROTS kernel (Figure 3(a)), user applications running on a modified kernel (Figure 3(b)) are able to access I/O via high layer I/O drivers, which are independent from FreeRTOS kernel. User applications designed for the original OS kernel can



(a) Traditional FreeRTOS Kernel



(b) Modified FreeRTOS Kernel

Fig. 3. Traditional and Modified FreeRTOS Kernels

be ported to the modified kernel directly (without any modification), since we have not modified the OS interfaces (OS APIs).

The architecture builds upon three existing technologies, the Virtualized Complicated Device Controller (VCDC) [37], GPIOCP [36] and BlueTree [30–32] which are introduced in Section 4.1. The full implementation of the BlueIO architecture is described in Section 4.

4 BLUEIO IMPLEMENTATION

BlueIO is included in the BlueTiles 2D mesh type open source NoC [47]. The use of a NoC is not required by BlueIO, because it is a general-purpose I/O system, which is agnostic to the type of bus and the software running on CPUs. To support a complete BlueIO system, the platform requires:

- Communication channels between BlueIO and CPUs;
- A global synchronization timer;
- A memory access interface – in the proposed design, BlueTree [31] is adopted as the memory access interface (see Section 4.5).

The use of BlueIO within BlueTiles is shown in Figure 4. BlueIO is physically connected to the home port (via the physical link) of a router, the global timer T, and the memory access interface (BlueTree).

4.1 Structure of BlueIO

BlueIO contains four main modules (see Figure 5):

- BlueGrass — is a communication interface between application CPUs, VCDC [36], I/O controllers and external memories (DDR);
- Virtualized Complicated Device Controller (VCDC) [36] — integrates functionalities of I/O virtualization and low layer I/O drivers. Note that the VCDC is the component mainly handling the scheduling policy of I/O requests in BlueIO (for more details, see Section 4.3 and [37]).
- GPIO Command Processor (GPIOCP) [36] — is a programmable real-time I/O controller, that permits applications to instigate complex sequences of I/O operations at an exact single clock cycle;
- BlueTree [30–32] — provides an interface to access memory and DMA.

These four modules are now introduced in the following sections.

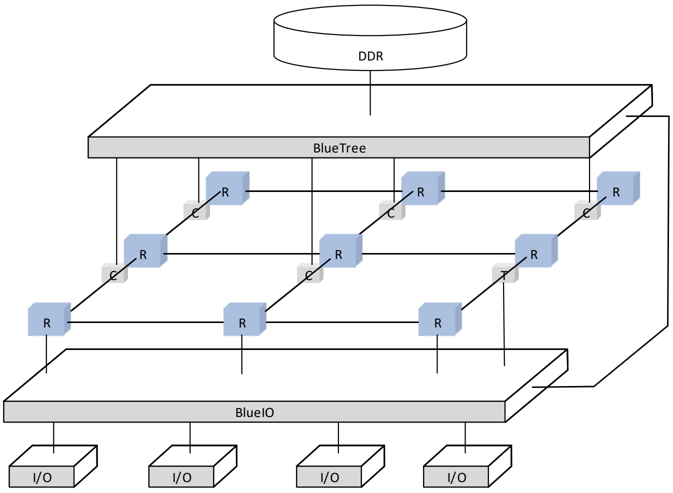


Fig. 4. Platform Overview
C - Core; R - Router / Arbiter; T - Global Timer

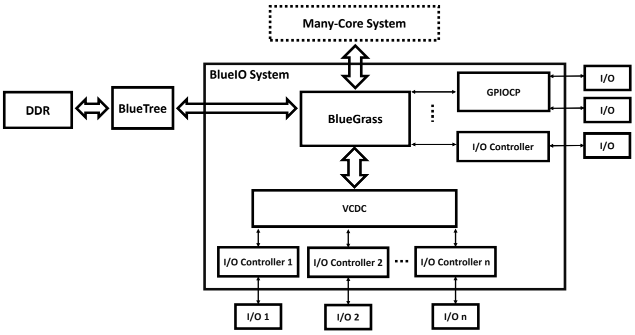


Fig. 5. The Structure of the BlueIO

4.2 BlueGrass

BlueGrass is the communication interface between application CPUs and BlueIO, and includes four communication interfaces:

- Interface from/to application CPUs;
- Interface from/to I/O controllers;
- Interface from/to VCDC;
- Interface from/to the external memory.

BlueGrass is physically connected to the NoC mesh (BlueTiles) and the memory access interface (BlueTree). Additionally, I/O controllers can be directly connected to the Bluegrass to maintain original functionality, or indirectly connected to the VCDC to acquire I/O virtualization.

The structure of BlueGrass (see Figure 6) contains two parts: downward path and upward path. The downward path is responsible for sending either I/O control commands or transferring data to

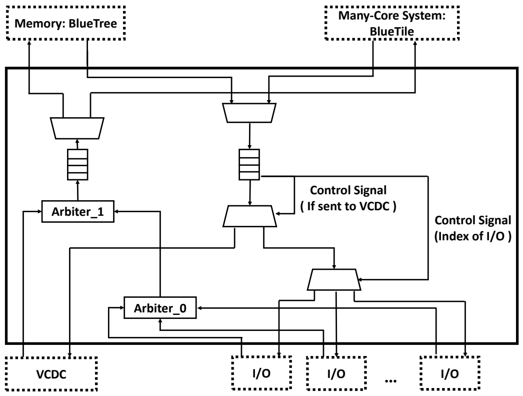

Memory: BlueTree
Many-Core System: BlueTile
Control Signal (If sent to VCDC)
Control Signal (Index of I/O)
Arbiter_1
Arbiter_0
VCDC
I/O
I/O
...
I/O

Fig. 6. The Structure of the BlueGrass

I/O devices. The upward path is responsible for sending I/O responses back to application CPUs, and data from I/O to CPUs or external memories.

Implementationally, the downward path consists of three half-duplex multiplexers and a FIFO. The 2-into-1 multiplexer connected to BlueTile[47] and BlueTree[30] is designed to receive, and then queue the I/O requests and data fetched from memory to the downward FIFO. The downward FIFO allocates these queued I/O requests and data to a specified I/O according to the format of packets (see [20]). The upward path consists of two arbiters, one half-duplex multiplexer and one FIFO. The arbiters determine the served sequence of I/O response and memory requests sent from each I/O. In order to prevent one single I/O dominating the upward path, and to be able to satisfy the requirement that the I/O system can be time-predictable, we have provided multiple real-time scheduling policies to both arbiters, including Round-Robin, fixed priority, and FIFO. In addition, users are also allowed to add a custom scheduling policy to the arbiters via our provided interface (see [20]). The upward FIFO and the connected 1-into-2 multiplexer are responsible for sending I/O responses and memory requests out of the BlueIO system.

4.3 Virtualized Complicated Device Controller (VCDC)

In [37], the Virtualized Complicated Device Controller (VCDC) was proposed, to implement I/O virtualization and I/O drivers into hardware. The VCDC can be physically connected to a many-core system, which is composed by two main parts (see Figure 7):

- I/O VMM – maintains the virtualization of I/O devices.
- Low Layer I/O Drivers – encapsulates specific I/O drivers for a specific I/O controller (eg. read the data from a specific address of the SPI NOR-flash).

The I/O VMM has two main responsibilities:

(1) Interpreting I/O requests (sent from a guest OS) to the actually I/O instructions (used to control a physical I/O);
(2) Scheduling and allocating the interpreted I/O instructions to physical I/O.

Considering that the functionalities and features of I/O devices are different, it is challenging to build a general purpose module to achieve virtualization for all kinds of I/O devices. Therefore, we create some specific-purpose I/O VMM for those commonly used I/O devices, including UART, VGA, DMA, Ethernet, etc. Additionally, users can also easily add their custom I/O VMM into VCDC via our provided interfaces (see technical report [20]).

Further details of VCDC design and implementation can be found in [20, 37].

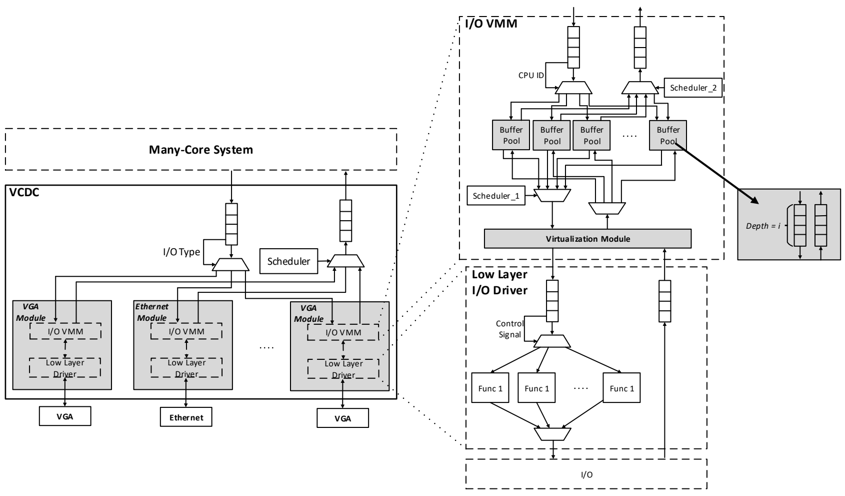


Fig. 7. Structure of VCDC

4.4 GPIO Command Processor (GPIOCP)

In [36], the GPIO Command Processor (**GPIOCP**) was proposed. It is a resource efficient programmable I/O controller that permits applications to instigate complex sequences of I/O operations (off-line) at an exact time, so providing timing-accuracy of a single clock cycle. This is achieved by loading application specific programs into the GPIOCP which then generates a sequence of control signals over a set of General Purpose I/O (GPIO) pins, eg. read/write. Applications then are able to invoke a specific program at run-time by sending the GPIO command, e.g. *run command X at time t (in the future)*.

GPIOCP achieves cycle level timing-accuracy as the latencies of I/O virtualization and communication bus are eliminated. For example, a periodic read of a sensor value by an application can be achieved by loading the GPIOCP with an appropriate program, then at run-time the GPIOCP issues a command such as *run command X at time t and repeat with period Z* – the values are read at exact times, with the latency of moving the data back to the application considered within that application's worst-case execution time.

The GPIOCP can be physically connected to a many-core system or VCDC, which is composed of four main components (see Figure 8):

- Hardware manager – Communicates with application CPUs, allocating incoming messages to either the command memory controller (to store new commands) or the command queue (to initiate an existing command).
- Command memory controller – Stores a new GPIO command into the storage units; and accesses an existing GPIO command for execution by a GPIO CPU (within the command queue).
- Command queue – Allocates GPIO commands to GPIO CPUs for execution (cooperate with command memory controller). Each GPIO CPU is a simple finite state machine, with guaranteed execution time so achieving timing-accuracy.

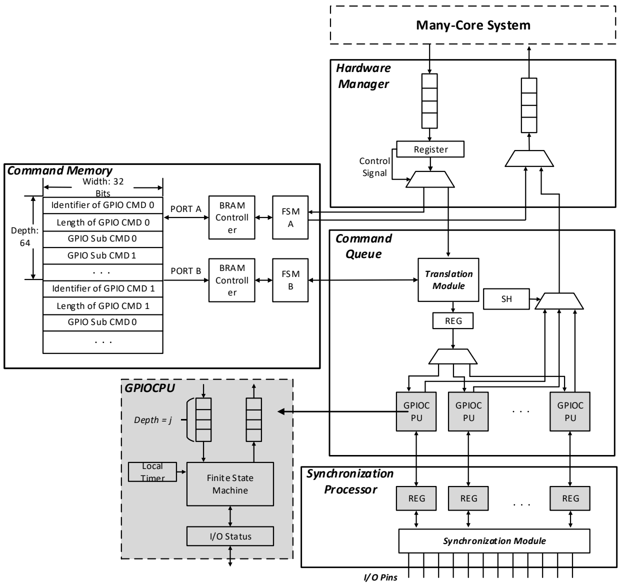


Fig. 8. Structure of GPIOCP

- Synchronization processor – Synchronises the values of I/O pins, which may be written by different GPIO CPUs and I/O devices.

Further details can be seen in [36] and [5].

4.5 BlueTree

BlueTree is a tree-like memory interconnect built for many-core systems, which enables time-predictable memory read/write from a scaled number of CPUs and I/Os [31] [30]. The BlueTree memory interconnect is designed to support the memory requirements of modern systems, leaving the NoC communications mesh for core-to-core communication only. BlueTree distributes memory arbitration across a set of 2-into-1 full-duplex multiplexers, each with a small arbiter (see Figure 9), rather than using a large monolithic arbiter next to memory. This allows the BlueTree to be scalable and enable a larger number of requesters at a higher clock frequency than would be available with a single monolithic arbiter.

In order to prevent a single core dominating the tree, and to be able to satisfy the requirement that the memory subsystem is time-predictable, each multiplexer contains a blocking counter, which encodes the number of times that a high-priority packet (i.e., a packet from the left) has blocked a low-priority packet (i.e., a packet from the right). When this counter becomes equal to a fixed value m, the counter is reset and a single low-priority packet is given service. This ensures that there is an upper bound for the WCET of a memory transaction. Note that the memory accesses in a BlueIO-based system may affect the real-time properties of the whole system. This paper focusses on the timing of I/O – specific timing analysis of BlueTree is given in [30, 31, 50].

5 HARDWARE CONSUMPTION ANALYSIS

In this section, the hardware consumption of BlueIO is analyzed regarding its scalability. Firstly, an analysis is given to describe the hardware consumption of BlueIO; secondly, actual hardware consumption of BlueIO in VLSI (logic gates) and FPGA (LUTs, registers, and BRAMs) is given.

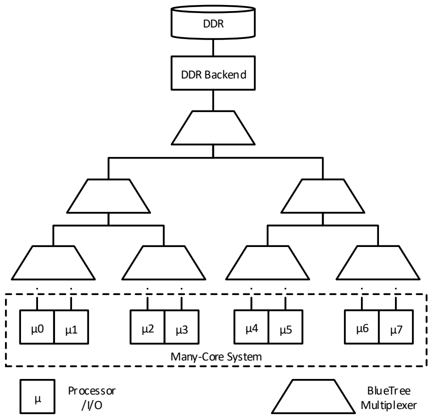


Fig. 9. BlueTree Memory Hierarchy

In this hardware consumption analysis, we assume:

- Only BlueIO is included – hence BlueTree is not included (as the functionality of BlueTree is memory access, which is not necessary for I/O);
- An independent I/O request buffer (buffer pool in VCDC) and an independent I/O request execution unit (GPIOCPU in GPIOCP) are allocated to each CPU – therefore, the number of buffer pools and GPIOCPUs in BlueIO equals to the number of CPUs in the whole system.

The following terms are introduced:

- Number of CPUs in the system: m;
- Number of I/Os in the system: n;
  - I/Os are indexed as from IO_1 to IO_n: UART — IO_1, SPI flash — IO_2, VGA — IO_3, and Ethernet — IO_4.
- Hardware consumption: C, where $C_x^{m,n}$ gives the hardware consumption of module x dependent on the number of CPUs (m) and I/Os (n) respectively.

In the analysis, we define the hardware consumption of a 1-CPU BlueIO system with GPIOCP ($C_{BIO}^{m=1,n=0}$) as the basic BlueIO system. We also define the difference between the m-CPU and n-IO BlueIO ($C_{BIO}^{m,n}$) and the basic BlueIO system as $\Delta C_{BIO}^{m,n}$. Therefore, the hardware consumption of a m-CPU and n-IO BlueIO system can be calculated as:

$$C_{BIO}^{m,n} = C_{BIO}^{m=1,n=0} + \Delta C_{BIO}^{m,n} \tag{4}$$

Similarly, the variation of hardware consumption of the m-CPU and n-IO VCDC and GPIOCP compared with the basic systems are $\Delta C_{VCDC}^{m,n}$ and $\Delta C_{GPIOCP}^{m,n}$ respectively:

$$C_{GPIOCP}^{m,n} = C_{GPIOCP}^{m=1,n=0} + \Delta C_{GPIOCP}^{m,n} \tag{5}$$

$$C_{VCDC}^{m,n} = C_{VCDC}^{m=1,n=0} + \Delta C_{VCDC}^{m,n} \tag{6}$$

BlueIO is comprised by BlueGrass, VCDC, and GPIOCP (See Figure 5). Since the hardware consumption of BlueGrass is constant, the variation of hardware consumption in BlueIO ($\Delta C_{BIO}^{m,n}$) equals to the sum of the variation of hardware consumption occurred in VCDC ($\Delta C_{VCDC}^{m,n}$) and GPIOCP ($\Delta C_{GPIOCP}^{m,n}$):

$$\Delta C_{BIO}^{m,n} = \Delta C_{VCDC}^{m,n} + \Delta C_{GPIOCP}^{m,n} \tag{7}$$

The hardware consumption of the VCDC (see Figure 7) is dominated by I/O VMMs and buffer pools (around 99%). Hence we consider VCDC hardware consumption as the summation of I/O VMMs (C_{VIO_i}) and buffer pools (C_{BP}) and ignore the effects from the other variables. In the proposed design, the hardware consumption of an I/O VMM (C_{VIO_i}) and a buffer pool (C_{BP}) are constant. Additionally, the number of I/O VMMs equals the number of I/Os, meanwhile, the number of buffer pools equals the number of CPUs. Therefore, the increased hardware consumption of VCDC ($\Delta C^{m,n}_{VCDC}$) is calculated as:

$$\Delta C^{m,n}_{VCDC} \approx \sum_{i=1}^{n} (C_{VIO\_i} + m * C_{BP}) \tag{8}$$

For the GPIOCP (see Figure 8), the only variation in its hardware consumption is the number of GPIOCPUs (C_{GCPU}), which equals the number of CPUs in the system. Therefore, the variation of hardware consumption of GPIOCP (ΔC_{GPIOCP}) is calculated as:

$$\Delta C^{m}_{GPIOCP} = (m-1) * C_{GCPU} \tag{9}$$

Combining equations 4, 7, 8, and 9 gives the hardware consumption of BlueIO to be:

$$C^{m,n}_{BIO} = C^{m=1,n=0}_{BIO} + \sum_{i=1}^{n} (C_{VIO\_i} + m * C_{BP}) + (m-1) * C_{GCPU} \tag{10}$$

Expanding gives:

$$C^{m,n}_{BIO} = C^{m=1,n=0}_{BIO} + \sum_{i=1}^{n} C_{VIO\_i} + (m-1) * C_{GCPU} + m * n * C_{BP} \tag{11}$$

Equation 11 shows that the hardware consumption of implementing BlueIO:

- scales linearly in the number of I/Os (n), while the number of CPUs (m) is constant;
- scales linearly in the number of CPUs (m), while the number of I/Os (n) is constant.

5.1 Implementing BlueIO in VLSI

This section shows that the implementation of BlueIO in VLSI has scalable hardware consumption at the gate level. Firstly, we use the Cadence RTL encounter compiler (v11.20) [6] to synthesise and provide gate level hardware consumption of each basic component in BlueIO, i.e. $C^{m=1,n=0}_{BIO}$, C_{GCPU}, C_{BP}, and C_{VIO_n} (see Table 3). Secondly, we synthesis BlueIO with different numbers of CPUs and I/Os and give their gate level hardware consumption in Table 4. Note that $OSU\_SOC\_v2.5$ [7] is the open source MOSIS SCMOS TSMC 0.25um library used in the synthesis. The consumption of logic gates may be varied by a specific synthesis compiler and adopted synthesis library.

Table 3. Hardware Consumption of Basic Modules (Gate Level)

Component	$C^{m=1,n=0}_{BIO}$	C_{GCPU}	C_{BP}	C_{VIO_1}	C_{VIO_2}	C_{VIO_3}	C_{VIO_4}
AND	201	64	47	328	621	512	981
AOI	1,085	369	36	1,502	2,381	2,201	4,523
DFFPOS	1,020	382	54	1,196	2,021	1,981	3,708
HA	12	6	1	13	18	15	60
INV	1,346	666	59	1,621	2,531	2,512	5,128
MUX2	7	5	0	10	14	16	80
NAND	745	477	70	1,253	1,573	1,789	3,001
NOR	572	248	25	7,61	1,221	1,201	2,401
OAI	633	420	35	1,066	1,652	1,602	3,101
OR	115	35	2	62	141	142	250
XNOR	9	10	0	26	40	36	32
XOR	10	6	3	21	20	20	52
Total	5,755	2,688	332	7,859	12,233	12,027	23,317

Table 3 shows that I/O VMM (C_{VIO_n}) consumes more gates when compared with GPIOCPU (C_{GCPU}) and buffer pool (C_{BP}). Therefore, even though the hardware consumption of BlueIO scales linearly in the number of CPUs (m) and I/Os (n) (see Equation 10), the number of I/Os (n) and the specific implementation of corresponding I/O VMMs (C_{VIO_n}) dominate overall hardware consumption.

Table 4. Hardware Consumption of BlueIO (Gate Level)

	$C_{BIO}^{m=1,n=0}$	$+IO\_1$			$+IO\_2$			$+IO\_3$			$+IO\_4$		
Numb. CPUs	1	1	2	4	1	2	4	1	2	4	1	2	4
AND	201	292	381	482	529	550	680	1,006	1,205	1,379	1,921	2,025	2,150
AIO	1,085	1,579	1,996	2,852	2,573	2,988	3,925	4,769	5,268	6,233	9,222	9,852	10,850
DEEPOS	1,020	1,288	1,695	2,512	2,188	2,776	3,752	3,988	4,520	5,425	7,588	7,992	8,895
HA	12	47	52	68	27	34	48	39	46	59	98	106	120
INV	1,346	1,801	2,623	4,156	2,909	3,650	5,125	5,371	6,210	7,685	10,307	11,125	12,650
MUX2	7	16	21	32	16	20	33	31	38	48	113	125	141
NAND	745	972	1,525	2,487	1,876	2,501	3,602	3,449	4,000	5,153	6,330	6,952	8,053
NOR	572	729	1,051	1,753	1,233	1,666	2,325	2,350	3,052	3,752	4,661	5,125	6,002
OAI	633	775	1,325	2,423	1,694	2,050	3,112	3,241	3,825	4,057	6,337	6,925	8,125
OR	115	83	125	193	182	252	388	312	388	412	579	628	755
XNOR	9	7	19	43	29	41	65	64	79	102	96	113	141
XOR	10	16	28	49	27	39	57	46	55	71	91	102	115
Total	5,755	7,605	10,841	17,050	13,283	16,567	23,112	24,666	28,686	34,376	47,343	51,070	57,997

Table 4 shows that the hardware consumption of BlueIO increases linearly with the number of CPUs (m) and I/Os (n) respectively. Specifically, if the number of I/Os (n) is fixed, hardware consumption may increase slightly as the number of CPUs (m) scales. Similarly, if m is fixed, the hardware consumption increases linearly as the number of I/Os increase. Additionally, the types of I/O included can also affect the hardware consumption — the logic gates required for a simple I/O (eg. $C_{BIO}^{m=1,n=0}$ with IO_1) is far less than a complicated I/O (e.g. $C_{BIO}^{m=1,n=0}$ with IO_4).

5.2 Hardware Consumption in RTL Level (FPGA)

Vivado (v2016.2) was used to synthesis and implement BlueIO on Xilinx VC709 FPGA board [14] with increasing numbers of I/Os and CPUs. The hardware consumption of BlueIO was recorded at the RTL level in terms of LUTs, registers, BRAMs, power consumption and maximum working frequency.

Table 5. Hardware Consumption of 2-CPU BlueIO with Different I/Os on FPGA (RTL Level)

Added I/O	Hardware Consumption								Power (mW)	Maximum Frequency (Mhz)
	LUTs	% of VC709	Register	% of VC709	BRAMs	% of VC709	DSP	% of VC709		
+ UART	2192	0.12%	1471	0.17%	0	0%	0	0%	13	221.8
+ VGA	4566	0.51%	2315	0.27%	0	0%	0	0%	19	221.8
+ SPI Flash	6120	1.41%	4225	0.49%	0	0%	0	0%	29	221.8
+ Ethernet	9723	2.24%	9035	1.04%	0	0%	0	0%	75	192

Table 6. Hardware Consumption of BlueIO (+GPIOCP) with Different Numbers of CPUs on FPGA (RTL Level)

Number of CPUs	Hardware Consumption								Power (mW)	Maximum Frequency (Mhz)
	LUTs	% of VC709	Register	% of VC709	BRAMs	% of VC709	DSP	% of VC709		
1	632	0.146%	962	0.111%	16	1.09%	0	0%	19	318
2	886	0.205%	1156	0.113%	16	1.09%	0	0%	20	303
4	1314	0.303%	1468	0.169%	16	1.09%	0	0%	22	291
8	1942	0.448%	2094	0.242%	16	1.09%	0	0%	25	284
16	3236	0.747%	3346	0.386%	16	1.09%	0	0%	31	249
32	5065	1.169%	5311	0.613%	16	1.09%	0	0%	37	236
64	8698	2.008%	8449	0.975%	16	1.09%	0	0%	50	204

The resource efficiency of BlueIO is shown by the Table 5 and 6, eg. a full featured 2-CPU BlueIO only consumes 2.24% LUTs and 1.04% Registers of the VC709 FPGA board. As shown, DSP slice is not required by the implementation of BlueIO on FPGA. Additionally, the slices of LUTs and registers are linearly increased by the number of I/Os and CPUs, respectively. Furthermore, the increased hardware consumption also leads to a linear increment in power consumption and a decrement in maximum working frequency.

6 EVALUATION

BlueIO was implemented using Bluespec[1] and synthesized for the Xilinx VC709 development board[14] (further implementation details are given in a technical report [20]). The following evaluation focusses on I/O devices and I/O systems, we do not consider the effects caused by NoC and routing protocols. In the evaluation, the BlueIO system was connected to a 4 x 5 2D mesh type open source real-time NoC[47] containing 16 Microblaze CPUs[11] running the modified guest OS (FreeRTOS v9.0.0) in the guest VM (see Section 3.2). The architecture is shown in Figure 4. To enable comparison, a similar hardware architecture without BlueIO was built – note that this architecture requires I/O operations requested by Mircoblaze CPUs to pass through the mesh to the I/O rather than being controlled by BlueIO. Both architectures run at 100 MHz.

6.1 Memory Footprint

In this section, the memory footprint of BlueIO is evaluated. It considers different versions of FreeRTOS running on Microbalze CPUs and uses the size tool of the Xilinx Microblaze GNU Tool chain. In the measurement, the native version of FreeRTOS (nFreeRTOS) is full-featured [8], which is the foundation of the other versions [1] [2] [3]. Table 7 presents the collected measurements.

Table 7. BlueIO Memory Footprint (Bytes)

Software	Memory Footprint			
	.text	.data	.bss	Total
BlueIO	0	0	0	0
nFreeRTOS	121,309	1,728	35,704	158,741
nFreeRTOS + I/O	179,652	1,852	36,250	217,754
vFreeRTOS + I/O	189,556	1,882	36,450	227,888
BV_vFreeRTOS + I/O	131,969	1,732	35,723	169,424

As it can be seen, the memory overhead introduced by the hypervisor (BlueIO) is zero, resulting from its pure hardware implementation. The native full-featured FreeRTOS (nFreeRTOS) requires 158741 bytes – with I/O module added, the memory footprint increases 37.18%, owing to the addition of I/O manager and I/O drivers. When it comes to the vFreeRTOS + I/O, the introduction of software implemented virtualization increases the memory footprint to 227, 888 bytes. However, BV_vFreeRTOS + I/O only consumes 169, 424 bytes of memory, which is increased by 6.73% compared to the native FreeRTOS, as well as 77.81% and 74.35% of the nFreeRTOS + I/O and vFreeRTOS + I/O, respectively. The main reason behind such a low memory footprint is that the implementation of para-virtualization (described in Section 3.2), has removed the software overhead significantly.

6.2 Timing Accuracy

This section compares the timing accuracy of the I/O operations in BlueIO and non-BlueIO systems. In both architectures, 9 CPUs were active, whose coordinates are from (0, 0) to (0, 2), (1, 0) to (1,

[1]FreeRTOS + I/O involves UART, VGA, and corresponding drivers.
[2]vFreeRTOS is a simply implemented software virtualized FreeRTOS for many-core systems, see [37].
[3]BV_vFreeRTOS is the virtualized FreeRTOS in BlueIO system.

2) and (2, 0) to (2, 2). When CPUs were required to access and read the GPIO at a specific time, then in the non-BlueIO architecture the CPU instigated the I/O operation, whilst in the BlueIO architecture, this was performed by BlueIO (ie. GPIOCP) to achieve timing accuracy. This was shown by connecting a timer to the GPIO (updating its value every cycle), with every CPU needing to read the value simultaneously.

Results of 1000 experiments are given in Table 8, showing that the latencies and variance for the non-BlueIO architecture are significant (errors calculated according to equation 1); in contrast, the BlueIO architecture is timing accurate at the cycle level.

Table 8. I/O Operation Timing Variance

CPU Index	Non-BlueIO								BlueIO							
	E (unit: ns)				E (unit: clock cycle)				E (unit: ns)				E (unit: clock cycle)			
	Min	Med	Mean	Max	Min	Med	Mean	Max	Min	Med	Mean	Max	Min	Med	Mean	Max
(0,0)	3140.0	3140.0	3145.8	3160.0	314	314	315	316	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(0,1)	3000.0	3000.0	3005.8	3020.0	300	300	301	302	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(0,2)	2790.0	2790.0	2795.8	2810.0	279	279	280	281	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(1,0)	2720.0	2720.0	2725.8	2740.0	272	272	273	274	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(1,1)	3070.0	3070.0	3075.8	3090.0	307	307	308	309	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(1,2)	2860.0	2880.0	2899.4	2940.0	286	288	289	294	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(2,0)	2580.0	2580.0	2585.8	2600.0	258	258	259	260	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(2,1)	2650.0	2650.0	2655.8	2670.0	265	265	266	267	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(2,2)	2860.0	2930.0	2902.0	2950.0	286	293	290	295	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

6.3 I/O Performance

The I/O performance evaluation considers I/O response time and I/O throughput separately in the following sections.

6.3.1 I/O Response Time. This experiment was designed to evaluate the I/O response time whilst CPUs and measured I/O were fully loaded within BlueIO and non-BlueIO systems. In both architectures, all active CPUs executed an independent application. The application continuously read data from an SPI NOR-flash (model: S25FL128S). The experiment was divided into four groups, depending on the number of bytes read: ie. 1, 4, 64 and 256 bytes. All experiments were run 1,000 times. We name the experiments according to the scheduling policy and the number bytes of data read in one I/O request. For example, *non-BlueIO-RR-4B* refers to a non-BlueIO system with Round-Robin global scheduling policy; and 4 bytes of data read from the NOR-flash in one I/O request.

For the non-BlueIO architecture, FreeRTOS was modified to be suitable for many-core systems[4]. In both architectures, while the user applications on different CPUs were requesting the I/O at the same time point, the scheduling policy could be set as local FIFO (non-BlueIO-FF and BlueIO-FF) and global Round-Robin (non-BlueIO-RR and BlueIO-RR) respectively. Experimental results showing the worst case and variation of each group of experiments are summarised in Table 9 (complete experimental results are given in [20, 37]).

Table 9 shows that the worst-case response time of I/O requests in the non-BlueIO architecture is significantly high for the reading of 1, 4, 64 or 256 byte(s) from the NOR-flash, especially whilst global Round-Robin scheduling was used – noting that a lower I/O response time indicates a higher I/O performance. In experiments with the number of read bytes increased, BlueIO system maintains its superior performance. Additionally, when it comes to variation, BlueIO systems have a better performance than non-BlueIO systems. For example, in the non-BlueIO-FF-1B, the variation is greater than 1, 500 clock cycles; and in non-BlueIO-RR-1B, the variation reaches to 60, 000 clock cycles. Conversely, in both BlueIO-FF-1B and BlueIO-RR-1B, the highest variance is less than 60

[4]FreeRTOS is designed for a single-core system; in our experiments, we modified it to be suitable for many-core systems [37]

Table 9. I/O Response Time in BlueIO and non-BlueIO Systems (unit: clock cycle) (Summarized Version)

Written Bytes	Non-BlueIO (FIFO)		Non-BlueIO (Round-Robin)		BlueIO (FIFO)		BlueIO (Round-Robin)	
	Worst Case	Variation	Worst Case	Variation	Worst Case	Variation	Worst Case	Variation
1	9,357	1,541	65,885	59,736	532	57	403	46
4	58,844	7,061	327,813	286,733	1,785	368	1,569	276
8	936,166	98,026	4,555,159	3,823,104	25,053	3,667	23,032	3,542
16	3,702,565	284,142	17,345,151	15,475,355	92,153	15,225	89,708	13,711

clock cycles. Therefore, the evaluation results reveal that a system with BlueIO provides more predictable I/O operations with lower response time.

6.3.2 I/O Throughput. The I/O throughput was evaluated using two architectures – one with BlueIO and one without BlueIO. In the experiments, we used the same NOR-flash described in the previous section. Additionally, the scheduling policy in both architectures was set as local FIFO and global Round-Robin respectively.

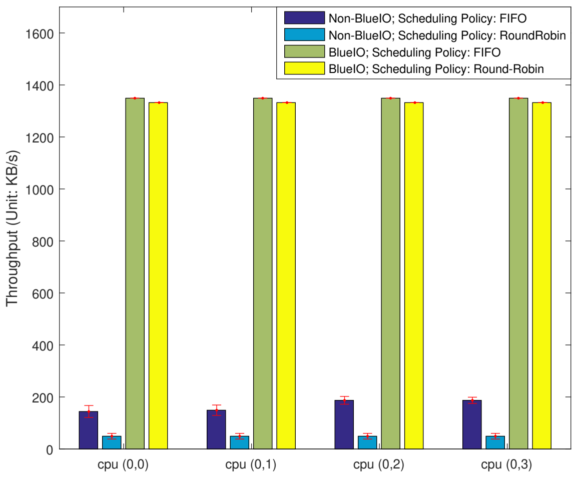


Fig. 10. I/O Throughput

In both architectures, an independent application was executed on each of 4 Microblaze CPUs (coordinates are from (0,1) to (0,3)) that continuously wrote to the NOR-flash — one byte written per I/O request. The number of bytes written from each CPU per second was recorded as I/O throughput (unit: KB/s). The result of higher I/O throughput implies a better performance. All the evaluations were implemented 1,000 times. The evaluation results are shown in Figure 10. In the figure, the four bar chart groups present the average I/O throughput in the BlueIO system and the non-BlueIO system for each CPU; and the error bar on each bar chart presents the variance of the I/O throughput during these 1,000 experiments. As shown, on all CPUs considered, no matter which scheduling policy is used, the BlueIO system always provides higher I/O throughput (nearly 7 times) and less variance.

6.4 Timing Scalability

This section provides an evaluation of the timing scalability of the BlueIO system when connected to a complex device, ie. Ethernet. The evaluation was implemented by measuring the I/O response time of Ethernet packets sent from different CPUs in single-core, 4-core, 8-core, and 16-core systems respectively. The implementation of the Ethernet virtualization in BlueIO system is given in [37].

The experiment was divided into two parts, dependent on the global scheduling policy of the BlueIO: Round-Robin (named BlueIO-RR) and fixed priority (named BlueIO-FP). For BlueIO-RR and BlueIO-FP, the experiments were further divided into four parts, according to the number of active CPUs. In these four parts of the experiments, we activated 1, 4, 8 and 16 Microblaze CPUs respectively. The experiments are named according to the global scheduling policy of the experiment plus the number of active CPUs. For example, in a 4-core BlueIO system with Round-Robin global scheduling policy, the experiment is labeled as BlueIO-RR-4.

Table 10. **Average Response Time of Loop Back 1KB Ethernet Packets in BlueIO System (Global Scheduling Policy: Fixed Priority; Unit: us)**

Number of CPUs	CPU Coordinate																Δr
	(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)	(0,3)	(1,3)	(2,3)	(3,3)	
1	11.5	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0
4	12.0	25.5	36.9	48.3	X	X	X	X	X	X	X	X	X	X	X	X	1.29
8	12.1	25.5	36.9	48.3	59.7	71.2	82.6	94.0	X	X	X	X	X	X	X	X	0.96
16	12.0	25.5	36.9	48.3	59.7	71.1	82.6	95.0	105.4	116.9	128.3	139.7	151.1	162.5	174.0	185.4	0.8

The software application running on each active CPU was the same, and continuously sent 1 KB Ethernet packets via BlueIO to a dedicated component. The 1 KB Ethernet packets sent from different CPUs were exactly the same. A dedicated component was used to monitor the response time of these Ethernet packets by recording the reach time and analysing the virtual source IP address of the packets. All the experiments were run 1000 times.

In BlueIO-FP, CPU (0, 0) was always set to the highest priority, followed by CPU (1, 0), (2, 0), (3,0) and (1, 0) etc. The experiment results are shown in Table 10. As shown, for all many-core systems, the I/O response time from the CPU with the highest priority is always fixed around 12μs; and the I/O requests from the CPUs with the lower priorities were always blocked by I/O requests with higher priorities, which guarantees the execution of the I/O requests with higher priorities. For example, in BlueIO-FP-8, the average response time of the I/O requests from CPU (0,0) (the highest priority) is kept to 12μs, which means it can never be blocked by others. When it comes to the I/O requests from CPU (3, 1) (the lowest priority), the I/O response time is always around 94μs, which is 8 times of the highest priority I/O requests. In an 8-core system, the theoretical optimal response time of the lowest priority I/O request should be 8 times the highest priority I/O request, which means that the BlueIO system does not introduce an extra delay for the lowest priority I/O request –as shown by the experimental results. In addition, with the number of CPUs increased, there is no obvious increment in Δr, which implies the loss of I/O performance is not significant as the number of CPUs is increased, showing good scalability of the BlueIO system (with the fixed priority scheduling policy).

For BlueIO-RR, the experiment results are shown in Table 11. As shown, with an increment in the number of CPUs, the I/O response time of each CPU is proportional to the number of CPUs. For example, the average response time of an I/O request in BlueIO-RR-4, BlueIO-RR-8, and BlueIO-RR-16 is close to their theoretical optimal values, which are around 4, 8 and 16 times of the one in a single-core system (BlueIO-RR-1). In addition, with the number of CPUs increased, there is no obvious increment in Δr, which also shows the good scalability of the BlueIO system (with the Round-Robin scheduling policy). Note that, Δr gives the average I/O performance loss suffered by each CPU, with ΔR showing the total I/O performance loss (See Formulas 2 and 3). Therefore, even as Δr decreases with increasing number of CPUs (n), the total I/O performance loss ($\Delta R = \Delta r * n$) increases significantly.

Table 11. **Average Response Time of Loop Back 1KB Ethernet Packets in BlueIO System (Global Scheduling Policy: Round Robin; Unit: us)**

Number of CPUs	CPU Coordinate																Δr
	(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)	(0,3)	(1,3)	(2,3)	(3,3)	
1	11.0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0
4	46.7	47.2	47.6	48.1	X	X	X	X	X	X	X	X	X	X	X	X	0.84
8	90.5	90.8	91.2	91.5	91.9	92.2	92.6	92.9	X	X	X	X	X	X	X	X	0.44
16	180.1	180.7	179.9	180.6	180.0	180.7	180.0	180.7	180.0	180.7	180.0	180.7	180.0	180.7	180.0	180.7	0.25

6.5 On-chip Communication Overhead and Scalability

In NoC-based many-core systems, all the I/O requests are transmitted as on-chip packets, with larger packets bringing a higher on-chip communication overhead. In this section, we compare the on-chip communication overhead while invoking commonly used I/O requests in BlueIO and non-BlueIO systems by recording the number of packets on the NoC. In the NoC [47], the width of all the on-chip packets is 32 bits. The evaluation results are demonstrated in Table 12. Results show that whilst the invoked I/O request is simple, the on-chip communication overhead is similar in all the systems, eg. displaying one pixel via the VGA in a single-core system. When the I/O operations become complicated or the number of CPUs is increased, the on-chip communication overhead in non-BlueIO architecture is significant; in contrast, the BlueIO architecture has a lower on-chip communication overhead, for example, reading 10 bytes data from the SPI flash in 10-core systems.

Table 12. On-chip Communication Overhead

I/O Device	I/O Operation		Number of on-chip Packets (Each Packet: 32-bit)		
			Non-BlueIO FIFO	Non-BlueIO Round-Robin	BlueIO
VGA	Display 1 Pixel	1 CPU	6	6	3
		4 CPUs	24	33	12
		10 CPUs	60	87	30
	Display 10 Pixels[5]	1 CPU	60	60	30
		4 CPUs	240	357	120
		10 CPUs	600	897	300
SPI Flash	Read 1 Byte	1 CPU	12	12	4
		4 CPUs	48	57	16
		10 CPUs	120	237	40
	Read 10 Bytes	1 CPU	120	120	40
		4 CPUs	480	597	160
		10 CPUs	1200	1497	400

7 RELATED WORK

This section presents the background, related research, and projects on real-time I/O virtualization, followed by corresponding analysis and discussion.

7.1 NoC-based Many-core Systems

Typical NoC based architectures (i.e. Figure 11) that have been implemented in silicon contain integrated devices connected to the edge of the mesh, e.g. Tilera's TILE64[13] and Kalray's MPPA-256[26], as well as I/Os (connected to the mesh).

The TILE64 requires CPUs within the mesh to instigate I/O operations, with a shared I/O controller passing the operation to the actual I/O device — hence significant latencies will occur between I/O command instigation and actual I/O occurring, which detracts from timing-accuracy and predictability of I/O operations. The MPPA-256 provides 4 I/O subsystems, with I/O operations instigated by the CPU passed to the Resource Manager (RM) cores within one of these I/O systems depending which device is required. The MPPA-256 RM cores are essentially Linux based CPUs controlling many devices (even though RTEMS [2] can also be used), hence timing accurate and

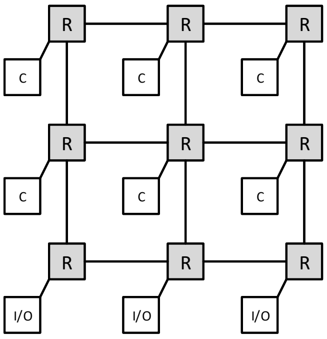
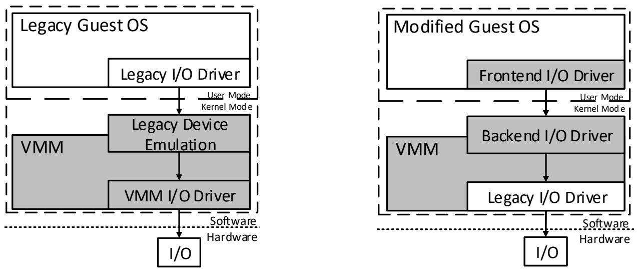


Fig. 11. A Typical Structure of NoC-based Many-Core System
(C - Core; R - Router / Arbiter)

predictable control of many external devices connected to the GPIO pins is not possible — also the approach is not resource efficient as a CPU is required for I/O control. Additionally, among these systems, there is no extra I/O virtualization support existed.

7.2 Software I/O Virtualization

Software virtualization can be classified into full virtualization and para-virtualization (see Figure 12, the grey parts are involved in virtualization implementation).



(a) Full Virtualization (b) Para-virtualization

Fig. 12. Classification of Software I/O Virtualization

In full virtualization, in order to maintain the guest OS being unmodified, I/O virtualization is implemented by the VMM in kernel mode. This drives, multiplexes I/O devices, and emulates multiple virtual device interfaces e.g. VMware ESX Server[16], KVM[10] and VirtualBox[15]. I/O requests from guest OS always trap the VMM. Afterwards, the VMM decodes the trapped I/O requests and maps them into physical devices, in order to drive complete I/O operations. This approach requires VMM entirely controlling physical devices, which involves in complicated I/O access path and significant software overhead (see Section 1). Additionally, once a new I/O device added, the VMM (I/O drivers) requires to be also modified and updated. Quest-V designed by Boston University [42] is an optimized full-virtualized multi-kernel, which efficiently reduce the access path of I/O operations. Specifically, Quest-V shares certain driver data structures across sandboxes (VMs), to allow I/O requests and responses to be handled locally. This solution allows any sandbox (VM) to be configured for the corresponding device interrupts, rather than have a dedicated sandbox to be responsible for all communication with that device. This greatly reduces the communication and control paths necessary for I/O requests from applications in Quest-V. However, the software implementation of Quest-V determines unavoidable significant software overhead. In addition, Quest-V does not have any assist for real-time I/O control (predictable and timing-accurate).

In para-virtualization, guest OS is normally modified to gain more efficiency and performance. In I/O part, an I/O driver is split into front-end and back-end drivers. Specifically, the back-end driver is installed in VMM (kernel mode) for physical device access, and provide access interface for guest OS. The front-end driver installed in guest OS is responsible for handling I/O requests and passing them to a corresponding back-end driver e.g. Xen[19] and OKL4[33]. With para-virtualization, VMM does not require to fully drive physical devices, which reduces the software overhead efficiently. Xen is a widely used open source VMM, therefore a number of related works are proposed for its enhancement. For example, Kaushik Kumar Ram et al[48] introduced the technology of reducing the overhead in guest OS, including engaging Large Receive Offload (LRO), employing software pre-fetching and reducing buffer size to half-page (reducing the TLB miss rate). Diego Ongaro [45] improves the VMM scheduler to gain a better overall system performance and equity. Most of these works efficiently decrease software overhead and increase system performance, however, these works cannot improve I/O performance (compared to a bare-metal system), and increase predictability and timing-accuracy of I/O operations.

7.3 Hardware I/O Virtualization Assistance

In order to alleviate the penalties suffered by software I/O virtualization e.g. complicated I/O access paths and significant software overhead, hardware assistances are proposed for I/O virtualization. VMM-bypass direct I/O makes VM access hardware devices directly without VMM or driver domain interposing, which enhances performance and exposes all hardware functionality to VM directly. For example, Intel VT-d [34], AMD IOMMU[22] and SR-IOV[12]. Because VT-d and IOMMU are similar technologies, we only introduce TV-d as an example.

Virtualization Technology for Directed I/O (VT-d) is the hardware support for isolating and restricting device accesses to the owner of the partition managing the device, which is developed by Intel [34]. VT-d includes three key capabilities: 1). Allows an administrator to assign I/O devices to guest VMs in any desired configuration; 2). Supports address translations for device DMA data transfers; and 3). Provides VM routing and isolation of device interrupts. In general, VT-d provides a hardware VMM that allows user applications running in the guest VMs to access and operate I/O devices directly, which decreases path of I/O access, as well as off-loads most overhead of virtualization from software to hardware (see Figure 13). However, apart from original I/O drivers, extra drivers for VT-d are also required in the software layer, which results in an increment of software overhead and a loss of the I/O performance [?]. Additionally, VT-d cannot guarantee the real-time properties of I/O operations.

Single Root I/O Virtualization (SR-IOV) is a specification, which proposes a set of hardware enhancements for the PCIe device. SR-IOV aims to remove major VMM intervention for performance data movement to I/O devices, such as the packet classification and address translation. An SR-IOV-based device is able to create multiple “light-weight” instances of PCI function entities (also known as VFs). Each VF can be assigned to a guest for direct access, but still shares major device resources, achieving both resource sharing and high performance. Currently, many I/O devices have already supported the SR-IOV specification, such as [27], [28] and [38]. Similar to Intel VT-d, to support an SR-IOV-based I/O, more drivers are required in the software, which detracts from I/O performance and lacks in real-time properties.

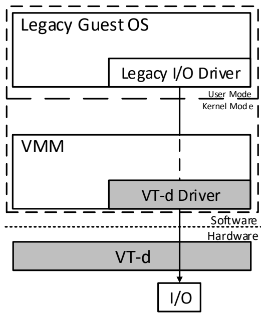


Fig. 13. VT-d Assisted I/O virtualization

7.4 Programmable Timely I/O Controller: PRU and TPU

TI Programmable Real-Time Unit (PRU) [23] and Freescale Time Processor Unit (TPU) [9] are programmable controllers that could be connected to a multi-core or many-core system for I/O control. The PRU contains two 32-bit RISC cores and ready built I/O controllers that are capable of real-time I/O, but the exact timing of I/O operations (e.g. at specific times in the future) is not possible, and the use of CPUs is not resource efficient. The TPU is essentially a RISC CPU with a timer subsystem and I/O controllers. Timing accuracy of I/O operations is not possible as I/O is instigated by a remote CPU; and the use of a CPU is not resource efficient. Both TPU and PRU, being sequential CPUs, cannot easily provide timing accuracy across a number of devices connected to GPIO.

8 CONCLUSION

In safety-critical real-time systems, I/O operations often require predictability, timing-accuracy, parallel access, isolation, and scalability simultaneously. This paper has proposed a scalable hardware-implemented real-time I/O system for multi-core and many-core systems — BlueIO, which satisfies the requirements at same time. BlueIO includes previous work (VCDC, GPIOCP,and BlueTree), extended by integration of I/O virtualization, low layer I/O drivers and the clock cycle level timing-accurate I/O controller (GPIOCP) in the hardware layer, meanwhile providing abstracted high-layer access interfaces to software layers (Guest VMs).

Evaluation reveals that BlueIO can virtualize a physical I/O to multiple virtual I/Os with significant performance improvements, including faster I/O response time, higher I/O throughput, less on-chip communication overhead, good scalability, and isolation. In addition, BlueIO can also handle multiple I/O operations with clock cycle accuracy, being in many cases totally timing-accurate and predictable. In the hardware consumption analysis, the paper has demonstrated that the hardware consumption of BlueIO scales linearly in the number of CPUs and I/Os respectively, evidenced by the implementation in VLSI and FPGA.

8.1 Future Work

There are several possible areas of future research based on this work presented in the paper. These include:

- Timing Analysis – In real-time systems, two commonly used methodologies are normally adopted to evaluate predictability — static analysis to identify the worst-case and measurement-based analysis [24]. In this paper, we only adopted measurement-based analysis to evaluate the predictability of I/O operations in Section 6. However, to find the worst-case path of a program is difficult in the general case. Further work is needed to find accurate WCET of applications using I/O operations in our system, and to include this within system schedulability analysis.

- Supporting I/O access preemption – Currently, our proposed BlueIO system cannot support I/O preemption. This is mainly resulted from the difficulties in achieving context switch among I/O operations in hardware level [40]. In order to overcome this drawback, it may be necessary to include further parts of the OS within hardware. This remains for future work.

REFERENCES

[1] 2015. Bluespec Inc. Bluespec System Verilog (BSV). (2015). Retrieved Sept. 27, 2015 from http://www.bluespec.com/products/
[2] 2015. RTEMS official website. (2015). Retrieved Aug. 26, 2015 from https://www.rtems.org/
[3] 2015. uCos official website. (2015). Retrieved Sept. 27, 2015 from https://www.micrium.com/rtos/kernels/
[4] 2015. Xilinx official website. (2015). Retrieved Jul. 5, 2015 from https://www.Xilinx.com
[5] 2016. (2016). Retrieved Aug. 27, 2016 from https://github.com/RTSYork/GPIOCP/
[6] 2016. Encounter RTL Compiler. (2016). Retrieved Oct. 16, 2016 from https://www.cadence.com/content/cadence-www/global/en_US/home/training/all-courses/84441.html
[7] 2016. Encounter RTL Compiler. (2016). Retrieved Oct 16, 2016 from https://vlsiarch.ecen.okstate.edu/flows/
[8] 2016. FreeRTOS official website. (2016). Retrieved Sept 27, 2016 from http://www.freertos.org/
[9] 2016. Freescale official website. (2016). Retrieved Aug. 27, 2016 from http://www.nxp.com/
[10] 2016. KVM official website. (2016). Retrieved Oct. 16, 2016 from https://www.linux-kvm.org/page/Main_Page
[11] 2016. Microblaze User Manual. (2016). Retrieved Aug. 27, 2016 from http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf
[12] 2016. SR-IOV official website. (2016). Retrieved Sept. 27, 2016 from http://pcisig.com/
[13] 2016. Tilera official website. (2016). Retrieved Aug. 27, 2016 from http://www.tilera.com/
[14] 2016. VC709 official Website. (2016). Retrieved Aug. 27, 2016 from https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html
[15] 2016. VirtualBox official website. (2016). Retrieved Oct. 16, 2016 from https://www.virtualbox.org/
[16] 2016. VMware official website. (2016). Retrieved Oct. 16, 2016 from https://www.vmware.com/
[17] 2016. VxWorks official website. (2016). Retrieved Aug. 27, 2016 from http://windriver.com/products/vxworks/
[18] 2016. VxWorks Timer Library. (2016). Retrieved Aug. 27, 2016 from http://www.vxdev.com/docs/vx55man/vxworks/ref/timerLib.html
[19] 2016. Xen official website. (2016). Retrieved Oct. 16, 2016 from https://www.xenproject.org/
[20] 2017. (2017). Retrieved Jan. 27, 2017 from https://github.com/RTSYork/BlueIO
[21] K. Adams and O. Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. ACM Sigplan Notices 41, 11 (2006), 2–13.
[22] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn. 2007. The Price of Safety: Evaluating IOMMU Performance. In The Ottawa Linux Symposium. 9–20.
[23] R. Birkett. 2015. Enhancing Real-time Capabilities with the PRU. In Embedded Linux Conference.
[24] A. Burns and A.J. Wellings. 2001. Real-time Systems and Programming Languages: Ada 95, Real-time Java, and Real-time POSIX. Pearson Education.
[25] Z. Cheng, R. West, and Y. Ye. 2017. Building Real-Time Embedded Applications on QduinoMC: A Web-connected 3D Printer Case Study. In IEEE Real-Time and Embedded Technology and Applications Symposium. IEEE, 13–24.
[26] B.D. de Dinechin, P.G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. 2013. A Distributed Run-time Environment for the Kalray MPPA-256 Integrated Manycore Processor. Procedia Computer Science 18 (2013), 1654–1663.
[27] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. 2012. High Performance Network Virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* 72, 11 (2012), 1471–1480.
[28] Y. Dong, Z. Yu, and G. Rose. 2008. SR-IOV Networking in Xen: Architecture, Design and Implementation. In Workshop on I/O Virtualization.
[29] C. Fetzer, U. Schiffel, and M. Süßkraut. 2009. AN-Encoding Compiler: Building Safety-critical Systems with Commodity Hardware. In International Conference on Computer Safety, Reliability, and Security. Springer, 283–296.
[30] J. Garside and N.C. Audsley. 2013. Prefetching Across a Shared Memory Tree within a Network-on-Chip Architecture. In Proceedings ISSoC. 1–4.

[31] M.D. Gomony, J. Garside, B. Akesson, N.C. Audsley, and K. Goossens. 2015. A Generic, Scalable and Globally Arbitrated Memory Tree for Shared DRAM Access in Real-time Systems. In Proceedings of DATE. EDA Consortium, 193–198.

[32] M. Gomony, J. Garside, B. Akesson, N.C. Audsley, and K. Goossens. 2016. A Globally Arbitrated Memory Tree for Mixed-Time-Criticality Systems. *IEEE Trans. Comput.* 66, 2 (2016), 212–225.

[33] G. Heiser and B. Leslie. 2010. The OKL4 Microvisor: Convergence point of Microkernels and Hypervisors. In Proceedings of the ACM Asia-Pacific Workshop on Workshop on Systems. ACM, 19–24.

[34] R Hiremane. 2007. Intel Virtualization Technology for Directed I/O (Intel VT-D). Technology Intel Magazine 4, 10 (2007).

[35] L.S. Indrusiak, J. Harbin, and M.J. Sepulveda. 2017. Side-channel Attack Resilience through Route Randomisation in Secure Real-time Networks-on-Chip. In International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC). 1–8. https://doi.org/10.1109/ReCoSoC.2017.8016142

[36] Z. Jiang and N.C. Audsley. 2017. GPIOCP: Timing-Accurate General Purpose I/O Controller for Many-core Real-time Systems. In Proceedings of DATE. EDA Consortium.

[37] Z. Jiang and N.C. Audsley. 2017. VCDC: The Virtualized Complicated Device Controller. In LIPIcs-Leibniz International Proceedings in Informatics, Vol. 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[38] J. Jose, M. Li, X. Lu, K.C. Kandalla, M.D. Arnold, and D.K. Panda. 2013. SR-IOV Support for Virtualization on Infiniband Clusters: Early Experiences. In IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). IEEE, 385–392.

[39] J.-H. Kim, S. Sukkarieh, and S. Wishart. 2003. Real-time Navigation, Guidance, and Control of a UAV using Low-cost Sensors. In Field and Service Robotics. Springer, 299–309.

[40] F. Kuhns, D.C. Schmidt, and D.L. Levine. 1999. The Design and Performance of a Real-time I/O Subsystem. In IEEE Real-Time Technology and Applications Symposium. IEEE, 154–163.

[41] J.A. Landis, T.V. Powderly, R. Subrahmanian, A. Puthiyaparambil, and J.R. Hunter Jr. 2011. Computer System Paravirtualization using a Hypervisor that is Implemented in a Partition of the Host System. (Jul. 2011). US Patent 7,984,108.

[42] Y. Li, M. Danish, and R. West. 2011. Quest-V: A Virtualized Multikernel for High-confidence Systems. arXiv preprint arXiv:1112.5136 (2011).

[43] M. Masmano, I. Ripoll, A. Crespo, and J. Metge. 2009. Xtratum: a Hypervisor for Safety Critical Embedded Systems. In 11th Real-Time Linux Workshop. 263–272.

[44] J. Mossinger. 2010. Software in Automotive Systems. IEEE Software 27, 2 (2010), 92.

[45] D. Ongaro, A.L. Cox, and S. Rixner. 2008. Scheduling I/O in Virtual Machine Monitors. In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. ACM, 1–10.

[46] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. 2017. LTZVisor: TrustZone is the Key. In LIPIcs-Leibniz International Proceedings in Informatics, Vol. 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[47] G. Plumbridge, J. Whitham, and N. Audsley. 2014. Blueshell: a Platform for Rapid Prototyping of Multiprocessor NoCs and Accelerators. ACM SIGARCH Computer Architecture News 41, 5 (2014), 107–117.

[48] K.K. Ram, J.R. Santos, Y. Turner, A.L. Cox, and S. Rixner. 2009. Achieving 10 Gb/s using Safe and Transparent Network Interface Virtualization. In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. ACM, 61–70.

[49] J. Sahoo, S. Mohapatra, and R. Lath. 2010. Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. In International Conference on Computer and Network Technology. IEEE, 222–226.

[50] M. Schoeberl, S. Abbaspour, B. Akesson, N.C. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, and R. Heckmann. 2015. T-CREST: Time-predictable Multi-core Architecture for Embedded Systems. Journal of Systems Architecture 61, 9 (2015), 449–471.

[51] S. Trujillo, A. Crespo, and A. Alonso. 2013. Multipartes: Multicore Virtualization for Mixed-criticality Systems. In Euromicro Conference on Digital System Design. IEEE, 260–265.

Received ?; revised ?; accepted ?