Task-Based Programming Models for Heterogeneous Recurrent Workloads

Jaume Bosch[1,2][0000−0002−4040−3416], Miquel Vidal[1][0000−0002−1973−8289], Antonio Filgueras[1][0000−0001−6071−3254], Daniel Jiménez-González[2][0000−0001−6064−7883], Carlos Álvarez[2][0000−0003−0536−5183], and Xavier Martorell[2][0000−0002−0417−3430] Eduard Ayguadé[1,2][0000−0002−5146−103X]

[1] Barcelona Supercomputing Center, 08034 Barcelona, Spain
{jaume.bosch,miquel.vidal,antonio.filgueras,eduard.ayguade}@bsc.es
[2] Universitat Politècnica de Catalunya, 08034 Barcelona, Spain
{djimenez,calvarez,xavim}@ac.upc.edu

Abstract. This paper proposes the extension of task-based programming models with recurrent workloads concepts. The proposal introduces new clauses in the OmpSs task directive to efficiently model recurrent workloads. The clauses define the task period and/or the number of task body repetitions. Despite the new clauses are suitable for any device, their support has been implemented using the capabilities of FPGA devices in embedded systems. These heterogeneous systems are common in industrial applications that usually develop recurrent workloads. The evaluation shows a huge gap in the applications' programmability, saving lines of code, and increasing the code readability. Besides, it shows the efficient management of recurrent tasks when performed in FPGA devices, which can support one order of magnitude finer tasks. All these improvements perfectly suit the needs of cyber-physical heterogeneous systems, which are frequently used in industrial environments to run recurrent workloads.

Keywords: Task-Based Parallelism · Recurrent Workloads · Heterogeneous computing · FPGA · OmpSs · OpenMP.

1 Introduction

Task-based parallel programming models have demonstrated to be a powerful tool to develop applications targeting heterogeneous platforms [12] [4]. There are new parallel programming models with native support for heterogeneous systems (e.g. OpenACC [8] and CUDA [7]). Also, different parallel programming models, which were initially designed for multicore processors, introduced new features to support such heterogeneous systems (e.g. OpenMP [9] and OmpSs [4]).

The importance of heterogeneous systems has grown in last years as co-processors can increase the computational power of systems while keeping the power consumption restricted. The FPGAs are reconfigurable co-processors that

Bosch, J. [et al.]. Task-based programming models for heterogeneous recurrent workloads. A: International Symposium on Applied Reconfigurable Computing. "Applied Reconfigurable Computing: Architectures, Tools, and Applications: 17th International Symposium, ARC 2021: virtual event, June 29–30, 2021: proceedings". Springer Nature, 2021, p. 108-122. ISBN 978-3-030-79025-7. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-79025-7_8.

allow implementing custom application logic within them. This logic can implement parts of applications, so they are executed faster and with a smaller power consumption than in the general purpose processor of the host.

Periodic systems (i.e., recurrent workloads) are a common workload in industrial environments and real-time applications. Those workloads use the task concept to define the different activities that must be executed periodically (after some amount of time). Thereby, task-based parallel programming models are great candidates to support recurrent workloads. We propose extending the current syntax of task-based parallel programming models to define the main recurrent task parameters. Therefore, modeling recurrent workloads can be accomplished efficiently in terms of code lines, and with all parallel capabilities of baseline programming models. Also, we propose using the reconfigurable heterogeneous platforms to efficiently manage these recurrent workloads. These platforms will provide an efficient management of recurrent tasks, keeping the great programmability provided by the parallel programming models.

The remainder of the paper is organized as follows. Section 2 describes the OmpSs@FPGA ecosystem. Section 3 describes the proposed programming model extension. Section 4 describes the implementation developed on top of OmpSs@FPGA to support the proposed extensions directly inside the FPGA co-processors. Section 5 presents the evaluation of the new design in an SoC. Finally, section 6 presents the related work and section 7 concludes.

2 Background

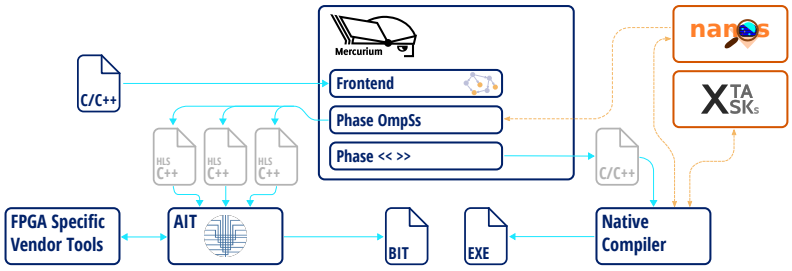


Fig. 1. OmpSs@FPGA compilation process

The OmpSs@FPGA ecosystem is an upgrade of the OmpSs [4] infrastructure (Mercurium source-to-source compiler and Nanos++ runtime) to incorporate FPGA support [2]. OmpSs is a task-based parallel programming model that extends the OpenMP 3.1 syntax and has been an active forerunner of the current tasking capabilities available in OpenMP 5.0. OmpSs@FPGA modifies Mercurium compiler, to support the generation of an independent source code file for each task with the FPGA target; and Nanos++ runtime, to handle the

data movements between the host and FPGA memory, and offload the tasks to the device. Moreover, the ecosystem is composed by other tools like AIT (Accelerator Integration Tool), which combines all FPGA files generated by Mercurium and calls specific vendor tools that generate the bitstream; and xTasks library, which provides a common API to Nanos++ for data and task management regardless the communication protocol between the host and the FPGA. The structure of all those tools is shown in figure 1.

```
1  #pragma omp target device(fpga) num_instances(2) \
2    copy_inout([BSIZE]buffer, [1]offset)
3  #pragma omp task
4  void read_sensor(float *buffer, unsinged int *offset) {
5    buffer[*offset] = ...;
6    *offset += 1;
7  }
8  int main(...) {
9    float buffer[BSIZE]; unsigned int offset = 0;
10   read_sensor(buffer, &offset);
11   #pragma omp taskwait
12 }
```

Listing 1: OmpSs@FPGA task example

The definition of an FPGA task in the OmpSs programming model can be seen in listing 1. In contrast to OpenMP, OmpSs supports the annotation of function declarations that result in a task spawn every time the function is called. In listing 1, we can see that function `read_sensor` is annotated with the task and target directives. The `device(fpga)` clause executes the task in an FPGA device instead of the host processor. The `num_instances(2)` clause defines the number of FPGA task accelerators that should be generated in the FPGA bitstream, which will be available to concurrently execute the instances of this task. The `copy_inout([BSIZE]buffer, [1]offset)` clause defines the data regions accessed by the task body, so the runtime manages a copy of them into FPGA memory before and after each task execution. Also, there are `copy_in(...)` and `copy_out(...)` clauses which only copy the data before or after each task execution respectively. In addition, data regions could be defined in the data dependence clauses (`in`, `inout`, and `out`) which also annotate the regions as task dependences to order the execution of tasks at runtime. The implementation supports task nesting within FPGA tasks. Therefore, FPGA task accelerators may spawn new FPGA/SMP/GPU tasks and synchronize them with the regular taskwait.

For each FPGA task in the application, Mercurium compiler isolates its source code in a separate source file and embeds it in a communication wrapper. This wrapper has a common external interface independent of task arguments, and it provides instrumentation capabilities and data caching optimization. Finally, the generated files are provided to AIT which generates an FPGA bitstream with all task accelerators and the management logic. The IP blocks placed in the FPGA design with their interconnections are shown in figure 2.

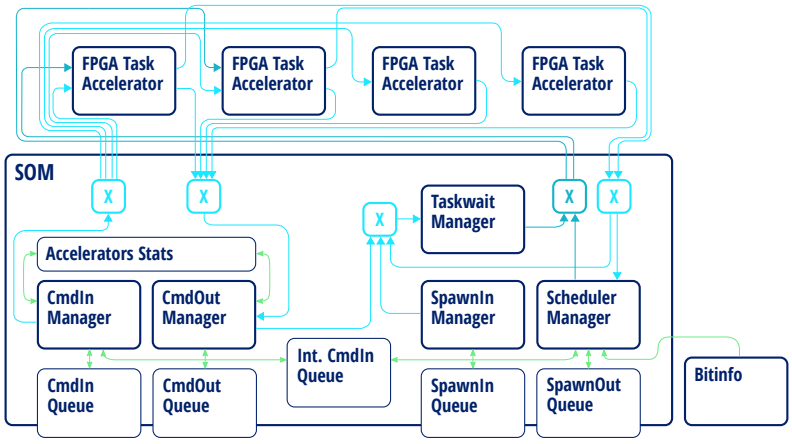


Fig. 2. OmpSs@FPGA bitstream organization

The FPGA task accelerators are connected with input and output streams to the SOM Hardware Runtime (HWR) [3]. The SOM module handles the commands sent by the host runtime and manages the messages sent by the FPGA task accelerators (task execution finished, task spawn, spawned tasks synchronization, etc.). The communication between the host runtime (through xTasks library) and the HWR is done by commands submitted into the 4 queues shown in figure 2. Each of the queues is managed by an IP block: `CmdIn Manager` forwards the commands to the FPGA task accelerators, `CmdOut Manager` notifies about executed commands, `Scheduler Manager` forwards the tasks spawned within the FPGA (into `Int. CmdIn Queue` when the task can be executed directly in the FPGA, into `SpawnOut Queue` otherwise), `SpawinIn Manager` handles executed task commands for reverse offloaded tasks, `Taskwait Manager` counts the tasks executed in each context for later synchronization.

3 Programming Model Extension

The OmpSs programming model has been extended with new clauses to annotate the recurrent tasks information. Although we present the clauses for the OmpSs programming model, they could also be incorporated in the OpenMP standard with the same semantics. The clauses are intended to provide the information of recurrent tasks which are periodically triggered by a timer. We keep as future work the support for recurrent tasks triggered by external events.

The `period` and `num_repetitions` clauses have been added to the existing task directive. They are optional and only one of both is needed to create a recurrent task. Both clauses take a single expression, which is evaluated as a 32 bits value at runtime. Therefore, the value may be unknown at compile time.

The `period(N)` clause defines the minimum amount of time between the beginning of two task executions. By default, the time is expressed in microseconds,

but this may be changed at compile time (using a compiler option). The default (implicit) clause value in a recurrent task is 0, which makes the task start again just after it finishes. The same happens when the task execution takes longer than the period: the next repetition starts just after the former. However, this behavior could be changed in the future with extra clauses or by a runtime option. Other approaches could be: aborting the current repetition, aborting the next repetition, schedule both repetitions in parallel.

The `num_repetitions(N)` clause defines the maximum number of times that the task body will be executed. The default (implicit) clause value in a recurrent task is an unlimited number of repetitions. Since 0 repetitions may be a valid amount, the unlimited number of repetitions is represented with the largest representable value (`0xFFFFFFFF`) which can be specified through `OMP_REPS_UNLIMITED` constant.

The semantics of other clauses in a recurrent task remain equal to a regular task. The same criteria apply to other programming model directives, like the taskwait. The recurrent tasks will not become finished until all repetitions have been run. Therefore, a taskwait after a recurrent task will not be accomplished until all task repetitions have been run. Also, the data dependences in a recurrent task only postpone the execution of the first repetition but not the others, and the successor tasks of a recurrent task are not ready until all repetitions are executed. However, a recurrent task can periodically spawn a set of child (nested) tasks with dependences between them. This makes possible having data dependences between periodically executed tasks.

```
1  #pragma omp target device(fpga) num_repetitions(reps) period(1000000) \
2    copy_inout([BSIZE]buffer, [1]offset)
3  #pragma omp task
4  void read_sensor(float *buffer, unsigned int *offset, const int reps);
5
6  int main(...) {
7    read_sensor(buffer, &offset, 10);
8    #pragma omp taskwait
9  }
```

Listing 2: OmpSs example with a recurrent task

Listing 2 shows the example of listing 1 extended with the new recurrent clauses. The `read_sensor` task has been annotated with the `run_repetitions` and `period` clauses in line 1. Therefore, the task's body will be executed `reps` times (10 as shown in function call of line 7) every 1 second (1000000 microseconds in the clause value). The taskwait in line 8 will be accomplished after all task body repetitions have been executed. Similarly, successor tasks that have a common data dependence on the same element (annotated with `in`, `inout`, or `out` clauses) will be ready to execute once recurrent tasks finish.

Two new APIs have been defined in the programming model to control the execution of recurrent tasks. Those APIs must be called within the recurrent tasks and apply to itself.

`omp_get_periodic_task_repetition_num` returns the current repetition number which is under execution. The first repetition is the number 1, since 0 is returned when the task is non-recurrent.

`omp_cancel_periodic_task` cancels the remaining repetitions of the current recurrent task. It does not cancel the remaining task code after the API call.

4 Implementation

The programming model extension has been implemented on top of OmpSs@FPGA. This section presents the modifications that we performed in order to support the recurrent tasks management in the FPGA task accelerators. In addition, the support for critical regions between task accelerators inside the same FPGA device has been introduced. This is very useful to coordinate the different accelerators during the execution as the recurrent tasks start their execution autonomously without any synchronization.

4.1 Mercurium Compiler

The OmpSs translation phase has been extended to support the new clauses. The compiler forwards the period and number of repetitions information to the runtime during the task creation if any of the clauses appear. The information is provided using a new runtime API. In contrast, the compiler emits the regular API calls for task creation if the task directive does not contain neither `period` nor `num_repetitions` clauses.

The wrapper generated by the compiler in the C++ HLS intermediate files has been extended. It has to support a new command (similar to the task execution command) with the number of repetitions and period information. The wrapper has to handle the calls to `omp_get_periodic_task_repetition_num` and `omp_cancel_periodic_task` APIs which could be called within the task code. Moreover, the support for critical regions in the FPGA task accelerators requires sending a message to the new `Lock Manager` of HWR.

4.2 Nanos++ runtime and xTasks library

The Nanos++ runtime has been extended with a new API that is called during the task spawn to set the recurrent information. The API is called `nanos_set_wd_recurrent` and it takes 3 arguments: the reference/pointer of the task being spawned, the minimum period between task executions, and the number of repetitions.

The xTasks library has been extended to support the new task information. When a task has to be offloaded to the FPGA and it is a recurrent task, the new command is issued instead of the previous task execution command.

4.3 FPGA Design Support

The support of the new features only requires minor changes in the FPGA task accelerators and HWR. The main change is the new command that must be supported by the HLS wrapper (generated by Mercurium) and the HWR. Also, the new `Lock Manager` has been added inside the HWR to manage exclusive access to critical regions.

The FPGA task accelerators have been extended to support the new capabilities. The wrapper generation has been modified to implement the repetitions management. In addition, two new ports have been added to delay the task body executions the specified amount of time. The ports are used to read a counter that is incremented every clock cycle, and the frequency of such increment. With this information, the wrapper is able to accurately wait the needed amount of time between repetitions.

The lock acquire and release messages, which are sent to support the critical regions on the FPGA task accelerators, are forwarded through the existing interconnections. These interconnections are the ones going between SOM and FPGA task accelerators, and vice-versa.

The `Lock Manager` is a new IP block developed in C++ using HLS tools and added inside the SOM HWR. It has input and output interfaces to receive and acknowledge the messages sent by the FPGA task accelerators. Those streams are connected like `Taskwait Manager` or `Scheduler Manager`, as shown in figure 2. The module's purpose is to keep the state of different locks that may be concurrently acquired and released by the FPGA task accelerators. To this end, the module has an internal table with the state of each lock. The module can receive two types of messages in the input stream: acquire messages and release messages. Whenever it receives a new message, it looks for the lock identifier (sent by the FPGA task accelerator) in the internal table and sets or clears the lock state. The response is successful in the acquire messages when the lock was not previously set and fails when the state is unchanged.

The number of entries in the internal lookup table is fixed during the AIT design stage. The current implementation has a direct mapping in the table for each lock ID. However, this could be changed and implement an N-way table. There must be enough entries in the table to ensure that nested locks do not map to the same entry. Otherwise, the system will enter in a deadlock state due to the impossibility of acquiring the inner lock.

The `CmdIn Manager` has been updated to consider the new command format that xTasks library may send to the FPGA task accelerators. The new command has the same format as the previous execute task command but with an extra word (number of repetitions and period) that also has to be forwarded.

5 Evaluation

The evaluation of the proposed design and implementation of recurrent tasks has been done in terms of programmability and productivity, limitations of tasks

management, and power savings. Section 5.1 presents the experimental setup used in the evaluation. Section 5.2 presents the evaluation of the limitations and management overheads of the proposed design using a synthetic benchmark. Section 5.3 shows the results for a sensor monitoring use case in embedded systems. Finally, section 5.4 presents the evaluation of the proposal for a face detection application.

5.1 Experimental Setup

The evaluation of the proposal modifications has been done in a Zedboard with real executions. The board contains a Xilinx Zynq®-7000 All Programmable SoC [1], and it is commonly used in embedded industrial systems as it offers high versatility with reduced power consumption and budget. The SoC is composed of 2 ARM Cortex-A9 cores, that run at 667 MHz, a Xilinx Zynq-7000 FPGA and a main DDR3 memory of 512 MB. The board is booted using the Ubuntu Linux 16.04 operating system. All FPGA bitstreams have been generated and executed at 100 Mhz.

The tools used to generate the application bitstream and binaries are: Vivado Design Suite 2020.1, GNU C/C++ Compiler 6.2.0, and PetaLinux Tools 2019.2. The modifications have been developed on top of OmpSs@FPGA release 2.3.0.

5.2 Synthetic benchmark

The synthetic benchmark executes a recurrent task `NUM_REPS` times every `PERIOD` microseconds. The task, called `foo`, lasts for `duration` microseconds. With these parameters, we can explore the runtime limits to manage fine-grain recurrent tasks. The `NUM_REPS` and `PERIOD` information can be provided into the programming model using the proposed clauses or handled at application level without the proposal extensions. Listing 3 shows the pseudo-code of benchmark tasks. The `foo` task is always used, but without the new clauses (`num_repetitions` and `period`) in the baseline implementation. The `foo_manager` task is only needed in the baseline implementation to launch and synchronize the recurrent task every `PERIOD` microseconds from the SMP host threads. In contrast, the `foo_manager` task is not needed with the proposed extensions, and `foo` is directly called.

Implementing the management logic of a recurrent task requires an effort from programmers and additional code. In contrast, the logic can be avoided by just using a couple of programming model clauses. The baseline system requires one extra task of 8 lines for each recurrent task (like `foo_manager` in listing 3). During the period wait, the taskyield directive [9] (line 12) is used to avoid blocking the host thread executing the manager task. However, the task consumes resources and adds pressure to the host runtime. In contrast, the proposal moves the management complexity into the FPGA task accelerator, which only needs to receive the task information with the number of repetitions and the period. This has a minimal footprint in the resources used by the FPGA task accelerator as shown in table 1 (for a 100 Mhz build). In terms of power consumption, the proposal removes the need to use an A9 core for recurrent task

```
#pragma omp target device(fpga) /* num_repetitions(NUM_REPS) period(PERIOD) */
#pragma omp task
void foo(int duration) { usleep(duration); }

#pragma omp task
void foo_manager(int duration) {
  for (unsigned int rep=0; rep<NUM_REPS; rep++) {
    double t_ini = wall_time_us();
    foo(duration);
    #pragma omp taskwait
    while ((wall_time_us() - t_ini) < PERIOD && rep < (NUM_REPS - 1)) {
      #pragma omp taskyield
} } }
```

Listing 3: Recurrent synthetic benchmark tasks pseudo-code

management, which consumes 277 mW. Indeed, it only increases the FPGA task accelerator power by 1 mW while doing the same work. Both powers are reported by Vivado in the post-implementation power summary.

Name	**BRAM**	**DSP**	**FF**	**LUT**	**Power**
Zedboard	280	220	106.400	53.200	-
ARM A9 Core	-	-	-	-	277 mW
`foo` baseline	0 (0%)	2 (0,9%)	593 (0,6%)	478 (0,9%)	7 mW
`foo` proposal	0 (0%)	4 (1,8%)	921 (0,9%)	792 (1,5%)	8 mW

Table 1. Vivado resources utilization and power report for Zedboard (100 Mhz)

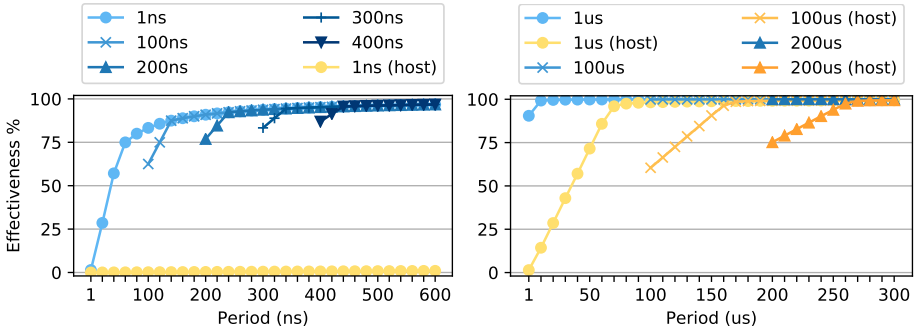
The time devoted to each repetition is `PERIOD`, or `duration` if the task execution lasts more than the task's period (concurrent repetitions are not allowed). Then, the optimal execution time (equation 1) is defined by `NUM_REPS` multiplied by the time devoted to each repetition, except for the last one that only needs to consider the task execution time (`duration`) as the benchmark can immediately finish. Moreover, the effectiveness of an execution (equation 2) is defined by the ratio between the optimal execution time and the real execution time (100% means optimal).

$$Time_{optimal} = (num\_reps - 1) * max(duration, period) + duration \quad (1)$$

$$Effectiveness_{exec} = Time_{optimal}/Time_{exec} \quad (2)$$

Figure 3 shows the effectiveness (y-axis) of the synthetic benchmark when changing the task period (x-axis). Figure 3a shows the results at nanoseconds scale, and figure 3b shows the results at microseconds scale. Each chart has different series that correspond to different task durations. The task durations are labeled in the legend and are only shown for periods equal or larger than them. The number of repetitions is constant in all results at the same scale and proportional between them (10.000.000 in figure 3a and 10.000 in figure 3b). The

FPGA device is configured at 100 Mhz. The blue series show the effectiveness of synthetic benchmark implemented with the proposal enhancements and having the recurrent task management in the FPGA task accelerator. On the other hand, the orange series show the effectiveness of baseline implementation using a manager task in the host threads. These baseline series are labeled in the legend with *(host)*.



(a) Nanoseconds scale (b) Microseconds scale

Fig. 3. Effectiveness of synthetic benchmark for different task duration values

The effectiveness results in figure 3 show that the proposal implementation always obtains higher effectiveness than the baseline implementation. All series behave similarly, the effectiveness increases the higher the period until the difference between the period and the task duration is at least 60 ns for the proposal and 70 us for the baseline. After these offsets, the effectiveness stabilizes near to 99%. The 70 us in the orange series are due to the communication round-trip between the host and the FPGA task accelerator. In contrast, the management inside the FPGA task accelerators (blue series) only needs 60 ns to check the number of remaining repetitions and whether the period has elapsed or not. Therefore, the integrated management of recurrent tasks avoids host-FPGA communications with overheads 1167x smaller than the ones in the baseline.

5.3 Sensors Monitoring

The sensors monitoring is a common workload in industrial environments. Its purpose is collecting information from any kind of sensors with a given frequency to know the elements state and keeping track of it, or even reacting if some irregularity is detected. We developed a benchmark that simulates this behavior based on a real and complex application. We defined a recurrent task for each sensor and one to handle the samples, all of them are FPGA tasks. The first tasks are responsible for retrieving and storing the sensor data into intermediate buffers. The handling task merges all data from sensors into a sample and

orchestrates a parallel sample processing. Therefore, the benchmark scalability can be analyzed based on the sensors' reading period and the number of sensors that are being read. We defined 3 different implementations depending on which new capabilities are being used:

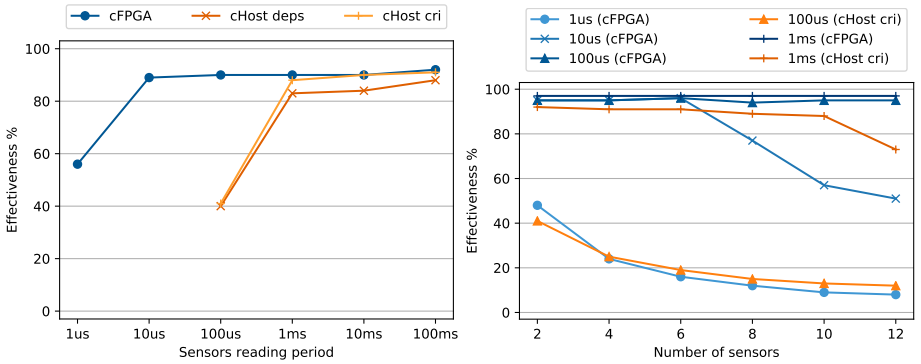
- `cHost deps`. Recurrent tasks managed at application level (without the new clauses) and using data dependences to synchronize the recurrent tasks.
- `cHost cri`. Recurrent tasks managed at application level (without the new clauses) and using critical regions to synchronize recurrent tasks.
- `cFPGA`. Recurrent tasks managed by the programming model (with the new clauses) and using critical regions to synchronize recurrent tasks.

The manager task that implements the recurrent support at application level requires a significant amount of extra logic. It has to keep track of the last submit timestamp and the number of repetitions executed for each recurrent task. Also, a finalization condition has to be maintained to know when all recurrent tasks have finished and then break the manager loop. The loop checks if the current timestamp is behind the last start timestamp plus the specified period for each recurrent task. If so, the task is launched, and their state inside the manager is updated. Once all recurrent tasks have been checked, the manager waits for them and yields until more tasks have to be submitted.

The management without the proposed capabilities assumes that the duration of recurrent tasks is smaller than the smallest period or recurrent tasks. This is due to the taskwait after the task spawns (similar to line 6 of listing 3). For durations longer than the smallest period, the management will delay some task spawns. In contrast, the proposal implementation does not have such limitation as the recurrent tasks are managed independently. Also, the host manager only can ensure the period between task spawns but not task starts, which may be delayed by the runtime when no executors are available or due to runtime overheads. The recurrent tasks management implemented in the proposal has a good period precision as the repetitions management is kept together with the executor (the FPGA task accelerator).

Figure 4a shows the effectiveness (y-axis, computed like in synthetic benchmark) for the 3 implementations among different base periods (x-axis). The number of repetitions for the sensors reading task is fixed to 10, and there are 2 sensors in all executions. The host implementations only maintain the effectiveness above 1 millisecond base period. In contrast, `cFPGA` maintains the effectiveness above 10 microseconds base period, which is 100 times smaller. Moreover, the host implementations fail to execute with base periods below 100 microseconds due to an excessive memory consumption generated by the huge number of created tasks in Nanos++ runtime.

Figure 4b shows the effectiveness (y-axis) for the small base periods of figure 4a, where effectiveness starts to decay, among different amounts of read sensor tasks. Figure 4b helps to see how the proposal and the baseline behave with different amounts of recurrent tasks (the read sensors in this case). The results confirm the 100x difference between both implementations seen in previous



(a) Base period (b) Num. accelerators

Fig. 4. Effectiveness of sensors monitoring benchmark

results. The proposal implementation effectively handles all amounts of sensors for 1 ms and 100 us periods. Meanwhile, `cHost cri` loses some effectiveness with 12 sensors for 1 ms period, and it suffers with the 100 us period regardless of the number of sensors. In contrast, `cFPGA` shows a similar effectiveness decrease with 1 us period. It keeps the effectiveness for periods above 1 us.

5.4 Face Detection

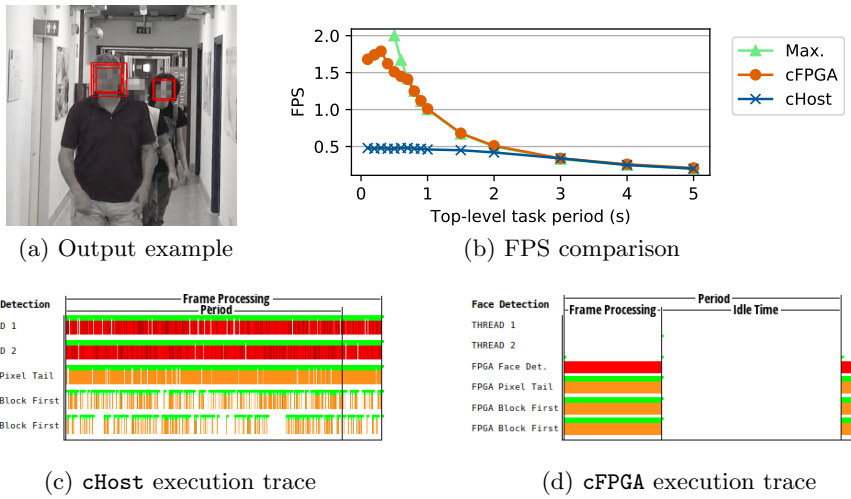
The detection of faces on pictures is a common workload in several scenarios. One of them is the analysis of an image stream in real-time, which could benefit from the new FPGA capabilities. The implementation is based on a previous port to OmpSs@FPGA developed in AXIOM project [5]. The application periodically analyzes a frame buffer with 2 sets of filters implemented with FPGA tasks. The first set is performed by blocks (144x144 pixels including the needed padding) for all the frame and using 2 instances of the FPGA task accelerator. The second set is only performed to pixels with a minimum score after the first pass using 1 FPGA task accelerator. After the filters and if some face is detected, an SMP task is triggered with the list of coordinates to render the frame as shown in figure 5a.

We defined two implementations to compare the management of the application with the new recurrent task capabilities against equivalent management in the baseline. The `cFPGA` implementation, which periodically executes the top-level task in the FPGA using the `period` and `num_repetitions` clauses. The `cHost` implementation, which executes the top-level task in the SMP threads and supports the periodic behavior at application level without using programming model extensions. The number of frames processed (number of repetitions of top-level task) in each execution is 30, and they have been extracted from an input video that contains three people walking in a row. This amount of

frames provides long enough executions to gather significant executions for the performance analysis.

Figure 5b shows the FPS (y-axis) handled by `cFPGA` and `cHost`. The values are shown for different period values of top-level task (x-axis). The maximum FPS value for each period ($FPS_{max}(period) = 1/period$) is also shown by the `Max.` series.

The results show that the `cHost` is not a performant option to handle the frames as the best rate does not reach 0.5 FPS (2 seconds per frame). The performance is limited by the data movements, which gather the first pass scores needed to conditionally spawn the tail tasks. In contrast, the `cFPGA` implementation reaches up to 1.79 FPS as all management can be kept inside the FPGA. The peak is achieved with a 0.3 seconds period, which is approximately the duration of one frame analysis when there are no active pixels for the tail pass. The `cFPGA` performance is the maximum value for periods above 0.7 seconds, which is the average duration of the analysis. Between 0.3 and 0.7 seconds for the period, some frames are processed within the period time and some are not. Therefore, the performance scales but not at the maximum values.



(a) Output example (b) FPS comparison

(c) `cHost` execution trace (d) `cFPGA` execution trace

Fig. 5. Face Detection execution results

Figures 5c and 5d show two execution traces of Face Detection (`cHost` and `cFPGA` respectively) for a 2 seconds period. Both have the same elapsed time (x-axes) and show the activities (colors) in the different computing elements (y-axis). The activities shown are: orange for execution of tasks in FPGA task accelerators, red for execution of top-level task, dark-red for task offload from host to FPGA, and white for idle time. Traces clearly show the sparsity of FPGA tasks in `cHost` and that they are compacted in `cFPGA`. The baseline implementation takes too long due to the task management which occupies the

2 host threads all the time with the execution of top-level task (red regions) and the offloading of FPGA tasks (dark-red regions). The huge overheads result in an under-utilization of FPGA task accelerators and a poor frame rate. The `cHost` requires all the trace time (approximately 2.25 seconds) to handle the frame which is above the desired period of 2 seconds. In contrast, the `cFPGA` lasts for 700 ms, and the system remains idle until the next repetition is launched. Therefore, the FPGA management is able to support more frames per second as shown in figure 5b chart. Another relevant fact is that the host threads are not needed almost any of the time. That optimizes the power budget of the application without impacting the overall performance as both threads could be switched off and turned on when needed.

6 Related Work

There are previous works that consider extending task-based parallel programming models with real-time system capabilities. Serrano et al. [11] analyze the usage of OpenMP to develop critical real-time systems. They analyze the timing constraints, which are not considered in this thesis but are fully compatible. They briefly introduce an event clause that is used to define when a recurrent task must start executing. Besides, Pop et al. [10] propose an OpenMP extension to define persistent tasks that work in a stream fashion. Those tasks are similar to the recurrent tasks proposed in this paper but they are activated by the availability of input data instead of by a timer. Also, we propose the support of those tasks using the FPGA capabilities to have accurate and low-latency management.

Other previous efforts try to extend the capabilities of accelerators and optimize their usage in heterogeneous systems. CUDA Dynamic Parallelism [7] adds support for nested execution of CUDA kernels. Vesely et al. [13] discuss the support of operating system calls in GPGPUs. Heinz et. al. [6] discuss the TaPaSCo runtime optimization to increase the execution rates in FPGA-accelerated jobs. All of them also extend and optimize the heterogeneous systems but without targeting real-time workloads or efficiently allowing their definition.

7 Conclusion

This paper proposes the extension of task-based programming models with recurrent workloads concepts. The proposal introduces new clauses in the task directive to efficiently model recurrent workloads. The clauses have been introduced in the OmpSs programming model but they are suitable for standard task-based parallel programming models like OpenMP. The evaluation of different benchmarks shows a huge gap in the application programmability. Besides, it shows the efficient management of recurrent tasks when performed in FPGA devices, which is able to support one order of magnitude finer tasks. The reduction of management overheads provides big idle gaps during the execution

of the application. This allows smaller periods of recurrent tasks than the baseline, or reducing the power consumption as the elements could be switched off. All these improvements perfectly suit the necessities of new cyber-physical and heterogeneous devices when running recurrent workloads.

Acknowledgments

This work is partially supported by the European Union H2020 Research and Innovation Action (project 801051), by the Spanish Government (projects SEV-2015-0493 and PID2019-107255GB, grant BES-2016-078046), and by the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328).

References

1. Avnet: ZedBoard Technical Spec. (2020), http:/zedboard.org/content/zedboard-0
2. Bosch, J., Tan, X., Filgueras, A., Vidal, M., Mateu, M., Jiménez-González, D., Álvarez, C., Martorell, X., Ayguadé, E., Labarta, J.: Application Acceleration on FPGAs with OmpSs@FPGA. In: 2018 International Conference on Field-Programmable Technology (FPT). IEEE (12 2018)
3. Bosch, J., Vidal, M., Filgueras, A., Álvarez, C., Jiménez-González, D., Martorell, X., Ayguadé, E.: Breaking master-slave model between host and FPGAs. In: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 419–420 (02 2020)
4. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel processing letters **21**(02), 173–193 (2011)
5. Filgueras, A., Gai, P., Garzarella, S., Oro, D., Hernando, J., Bettin, N., Pomella, A., Procaccini, M., Giorgi, R., Vidal, M., Mateu, M., Jiménez-González, D., Álvarez, C., Martorell, X., Ayguade, E., Theodoropoulos, D., Pnevmatikatos, D.: The AXIOM Project: IoT on Heterogeneous Embedded Platforms. IEEE Design & Test **PP**, 1–1 (11 2019)
6. Heinz, C., Hofmann, J.A., Sommer, L., Koch, A.: Improving job launch rates in the tapasco fpga middleware by hardware/software-co-design. In: 2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS). pp. 22–30 (2020). https://doi.org/10.1109/ROSS51935.2020.00008
7. NVIDIA: CUDA Dynamic Parallelism Programming Guide (2019)
8. OpenACC Consortium: The OpenACC application programming interface (2011)
9. OpenMP ARB: OpenMP API Specification (2018), https://www.openmp.org/spec-html/5.0/openmp.html
10. Pop, A., Cohen, A.: A Stream-Comptuting Extension to OpenMP. pp. 5–14 (01 2011)
11. Serrano, M.A., Royuela, S., Quiñones, E.: Towards an OpenMP Specification for Critical Real-Time Systems, pp. 143–159 (01 2018)
12. Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., Gschwandtner, P., Lemarinier, P., Markidis, S., Jordan, H., et al.: A taxonomy of task-based parallel programming technologies for high-performance computing. The Journal of Supercomputing **74**(4), 1422–1434 (2018)

13. Veselý, J., Basu, A., Bhattacharjee, A., Loh, G.H., Oskin, M., Reinhardt, S.K.: Generic system calls for GPUs. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). pp. 843–856. IEEE (2018)