GPIOCP: Timing-Accurate General Purpose I/O Controller for Many-core Real-time Systems

Zhe Jiang and Neil C. Audsley
Department of Computer Science, University of York, York, UK.
email: {zj577, neil.audsley}@york.ac.uk

***Abstract*—Modern SoC / NoC chips often provide General-Purpose I/O (GPIO) pins for connecting devices that are not directly integrated within the chip. Timing accurate control of devices connected to GPIO is often required within embedded real-time systems – ie. I/O operations should occur at exact times, with minimal error, neither being significantly early or late. This is difficult to achieve due to the latencies and contentions present in architecture, between CPU instigating the I/O operation, and the device connected to the GPIO – software drivers, RTOS, buses and bus contentions all introduce significant variable latencies before the command reaches the device. This is compounded in NoC devices utilising a mesh interconnect between CPUs and I/O devices.**

The contribution of this paper is a resource efficient programmable I/O controller, termed the GPIO Command Processor (GPIOCP), that permits applications to instigate complex sequences of I/O operations at an exact time, so achieving timing-accuracy at a single clock cycle level. Also, I/O operations can be programmed to occur at some point in the future, periodically, or reactively. The GPIOCP is a parallel I/O controller, supporting cycle level timing accuracy across several devices connected to GPIO simultaneously.

The GPIOCP exploits the tradeoff between placing using a full sequential CPU to control each GPIO connected device, which achieves some timing accuracy at high resource cost; and poor timing-accuracy achieved where the application CPU controls the device remotely. The GPIOCP has efficient hardware cost compared to CPU approaches, with the additional benefits of total timing accuracy (CPU solutions do not provide this in general) and parallel control of many I/O devices.

I. INTRODUCTION

Real-time applications often need to access I/O devices at specific times in order to achieve accurate control over I/O - eg. the control of an automotive engine often requires I/O at accurate times in order to inject fuel at the optimal time [12]. Thus I/O operations may need to occur at an exact time (within small error margins), ie. be *timing-accurate* – it can be neither significantly late or early. In a single-core system, latencies caused by device drivers and application process scheduling make timing-accurate I/O control problematic – often leading a dedicated CPU for the I/O application, or that application being made the highest priority – neither solutions are scalable or offer good resource utilisation.

In a many-core systems, these issues are compounded. Whilst an application can invoke an I/O operation accurately via the interrupt of a high-resolution timer (e.g., the nanosecond timer provided by an RTOS[7] [8]), 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. Hence, it becomes difficult for an application to issue an I/O operation that will result in a low-latency timing-accurate device level I/O operation.

The specific architectural context of this paper is timing-accurate control of I/O devices that are off-chip, accessed via General-Purpose I/O (GPIO) pins, potentially using some bus protocol over those pins. This is in contrast to devices that are integrated within a chip (ie. a SoC or NoC chip) – such integrated devices have their latencies and timing-accuracy largely fixed by the existing architecture.

The contribution of this paper is a resource efficient programmable I/O controller, termed the *GPIO Command Processor (GPIOCP)*, that permits applications to instigate complex sequences of I/O operations at an exact time, so achieving timing-accuracy of a single clock cycle. This is achieved by loading application specific programs into the GPIOCP which generate a sequence of control signals over a set of General Purpose I/O (GPIO) pins, eg. for read / write. Applications then invoke a specific program at run-time by sending the GPIOCP commands such as *run command X at time Y* (where Y is some future time). This achieves cycle level timing-accuracy as the latencies of the bus or NoC are removed. 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 application issues a command such as *run command X at time Y 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 execution time.

The GPIOCP is a parallel multi-functional controller, supporting many different I/O devices in parallel – so can provide timing-accurate I/O for several applications simultaneously. The GPIOCP can also be reprogrammed at run-time to control an I/O device in a different way, or potentially allowing hot-swap of I/O devices (noting that the program needs to be moved to the controller, requiring that traffic to be included in any system timing analysis).

The paper is organized as follows: Section II presents our motivation; Section III describes the operation and implementation of the GPIOCP. Section IV evaluates the performance of GPIOCP. Section V presents related work,

TABLE I. Errors in Timing-accuracy of I/O Operations

CPU Index	E (unit: ns)			
	Minimum	Median	Mean	Maximum
Single-core: Architecture	2090.0	2090.0	2012.5	2100.0
NoC-based Multi-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

with conclusions offered in section VI.

II. Motivation

The error in the timing-accuracy of I/O operations is defined as the absolute time difference between the time at which I/O operation is required (T_x) and the actual time that the I/O operation (eg. read) actually occurs (T_y):

$$E = |T_x - T_y| \tag{1}$$

Thus a smaller E implies a higher timing-accuracy of the I/O operation. If E equals 0, this I/O operation occurs at the expected time – it is 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 that can be achieved in existing single and multi-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 are given in Table I (further experiment design is described in Section IV). It is clear that, even in a single-core system, the E is not close to a single cycle, with the timing error in many-core systems considerably worse due to the communication bottlenecks and contention of the system. 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 code), scheduling (amongst competing software tasks) and the Real-Time OS system calls would add considerably to the overall latencies in Table I.

III. GPIO Command Processor (GPIOCP)

The GPIO Command Processor (GPIOCP) proposed within this paper enables:

- *Cycle level timing-accuracy* – all I/O operations over the GPIO pins can be issued with an accuracy of a single cycle.
- *Programmabilty* – the GPIOCP holds small programs designed to control connected devices. They are loaded into GPIOCP memory by the application during system initialisation (so that loading does not interfere with normal execution and timeliness of the system). Importantly, commands within the program can be executed at exact times (cf. conventional CPU instructions).
- *Control of multiple connected devices in parallel* – multiple I/O devices connected to the GPIO pins can be controlled in parallel, whilst maintaining timing-accuracy of a single cycle.

Typical use of the GPIOCP within a NoC architecture is shown in Figure 1 – low level driving of the I/O device is performed by the GPIOCP rather than remotely by the application. At run-time an application can invoke a command program on the GPIOCP to achieve required I/O. This can execute immediately or at some time in the future; can be periodic; and can return data to the application CPU.

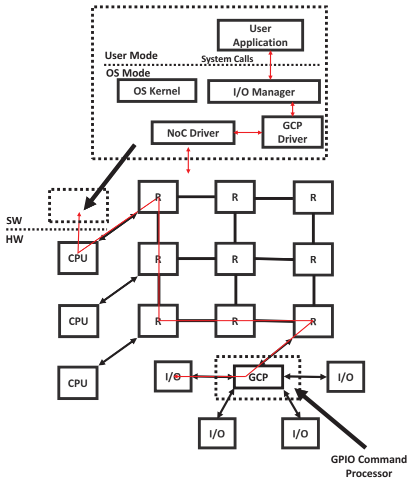

Fig. 1: GPIOCP Connected to a NoC
(R - Router / Arbiter; GCP - GPIOCP)

A. Architecture of GPIOCP

The architecture of the GPIOCP consists of the following main parts (see Figure 2):

- *Hardware Manager* – provides the interface to/from application CPUs via the NoC mesh;
- *Command Memory Controller* – manages internal GPIOCP memory to store/retrieve commands and data;
- *Command Queue* – manages GPIO CPUs which execute commands;
- *Synchronisation Processor* – provides synchronisation between the GPIO CPUs and external GPIO pins.

These architectural elements are detailed in the following subsections.

1) 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). The architecture of hardware manager is shown in Figure 3, with the left part allocating incoming requests; the right part takes ending back data from GPIOCP to CPUs.

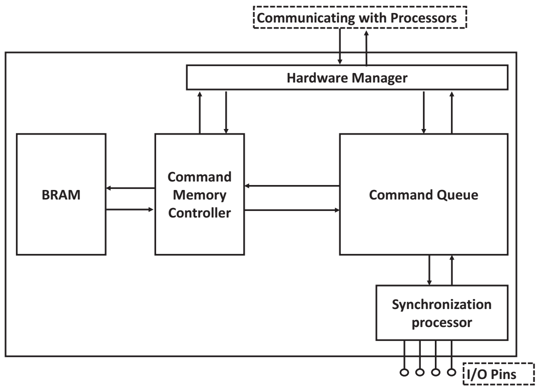
Communicating with Processors
Hardware Manager
BRAM
Command Memory Controller
Command Queue
Synchronization processor
I/O Pins

Fig. 2: Architecture of GPIOCP

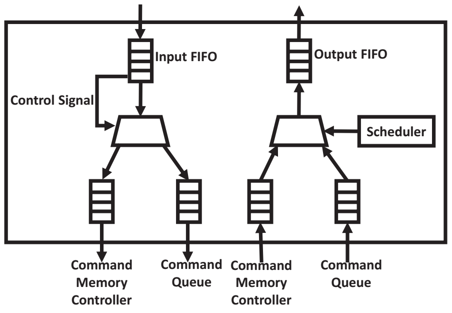

Fig. 3: Architecture of Hardware Manager

The GPIOCP receives two forms of request:

- Type 1: creating a new GPIO command – allocated to the output FIFO which is connected to the Command Memory Controller;
- Type 2: invoking a ready-built GPIO command Type 1 requests – allocated to the output FIFO connected to the Command Queue.

Similarly, the right part of hardware manager is mainly comprised by two input FIFOs, a multiplexer, a output FIFO and a scheduler. The two input FIFOs are respectively connected to the Command Memory Controller and the Command Queue, in order to receive the data to be sent back to the CPUs. The scheduler controls the multiplexer to choose which input FIFO can transmit data into the output FIFO (if both input FIFOs are not empty the FIFOs are chosen in a round-robin manner).

2) Command Memory Controller: Stores a new GPIO command into the BRAM (FPGA Block RAM); and accesses an existing GPIO command for execution by a GPIO CPU (within the Command Queue). The architecture of command memory controller is shown in Figure 4. Memory is divided into pages, with one GPIOCP command per page; each page containing command identifier (integer 4 bytes), command length (4 bytes) and the commands themselves (GPIO commands discussed in section III-B. Eg. a 32KB BRAM can be split into 128 pages, each able to store identifier, length and upto 62 commands (each of 32 bits). Finally, the dual-ported nature of BRAM is exploited to provide separate interfaces to the Command Memory Controller and Command Queue to improve performance.

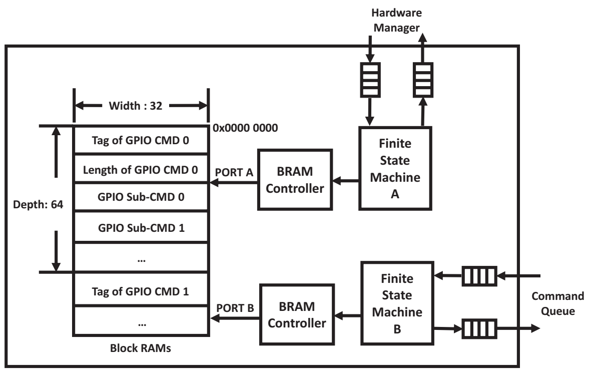

Fig. 4: Architecture of Command Memory Controller

3) Command Queue: Allocates GPIOCP commands to GPIO CPUs for execution (architecture is shown in Figure 5). The Command Translation Module requests commands from the internal memory via the Command Memory Controller, sending the commands to a GPIO CPU for execution.

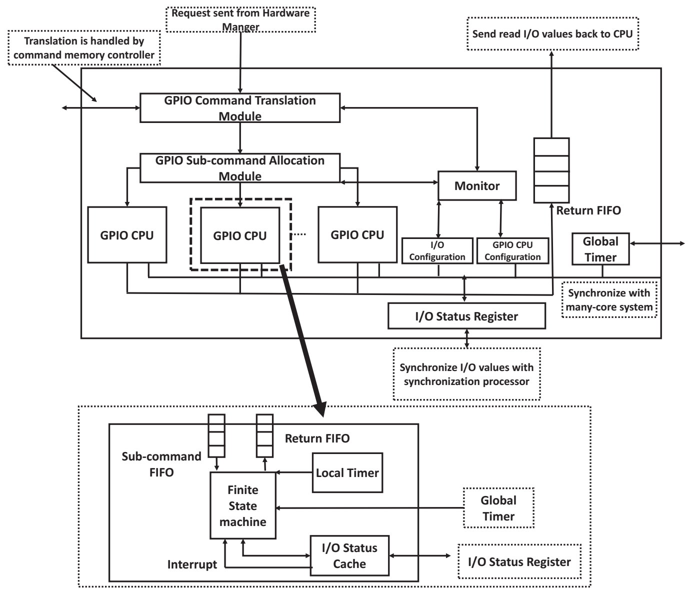
Request sent from Hardware Manger
Translation is handled by command memory controller
Send read I/O values back to CPU
GPIO Command Translation Module
GPIO Sub-command Allocation Module
Monitor
Return FIFO
GPIO CPU
GPIO CPU
GPIO CPU
I/O Configuration
GPIO CPU Configuration
Global Timer
Synchronize with many-core system
I/O Status Register
Synchronize I/O values with synchronization processor
Sub-command FIFO
Return FIFO
Local Timer
Finite State machine
Global Timer
I/O Status Cache
I/O Status Register
Interrupt

Fig. 5: Architecture of GPIO Command Queue

Each GPIO CPU is a simple finite state machine, with guaranteed execution time so achieving timing accuracy. Each GPIO CPU has a dedicated I/O status cache (4 bytes), which only stores the status of I/O pins belonged to this GPIO CPU. This dedicated cache synchronises its I/O status with a global shared register at a fixed frequency. The status of all I/O pins are stored in this shared register.

A Global timer is connected to all GPIO CPUs so providing time synchronisation – eg if several GPIO CPUs all need to execute a command at time t, the global timer enables this.

4) Synchronisation Processor: is responsible for synchronising the values of I/O pins, which may be written by different GPIO CPUs and I/O devices. The architecture of the synchronisation processor is shown in Figure 6.

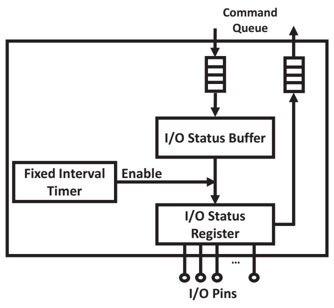

Fig. 6: Architecture of Synchronization Processor

The modified values of I/O pins received from the input FIFO are stored in the I/O status buffer, rather than being updated immediately. Fixed interval timer enables the synchronisation between I/O status buffer and I/O status register every 5 clock cycles. Once the value of I/O status register changed, the changed value will be sent back to the command queue via the output FIFO.

B. GPIOCP Commands

A set of composable I/O control instructions, termed *sub-commands* are provided, consisting of I/O control sub-commands, timing control sub-commands and a loop control sub-command. Application specific programs can thus be built from the sub-commands and stored in a page in GPIOCP internal memory (see section III-A2).

GPIO writing sub-commands supported are:

1) Execute the next write sub-command at a specific time;
2) Set a specific I/O pin to high/low;
3) Set a group of I/O pins to specific values;
4) Delay for a specified time (in clock cycles).

GPIO read sub-commands supported are:

1) Execute the next reading sub-command at a specific time;
2) Read the value(s) of an specified I/O pin(s);
3) Read the value(s) of an specified I/O pin(s) while a predefined I/O pin triggered high/low.

The GPIO loop control sub-command supported is:

1) Goto to a specified GPIO sub-command.

The timing control sub-commands (1, 4, 5 and 8 above) provides timing constraints for I/O control sub-commands (2, 3, 6 and 7 above) which guarantees an I/O device can be operated accurately to a clock cycle. Running a sub-command 4 or sub-command 8 may take more than one clock cycle, but can be bounded by users. Any other seven sub-commands always use exactly 1 clock cycle. Therefore, the running time of GPIO commands is predicable, as they are comprised by GPIO sub-commands.

We define a GPIO sub-command as a 32 bit instruction, defined as follows (see Figure 7):

- Bit 0 - Bit 15: Operation parameters of GPIO sub-command, which provided different information for different sub-commands. Specifically, in sub-commands 1, 4, 5 and 8, information regards timing; in sub-commands 2, 3, 6 and 7, information regards the specific I/O pins; in sub-command 9, information includes the index of the sub-command to goto.
- Bit 16: The function type of GPIO sub-command - read/write; '1' stands for writing function and '0' represents reading function;
- Bit 24 - Bit 31: The index of GPIO sub-command.

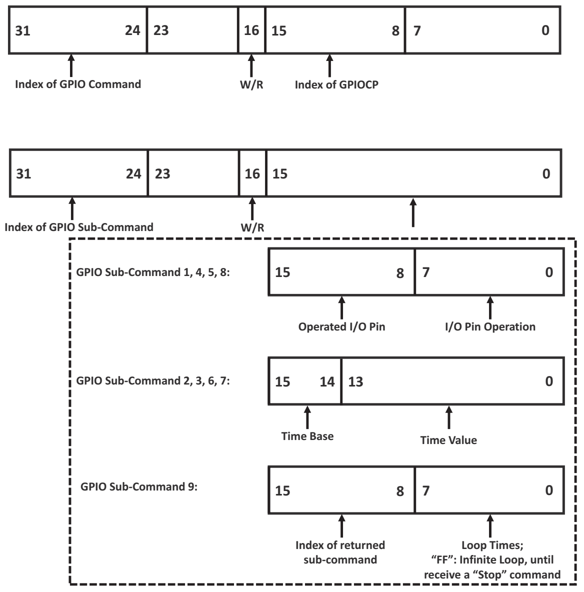
31
24
23
16
15
8
7
0
Index of GPIO Command
W/R
Index of GPIOCP
Index of GPIO Sub-Command
GPIO Sub-Command 1, 4, 5, 8:
Operated I/O Pin
I/O Pin Operation
GPIO Sub-Command 2, 3, 6, 7:
14
13
Time Base
Time Value
GPIO Sub-Command 9:
Index of returned sub-command
Loop Times; "FF": Infinite Loop, until receive a "Stop" command

Fig. 7: Format of GPIO Subcommand

C. Example: PWM

To achieve a PWM signal on I/O pin #6 with 50% duty cycle the following sub-commands can be used:

1) Execute the next sub-command at time 200ns (ie. delay until start of PWM signal): $0x010100C8$
2) Pull I/O pin #6 high: $0x02010601$
3) Wait for 20ms: $0x04018014$
4) Pull I/O pin #6 low: $0x02010601$
5) Wait for 20ms: $0x04018014$
6) Go back to 2 and infinite loop: $0x090102FF$

D. Invoking a GPIOCP Command

Requesting the GPIOCP to execute a command stored in a GPIOCP internal memory page requires the unique index of that command to be sent from the user application CPU to the GPIOCP. The format of the request is given in Figure 8:

- Bit 8 - Bit 15: The index of GPIO CPU which will execute this GPIO command.
- Bit 16: Read (0) / write (1)
- Bit 24 - Bit 31: The index of GPIO command;

For example, to execute command with index #2 on GPIOCP #3 would be: $0x02010300$

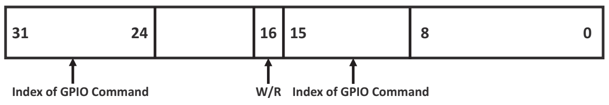

Fig. 8: Format of GPIO Command

IV. Evaluation

The GPIOCP was implemented using Bluespec[1] and synthesised for the Xilinx VC709 development board [6](further implementation details a technical report[1] The GPIOCP was connected to a 4*3 size 2D mesh type open source NoC[2] containing 9 Microblaze CPUs [2] running the uCosII RTOS (v1.41)[5]. The architecture is shown in Figure 9. To enable comparison, a similar hardware architecture was built, without the GPIOCP – note that this architecture requires I/O operations requested by CPUs to pass through the mesh to the GPIO rather than being controlled by a GPIOCP. Both architectures run at 100 MHz.

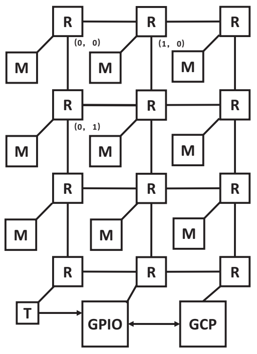

Fig. 9: Experimental Platform

A. Accuracy of I/O Operations

When CPUs are required to access and read the GPIO at a specific time, then for a non-GPIOCP architecture the CPU has to instigate the I/O operation, for the GPIOCP architecture, this can be delegated to the 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 II, showing that the latencies and variance for the non-GPIOCP architecture are significant (errors calculated according to equation 1); in contrast the GPIO architecture is accurate at the cycle level.

B. Hardware Overhead

The resource efficiency of the GPCPIO when implemented on the Xilinx VC709 FPGA development board is shown in Table III. The GPIOCPU is compared with FPGA (ie. softcore) implementations of a CPU and an SPI controller. The former enables comparison against approaches that use a dedicated CPU as an I/O controllers, and a dedicated core for the specific I/O device. The GPIOCP utilises significantly less hardware than the CPU, but more than the dedicated controller. Thus a parallel reprogrammable I/O controller can be achieved in less resource than a CPU, and offers true timing accuracy across multiple GPIO connected external devices; but more resources than a dedicated controller that is useful for only one I/O device. Note that in this comparison, the GPIOCP is configured with 2 GPIO CPUs and 8 KB storage units.

C. Case Study

The effectiveness of the GPIOCP approach is illustrated by considering the control of a 3D printer which requires X and Y co-ordinates (via multiple motors) updating at a 5Mhz frequency[3]. The printer is required to print out the following patterns:

- Pattern A: $f_A(x) = 8$
- Pattern B: $f_B(x) = x$
- Pattern C: $f_C(x) = 128/x + sin(x) - x * cos(x)$
- Pattern D: $f_D(x) = 80 * (sin(x))^5$
- Pattern E: $f_E(x) = 128/x$
- Pattern F: $f_F(x) = \sqrt{((100^2 - (x-100)^2) + 108)}$

Control of the motors to draw the above patterns is required at a frequency of 5 MHz. No time was measured for the calculation of values by CPU, these were pre-calculated.

This was implemented with and without GPIOCP support. The non-GPIOCP implementation used a single CPU accessing the GPIO directly; the GPIOCP implementation placed the entire control for generating the pattern in the GPIOCP as a single program.

To evaluate the patterns generated by the two implementations, the GPIO pins were monitored (by as separate core) that compared the generated values against expected values (stored as a pre-calculated table within the monitor). Table IV shows the miss-rate (values that were not written at the correct frequency) and variance (RMS error of output value compared with expected value at that time) – both are expressions of timing accuracy defined by equation (1).

The non-GPIOCP implementation has high miss rate, even for simple patterns (even the constant, pattern A), showing the latency of controlling GPIO from a CPU. Where the pattern is simple, variance is low for the non-GPIOCP showing that if the pattern value does not change quickly, then outputting the wrong value (for the time) has less effect. However where the pattern varies more over time, variance increases. The GPIOCP implementation has zero miss rate and variance for all patterns – hence is timing-accurate.

V. Related Work

Related approaches for accurate I/O over a multi-processor NoC architecture can be divided into those that utilise a standard architecture and those that introduce a

[1]Citation to technical report removed for double blind review.

[2]Citation for NoC removed for double blind review.

[3]This is in excess of usual 3D printer motor control frequencies but illustrates the effectiveness of the GPIOCP approach in that higher control frequencies are possible – offering potentially more accurate printing.

	Non-GPIOCP								GPIOCP							
CPU Index	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
(0,1)	3000.0	3000.0	3005.8	3020.0	300	300	301	302	0.0	0.0	0.0	0.0	0	0	0	0
(0,2)	2790.0	2790.0	2795.8	2810.0	279	279	280	281	0.0	0.0	0.0	0.0	0	0	0	0
(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
(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
(1,2)	2860.0	2880.0	2899.4	2940.0	286	288	290	294	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	2600	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	00.0	0.0	0.0	0.0	0	0	0	0
(2,2)	2860.0	2930.0	2902.2	2950.0	286	293	290	295	0.0	0.0	0.0	0.0	0	0	0	0

TABLE II. I/O Operation Timing Variance

Microblaze Softcore CPU			SPI Softcore			GPIOCP 2 GPIO CPUs		
L	R	B	L	R	B	L	R	B
1170	1568	8	326	501	0	886	615	4

TABLE III. FPGA Hardware Usage:
(L - Look Up Tables; R - Registers; B - Block RAMs)

Pattern Index	Miss Rate		Variance	
	GPIOCP	non-GPIOCP	GPIOCP	non-GPIOCP
1	0.00%	81.67%	0	0.0000
2	0.00%	83.33%	0	2.8735
3	0.00%	91.67%	0	18.6886
4	0.00%	88.33%	0	25.7971
5	0.00%	86.67%	0	18.8541
6	0.00%	85.00%	0	3.4698

TABLE IV. Miss Rate and Variance

dedicated unit for handling I/O. Technologies such as Programmable Logic Controllers (PLCs) and associated I/O controllers are out-of-scope.

Typical NoC based architectures that have been implemented in silicon contain integrated devices connected to the edge of the mesh, eg. Tilera's TILE64[11] and Kalray's MPPA-256[10], as well as GPIO (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 device (including GPIO) – hence significant latencies will occur between I/O command instigation and actual I/O occurring, which detracts from timing-accuracy. 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 (although RTEMS[3] can also be used), hence timing accurate control of many external devices connected to the GPIO is not possible – also the approach is not resource efficient as a CPU is required for I/O control.

Other approaches include the TI Programmable Real-Time Unit (PRU)[9] and the Freescale Time Processor Unit (TPU)[4] – these are programmable controllers that could be connected to a NoC mesh for GPIO control. The PRU contains two 32-bit RISC cores and ready built I/O controllers that are capable of real-time I/O, but exact timing of I/O operations (eg. 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.

VI. Conclusion

In this paper, we presented the concept of a programmable I/O controller (GPIOCP) with a clock cycle level granularity. It enables application specific I/O control protocols, as well as operating multiple I/O devices in parallel with clock cycle level accuracy, all with timing accuracy appropriate to demanding real-time systems.

Evaluation reveals that GPIOCP can handle multiple I/O operations with clock cycle accuracy, in many cases totally timing accurate. However, the hardware overhead was 50% less compared to a testbed with the same functionality build using a minimalistic version of the soft core microprocessor; Microblaze instead of GPIOCP.

References

[1] Bluespec Inc. Bluespec System Verilog (BSV). http://www.bluespec.com/products/. Accessed 27/9/15.
[2] Miroblaze User Manual. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf. Accessed August 27, 2016.
[3] RTEMS. https://www.rtems.org/. Accessed 26/8/15.
[4] TPU User Manual. http://www.nxp.com/files/microcontrollers/doc/ref_manual/TPURM.pdf?fasp=1&WT_TYPE=Reference%20Manuals&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation&fileExt=.pdf. Accessed 27/8/16.
[5] uCos. https://www.micrium.com/rtos/kernels/. Accessed 27/9/15.
[6] VC709. https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html. Accessed 27/8/16.
[7] VxWorks. http://windriver.com/products/vxworks/. Accessed 27/8/16.
[8] VxWorks Timer Library. http://www.vxdev.com/docs/vx55man/vxworks/ref/timerLib.html. Accessed 27/8/16.
[9] R. Birkett. Enhancing Real-time Capabilities with the PRU. In *Embedded Linux Conference*, 2015.
[10] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.
[11] D. Jagtap, K. Bahulkar, D. Ponomarev, and N. Abu-Ghazaleh. Characterizing and Understanding PDes Behavior on Tilera Architecture. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 53–62. IEEE Computer Society, 2012.
[12] J. Mossinger. Software in Automotive Systems. *IEEE software*, 27(2):92, 2010.