FPGA Checkpointing for Scientific Computing

1st Marc Perelló Bacardit
Barcelona Supercomputing Center
Barcelona, Spain
marc.perello@bsc.es

2nd Leonardo Bautista-Gomez
Barcelona Supercomputing Center
Barcelona, Spain
leonardo.bautista@bsc.es

3rd Osman Unsal
Barcelona Supercomputing Center
Barcelona, Spain
osman.unsal@bsc.es

***Abstract*—The use of FPGAs in computational workloads is becoming increasingly popular due to the flexibility of these devices in comparison to ASICs, and their low power consumption compared to GPUs and CPUs. However, scientific applications run for long periods of time and the hardware is always subject to failures due to either soft or hard errors. Thus, it is important to protect these long running jobs with fault tolerance mechanisms. Checkpoint-Restart is a popular technique in high-performance computing that allows large scale applications to cope with frequent failures.**

In this work we approach the fault tolerance of CPU-FPGA heterogeneous applications from a high level by using OmpSs@FPGA environment and a multi-level checkpointing library. We analyse the performance of several different applications and we understand what kind of overheads we can expect from checkpointing computational workloads running on FPGAs. Our results demonstrate overheads as low as 0.16% and 0.66% when checkpointing very frequently, indicating that this technique is efficient and does not add a significant amount of overhead to the system. In addition, we showcase a proof of concept for checkpointing partial data of the FPGA task itself. This can prove useful for workloads in which most data is offloaded to the FPGA memory at once and do not constantly move all the data between the accelerator and the CPU.

***Index Terms*—FPGA, FTI, fault tolerance, accelerator, resilience, checkpointing, reliability**

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are computational devices known for their energy efficiency in comparison to instruction set architecture (ISA)-based devices like CPUs and GPUs. However, FPGAs are also more challenging for application development. The workload to be executed has to be converted into a set of bits containing the information about the configuration of its logic cells, Block RAM, etc. Originally, this had to be done by developing the application in hardware description languages like VHDL.

Fortunately, there are now tools like Vivado High Level Synthesis (HLS) [1] which allow developing for such devices in high level languages. Also, frameworks such as OmpSs@FPGA [2] make it even more straightforward and do most of the extra work needed to develop and build FPGA applications. Because of this, heterogeneous systems with FPGAs are becoming increasingly popular and are being considered as accelerators for High Performance Computing (HPC) systems.

Multiple works in the literature have demonstrated the energy efficiency of FPGAs. For instance, a recent study [3] evaluated the performance and energy efficiency of the Gaxpy kernel (matrix-vector multiplication) implementation on CPU, GPU and FPGA platforms, demonstrating that the FPGA implementation is an order of magnitude lower in terms of power consumption, while also being very competitive in terms of performance.

FPGAs are ideal devices for machine learning purposes, since they can achieve close to ASIC performance and efficiency while not being stuck with a single algorithm-specific hardware design. A Xilinx study [4] shows a comparison between a NVidia Tesla V100 GPU and a Xilinx Virtex Ultrascale+ XCVU13P FPGA in terms of number of operations per Watt. The FPGA device almost matches the efficiency of the Tensor Cores hardware from the Tesla V100 GPU and more than triples its standard efficiency (without Tensor Cores acceleration), which demonstrates its energy efficiency.

On the other hand, supercomputers are continuously growing in computational complexity, power and storage resources. Because of this, their mean time between failures (MTBF) is becoming shorter. Therefore, fault tolerance tools are a vital part of the HPC software stack nowadays, not only to save time but also for substantial energy savings. Since current supercomputers are in need of efficient fault tolerance systems due to their sheer amount of computing units; FPGAs need to be able to take advantage of those resilience techniques as well. Thus, evaluating the performance of fault tolerance schemes in HPC systems with FPGA devices is crucial for the future of large scale supercomputers.

Checkpointing is the preferred software resilience mechanism for scientific computing which typically features long running applications. In this work, we propose to implement Checkpoint-Restart (CR) for FPGA applications running on heterogeneous systems (i.e., CPU-FPGA). For that, we adapt a multilevel checkpoint library called Fault Tolerance Interface (FTI) [5], in order to be able to checkpoint FPGA applications. In addition, we implement two types of checkpoint strategies, i) checkpointing from the host in between FPGA task executions, and ii) checkpointing inside long-running FPGA tasks. We describe the design of both techniques and evaluate them with multiple applications. We use the term *FPGA task* to describe the portion of the application which is implemented in the FPGA's configurable logic.

The contributions of this paper are summarized as follows:

- Design of a CR technique capable of saving the state of FPGA applications and restart from it.
- Implementation of a checkpointing method involving the checkpointing of partial work of an FPGA task, as

opposed to the standard method of checkpointing the entire work in the main loop.
- Performance evaluation of the FTI library on a CPU/FPGA heterogeneous system, with several applications.

The rest of this paper is organized as follows. Section II discusses the motivation for this research. Section IV explains the methodology used to implement checkpointing on FPGAs. Section V shows the evaluation details and results. Section III presents the related work. Finally, Section VI discusses the conclusions and future work suggestions.

II. Motivation

A. Energy Efficiency and Reliability in HPC

Several reports on extreme scale computing [6]–[8] have studied the main challenges to achieve exascale computing, and most of them agree that energy efficiency and reliability are among the top three most pressing problems. Therefore, exploring low-power computing devices is paramount in the road ahead to achieve energy-efficient exascale computing.

The LEGaTO project [9] aims to address this issue by leveraging heterogeneous systems. The main objective is to provide a software stack which is not only developed for high performance, but also has support for energy-efficient computing. Fault tolerance plays an important role on this endeavour. Since supercomputers use a lot of energy for computational purposes, restarting a failed computation from a checkpoint, instead of restarting from scratch, increases dramatically the energy efficiency. Thus, clearly reliability plays an important role in the decrease of wasted CPU hours and therefore wasted energy.

Unfortunately, classic checkpointing is not enough. The resilience at exascale report [10] highlights the importance of leveraging novel storage devices, as well as developing new interfaces and runtime libraries to optimize and delegate most of the fault tolerance tasks that scientific applications need to execute. Therefore, it is important to use and evaluate scalable fault tolerance techniques and adapt those to the new low-power heterogeneous architectures.

A recent study collected and analyzed a dataset of 44-month failures of the China Meteorological Administration HPC system, divided into two subsystems [11]. The study shows that up to 50 failures can be encountered in one of the subsystems in a single month, approaching the average of 2 failures each day. This shows that failures in HPC systems are not rare, but instead are a daily occurrence. Since exascale systems are going to have a lot more resources than current HPC systems, they are expected to fail much more frequently. Because of this, efficient checkpoint methodologies are needed to minimize impact on performance and energy consumption.

B. Scalable Checkpointing

Given the importance of resilience for extreme scale computing, there has been a large body of work on scalable fault tolerance techniques for HPC, CR being the most popular. However, CR has evolved through recent years in order to overcome important bottlenecks and leverage new storage devices. Multi-level checkpointing is today the state-of-the-art technique for fault tolerance at large scale.

In this paper we focus on the FTI multi-level checkpoint library. FTI provides efficient multi-level checkpointing in large scale supercomputers. It leverages local storage plus data replication and erasure codes to provide several levels of reliability and performance [12]. There are a total of four checkpoint levels: i) L1: Local checkpoint; ii) L2: Partner copy; iii) L3: Reed-Solomon (erasure code); and iv) L4: Parallel File System (PFS). Checkpoints can be taken either manually (*FTI_Checkpoint*) or periodically (*FTI_Snapshot*).

The FTI library has been evaluated using CPU and CPU/GPU systems [5], [13]. However, it has not been adapted to work on CPU/FPGA heterogeneous systems. Given the raising popularity of FPGAs for computational workloads, knowing how such a library performs with these devices is useful to estimate the overhead one can expect from such fault tolerance solutions on a certain class of applications. The aim of this work is to analyze and evaluate multi-level checkpointing on CPU/FPGA heterogeneous systems.

It is important to highlight that the work presented in this paper is not limited to FTI, but applicable to other multi-level checkpointing libraries, such as SCR [14] and Veloc [15]. By analyzing the performance of FPGA applications with the different checkpoint levels that the FTI library offers, we can also have a reasonable approximation of what performance other multi-level checkpointing libraries can achieve, due to being based on similar principles. Of course, library specific testing is needed for a more precise analysis of their behavior.

III. Related Work

A. FPGA-powered HPC

The ExaNoDe project [16] develops technologies for compute nodes leading towards Exascale capability. The node comprises a multi-chip module composed of ARM-v8 cores, low power processors, CNN accelerators, FPGAs and thermal and power management. Their objective is to achieve energy efficiency, scalability, heterogeneity and specialization. The EuroEXA project [17] brings the research and technology from other HPC projects [16], [18] and has the objective of co-designing an exascale-capable platform with peak performance of 400 PFLOPS with a peak power consumption of 30MW, over four times the performance and over four times the energy efficiency of current HPC platforms.

B. FPGA Checkpointing studies

Several checkpointing strategies have been proposed in the literature. A tree based checkpointing architecture called CPRTree [19] was proposed. Their approach was based on Hardware Description Language (HDL) and consisted in saving and restoring the state of all elements that define the context (registers, RAM and wires). Another study [20] evaluates the hazards of interrupting a running FPGA task by building a context saving and restoring simulator. The authors create several basic designs that contain most sources of interruption hazards and simulate task interruption and identify the hazards in each of those designs. To solve the

identified issues, the authors propose the use of Task Interruption (TI) wrappers along with a TI controller that controls their behavior. Both approaches, rely on some low level context saving that includes all the aspects of the FPGA tasks involved. However, application-level checkpointing has shown that many HPC applications need to checkpoint less than 50% of the application state in order to recover successfully [5].

Another study [21] focuses on the usage of FPGA soft processors in real-time applications. The proposed approach involves scanning the configuration memory periodically to detect and correct transient faults along with checkpointing for fault recovery. The experiments showed that the checkpoint frequency has a great impact on the task deadline fulfillment. This is a known result in the HPC resilience literature, which led to the development of the optimal checkpoint interval [22], [23].

A minimally-invasive monitoring infrastructure for message-passing computing systems is presented in [24]. This infrastructure enables monitoring and has the ability to request status, context retrieval and issuing of checkpoint/restart commands of FPGA worker nodes. What all these studies have in common is that they tackle the resiliency challenge at the logic/hardware description level. Because of that, they require the FPGA developer to use a hardware description language as their programming framework. In this paper, we instead tackle the problem at a high level using Vivado HLS and OmpSs@FPGA combined with a multi-level checkpoint library that is easy to link and use. This way, developers programming in high-level languages such as C/C++ can also obtain the advantages of fault tolerance and resiliency, without the burden of implementing reliability at the hardware description level.

C. Other accelerator checkponting studies

Sudarsun Kannan et al. presented a novel approach for CPU-GPU checkpointing [25], which addresses end-to-end data movement from the GPU to persistent storage, in order to tackle all aspects of the data transfer. Another study [13] proposes the integration of GPU checkpointing in the Fault Tolerance Interface library, using generic API calls not dependent on the location of the data. The main difference between these GPU checkpointing techniques and our approach on FPGA checkpointing is that we propose a technique for checkpointing partial progress of the FPGA task while it is currently running, as opposed to the cited GPU techniques which rely on CPU loop iterations of GPU kernel executions (referred in this paper as host-based checkpointing).

IV. Methodology

In this section we will explain the methodology which we followed to implement FPGA checkpointing. The work is separated into two different main approaches when it comes to implementing FTI functionality: host-based checkpointing and checkpointing of partial data from the FPGA task.

A. Host-based Checkpointing

As the name implies, this implementation approach is completely transparent to the FPGA, and is only done from the host side, in between FPGA tasks execution. It is the most straightforward implementation, where the FTI annotations are usually put in the main loop and keep track of the iteration number and the data needed to recover the execution upon a failure. This is the ideal method when the FPGA tasks do not take a long time to execute. This implementation is also most useful when the FPGA tasks are repeated several times, in which case it gives the user enough flexibility for setting the checkpoint frequencies.

```
#pragma omp target device(fpga) copy_in([1]sizex, [1]sizey) \
        copy_out([1]error)
#pragma omp task in([MAX_BSIZE*MAX_BSIZE]h_mem, [1]sizex, [1]sizey) \
        inout([MAX_BSIZE*MAX_BSIZE]g_mem, [1]error)
void jacobi_fpga(float g_mem[MAX_BSIZE*MAX_BSIZE], float h_mem[MAX_BSIZE*MAX_BSIZE],
                                        float error[1], int sizex[1], int sizey[1]){
        int nbLines = *sizex;
        int M = *sizey;
        float localerror = 0.0f, diff;
        float h[(FPGA_BLOCK_ROWS+2)*362];
        float g[(FPGA_BLOCK_ROWS+2)*362];
        int nb = nbLines/FPGA_BLOCK_ROWS + 1;
        for(int b = 0; b < nb; b++){
                int lines = ((b+1)*FPGA_BLOCK_ROWS < nbLines) ?
                                FPGA_BLOCK_ROWS : nbLines-(b*FPGA_BLOCK_ROWS);
                if(b > 0) lines += 2;
                int start = (b == 0) ? 0 : b*M*FPGA_BLOCK_ROWS - 2*M;
                int size = lines*M*sizeof(float);
                memcpy(h, &h_mem[start], size);
                memcpy(g, &g_mem[start], size);
                for(int i = 1; i < (lines-1); i++){
                        for(int j = 1; j < (M-1); j++){
                                g[(i*M)+j] = 0.25*(h[((i-1)*M)+j]+h[((i+1)*M)+j]+
                                                h[(i*M)+j-1]+h[(i*M)+j+1]);
                                diff = g[(i*M)+j] - h[(i*M)+j];
                                localerror += diff * diff;
                        }
                }
                memcpy(&g_mem[start], g, size);
        }
        *error = localerror;
}
```

Fig. 1: Implementation of the FPGA task for the Jacobi Solver

1) Add FTI support: The change mainly consisted in protecting the variables necessary for the recovery of the execution in case of failure, and adding the FTI annotations inside the main loop to decide when it is time to perform a checkpoint or recovery. The important data is protected by using `FTI_Protect`, which will be recovered in the case of a failed execution. Note that in addition to the main datasets, we also need to protect the iteration number since it is crucial to properly resume the execution.

2) Host-Based Checkpointing Task Implementation: Fig. 1 shows the C implementation of the FPGA task for a Jacobi Solver application used for host-based checkpointing. Since in this first approach checkpoints are performed inside the main loop of the application, the FPGA task does not contain any checkpoint specific logic inside.

In the implementation, the host copies all the matrix data to DRAM and calls the FPGA task. The FPGA task gradually copies the data from the DRAM to local variables (which are stored in the BRAM), and performs the computation. The task also copies the important data back to memory so that the host can gather the results after the task has finished.

B. Checkpointing partial work of the FPGA task

The intuition of this alternate implementation is to be able to checkpoint the progress of the FPGA task at certain points of its execution, without the need for the FPGA task to send its intermediate results to the host. Note that checkpointing inside the FPGA task is not necessary for all applications, however,

this technique could prove useful when used on FPGA tasks which run for a long time before returning the final result.

1) Adapting application to handle partial checkpointing: Because the host is unable to access the Block RAM of the FPGA, every data transmission or communication has to be done through the DRAM memory. The first approach we took was to make the FPGA task and the host synchronize and communicate when a checkpoint is needed. That is, after a certain amount of work the FPGA would copy the work back to DRAM and set a flag to tell the host that data is ready to be checkpointed.

The host would then copy the data that is relevant to host memory and perform a `FTI_Checkpoint`. While the host is copying the data and performing the checkpoint, the FPGA waits to avoid overwriting data and leaving the checkpoint in an inconsistent state. When the host finishes, it tells the FPGA task to resume its execution by setting another flag in DRAM.

However, for performance objectives, it is better to avoid synchronizations whenever possible.

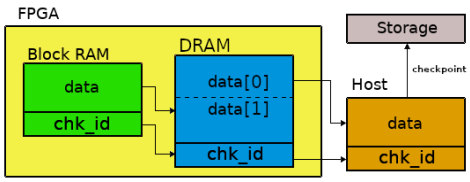

Fig. 2: Data movement of the asynchronous partial checkpointing implementation

Thus, the next step was to implement a modified version of the previous approach which does not need synchronization between the host and the FPGA task. In this implementation, the host checks the progress of the FPGA in DRAM while the FPGA task is constantly running.

The idea behind this new approach is the following: if the FPGA task has started computing the data for checkpoint *i*, the data for checkpoint *i-1* is ready to be copied. In order to have a secure way to copy data without risking overrides, some structures will need to be replicated. An asynchronous double buffering strategy in the DRAM suffices, as shown in Fig. 2. Given that checkpoint intervals are always much larger than the checkpoint cost (otherwise the overhead will be too high), a double buffering is enough to avoid data overwrites. By using Young [22] and Daly [23] formula to obtain the optimal checkpoint period with a reasonable MTBF (e.g., 12 hours for petascale supercomputers), it results in a much larger checkpoint interval than checkpoint cost, often over an order of magnitude. An order of magnitude is enough even under scenarios of CPU scheduling variance, cache/memory latency and CPU/FPGA communication.

2) Partial work Checkpointing Task Implementation: As can be seen in Fig. 3, the implementation of the FPGA task for the checkpointing of partial work adds the required extra logic to handle the checkpointing and the recovery of the FPGA task progress. If *param[0]* is found to be bigger than zero, it means that the FPGA task is resuming from an execution failure. In that case, it recovers the local error and the block iteration and continues the computation from there. Otherwise, it is assumed to be a normal execution and the FPGA task starts from the beginning. The recovered matrices are already given to the FPGA task by the host, and so does not require additional logic. Unlike in the previous version, the local error needs to be copied to memory periodically to tell the host that there is a new checkpoint procedure available. Since the host has to be actively waiting for changes in memory, a *taskwait* pragma does not suffice as a way to know that the task has finished, and so *param[1]* is used instead.

```
#pragma omp target device(fpga) copy_in([1]sizex, [1]sizey)
#pragma omp task in([MAX_BSIZE*MAX_BSIZE]h_mem, [1]sizex, [1]sizey) \
        inout([MAX_BSIZE*MAX_BSIZE]g_mem, [MAX_BSIZE]error, [2]params)
void jacobi_fpga(float g_mem[MAX_BSIZE*MAX_BSIZE], float h_mem[MAX_BSIZE*MAX_BSIZE],
                 float error[MAX_BSIZE], int sizex[1], int sizey[1], int params[2]){
        int nbLines = *sizex;
        int M = *sizey;
        float localerror = 0.0f, diff;
        float h[(FPGA_BLOCK_ROWS+2)*362];
        float g[(FPGA_BLOCK_ROWS+2)*362];
        int nb = nbLines/FPGA_BLOCK_ROWS + 1;
        //if recovery is available, set localerror to the last protected block number
        if(params[0] > 0){
                localerror = error[params[0]-1];
        }
        for(int b = params[0]; b < nb; b++){
                params[0] = b; //update block number to global memory
                int lines = ((b+1)*FPGA_BLOCK_ROWS < nbLines) ?
                                FPGA_BLOCK_ROWS : nbLines-(b*FPGA_BLOCK_ROWS);
                if(b > 0) lines += 2;
                int start = (b == 0) ? 0 : b*M*FPGA_BLOCK_ROWS - 2*M;
                int size = lines*M*sizeof(float);
                memcpy(h, &h_mem[start], size);
                memcpy(g, &g_mem[start], size);
                for(int i = 1; i < (lines-1); i++){
                        for(int j = 1; j < (M-1); j++){
                                g[(i*M)+j] = 0.25*(h[((i-1)*M)+j]+h[((i+1)*M)+j]+
                                                   h[(i*M)+j-1]+h[(i*M)+j+1]);
                                diff = g[(i*M)+j] - h[(i*M)+j];
                                localerror += diff * diff;
                        }
                }
                memcpy(&g_mem[start], g, size);
                error[b] = localerror;
        }
        params[1] = 1; //notify finish
}
```

Fig. 3: Adaptation of the FPGA task to support partial work checkpointing

Additional code in the host is needed to receive the data from the FPGA task. When the host detects a change in the checkpoint number in DRAM, it proceeds to copy the necessary data that the FTI library needs. The `FTI_Snapshot` function is then called, which performs a checkpoint if deemed necessary.

V. Evaluation

A. Environment

1) Hardware & Libraries: We performed the experiments on a cluster of 4 Xilinx Zynq-7000 SoC nodes. Each node is composed of a 32bit 2-core ARM processor with 1GB of DDR3 memory with an integrated Xilinx FPGA device. The FPGA device has the following specifications [26]:

- Logic Cells: 85K
- Block RAM: 4.9Mb
- DSP Slices: 220
- Maximum I/O Pins: 200

We used version 2.2.0 of OmpSs@FPGA environment, along with Vivado Tools 2017.3 for the synthesis and generation of the FPGA bitstream. The FTI version used was 1.4.1.

2) Applications: In this section we explain the applications that were used for our evaluation.

i) K-means clustering [27]: iterative algorithm that assigns n observations into k clusters, k being a fixed parameter. The procedure is the following:

- k initial means are generated
- k clusters are generated from the observations based on the nearest mean
- The centroid of each cluster becomes the new mean
- Process is repeated until it converges

ii) Jacobi Solver for heat equation: consists in an iterative propagation of the heat represented as a 2D matrix. For every iteration, the average of the 4 neighbor positions (up, down, left, right) is assigned to every position (excluding the matrix borders). The algorithms stops when reaching a small enough gradient or when reaching a maximum number of iterations.

iii) YUV Filter: consists in applying a YUV based filter to an image given as input. The steps are the following:

- Convert RGB image to YUV format
- Apply YUV filter to image
- Convert image back to RGB format
- Perform these steps a fixed amount of times

B. Host-Based Checkpointing

1) K-Means: Two versions of the application were evaluated, the first using a ramdisk as the location for the checkpoint and the second one using a drive mounted by the Network File System (NFS). These experiments are situated on the opposite sides of the performance versus fault coverage tradeoff space. Taking the checkpoint to ramdisk is fast and thus high performance but since ramdisk is volatile on-device memory, this option does not provide fault coverage for FPGA node faults, but rather just for detected unrecoverable faults such as a double bit flip on the SECDED ECC implementations that are found in modern FPGAs. On the other hand, taking the checkpoint to NFS is slower but provides coverage for FPGA node faults. In both versions, the application performs a single checkpoint at approximately the middle of the execution. Each version is executed a total of 20 times, and we evaluate the overhead against the non-FTI execution.

The input used for the execution was 43000 points of 1024 dimensions each, stored in a file (173MB). The variables to be protected by FTI were the centroids, the labels, the current iteration, the current error and the error of the previous iteration. Taking this in consideration, the size of a L1 checkpoint is 0.20MB per process.

The average execution time of the ramdisk and NFS versions were 13.71 and 14.45 seconds respectively. Considering that the average execution time of the non-FTI version was 13.59 seconds, the ramdisk and NFS version had an overhead of 0.78% and 6.28% respectively. As expected, the overhead of writing the checkpoint in a ramdisk is much lower than when using a NFS drive. The FTI library having 4 levels of checkpointing can help mitigate different levels of performance and resilience. Note that these overheads are very low despite the short execution time of the application.

2) Jacobi Solver: For the Jacobi solver, we analysed the performance of the FTI library using checkpoint levels 2, 3 and 4 in separate experiments. For each experiment, we executed the application with FTI functionality for a total of 20 times. After the executions, we took the averages of each one and compared it to the execution of the application without FTI support. We interleaved the order of the executions to account for possible system slowdown over time. For this application, we set a precision limit of 0.005 and a maximum number of 35000 iterations, which made the executions run for almost 10 minutes each.

As mentioned previously, this application makes use of the `FTI_Snapshot` functionality instead of using a manual checkpoint. We set an interval of 5 minutes of the desired checkpoint level in each experiment in the FTI configuration file, while disabling the rest. According to Young [22] and Daly [23] formula, a checkpoint period of 5 minutes is the optimal checkpoint interval for a checkpoint cost of 1 second and a MTBF of 12 hours, which is a realistic MTBF for petascale supercomputers. In this case, the FTI library had to protect the iteration counter and both g and h matrices. The total size of a checkpoint is 0.26MB per process for this problem size. The results of the experiments are shown in Fig. 4. The lines represent the average execution time of each version. Note that Fig.4 and Fig.5 have the Y axis zoomed in, in order to emphasize the differences between each version. We observe that, on average, the overhead of `FTI_Snapshot` with the L4 checkpoint in this application is 0.16%.

The experiment with the L2 checkpoint, also known as partner copy, still shows a very low overhead of 0.25%. While the L2 checkpoint seems to have slightly higher overhead than L4, it is expected because of the extra communication and I/O usage from saving an extra checkpoint copy on another node.

As with the previous experiments, we can observe a slightly higher overhead of 0.32% by using the L3 checkpoint. This level of checkpointing consists in the generation and communication of Reed-Solomon erasure codes in order to tolerate failures in multiple nodes, so the slight overhead increase is due to the extra computational workload from the nodes and the increased communication needed for the computation. In this case we only use 4 nodes, so there is not much difference between writing in local storage vs the NFS. However, at large scale with many nodes writing at the same time, it is advisable to checkpoint in local storage (L1) instead of a NFS. All 3 experiments show very low overheads in comparison to the original application. Thus, we think this technique is a viable option for host-based checkpointing in FPGA applications.

3) YUV Filter: Just like with the previous application, the YUV filter was executed 20 times with FTI and another 5 times without FTI. The `FTI_Snapshot` configuration was also the same as the previous application. For this application, we set it to a total of 80000 iterations in order to have a long enough execution time. We set the L4 checkpoint interval to 5 minutes and disabled the rest of the checkpoint levels. The FTI library protected the iteration counter and the image data (in which the filters are applied). The total size of a checkpoint

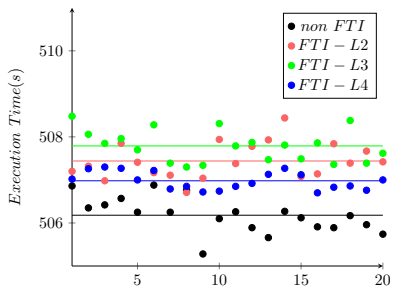

Fig. 4: Performance comparison of Jacobi Solver FTI implementation using L2/L3/L4 checkpoints

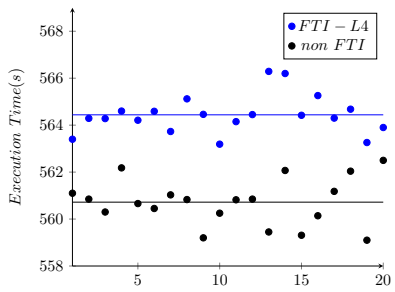

Fig. 5: Performance comparison of YUV Filter with and without FTI support

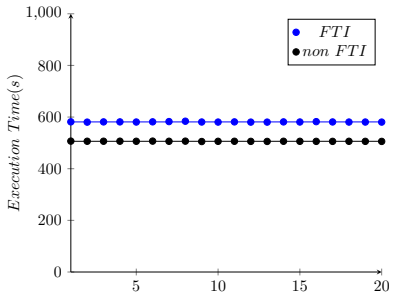

Fig. 6: Performance comparison of Jacobi Solver implementation with partial FPGA task checkpointing

is 0.16MB per process. The results are shown in Fig. 5. We can observe an overhead of 0.66% by using `FTI_Snapshot` and performing the L4 checkpoint. This is still a very low overhead.

C. Partial FPGA Task Checkpointing

1) Jacobi Solver: We ran the FTI version of the Jacobi Solver that checkpoints partial work of the FPGA task a total of 20 times, the same as the previous versions. In Fig. 6 we compare the original application with no FTI, versus this new version where the checkpoint is taken during the FPGA task execution. In this case, we can see an overhead of 14.82% by using this approach of FPGA checkpointing in comparison to the original non-FTI application.

While the overhead in this case is certainly larger than the host-based approach, it is worth pointing out that this application is not well suited to benefit from this method of checkpointing due to its FPGA task having short execution time. If the FPGA task finishes in a short period of time, it has enough granularity to benefit from host-based checkpointing instead, with lower overhead. A better application for evaluating this method would be one with an FPGA task that takes significantly longer to give a final result, so that a host-based checkpoint implementation does not have enough granularity to be effective. Moreover, despite non being a good candidate for partial task checkpointing, the overhead is still under 15%.

D. Resource overhead

In order to measure the impact of our implementations in the utilization of FPGA resources, we used Vivado to extract the resource usage of the 4 different components of the FPGA logic (BRAM, DSP, FF and LUT). We also measured the DRAM used by checking the size of all the function parameters, since we stored all the parameters in this memory.

The comparison of Jacobi Solver implementation without FTI and with partial checkpointing is shown in Table I. We did not include the Host-Based version since it uses the exact same design as the non-FTI. The Partial Checkpointing implementation requires extra logic for the recovery and copying of partial progress to the FPGA memory, so we can see overheads of 15.38% and 13.45% on LUT and FF usage respectively. In terms of BRAM, however, there is a low overhead of 2.00% due to not having the need to store more variables. In our particular case, our DRAM usage was not severely impacted by our implementation because we only needed to replicate the error variable. However, note that DRAM usage can experience up to a 2x overhead in a worst case scenario where all the parameters are in risk of being overwritten and need to be replicated.

TABLE I: Resource utilization comparison of Jacobi Solver

Resource	non-FTI	Partial Chkp. FTI
DSP	18(8.18%)	18(8.18%)
LUT	10454(19.65%)	12062(22.67%)
FF	13717(12.89%)	15562(14.63%)
BRAM	50(35.71%)	51(36.43%)
DRAM	1057052 B	1057112 B

VI. Conclusion

Host-based checkpointing with the FTI library provides an efficient way to give reliability to CPU/FPGA applications. The FTI_Snapshot functionality has been observed to have very low overhead even checkpointing at high frequency. Although our evaluation of the checkpointing of partial work turned out to have a higher overhead than the host-based checkpointing strategy, the application used was not fit for this kind of checkpointing given the short execution time of the FPGA tasks. As future work, this research could be complemented by a more extensive analysis of the checkpointing of partial work with other applications that benefit from that type of checkpointing, in order to evaluate the overhead in a more beneficial scenario. The paper could be further complemented by exploring the use of different on-chip memory hierarchies in larger FPGAs, such as UltraRAM on UltraScale+ Xilinx devices, to improve checkpointing performance.

VII. Acknowledgements

This research has received funding from the European Union's Horizon 2020 research and innovation programme under projects EuroEXA (grant agreement nº 754337) and eProcessor (grant agreement nº 956702).

REFERENCES

[1] "Vivado high-level synthesis," www.xilinx.com/hls.

[2] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, "Application acceleration on fpgas with ompss@fpga," in *International Conference on Field-Programmable Technology, FPT 2018, Naha, Okinawa, Japan, December 10-14, 2018*. IEEE, 2018, pp. 70–77. [Online]. Available: https://doi.org/10.1109/FPT.2018.00021

[3] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE computer society annual symposium on VLSI*. IEEE, 2010, pp. 288–293.

[4] "Xilinx all programmable devices: A superior platform for compute-intensive systems," https://www.xilinx.com/support/documentation/white_papers/wp492-compute-intensive-sys.pdf.

[5] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: high performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–32.

[6] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[7] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *International Conference on High Performance Computing for Computational Science*. Springer, 2010, pp. 1–25.

[8] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the" new normal"'for computer architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[9] "Legato: Low energy toolset for heterogeneous computing," https://legato-project.eu/.

[10] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[11] X. Chen and J. Sun, "Study and analysis of the high performance computing failures in china meteorological field," in *Journal of Geoscience and Environment Protection*, vol. 5, no. 12. SCIRP, 2017, pp. 28–40.

[12] "Components — legato," https://legato-project.eu/software/components.

[13] K. Parasyris, K. Keller, L. Bautista-Gomez, and O. Unsal, "Checkpoint restart support for heterogeneous hpc applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 242–251.

[14] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[15] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 911–920.

[16] "European exascale processor memory node design," https://exanode.eu/.

[17] "Innovative and scalable fpga-based system for extreme scale," https://euroexa.eu/.

[18] "European exascale system interconnect and storage," https://exanest.eu/.

[19] H. G. Vu, S. Kajkamhaeng, S. Takamaeda-Yamazaki, and Y. Nakashima, "Cprtree: A tree-based checkpointing architecture for heterogeneous fpga computing," in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2016, pp. 57–66.

[20] S. Attia and V. Betz, "Feel free to interrupt: Safe task stopping to enable fpga checkpointing and context switching," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 1, pp. 1–27, 2020.

[21] A. Sari, M. Psarakis, and D. Gizopoulos, "Combining checkpointing and scrubbing in fpga-based real-time systems," in *2013 IEEE 31st VLSI Test Symposium (VTS)*. IEEE, 2013, pp. 1–6.

[22] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[23] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future generation computer systems*, vol. 22, no. 3, pp. 303–312, 2006.

[24] A. G. Schmidt, B. Huang, R. Sass, and M. French, "Checkpoint/restart and beyond: Resilient high performance computing with fpgas," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 162–169.

[25] S. Kannan, N. Farooqui, A. Gavrilovska, and K. Schwan, "Heterocheckpoint: Efficient checkpointing for accelerator-based systems," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 738–743.

[26] "Xilinx zynq-7000 soc zc702 evaluation kit," https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html.

[27] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.