
Proceedings Paper:
Zhao, Shuai, Jiang, Zhe, Dai, Xiaotian et al. (Accepted: 2020) Timing-accurate general-purpose I/O for multi- and many-core systems: scheduling and hardware support. In: Design Automation Conference (DAC). (In Press)

Timing-Accurate General-Purpose I/O for Multi- and Many-Core Systems: Scheduling and Hardware Support

Shuai Zhao, Zhe Jiang, Xiaotian Dai, Iain Bate, Ibrahim Habli, Wanli Chang
Department of Computer Science, University of York, UK
{shuai.zhao, zhe.jiang, xiaotian.dai, iain.bate, ibrahim.habli, wanli.chang}@york.ac.uk

***Abstract*—General-purpose I/O widely exists on multi- and many-core systems. For real-time applications, I/O operations are often required to be timing-predictable, i.e., bounded in the worst case, and timing-accurate, i.e., occur at (or near) an exact desired time instant. Unfortunately, both timing requirements of I/O operations are hard to achieve from the system level, especially for many-core architectures, due to various latency and contention factors presented in the path of instigating an I/O request. This paper considers a dedicated I/O co-processing unit, and proposes two scheduling methods, with the necessary hardware support implemented. It is the first work that guarantees timing predictability and maximises timing accuracy of I/O tasks in the multi- and many-core systems.**

I. Introduction

The increasing demand of computation power in emerging real-time application scenarios (e.g., automotive, aerospace and robotics) has necessitated the transition from signal-core to multi- and many-core systems, where general-purpose I/O (GPIO) is usually provided for connection to external devices. The I/O operations are often required to be both timing-predictable — with an analytical bound for the worst case, and timing-accurate — getting executed at exact time instants (or at least within a small time range) [1], [2]. For instance, in an autonomous control system, the engine requires a periodic I/O to occur at accurate time instants, for the optimal fuel injection [3]. The timing accuracy of an I/O operation can be quantified by the absolute value of the difference between the time instant when the I/O operation is ideally expected to occur and the time instant when it actually occurs [2].

It is very challenging to provide timing predictability and accuracy for I/O operations from the system level, especially for the many-core architectures, such as Network-on-Chip (NoC). There are various latency, interference and resource contention presented in the path of instigating an I/O request, from the application to the underlying hardware [4]. For instance, the on-chip communication latency for sending an I/O request from a CPU to an I/O controller can be substantial due to the arbitration of the on-chip data flows across the communication mesh [5]. In addition, the contention (as well as interference) from the application, the underlying (real-time) operating system, software drivers and devices connected to the GPIO can vary significantly. Furthermore, potential competition between I/O requests for accessing the same I/O device can elevate the difficulty for satisfying these timing requirements towards I/O operations [6].

Related work: Research efforts aiming to achieve real-time I/O operations in many-core systems have been conducted in various perspectives, from the system level to the dedicated I/O controllers [2], [7]–[11]. At the system level, research mainly focuses on I/O scheduling, I/O contention-aware task mapping and communication latency bounding, for predictable I/O operations with minimised latency [7]–[9], potentially applying certain resource sharing protocol to manage the contention of accessing the same I/O device. Unfortunately, as discussed above, it is difficult to provide timing predictability from the system level, and even more so for timing accuracy. The reason is the occurrence of an I/O operation depends on the actual execution on the underlying hardware, especially in the case where intensive I/O requests compete for accessing the same I/O device.

At the hardware level, programmable I/O controllers have been developed and manufactured by various semiconductor vendors. Among these products, TI's Programmable Real-Time Unit (PRU) [10] and NXP's Time Processor Unit (TPU) [11] are particularly designed for the real-time context. However, timing accuracy of I/O operations is not possible in either product as I/O is instigated by a remote CPU. A GPIO command processor (GPIOCP) is proposed in [2] for handling I/O requests via a dedicated co-processor, which could significantly reduce the communication latency by pre-loading timed (e.g., periodic) I/O operations into GPIOCP before run-time. GPIOCP is a step towards timing-accurate I/O control via specifying the exact start time of each I/O command. However, as GPIOCP relies solely on FIFO queues for ordering the execution of I/O operations, its performance largely depends on the arrival order of the I/O requests. Therefore, GPIOCP cannot guarantee either of the timing requirements.

Main contributions: This paper focuses on dedicated I/O co-processing units and proposes a scheduling model for timed I/O operations. Two scheduling methods are presented to provide predictability guarantee and maximised timing accuracy of timed I/O operations for hard real-time systems. The first method relies on task allocation heuristics and aims to maximise the number of I/O operations with exact timing-accurate control. The second method maximises both the number of exact timing-accurate I/O operations and the overall I/O performance of the system in terms of timing accuracy (i.e., generally the I/O operations are close to the desired occurring time instants), based on a genetic algorithm solver. Both methods guarantee timing bounds to be satisfied, hence predictability. Necessary hardware support for realising the proposed scheduling model in I/O controllers is presented, with a reference implemented provided. Experiments are conducted to evaluate the schedulability, timing accuracy, and hardware resource efficiency.

Organisation of this paper: Section II describes the system and task model for scheduling timed I/O operations. Section III details the proposed scheduling methods for dedicated I/O controllers. Section IV presents the hardware requirements to enable the proposed I/O scheduling and provides a reference implementation. Experimental results are given in Section V. Section VI concludes the paper.

II. System and Task Model

The system and task model is derived from a typical I/O system in [2], where the system contains a set of timed I/O commands that are pre-loaded into the I/O controller before run-time. Each I/O command is executed on its designated I/O device, based on a pre-defined time instance with a time interval i.e. user can request the controller to *execute I/O command X on device D at time Y, and repeats Z times with an interval T*. As described in [2], each

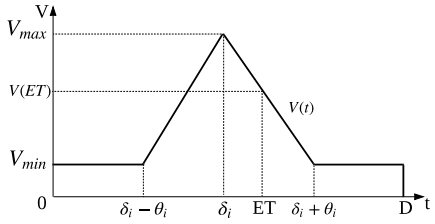


Fig. 1: An example quality curve of an I/O task τ_i

application processor is associated with one GPIOCP instance, which handles the I/O requests issued from that processor. During run-time, a fired I/O request is firstly queued into a FIFO queue, and is then executed with its requesting I/O device when it becomes the head of the queue. However, as stated in Section I, such an approach is insufficient to provide guarantee to either schedulability or timing accuracy, because I/O requests are executed by their arrival order, regardless of their ideal start times and deadlines.

The timed I/O requests are modelled as a set of periodic I/O tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. Each I/O task (i.e., a periodic I/O request) τ_i is defined by a 6-tuple $\{C_i, T_i, D_i, P_i, \delta_i, \theta_i\}$, indicating its worst-case computation time for operating the I/O device (C_i), period (T_i), implicit deadline ($D_i = T_i$) and a deadline-monotonic priority P_i ($D_1 > D_2$ so that $P_1 < P_2$), respectively. Notation δ_i indicates the relative ideal time that the I/O task is expected to occur, to achieve the exact timing accurate I/O control. During execution, each task can give raise to a set of jobs $\Lambda_i = \{\lambda_i^0, \lambda_i^1, ..., \lambda_i^m\}$ in a hyper-period. For each job λ_i^j, its ideal start time is given by $T_i \times j + \delta_i$, and hence, the timing boundary for its quality curve can be obtained as well. κ_i^j denotes the actual start time of λ_i^j, which is decided by the scheduler. All I/O jobs are executed in a non-preemptable fashion.

In contrast to [2], a more realistic and flexible timing-accuracy model is defined. Besides the exact timing-accurate control, i.e., starting at δ_i with the maximum quality V_{max}, we allow a limited timing margin θ_i, where I/O operations are also beneficial if they are executed within the given boundary, but subject to certain quality decay. The timing boundary is defined as $[\delta_i - \theta_i, \delta + \theta_i]$ for τ_i in each release. If the task is not executed within the timing boundary, a minimal quality V_{min}[1] can be obtained as long as the I/O operation is finished before its deadline, i.e., being schedulable. Figure 1 gives an example quality curve of τ_i, where it is released at time AT, and executed at time ET with a quality $V(ET)$. Note that the exact quality curve of an I/O job is highly application-dependent and could lead to varying performance. The focus of this work is not to investigate the performance impact from different value curves. We assume a common linear curve and use it to evaluate the existing and proposed solutions. With the quality curve, the I/O scheduling can be formulated as a typical value-driven optimisation problem [12]–[14]. However, previous works cannot be directly applied as they do not consider either the ideal start time (i.e., the notion of timing accuracy) or the hard real-time constraints.

III. Timing-Accurate I/O Scheduling Methods

With the system and task model established, scheduling methods can be applied to achieve both schedulability and timing accuracy. However, matured real-time scheduling techniques (e.g., FPS and the method in [15]) are not appropriate to schedule I/O operations with timing-accuracy requirements. For instance, under intensive I/O requests, exact timing-accurate control (i.e., executing at the desired time instant) may not be possible for each I/O request due to the overlapped ideal execution time. In such cases, these scheduling methods do not maximise I/O performance of the system, as they solely focus on guaranteeing the system schedulability. A similar system model is considered in [16], which further incorporates the concept of gravity to guarantee the utility of tasks with higher weight. It optimises neither the number of jobs with exact timing-accurate control nor the overall timing-accurate performance.

In this section, two scheduling methods are proposed to provide solutions that can guarantee both the real-time requirements and timing accuracy of I/O requests. More importantly, a key objective of the proposed schedule is to handle the situation where exact timing accuracy for all I/O operations is not possible, i.e., the intensive I/O requests case. Under this case, the methods aim at maximising the overall timing accuracy for the system, base on the I/O performance metrics defined later in this section. In contrast to [2], we assume a global I/O controller with a fully-partitioned I/O scheduling model, in which each processor in the controller is associated to one I/O device. Accordingly, the pre-loaded I/O tasks are allocated to each partition based on the I/O devices they access. By doing so, we avoid potential contentions between the I/O requests on different processors accessing the same I/O device. A schedule is then produced for the jobs released by the I/O tasks in each partition.

A. Heuristic-based I/O scheduling

The first scheduling method is based on task allocation heuristic and aims at maximising the percentage of exact time-accurate I/O jobs i.e., $\Psi = |E|/|\lambda|$, in which E denotes the number of exact time-accurate I/O jobs (Equation (1)), λ gives the set of input jobs that access one I/O device and $|\cdot|$ returns the size of a given set.

$$E = \left\{ \lambda_i^j \middle| T_i \times j + \delta_i - \kappa_i^j = 0, \forall \lambda_i^j \in \lambda \right\} \quad (1)$$

For each partition, the algorithm takes all jobs in one hyper-period as the input, and returns an explicit schedule by computing the actual start time κ_i^j for each job, along with the final Ψ. To facilitate the scheduling, each job λ_i^j is assigned with a penalty weight ψ_i^j, indicating the number of jobs that cannot be exact timing accurate if λ_i^j is executed at the ideal start time instant. Unlike the traditional FPS, this algorithm allows: i) priorities to be overruled, and ii) tasks to be delayed in certain situations (even if the I/O device is currently idle), so Ψ can be maximised.

Algorithm 1 outlines the scheduling method. Essentially, the algorithm examines the executions of all I/O jobs in their ideal cases and identifies potential execution conflicts between the jobs by forming *dependency graphs* (phase one). Then in phase two, the algorithm decomposes the *dependency graphs* to resolve the execution conflicts, via sacrificing the jobs that can affect the exact timing accuracy of the most jobs i.e., with a high ψ_i^j. By doing so, the number of jobs that could achieve exact timing accuracy can be maximised. Finally in the third phase, the sacrificed jobs are allocated to the free slots within their release periods to guarantee the system schedulability, using the Least Contention and Capacity Decreasing (LCC-D) allocation proposed later in this section.

Dependency graph formation: To identify and resolve the potential conflicts between jobs in their ideal executions (i.e., starting at their δ_i^j), *dependency graphs* are introduced to depict the relation of the conflicting jobs, denoted as $G = \{G_1, G_2, ..., G_n\}$ (Line 1 in Algorithm 1). A dependency graph contains a set of jobs that have continuously execution overlaps in their ideal execution case.

[1] V_{min} can be set to zero in systems where I/O operations are not beneficial if being executed outside $[\delta_i - \theta_i, \delta_i + \theta_i]$. For safety-critical systems, a large penalty value (e.g., -1000) could be applied.

Algorithm 1: Job-level I/O scheduling for maximising Ψ

```
1  create dependency graphs G based on the input jobs;
2  λ* = λ¬ = ∅;
3  for each G_n ∈ G do
4      while |G_n| > 1 do
5          take λ_i^j with the highest ψ_i^j;
6          λ¬ = λ¬ ∪ {λ_i^j};
7      end
8      λ* = λ* ∪ G_n;
9  end
10 identify free slots s_1..s_n between each job in λ*;
11 for each λ_i^j ∈ λ¬, largest P_i first do
12     if ∃s_n in λ_i^j's release period that s_n ≥ C_i then
13         allocate λ_j^i to s_n with least contention and capability;
14     else
15         if ∃s_1..s_n that Σ_{s_1..s_n} ≥ C_i then
16             allocate τ_i^j by shifting least tasks in λ*;
17             update λ*;
18         else
19             return {infeasible, 0}
20         end
21     end
22 end
23 return {schedule, |λ*|}
```

Figure 2 provides an example for grouping nine jobs (Figure 2(a)) into four dependency graphs (Figure 2(b)), where the up-arrows denotes their δ_i^j. We first note that 'Job 1' is grouped by itself into G_1 as it does not conflict with other tasks. Then, jobs 2 and 3 are linked as G_2 due their overlapped executions. Similarly, Jobs 3, 4 and 5 are grouped as G_3, but note, jobs 4 and 6 are not linked as their executions do not overlap. The last three tasks are linked to each other as G_4 due to their mutual execution conflicts. With the graphs established, it becomes clear that the goal of the algorithm is to completely decompose each graph until all jobs are discrete to each other. In addition, the penalty weight ψ_i^j can also be obtained directly, where it equals to the number of lines linked to λ_i^j (e.g., 'Job 5' has a penalty weight of 2). Note, both G and ψ_i^j changes dynamically during the graph decomposition phase.

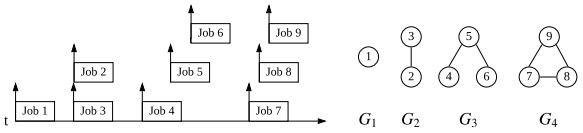
Decomposing graphs: With the dependency graphs, the algorithm aims to remove the links between jobs i.e., to eliminate overlapped job executions (Line 2-9). For each graph, the algorithm removes the job with the highest ψ i.e., the one that affects the most jobs if it is executed at its ideal start time (Line 5). The algorithm breaks the tie by P_i, where the job with the lowest priority is taken. Intuitively, jobs with a lower P_i (i.e., with a wider release period) has a wider range of free time slots to be allocated. The removed jobs are added into set $\lambda^\neg$, which will be allocated later on the in the final phase. If a graph contains two discrete jobs during the graph decomposition phase i.e., with all conflict jobs removed, it splits into two independent graphs. For instance, G_3 will split into two graphs with 'Job 5' removed. This process repeats until the graph is completely discomposed i.e., with only one job remain, denoted as $|G_n|= 1$. At last, the remaining job in each graph is added into λ^* (Line 8), which contain jobs that could be exact timing accurate.

LCC-D allocation: With λ^* and $\lambda^\neg$ obtained, the algorithm aims at guaranteeing the schedulability of the jobs in $\lambda^\neg$ i.e., each job must be finished before their deadline. Assuming all jobs in λ^* are executed at their δ_i^j, it forms a allocation problem that is similar to the classical bin-packing problem i.e., allocate each job λ_i^j in $\lambda^\neg$ into a set of free slots, but must within its release period $[T_i \times j, T_i \times j + D_i]$. To achieve this, the Least Contention and Capacity Decreasing (LCC-D) task allocation is proposed. First, the algorithm iterates through all jobs in $\lambda^\neg$ and identifies the time slots in their release periods. Then, for each λ_i^j, highest P_i first, this allocation method considers two cases:

1) There exist one or more slots (denoted by s) in $[T_i \times j, T_i \times j + D_i]$ that can allocate λ_i^j directly (Line 12).
2) Neither slot can allocate λ_i^j directly, but the sum of their capacity is equal to or higher than C_i (Line 15).

For case 1), LCC-D allocates λ_i^j to the slot that can be used to allocate the least number of jobs, for minimised contention. If there exist two or more such slots, the slot that provides the least capability will be used, and hence the name of the method. The rationale is similar to the Best-Fit algorithm, which aims at maximising number of fitted tasks. For case 2), the algorithm iterates through each empty slot by time, and selects the slots that can fit λ_i^j with the least number of timing accurate jobs in between. Then, λ_i^j is allocated in these slot, by shifting all allocated jobs between the selected time slots.

If it is neither case, we acknowledge the possibility that a feasible allocation could still be achievable by replacing certain allocated task(s), and subsequently, to allocate the replaced tasks in other feasible slots. However, we decide not to go further as this could cause the algorithm not to terminate. Thus, the algorithm ends in this situation (Line-19) with no feasible schedule being found. Note that essentially, the complexity of this I/O scheduling problem is identical to the NP-hard bin-packing problem, where no optimal solution can be achieved in pseudo-polynomial time.



(a) example jobs (b) dependency graphs

Fig. 2: An example illustrating jobs to dependency graphs

B. Multi-objective GA-based searching

Although Algorithm 1 provides a static scheduling solution that can maximise Ψ, this is often achieved by sacrificing the timing accuracy of other jobs, where the I/O performance of these jobs are highly likely to be minimal i.e., with a quality of V_{min}. In addition, the algorithm relies on heuristic-based task allocation and cannot consider all corner cases. To provide more balanced scheduling solutions, an multi-objective Genetic Algorithm (GA)-based I/O scheduling method is proposed, which aims at improving both Ψ and the overall I/O performance of the system, denoted by Υ in Equation (2), which gives the sum of the normalised quality of all jobs in λ. Function $V_i^j(t)$ returns the quality of job λ_i^j that is executed at time instant t, and follows the value curve given in Figure 1.

$$\Upsilon = \sum_{\lambda_i^j \in \lambda} V_i^j(\kappa_i^j) \;/\; \sum_{\lambda_i^j \in \lambda} V_i^j(\delta_i^j) \tag{2}$$

With both objective functions defined, the GA-based I/O scheduling problem can be formalised, as follows.

$$\begin{array}{ll} \text{Given:} & \text{a set of input jobs } \lambda \\ \text{Maximise} & \{\Psi, \Upsilon\} \\ \text{On} & \{\kappa_i^j \mid \forall \lambda_i^j \in \lambda\} \end{array} \tag{3}$$

The tuning parameter κ_i^j for each input job is encoded as one sequence to form the chromosomes of each individual solution. In addition, two constraints are derived to regulate the GA search, to guarantee schedulability and correct execution behaviours. First, each job should be executed within its release period, and must be finished before its deadline, which leads to the following constraint for schedulability concern.

Constraint 1. $\forall \lambda_i^j \in \lambda : T_i \times j \leq \kappa_i^j \leq T_i \times j + D_i - C_i$

Then, the execution of the jobs should not be conflicted with each other, as defined by the following constraint. For a given job λ_i^j and another job λ_x^q released by a different task, λ_i^j can execute either before or after the execution of λ_x^q.

Constraint 2. $\forall \lambda_i^j, \lambda_x^q \in \lambda, x \neq i : \kappa_i^j \leq \kappa_x^q - C_i$ or $\kappa_i^j \geq \kappa_x^q + C_x$

Because of this constraint, the typical Mixed Integer Linear Programming is difficult to apply as the solution space is not continuous. Constraint 2 can be further refined by identifying the exact jobs that can be released by other tasks during the release period of λ_i^j. For jobs of another task τ_x, the index of its first job released in the release period of λ_i^j can be bounded by:

$$\alpha_{x,\lambda_i^j} = \max\{\left\lfloor \frac{T_i \times j}{T_x} \right\rfloor - 1, 0\} \tag{4}$$

and the index of its last job in the given release period can be safely bounded as:

$$\beta_{x,\lambda_i^j} = \left\lceil \frac{T_i \times j + D_i}{T_x} \right\rceil \tag{5}$$

Accordingly, constraint 2 can be further refined as:

Constraint 2^***.** $\forall \lambda_i^j, \lambda_x^q, i \neq x, \alpha_{x,\lambda_i^j} \leq q \leq \beta_{x,\lambda_i^j} : \kappa_i^j + C_i \leq \kappa_x^q$ **or** $\kappa_i^j \geq \kappa_x^q + C_x$

With above constraints, κ_i^j for each job can be bounded effectively to provide correct execution behaviours and to guaranteed schedulability of the I/O jobs. During the GA search, constraint 1 is ensured during population initialisation and gene mutation, where κ_i^j of each job is generated randomly in $[T_i \times j + \delta_i - \theta_i, T_i \times j + \delta_i + \theta_i]$ i.e., the timing boundary that λ_i^j has a value above the minimum. Constraint 2 is ensured by a reconfiguration function (applied before the objective functions), which resolves potential execution conflicts[2] while preserving the resulting execution order. In addition, the reconfiguration function examines each job and tries to execute them at their ideal starting times (if possible). The weights of both objectives for all individuals is spread uniformly from $[1.0, 0]$ to $[0, 1.0]$. If an individual solution is not schedulable after the reconfiguration, -1 is returned for both objectives. At last, the algorithm returns all the non-dominated solutions being found during the search.

C. Further discussion

With both methods, the schedule is statically decided offline so that the actual finish time of each I/O task (i.e., the longest execution time among all its jobs) can be obtained. Higher-level systems could integrate this value to their analysis (e.g., schedulability tests in [4] for NoC systems) and form complete I/O-aware schedulability tests. In the case where jobs execute less than their WCETs, the scheduling decisions can be preserved by making the processor idle until the execution time of the next task arrives. In addition, the proposed methods can also be applied to I/O tasks with different release offsets, where both methods can produce explicit schedule for different hyper-periods of the input jobs, until the schedule can repeat in future execution.

[2]If two jobs are assigned with the same start time, the job with a higher priority will be executed first.

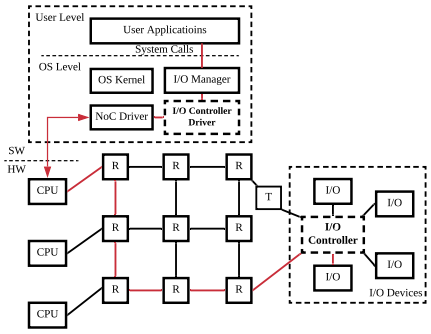


Fig. 3: I/O controller in an NoC system (R - Router / Arbiter)

IV. Hardware Support for I/O Scheduling

As described in Section III, the proposed system is fully-partitioned based on each I/O device, with scheduling decisions produced offline for each partition. Existing I/O controllers (including GPIOCP) cannot be directly applied as they do not support the offline job-level I/O scheduling and the execution model described in this work. Thus, additional requirements are derived for the underlying I/O controllers so that the proposed schedule can be enabled and correctly executed during run-time.

The use of the I/O controller within a NoC system is shown in Figure 3. As given in the figure, the controller is connect to the home port of a router and the global timer T via the physical links, to provide the communication channel (e.g., for loading I/O tasks) and synchronised time (i.e., for executing timely triggered I/O tasks) with the application processors. In addition, the controller is physically connected and synchronised with the I/O devices, so that the timing accuracy of a single I/O operation can always be achieved [2]. These facilities provide the basis for enabling the proposed off-line schedule and timing accurate I/O control. Note, a NoC is not mandatory as general purpose I/O systems are agnostic to the bus type, CPU architecture, and executing software.

Essentially, an I/O controller that can support the proposed schedule contains three major phases before and during system executions, and each phase requires the I/O controller to provide certain functionalities to realise the complete scheduling routine.

- Phase 1: *Pre-loading tasks* – The continuous I/O commands are grouped as one I/O operation (i.e., a timed I/O task), and all I/O tasks are stored into the controller before run-time.
- Phase 2: *Offline scheduling* – The start time of jobs released by the pre-loaded I/O tasks is calculated offline by the proposed methods, with the scheduling decisions passed and stored in the I/O controller.
- Phase 3: *Task execution* – Based on the stored scheduling, the pre-loaded I/O tasks are executed at their start time instants.

This is different from the design philosophy of GPIOCP, which only considers the pre-loading phase and applies a simple execution phase based on the FIFO scheduling [2]. Supporting the above functionalities in each phase requires the hardware implementations of two major components in the I/O controller: the Controller Memory and the Controller Processor(s).

- *Controller Memory* – manages the external (and internal) accesses to store (and retrieve) the pre-loaded I/O tasks;
- *Controller Processor(s)* – stores scheduling decisions; translates pre-loaded tasks to executable I/O commands; and executes the commands at the start time instants, based on a global timer.

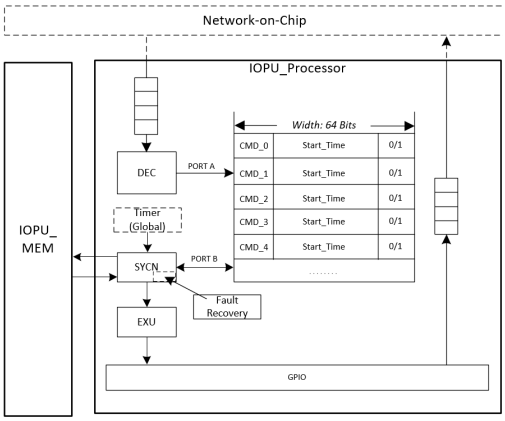


Fig. 4: Design of an I/O controller processor unit

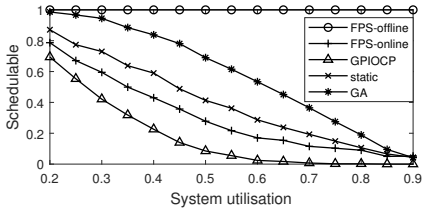


Fig. 5: System schedulability of each scheduling method

Among these phases, Phase 1 is handled by the controller memory, with communications between the application processors for I/O tasks. Phase 2 is performed off-line based on the proposed methods, with the scheduling decisions sent from the communication channel and stored in the controller processors. At last, Phase 3 is performed by all controller processors and the controller memory, with the communications between the application processors for initiating the I/O tasks and sending results.

As the required functions (i.e., to pre-load and to retrieve I/O tasks) of the controller memory is identical with that of the GPIOCP, the memory unit implemented in [2] can be applied directly to the proposed I/O controller. However, new facilities are required by the controller processors to provide the functions described above. The design of a controller processor is generic, which can be duplicated in the system integration, to provide partitioned scheduling for multiple connected I/O devices. Each processor is connected with one I/O device. The architecture of a controller processor is shown in Figure 4, which can be divided as the scheduling table, the request channel, the execution module and the response channel.

Specifically, the scheduling table records the identifier and the start time of the I/O tasks produced by the offline scheduling methods, which is received from application processors via "Port A" (i.e., Phase 2). During run-time, I/O requests received from the request channel set the corresponding bits of the requesting I/O tasks in the scheduling table to '1', indicating the schedule of the task is enabled. The execution module (Phase 3) consists of a global timer, a synchroniser, a fault recovery unit and a execution unit. The global timer is connected to the synchroniser and triggers the timed execution of I/O tasks, based on their start time instants encoded in the scheduling table. Once a I/O job is selected to execute, the synchroniser translates I/O tasks to the corresponding I/O commands via accessing the controller memory and sends the translated commands to the executor of the execution module i.e., "EXU". Moreover, run-time fault recovery is provided inside the synchroniser to handle the run-time exceptions (e.g., an I/O task is not received) and to ensure the correctness of the scheduling behaviours. At last, the responses (e.g, read data) from I/O devices are sent back to the application CPU via the response channel.

V. Experimental Results

In this section, evaluations are performed i) to investigate schedulability and timing accuracy of the proposed scheduling methods and ii) to demonstrate the resource efficiency of the proposed I/O controller implementation (i.e., the feasibility to deploy the described I/O controller to real-world applications), against the state-of-art I/O processing techniques and the mainstream I/O controllers.

A. Schedulability and timing accuracy

The schedulability and timing accuracy of the proposed scheduling methods are evaluated by randomly generated systems with the system utilisation incremented by 5%. The Utilisation of tasks is generated by the UUniFast algorithm [17], with a total system utilisation given by $U = 0.05 \times |\Gamma|$. For each task $\tau_i \in \Gamma$, T_i is generated randomly in a uniform distribution, from all periods that lead to a hyper-period of $1440ms$, with $D_i = T_i$ and P_i set by the DMPO. The quality curve range $[\delta_i - \theta_i,\ \delta_i + \theta_i]$ is set as half of its releasing period i.e., $\theta_i = T_i/4$, with δ_i set randomly between $[\theta_i,\ D_i - \theta_i]$. We enforce that $\theta_i \geq C_i$. At last, V_{max} is set to $P_i + 1$ for each task, and a global $V_{min} = 1$ is applied to all tasks. The population size and maximum iterations of the GA solver is set to 300 and 500 respectively. For each system configuration, 1000 synthetic systems are generated and tested by all examined methods.

Figure 5 presents the schedulability of each method, in which "static" denotes the proposed heuristic-based schedule (Algorithm 1), "GA" denotes the GA-based schedule and "GPIOCP" indicates the system and schedule presented in [2], with the assumption of a single I/O device in the system. In addition, the schedulability of the traditional non-preemptive FPS schedule is also presented as the baseline. "FPS-offline" is performed statically before run-time, which always executes the released job with the highest priority. "FPS-online" gives the worst-case schedulability of the dynamic FPS schedule during run-time, based on the schedulability test in [18].

From the figure, "FPS-offline" gives the best schedulability results, where all systems are schedulable under each configuration. However, the "FPS-online" demonstrates significantly lower schedulability due to the potential blocking imposed to the I/O tasks, and is outperformed by both proposed methods. This observation also justify the choice for performing off-line schedule, which yields better schedulability. As expected, the GPIOCP demonstrates the worst schedulability due to the simple FIFO queuing policy. Among the proposed methods, the GA-based schedule outperforms the static method, as more concern cases can be considered during the search.

Figures 6 and 7 present the I/O performance of the examined off-line scheduling methods among 1000 schedulable systems. For the GA-based method, the best result obtained for each objective (i.e., Ψ and Υ) is presented in respective figure. First, we observed that the traditional FPS (performed offline) is outperformed by other methods in all cases, due to lacking necessary consideration of timing accurate I/O control. In particular, no job is exact timing accurate under FPS for all configurations (i.e., $\Psi = 0$ for FPS under each configuration in Figure 6). This observation justifies the motivation for developing new scheduling methods for systems with timing accuracy requirements.

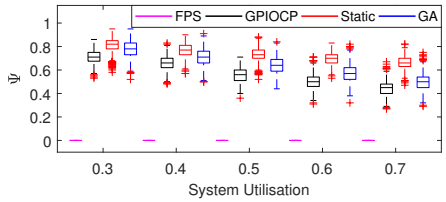


Fig. 6: Ψ of the offline scheduling methods

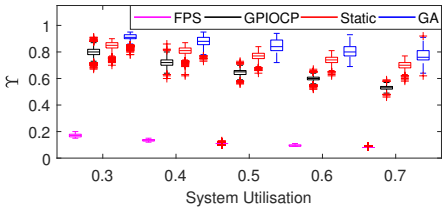


Fig. 7: Υ of the offline scheduling methods

With $U = 0.3$, GPIOCP can provide similar Ψ and Υ results as with the proposed methods, due to a relatively low scheduling pressure. However, with the increase of U, GPIOCP demonstrates the most pronounce fall in both objectives as it relies solely on the FIFO queues for scheduling I/O jobs, and is outperformed by the proposed methods. Among the proposed methods, the static schedule yields higher Ψ due to its explicit approach for maximising the number I/O jobs with exact timing accurate control. As for Υ, the GA-based schedule obtained better solutions than the heuristic-based method, as the sacrificed jobs in the static method are allocated only with the schedulability concern. However, with the increase of U, I/O performance obtained by the GA-based method demonstrates an obvious decreasing trend in both figure due to the increased search space while results of the static method are relatively stable.

B. Resource efficiency of the proposed I/O controller

To evaluated the hardware cost for supporting the proposed schedule, the proposed controller is evaluated against both basic and full-featured MicroBlazes (MB-B and MB-F), mainstream I/O controllers (i.e., UART, SPI, and CAN controllers), and the GPIOCP. All components are synthesised by Vivado (v2017.4) on Xilinx VC709 FPGA board, and are compared in terms of Look Up Tables (LUTs), registers, DSPs, BRAMs (Block RAMs) and power consumption required for implementation. As shown in Table I, the proposed I/O controller utilises significantly less hardware than a MB-F (i.e., 23.6% LUTs, 22.4% registers), and is similar to a MB-B (i.e., 135.4% LUTs, 185.6% registers). However, compared with the I/O controllers, more hardware resources are required to enable real-time scheduling and timing accuracy. At last, compared with GPIOCP, the proposed controller demands more hardware (i.e., additional 30.5% LUTs, 52.2% registers) to support the integration of the real-time scheduler. As for the power consumption, only 8.7% and 4.6% power is required, compared to the MB-B and MB-F respectively. To conclude, the proposed I/O controller requires slightly higher hardware resources to enable the proposed schedule, but is resource efficient compared to ones with generic CPUs in use.

VI. Concluding Remarks

In this paper, a scheduling model is proposed for dedicated real-time I/O processing units in multi- and many-core systems. Two scheduling methods are presented to provide I/O scheduling solutions that maximise I/O performance while guaranteeing system schedulability. The first method provides a heuristics-based solution that maximises the number of exact timing accurate I/O operations while the second method uses a GA-based approach that improves the overall I/O performance of the system. Necessary hardware support for realising the proposed schedule in I/O controllers is provided. Experiments show that the proposed schedule outperforms the state-of-art I/O processing techniques in terms of both schedulability and timing accuracy, and the proposed I/O controller is resource efficient.

TABLE I: Hardware overhead of evaluated I/O controllers

I/O Controllers	LUTs	Registers	DSP	RAM (KB)	Power (mW)
Proposed	1156	982	0	32	11
MB-B	854	529	0	16	127
MB-F	4908	4385	6	128	238
UART	93	85	0	0	1
SPI	334	552	0	0	4
CAN	711	604	0	0	5
GPIOCP	886	645	0	16	7

Acknowledgement

This work is partially funded by Huawei Technologies under the MOCHA project. The authors would also like to thank Dr. Xinwei Fang, for his assistance during the paper revision.

References

[1] W. Chang, "Resource-aware automotive control systems design," Ph.D. dissertation, Technische Universität München, 2017.
[2] Z. Jiang and N. C. Audsley, "GPIOCP: Timing-accurate general purpose I/O controller for many-core real-time systems," in *DATE*, 2017.
[3] J. Mössinger, "Software in automotive systems," *IEEE software*, 2010.
[4] L. S. Indrusiak, "End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration," *Journal of systems architecture*, pp. 553–561, 2014.
[5] B. Nikolić, S. Tobuschat, L. S. Indrusiak, R. Ernst, and A. Burns, "Real-time analysis of priority-preemptive NoCs with arbitrary buffer sizes and router delays," *Real-Time Systems*, vol. 55, no. 1, pp. 63–105, 2019.
[6] S. Zhao, J. Garrido, A. Burns, and A. Wellings, "New schedulability analysis for MrsP," in *RTCSA*, 2017.
[7] L. Abdallah, M. Jan, J. Ermont, and C. Fraboul, "I/O contention aware mapping of multi-criticalities real-time applications over many-core architectures," 2016.
[8] N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter, "Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks," in *RTNS*, 2018.
[9] J.-E. Kim, M.-K. Yoon, R. Bradford, and L. Sha, "Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems," in *38th Annual Computer Software and Applications Conference*, 2014.
[10] "PRU," http://www.ti.com/tool/pru-swpkg.
[11] "TPU," http://www.nxp.com/products/microcontrollers-and-processors.
[12] P. Dziurzanski, J. Swan, and L. S. Indrusiak, "Value-based manufacturing optimisation in serverless clouds for industry 4.0," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018.
[13] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini, "The meaning and role of value in scheduling flexible real-time systems," *Journal of systems architecture*.
[14] B. Khemka, R. Friese, L. D. Briceno, H. J. Siegel, A. A. Maciejewski, G. A. Koenig, C. Groer, G. Okonski, M. M. Hilton, R. Rambharos *et al.*, "Utility functions and resource management in an oversubscribed heterogeneous computing environment," *IEEE TC*, 2014.
[15] I. J. Bate, "Scheduling and timing analysis for safety critical real-time systems," Ph.D. dissertation, University of York, 1999.
[16] R. Guerra and G. Fohler, "A gravitational task model with arbitrary anchor points for target sensitive real-time applications," *Real-Time Systems*, vol. 43, no. 1, pp. 93–115, 2009.
[17] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.
[18] R. I. Davis, S. Kollmann, V. Pollex, and F. Slomka, "Controller area network (CAN) schedulability analysis with FIFO queues," in *23rd Euromicro Conference on Real-Time Systems*, 2011.