A Step toward stateful HW-SW migration: an architecture-agnostic checkpointing-rollback toolchain

MAXIME FRANCE-PILLOIS, The University of Edinburgh, United Kingdom
ZIHAN HUANG, The University of Edinburgh, United Kingdom
JIAXUN YANG, The University of Edinburgh, United Kingdom
EDSON HORTA, Virginia Tech, USA
BINOY RAVINDRAN, Virginia Tech, USA
ANTONIO BARBALACE, The University of Edinburgh, United Kingdom

FPGAs have become key players in data-centers. However, the integration of such accelerators poses several challenges related to Quality of Service (QoS). Herein we propose a compiler-based toolchain that increases FPGA flexibility by enabling dynamic stateful HW-SW migration. Task migration is instrumental in solving at least fault-tolerance and preemption in FPGA. Thus, we design, based on our toolchain, a checkpointing-rollback framework to enable restoring a task after a component failure (i.e., FPGA crash) and making a first step toward fault-tolerant data-center systems. Starting from the Xilinx HLS compilation workflow, we design 1) a set of LLVM optimization passes that instrument an FPGA-accelerated application with migration points, and 2) an asynchronous periodic data backup scheme for efficient context transfers. These together allow the FPGA-accelerated task to migrate statefully from the FPGA onto its host CPU where execution is resumed. We evaluate this proposal on several applications and show that, although reliability (inevitably) comes at a cost, our framework offers promising results by transforming a set of common benchmark kernels into *fault-tolerant* kernels with acceptable best-case runtime overheads (e.g., 1.1x for Gaussian Blur filter and MaxPool operation). We also show that FPGA task preemption allows HW-SW multitasking.

CCS Concepts: • **Software and its engineering → Compilers**; **Preprocessors**; **Runtime environments**.

Additional Key Words and Phrases: compiler-based HW-SW migration, Data-center QoS, Dependable HW acceleration

ACM Reference Format:
Maxime France-Pillois, Zihan Huang, Jiaxun Yang, Edson Horta, Binoy Ravindran, and Antonio Barbalace. 2026. A Step toward stateful HW-SW migration: an architecture-agnostic checkpointing-rollback toolchain. 1, 1 (February 2026), 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Data intensive workloads, such as deep learning and scientific applications, are amongst the most common applications in data-centers. Such workloads perform a reduced set of operations on large batches of data. To take advantage of this characteristic and reduce computing costs (i.e., $ per operation), many data-centers – such as cloud, edge, and HPC, incorporate heterogeneous

Authors' Contact Information: Maxime France-Pillois, maxime.france-pillois@club.fr, The University of Edinburgh, Edinburgh, United Kingdom; Zihan Huang, Z.Huang-31@sms.ed.ac.uk, The University of Edinburgh, Edinburgh, United Kingdom; Jiaxun Yang, jiaxun.yang@ed.ac.uk, The University of Edinburgh, Edinburgh, United Kingdom; Edson Horta, edson-horta@ieee.org, Virginia Tech, Blacksburg, VA, USA; Binoy Ravindran, binoy@vt.edu, Virginia Tech, Blacksburg, VA, USA; Antonio Barbalace, antonio.barbalace@ed.ac.uk, The University of Edinburgh, Edinburgh, United Kingdom.

processing elements, e.g., GPUs (Graphics Processing Units), FPGAs (Field Programmable Gate Arrays), Application-Specific Integrated Circuits (ASICs), etc. [41, 47] in their offer. Amongst others, FPGAs have demonstrated their ability to provide flexible and efficient computation [7, 17].

With the rising interest in incorporating efficient processing elements in data-centers, both industry and academia have made significant efforts to ease the integration and the usage of FPGAs. A significant step was the High-Level synthesis (HLS) tools that enable the generation of FPGA-compliant "executables" from high-level languages descriptions (C, C++, OpenCL, ...) [23, 43, 64]. As a result, many data-center providers integrate FPGA accelerators into their servers and adpot them as a "standard" service, also called FPGAs-as-a-Service [21, 44, 61, 72]. However, the integration of accelerators in data-centers raises several Quality of Service (QoS) related issues, such as fault tolerance and resilience, resource sharing and prioritization – not yet solved on FPGAs [9].

Checkpointing-rollback. Data-centers are multi-tenant environments, where hardware resources are securely shared between users. Multi-tenant systems increase the risk of security attacks, e.g., denial-of-service, which render FPGAs unusable, similar to the case of an FPGA hardware error [9, 12, 35, 40, 52]. Hardware component failures, resulting from security attacks or hardware errors, may occur anytime. Hence, to solve the issue of fault tolerance and resilience, the system must be robust against *unpredictable* failures such that the compute service always remains available. A common strategy to address this is the checkpoint-rollback paradigm: during the execution of an application, the system takes snapshots of the application's internal state and saves it outside the component executing it. If a component failure happens – e.g., an FPGA hardware error – the application can resume from the last valid snapshot and does not need to restart from scratch. This saves time and helps maintain the application running with minimal degradation.

The sharing of resources among multiple tenants revives the need for prioritization. An hardware task running on FPGA may have a lower priority than a newly spawned one, which may have to be executed immediately on the FPGA. The lower priority one could be killed or preempted. Checkpointing-rollback can also be used to instantly preempt a hardware task running on an FPGA, thus avoiding the loss of the already executed calculation.

Heterogeneous Checkpointing-rollback. A challenge of checkpointing-rollback is to save viable snapshots from which the application can safely resume, i.e., the input or output data must not be corrupted, and all the relevant information on the computing state must be recorded. In addition, taking snapshots should minimally impact the application execution time. *Hence, for performance and viability reasons, snapshots cannot be placed at any processing point.* Moreover, once an FPGA failure occurs or a preemption is triggered, the interrupted hardware task should be resumed as soon as possible to minimize service downtime – hence, providing fault tolerance and availability. However, resuming on the same or another FPGA may not be possible because of a lack of resources, or may require significant time, because of queuing, reset, and reprogramming (see Section 2). Therefore, unlike previous work [59], we are the first to propose the resumption (rollback) of the FPGA-accelerated task on its host CPU. This has the advantage of reducing data movements and competition for FPGA resources and providing minimum resume latency and downtime, but requires handling cross-architecture migrations between two *very different* architectures: CPU-based execution and custom hardware design tailored to user-defined applications.

We introduce ***HW-SW migration:*** a mechanism that allows moving a task from a hardware computation engine (FPGA) to software running on a processing unit (CPU) at fine granularity, *without losing the achieved intermediate computation progress.* HW-SW migration introduces the complex problem of migrating between FPGAs to CPUs that requires the identification of *equivalent processing points* at which an application state is transferable between architectures [6, 39, 56].

Fortunately, kernels running on FPGA have specific characteristics, such as idempotence[1], which make possible kernel migration from one architecture to another (see Section 4).

Contributions. We propose to increase the flexibility, reliability, and shareability of FPGAs through a compiler-based toolchain that inserts checkpoints at equivalence points between custom-hardware processing and CPU processing. Key contributions:

- An *open-source* compilation toolchain, based on LLVM, that automatically inserts checkpoints based on user directives and creates equivalence points to enable *cross-architecture live* task migration from FPGA to CPU. It requires *no source-code changes* and therefore supports legacy applications.
- A methodology to transform any offloaded task running on FPGA into a preemptible application via checkpoint-enabled FPGA to CPU task migrations. On top of that, we propose a system able to detect and mitigate Denial-of-Service attacks. Checkpoints are inserted in the offloaded kernel. Each time a checkpoint is successfully reached, the context of the task is sent to the host. When an FPGA crash is detected or a preemption is triggered, the host CPU resumes the task execution from the last valid received checkpoint.
- An evaluation of the proposed tool on real use-cases.

Outline. This paper is organized as follows: Section 2 further motivates this work; Section 3 discusses background and related work; Section 4 introduces our FPGA-CPU migration; Section 5 details the implementation of our work. Sections 6, 7 and 8 describe key use cases: fault-tolerance, preemption and modular redundancy; Section 9 presents and analyzes our experimental results. Section 10 discusses the work limitations, Section 11 concludes.

2 Motivational Use Cases

Fault Tolerance. FPGA failures can result from: organic hardware malfunctions [1], or external attacks [9, 12, 35, 40, 52]. Although the error rate of FPGAs is generally low (Intel FPGA claims a Mean Time to Failure of 22.83 years [19]), the error rate is highly dependent on the system environment [2], and radiation-induced soft errors can cause crashes every few hours in large-scale deployments [31]. Also, since cloud servers are multi-tenant, they are vulnerable to attacks from malicious actors [12, 29]. A body of work has demonstrated that FPGAs are prone to crashes triggered by either direct voltage-based attacks, or bitflips at specific locations [24, 34, 38]. La *et al.* [34] demonstrated that in a cloud environment, Denial-of-Service (DoS) attacks may disable (crash) an FPGA for several hours before a new instance can be started, leading to monetary losses and a poor customer experience.

In FPGA-acceleration, the loose coupling between host and accelerator does not allow the former to obtain information on the computation progress achieved by the latter, nor monitor the occurrence of failures. In many cases, the host CPU is unable to detect an FPGA crash, and will keep waiting for a reply from the crashed FPGA, causing the application, and even the host, to stall. For host CPUs that can detect and handle such faults, they must restart the entire accelerated task from scratch – including FPGA reset and reprogramming, which severely increases service latency. This issue is most detrimental to FPGA-accelerated applications with long runtimes such as DNA sequencing, where runtimes on large datasets can extend to thousands of minutes [36].

Previous work (see Section 3) has proposed FPGA to FPGA migration: this solves the issue of losing what has been computed so far. However, this may require that another identical FPGA is available in the same data center (see Figure 1a) or in the same machine (see Figure 1b). Then, in the first case, the full (hardware) state of the task must be moved from one machine to another, and

[1]The idempotence principle is that performing an operation multiple times should have the same effect as performing it once.

in the second case, users have to wait for FPGA to be available [9]. In both cases, reprogramming time will further slow down the task. Table 1 reports the reprogramming time necessary for a set of kernels we used in the evaluation. If no other FPGA to reprogram is available on the same machine – because not physically present or busy with other tasks – we may want to reset the the faulty FPGA. In the case of a Xilinx U50, the cost is about 4.8s in our setup (see Section 9). Moving the task to another server with available FPGA resources in the data center, requires a VM/container migration that may take seconds due to coordinations with orchestrators. Hence, this paper explores the possibility of restarting the FPGA task on CPU resources.

Table 1. Vanilla kernels' FPGA reprogramming time (ms)

Lud 1024	Lud 3072	Kmeans	Choles.	Blur	MiniMap	MaxPool
312	413	122	679	83	803	764

Fine-grain Sharing. The number of tasks that require FPGA acceleration in data centers may exceed the available FPGA hardware, creating potential resource contention [61]. Because of contention, FPGA tasks are queued for available hardware resources as FPGAs typically operate under a non-preemptive execution model and tasks must run to completion once started [15].

Such constraints may present difficulties when high-priority tasks arrive while the limited FPGA resources are occupied by longer-running, lower-priority workloads. For instance, scenarios involving long-running FPGA tasks such as machine learning training or genomic sequencing – these could potentially occupy scarce FPGA resources for extended periods, raising questions about how to balance resource allocation and system responsiveness. These considerations suggest exploring mechanisms for fine-grain FPGA sharing: preemption, addressing resource management challenges while maintaining efficient system operation. We believe that preemption may also enable "spot" FPGA instances in data centers, similar to spot VMs [70].

N-Modular redundancy. Safety-critical applications subject to stringent requirements, such as aerospace control systems[58], medical devices, or automotive safety functions[28]—require guaranteed error-free execution. These applications commonly employ N-modular redundancy (NMR) [32], where N instances of the same task execute simultaneously with continuous cross-validation. For the highest criticality levels, safety standards mandate implementation diversity, requiring each redundant instance use different approaches or heterogeneous hardware to avoid common-mode failures[27]. Silent data corruption (SDC) represents a critical failure mode in FPGAs, where computations complete without error detection but produce incorrect results. Large-scale studies show that SDC occurs in 60-90% of FPGA soft errors [31]. Unlike fail-stop failures that trigger protective responses, SDC undermines safety standards by allowing corrupted outputs to propagate undetected. In most FPGA-accelerated computation scenarios, the CPUs are paired with FPGAs in the system to act as the control plane. CPU-based computation also exhibits lower failure rates compared to FPGAs [20, 31], making the CPU a perfect candidate to act as a golden reference model and backup execution module.

3 Background and Related Work

Common FPGA architectures in data-centers are: a server connected to an FPGA (e.g. Microsoft Catapult v2 [13]), or a server connected to multiple FPGAs (e.g. AWS F1 [47]), see Figure 1. The choice of architecture is often dictated by its cost-effectiveness [9], or application requirements [53].

3.1 Compiler-based Checkpointing

To address cross-architecture migration, we propose a checkpoint-based approach. Periodically taking a snapshot of the current execution state of an application, and storing it in memory (i.e., creating a checkpoint) in order to restore and replay the execution from that point in the event of a failure, is a well-established technique. An extensive body of work exists on that topic. Among those, some proposals have taken advantage of compiler passes to create checkpoints. Zhao *et al.* [73] propose a software checkpointing framework. Users can specify checkpoint regions, then use compiler-driven transformations to instrument all relevant write/store operations with function calls to back up the values of the stored variables to memory, and insert special functions to handle system functions with implicit memory writes (e.g., `memcpy`). The backed-up data is stored in a dynamically-sizable checkpoint buffer in main memory. Choi *et al.* [16] propose Bolt, a compiler-based soft error recovery scheme that enables error recovery on transient faults via re-execution of errored code regions. Bolt partitions and transforms the program code into idempotent regions, and checkpoints each region at the region boundaries. While these proposals settle important bases for checkpointing – code instrumentation and idempotent-region partitioning – they are not sufficient to create the architecture-agnostic equivalence points that are needed for stateful HW-SW migration. Moreover, these solutions do not target FPGAs, which require special care regarding the way the computation state is captured.

3.2 Checkpointing an FPGA Kernel

Extensive literature has presented the opportunities to preempt and restore hardware tasks running on FPGAs [3, 11, 33, 45, 50]. Former solutions benefit from the read-back bitstream feature offered by many FPGA providers which snapshots the FPGA's state and reloads it when required [50]. This method has a few main drawbacks: 1) the internal state of BRAMs and DSP blocks cannot be saved or restored [4], 2) it is target dependant because a bitstream is only suitable for one target device, and 3) the bitstream read-back feature is very slow [33]. To solve the partial checkpoint problem, Attia *et al.* propose an extended multi-cycle hardware checkpoint mechanism that reveals the state embedded in BRAM or DPS blocks [3, 4]. Other works [11, 33, 45, 57] have preferred to perform selective context saving using checkpoint based alternatives. In these solutions, the useful (i.e., live) context is stored using additional hardware, like a scan chain [11, 33].

While these proposals enable a hardware task to stop and resume on the same FPGA, they do not support moving tasks from one FPGA to another. More recent papers addressed this issue and allow the preempted task to be migrated from one FPGA to another [55, 59]. Notably, in [55] the authors rely on the high-level OpenCL programming framework to move a task during its execution instead of a low-level hardware approach. They designed a full FPGA Manager engine that can allocate FPGA slots and relocate tasks from one FPGA to another. Also, this proposal does not store the live context of the task. Moreover, it relies on the OpenCL work item breakdown scheme to re-submit uncompleted work items to the new FPGA, which limits its applicability to scheme-compatible tasks only.

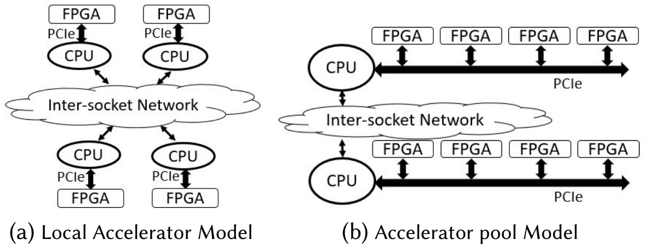
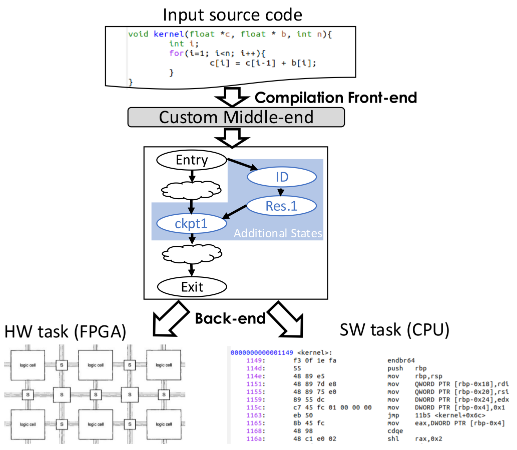


Fig. 1. Example of FPGA linking in datacenters.

4 Fine-grained live HW/SW task migration

Our proposed migration from a hardware-based FPGA task to a software task on a CPU avoids the limitations of previous works. It allows a task to resume its execution directly and quickly on its nearby host CPU in the event of a crash or preemption or a silent-fault; data need not be transferred wholesale between nodes, and other FPGAs need not be overloaded with more tasks.



```
void kernel(float *c, float * b, int n){
        int i;
        for(i=1; i<n; i++){
                c[i] = c[i-1] + b[i];
        }
}
```

```
0000000000001149 <kernel>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   rbp
    114e:       48 89 e5                mov    rbp,rsp
    1151:       48 89 7d e8             mov    QWORD PTR [rbp-0x18],rdi
    1155:       48 89 75 e0             mov    QWORD PTR [rbp-0x20],rsi
    1159:       89 55 dc                mov    DWORD PTR [rbp-0x24],edx
    115c:       c7 45 fc 01 00 00 00    mov    DWORD PTR [rbp-0x4],0x1
    1163:       eb 50                   jmp    11b5 <kernel+0x6c>
    1165:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
    1168:       48 98                   cdqe
    116a:       48 c1 e0 02             shl    rax,0x2
```

Fig. 2. Single input source approach for explicit checkpointing

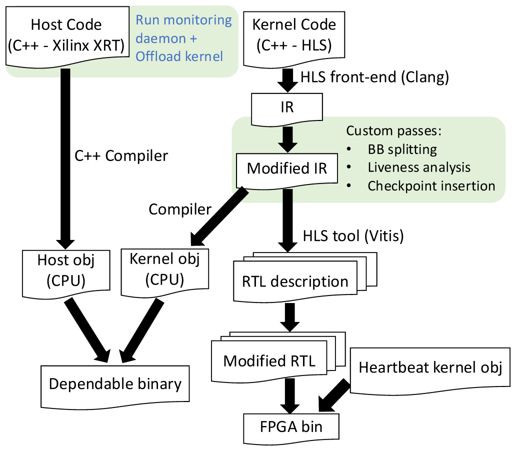


Fig. 3. Toolchain overview. Green background highlight the custom step we have added or modified

4.1 Hardware/Software migration challenges

The usual workflow for an offloaded kernel is: 1) read data inputs, 2) *progressively* compute the outputs based on these inputs and any intermediate computational states, and finally 3) commit the final computed outputs. Thus, to migrate a task between processing engines without losing the achieved progress requires moving *both* the task description (e.g., code) and the state of the computation, i.e., the execution context.

Since execution contexts are architecture-specific, some states are complex to translate from one processing engine to another (e.g., from x86 to ARM) [5, 6, 42, 69]. Consequently, in literature, cross-platform migration can only occur at specific, well-chosen points. For instance, Barbalace *et al.* [6] only allows heterogeneous-ISA (Instruction Set Architecture) migration at predefined points in the code. In our case, the micro-architectures of a hardware kernel running on an FPGA differ significantly from a CPU-based kernel execution. In CPU, execution context is held by $T_i =< L_i, S_i, R_i, H_i >$ with L_i the Local storage data, S_i the user-space stack, R_i the user-space visible state of the CPU (i.e., registers) and H_i the Heap data space [6]. In contrast, for FPGA custom hardware, the context is spread across logic registers, BRAMs, and other FPGA internal memories.. Clearly, the translation is non-trivial.

Previous work circumvented this issue by either restricting migration to function boundaries only [25], or targeting specific workloads that restrict the context to OpenCL work-item indexes to be able to re-dispatch the remaining work-items to a different target[55]. However, none of these are suitable for generic and dependable high-QoS systems. The first method does not enable fine-grain migration: in the event of a failure, the whole function must be replayed on the new computation engine, which can incur a significant rollback delay. The second method only applies to workloads that can be split into OpenCL work-items, rendering it dependent on the OpenCL programming model and inapplicable to legacy applications without code refactoring.

Unlike these approaches, we devise a solution that enables the fine-grained migration by creating on-demand migration points — called equivalence points — in the intermediate representation of an application. These equivalence points are created via custom subroutines inserted into a task. These subroutines: 1) capture the relevant live context of the task and 2) enable migration.

4.2 Building explicit equivalence points

FPGA kernels have two characteristics enabling the extraction of consistent task contexts with extra routines: 1) those are enclaved within the target and do not modify the global memory or access peripherals on their own. 2) Outputs are often written in the FPGA buffer and then copied to the central memory on the host's request. This execution model ensures that outputs are only committed on demand.

These characteristics allow us to ensure idempotent processing for offloaded kernels. Hence, by managing partial output committing, we can guarantee the consistency of the system when restarting the function and reloading its context on another processing engine.

Listing 1. Checkpoint BB

```
bb.ckpt1:
store %ID, memCkpt[0]
store %index, memCkpt[1]
store %matA, memCkpt[.]
...
%bb.ckpt1_end:
```

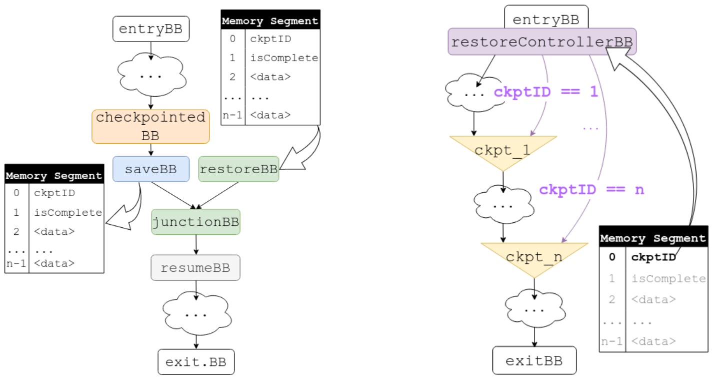
Listing 2. Restore BB

```
bb.restore1:
%ID = load memCkpt[0]
%index = load memCkpt[1]
%matA = load memCkpt[.]
...
br label %bb.ckpt1_end
```

Our framework adds custom sub-routines – checkpoints, into the kernel workflow. These checkpoints explicitly extract the live context of the task and store it into a dedicated memory area. Listing 1 shows a pseudo-code of the checkpointing process. The context of a task at a specific checkpointing point comprises all the live variables at that point. A variable is “live" at a point if it holds a value that will be used in the future. The restoration process consists of reading the task context from the checkpoint memory area and reloading live variables with their previous value, as shown in Listing 2. Unlike software, hardware does not have the notion of function calls (stack) and heap. This causes some restrictions when restoring the task onto the software (CPU) side: We must either reconstruct a consistent stack or ensure that checkpoint/restoring processes only occur at the first level of the function call tree. Our design employs the latter.

However, the main challenge remains: given the significant differences between the CPU-based software version of a task and its FPGA-based hardware version, there is no guarantee of finding natural equivalence points between them. Indeed, target-specific optimization passes on both sides can refactor the code and remove variables. For example, if the HLS tools implement a loop as an n-stage pipeline, the loop index as it appears in the software source code will not exist in the hardware task implementation [18].

We overcome this by designing our framework to use a single *shared* task representation. We leverage HLS tools to specify a single high-level language source code for both software (CPU) and hardware (FPGA) implementations of the task, into which we insert our dedicated sub-routines before any target-specific optimizations and compilations are performed. Figure 2 shows this methodology: Our single input source code (e.g., a C/C++ description of the task) is lowered to an Intermediate Representation (IR) by the compiler front-end. At this level, we perform analyses and code transformations to insert our custom sub-routines. Finally, the modified IR is sent to the target-specific compiler back-ends to generate separate executable files for CPU and FPGA. These



(a) Inserted Checkpoint BBs (b) Restore process BB

Fig. 4. Illustration of added BBs in the initial task CFG

executable files both implement the same explicit checkpoints at the same code locations, ensuring structural equivalence at such points.

4.3 A compilation-based checkpointing mechanism

For context extraction, our checkpoints must know all the live variables at the checkpoint locations. For ease of usage, checkpoints are inserted by the compiler. We perform this at IR level due to 1) the code analysis and transformation steps required (liveness analysis, etc.) and 2) the fact that it is the lowest task representation shared between hardware and software implementations.

Figure 3 shows the compilation toolchain, and highlights our proposed enhancements in green. It consumes two input files: the offloaded kernel's source code (right), and the host's code that manages the offloading of the kernel onto the target (left). The compilation is extended to automatic checkpoint/rollback code insertion in three steps: Basic Block (BB) splitting, liveness analysis, and (actual) checkpoint insertion, detailed in Section 5. With these steps, users need only to specify the checkpointing locations, and the compilation process takes care of outputting sound binaries for CPU and FPGA.

5 Implementation

Our toolchain is built upon the open-source Vitis HLS toolchain incorporating the LLVM-7[2] framework [63].

5.1 Custom Compilation Passes

Checkpoints are implemented by inserting new subroutines – new instructions and BBs – into the kernel IR at the chosen checkpoint locations. Figure 4 illustrates the insertion of these BBs into the initial Control-Flow Graph (CFG) of the task. The `save` and `restore` subroutines are inserted at each checkpoint (see Figure 4a). The `restoreManager` subroutine is inserted within the function's entry block (`entry BB`) to control which checkpoint to restore execution to, based on a checkpoint ID (`CkptID`) stored in the checkpoint memory segment (Figure 4b).

A 3-pass Compiler Support. We design our compiler support as a 3-pass compiler flow. The kernel code is first compiled by *clang++* into LLVM IR. The 3-pass flow consumes this IR and

[2]Our custom compilation passes are also LLVM-14 compatible.

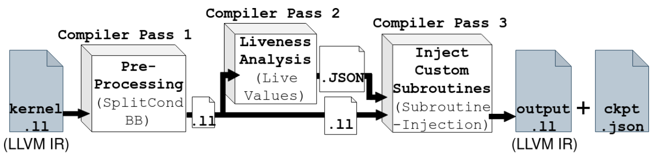

Compiler Pass 1
Compiler Pass 2
Compiler Pass 3
kernel .ll
(LLVM IR)
Pre-Processing (SplitCond BB)
.ll
Liveness Analysis (Live Values)
.JSON
.ll
Inject Custom Subroutines (Subroutine-Injection)
output .ll
(LLVM IR)
+
ckpt .json

Fig. 5. 3-pass Compiler Workflow

outputs the final checkpointed kernel IR, which can be lowered into binary for CPU execution, or synthesized into hardware bitstreams for FPGA execution. The stages of the 3-pass flow are shown in Figure 5, and are as follows:

(1) `SplitConditionalBB` *(IR pre-processing)*. Perform per-`Function` pre-processing transformations on an unmodified user's kernel IR file to facilitate subsequent subroutine injection. It extracts single-successor `BB`s from N-successor `BB`s by splitting before the terminator, allowing simpler subsequent insertions of checkpoint/restore `BB`s.
(2) `LiveValues` *(liveness analysis)*. Perform per-`Function` liveness analysis for each `BB` in the pre-processed IR `Module`. The results of the analysis are outputted as JSON files.
(3) `SubroutineInjection` *(checkpoint insertion)*. Consume the analysis data from `LiveValues`, and perform `Module`-level transformations on the pre-processed IR. The checkpoint locations are identified by locating user-directed `checkpoint()` directives in the kernel. This pass outputs the final checkpointed IR, and a metadata file used by the host to allocate the `ckpt_mem_seg` memory segment where checkpointed data is stored.

Inter-Pass Communication. A challenge in these passes is the transfer of data between `LiveValues` (`FunctionPass`) and `SubroutineInjection` (`ModulePass`). Each LLVM pass has a private memory space that is inaccessible from other passes; data transfer between passes must be mediated by the LLVM framework. The open-source Vitis HLS toolchain uses a legacy LLVM-7 framework that does not allow such data transfer between `FunctionPass`es and `ModulePass`es (unless in special situations). Although LLVM-7 allows such data transfer between passes of the same type, `LiveValues` uses dependencies that make it difficult to modify it into a `ModulePass`, and `SubroutineInjection` must be a `ModulePass` to perform `Module`-wide analysis and transformations. We therefore opt to perform this data transfer using an external JSON file.
Subroutine Injection Process. Listing 3 shows the algorithm to insert and populate checkpoint and restore-related BBs. It first performs checkpoint selection via `get_checkpointedBBs()`, then inserts a `restoreControllerBB`. If successful, it processes each checkpointed `BB` in turn, where it inserts a `saveBB`, `restoreBB` and `junctionBB` trio. Finally, for each tracked variable in the checkpointed `BB`, it computes its index position in `ckpt_mem_seg`, populates the `saveBB`, `restoreBB` and `junctionBB` with the appropriate instructions, and (if appropriate) propagates the `PHI` result from `junctionBB` across the `CFG` to maintain the IR's `SSA` form.
Type Conversions for Mixed Datatype Checkpoints. To simplify data saving and restoring, we adopt a conservative storing strategy that formats the checkpoint memory segment into the most memory-demanding data type. For example, if the context is composed of float and char variables, the checkpoint memory segment will be of float type, thus avoiding any memory segmentation issues. This also simplifies the calculation of the addresses (i.e., offsets) for the variables, as each variable occupies the same amount of memory. For cases where checkpointed variables are not all of `ckpt_mem_type`, we insert type conversion instructions (e.g. `sitofp`, `fptosi`) before storing into and after loading from `ckpt_mem_seg`.

Listing 3. Checkpoint Insertion Algorithm

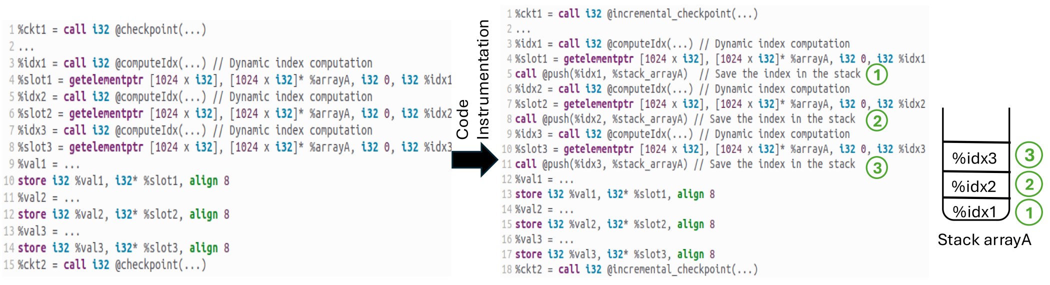
```
for Func in Module do
    checkpointed_BBs = get_checkpointedBBs(Func);
    if (checkpointed_BBs.isEmpty()) do return;

    success_rc = insert_restoreControllerBB();
    if (!success_rc) do return;

    for BB in checkpointed_BBs do
        success_s = insert_saveBB(BB);
        if (!success_s) do continue;

        success_j = insert_junctionBB(BB);
        if (!success_j) do
            erase_inserted_BBs(BB);
            continue;
        insert_restoreBB(BB);

        tracked_vars = get_BB_tracked_vars(BB);
        for tracked_var in tracked_vars do
            compute_ckpt_mem_idx(tracked_var);
            savedVar = populate_saveBB(tracked_var);
            restVar = populate_restoreBB(tracked_var);
            newVar = add_PHI_juncBB(savedVar, restVar);
            if (newVar is not array ptr) do
                propagate(newVar);
    populate_restoreControllerBB();
    assign_global_ckpt_IDs();
```

Listing 4. Checkpoint pseudo code

```
bb.checkpoint:
  n = 0
  // ArrayA is fully copied
  // into the checkpoint memory
  for elem in arrayA:
    store %elem, memCkpt[n++]
```

Listing 5. Incremental checkpoint

```
bb.incremental_checkpoint:
  n = 0
  // Only the modified element of ArrayA
  // are copied into the checkpoint memory
  while(!stack_arrayA.empty())
    %idx = call i32 @pop(stack_arrayA)
    store arrayA[idx], memCkpt[n++]
```



```
%ckt1 = call i32 @checkpoint(...)
...
%idx1 = call i32 @computeIdx(...) // Dynamic index computation
%slot1 = getelementptr [1024 x i32], [1024 x i32]* %arrayA, i32 0, i32 %idx1
%idx2 = call i32 @computeIdx(...) // Dynamic index computation
%slot2 = getelementptr [1024 x i32], [1024 x i32]* %arrayA, i32 0, i32 %idx2
%idx3 = call i32 @computeIdx(...) // Dynamic index computation
%slot3 = getelementptr [1024 x i32], [1024 x i32]* %arrayA, i32 0, i32 %idx3
%val1 = ...
store i32 %val1, i32* %slot1, align 8
%val2 = ...
store i32 %val2, i32* %slot2, align 8
%val3 = ...
store i32 %val3, i32* %slot3, align 8
%ckt2 = call i32 @checkpoint(...)
```

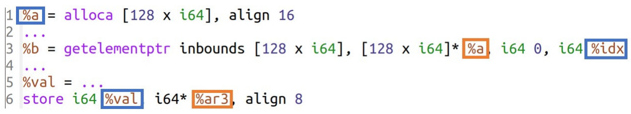
```
%ckt1 = call i32 @incremental_checkpoint(...)
...
%idx1 = call i32 @computeIdx(...) // Dynamic index computation
%slot1 = getelementptr [1024 x i32], [1024 x i32]* %arrayA, i32 0, i32 %idx1
call @push(%idx1, %stack_arrayA)  // Save the index in the stack
%idx2 = call i32 @computeIdx(...) // Dynamic index computation
%slot2 = getelementptr [1024 x i32], [1024 x i32]* %arrayA, i32 0, i32 %idx2
call @push(%idx2, %stack_arrayA) // Save the index in the stack
%idx3 = call i32 @computeIdx(...) // Dynamic index computation
%slot3 = getelementptr [1024 x i32], [1024 x i32]* %arrayA, i32 0, i32 %idx3
call @push(%idx3, %stack_arrayA) // Save the index in the stack
%val1 = ...
store i32 %val1, i32* %slot1, align 8
%val2 = ...
store i32 %val2, i32* %slot2, align 8
%val3 = ...
store i32 %val3, i32* %slot3, align 8
%ckt2 = call i32 @incremental_checkpoint(...)
```

(a) Original kernel code sample (b) Instrumented kernel code sample (c) Index stack

Fig. 6. Incremental checkpointing illustration. Green numbers show how indexes are pushed into stack.

Incremental Data Checkpointing. Since kernel's final computation results are usually computed incrementally during its execution, the difference between the contexts saved by two adjacent checkpoints is often small. We therefore perform *incremental checkpointing* — which only copies the context data that has been updated by the kernel since the previous checkpoint — to minimize costly accesses to the FPGA's external memory.

However, static analyses (such as the liveness analysis used) cannot determine which memory slots of arrays have been written if the indexes are defined dynamically at runtime. Even worse, static analyses struggle to calculate static indexes if the define-chain is too complex. As a result, only dynamic tracking of slot indexes that have been modified between two checkpoints can ensure the minimal amount of data has been checkpointed. We perform this dynamic tracking by instrumenting `store` instructions to arrays in the IR code as shown in Figure 6. Each time a `store` to an array occurs, the address offset is stored in a dedicated stack. Hence, for each `store` instruction, we extend the LLVM optimization pass to retrieve the base address of the array and the index by rewinding the SSA variable path as shown in Figure 7. Note that the base address is used to determine the stack on which the index should be pushed. Then, when performing the checkpoint, we pop the stack associated with an array and copy each corresponding array element to the checkpoint memory area as shown in Listing 5.



```
1 %a = alloca [128 x i64], align 16
2 ...
3 %b = getelementptr inbounds [128 x i64], [128 x i64]* %a, i64 0, i64 %idx
4 ...
5 %val = ...
6 store i64 %val, i64* %ar3, align 8
```

Fig. 7. Static `store` instruction analysis. Orange squares are intermediate variables that must be rewound. Blue squares designate the final values for our analysis: line 7 `%val` data value, line 3 `%idx` index value, line 1 `%a` array base address.

5.2 Enable fine-grain Host-Target Communications

Xilinx Low-Level Driver (XRT). Although FPGA providers offer a diverse suite of libraries to ease kernel handling, they often do not offer fine-grained support for kernel management. For example: the Xilinx Vitis OpenCL library does not allow users to transfer information during kernel execution, making periodic backup of checkpointed data to host memory non-trivial.

We thus directly use the Xilinx XRT driver API to manage 1) the user kernel, 2) checkpoint memory backup, and 3) failure detection. Data offloading to the kernel follows the classical OpenCL sequence. Checkpoint backup, failure detection and task resumption is managed by a daemon (Posix thread) running on the host. This daemon performs periodic synchronization of the checkpoint and heartbeat buffers to offload the saved context data and trigger the task restoration process if FPGA failure is detected.

Host-FPGA Synchronization. We decouple the onboard checkpoint saving (write) process from the host data backup (read) process for improved performance. To avoid data corruption from this asynchrony, we protect the data-write procedure with a mutex — set to “locked" at the start of the checkpoint data saving process, and to “unlocked" once complete — and use additional HLS directives to enforce the sequential in-order execution of checkpointing-related instructions (i.e., `#pragma HLS protocol` in our case).

6 Use case: Fault-tolerant FPGA Task

6.1 Failure model

The proposed framework targets FPGA accelerators used in cloud systems such as Amazon AWS [47] and Microsoft Catapult v2 [13]. These types of multi-tenant systems are prone to attack by malicious users [30]. These attacks can be carried out through different channels: Physical Attacks [52], Malicious IPs or reconfiguration [14], etc. Denial-of-Service (DoS) is definitely one of the objectives of these attacks [30, 71]. Indeed, previous works have demonstrated that FPGAs can be made completely unresponsive by these attacks [24, 34, 37]. Our failure model, therefore, considers the entire FPGA and its memory as the failure zone (Fig. 8). In the event of failure, this entire zone is out of service, and no data can be extracted from it. Since such DoS attacks (and the resultant crashes) are unpredictable, one cannot anticipate the precise failure point. Consequently, sensitive data must be periodically saved outside the failure zone to be reliably accessible after a crash.

6.2 Active/Standby approach

We adopt an Active/Standby approach, which is commonly used in data-centers to ensure service availability after component failure [10, 26, 51]. There are two instances of the same task: an active one running on the FPGA, and a *shadow task* waiting (standby mode) on the host CPU. Our shadow

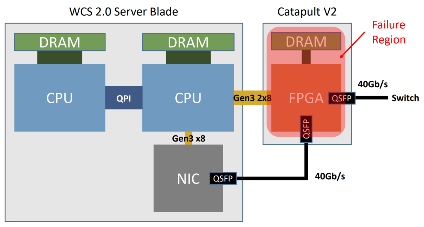


Fig. 8. Failure region, superimposed over the Microsoft Catapult node architecture [22]

task saves the progress achieved by the active task, i.e., stores the completed checkpoints from the FPGA into the CPU's host memory. When an FPGA failure is detected, the shadow task promotes itself to an active task by executing the program from the last valid stored checkpoint.

6.3 Implementation Details

Figure 9 illustrates the proposed framework applied to an FPGA-accelerated application. The upper part (Figure 9(a)) shows the system before the FPGA failure. The lower part (Figure 9(b)) shows the system when a failure occurs. The left side of each part represents the (host) CPU and its memory, while the right side depicts the FPGA with its Programmable Logic (PL) and its local memory (target memory). The host application relies on the drivers and APIs provided by the FPGA manufacturers to communicate with the FPGA. This diagram assumes that the FPGA is a Xilinx fabric due to its prominence in the FPGA market. Hence, the CPU driver is XRT and the internal communication bus is the AXI bus [66, 68]. The blue numbers represent runtime actions: The first step consists of running the host application on the CPU (Fig. 9(a).1). The host application launches the kernels' execution on the FPGA using XRT API/driver (Fig. 9(a).2). The *Compute kernel* corresponds to the user-offloaded kernel that has been instrumented with checkpoints. When a checkpoint is encountered, the task context is saved to the FPGA's local memory. The *Heartbeat kernel* is responsible for detecting failures by periodically updating a heartbeat variable that indicates that the system is alive. The *Daemon* thread on the host application monitors this heartbeat (Fig. 9(a).3), and also periodically copies the saved context out of the FPGA into the host memory (Fig. 9(a).3). If the *Daemon* detects that the heartbeat is no longer updated (i.e., because the FPGA has crashed (Fig. 9(b).5)), it interprets that as a component failure and starts the recovery process (Fig. 9(b).6). The recovery process proceeds in two phases: first the software version of the task is started (Fig. 9(b).6). Then, the last valid context of the task is reloaded from the host memory (Fig. 9(b).7).

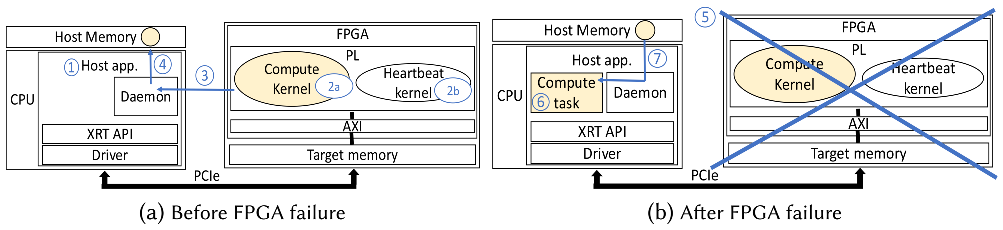


Fig. 9. Fault-tolerant FPGA accelerated application. Shadow yellow highlights the user task and its live context.

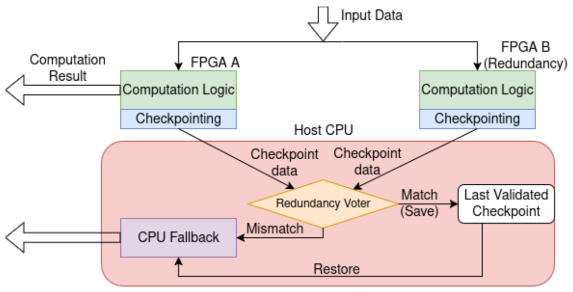


Fig. 10. Architecture of N-modular redundancy FPGA system

7 Use case: Preemptible FPGA Task

7.1 System Model.

Traditional FPGA systems operate under a non-preemptive execution model, where tasks occupy FPGA resources until completion, preventing dynamic resource sharing. This rigidity limits the ability to prioritize incoming workloads or overcommit FPGA resources, as high-priority tasks must wait indefinitely for hardware availability. To address this, our framework introduces stateful preemption, enabling FPGA tasks to be interrupted and migrated to the CPU when higher-priority workloads arrive. This capability transforms static FPGA provisioning into a flexible, elastic resource model akin to cloud computing paradigms like "spot instances," where lower-priority tasks temporarily occupy hardware but yield resources on demand.

7.2 Implementation Details

Tenants submit FPGA-accelerated tasks to the system, which are preprocessed by a checkpoint-enabled compilation toolchain. During submission, tasks are assigned priority levels (e.g., high for latency-sensitive workloads, low for batch processing).

When a high-priority task arrives and the FPGA is occupied by a lower-priority workload, the host initiates preemption: 1) The FPGA undergoes a soft reset to free resources, avoiding full reconfiguration. 2) The high-priority task is immediately deployed on the FPGA. 3) The preempted task resumes execution on the CPU from the latest checkpoint stored in host memory.

This workflow allows high-priority tasks to bypass queue delays while preserving progress for preempted workloads. Although CPU execution incurs performance penalties, the checkpoint-rollback mechanism eliminates the need to restart computations from scratch. In contrast, FPGA-to-FPGA migration strategies [46, 54] require costly bitstream reprogramming and state transfers between nodes. The proposed approach thus enables dynamic resource sharing aligned with cloud elasticity models, where FPGA resources are over-committed and prioritized on demand.

8 Use case: N-modudar redundancy for safety-critical systems

Our toolchain enables a novel implementation of dual modular redundancy that takes advantage of the heterogeneity of FPGA-CPU to further increase system reliability.

In the proposed framework, two FPGAs, *FPGA A* and *FPGA B* in Figure 10, execute an instance of the same kernel, both programmed with bitstreams containing our checkpoint-instrumented kernel. Each FPGA operates on the same input dataset, launched simultaneously through the XRT driver API. To ensure deterministic execution, we configure both FPGAs with identical initial conditions and disable any non-deterministic compiler features during HLS synthesis.

The data checking between data produced by the FPGA is handled by a dedicated verification thread (in light red in Figure 10). This thread runs on the host CPU, continuously monitoring checkpoint buffers from both FPGAs asynchronously. The module implements the following algorithm as shown in Listing 6.

In execution of the kernel, when an *equivalence point* is reached, the *verification module* checks that the instances have produced the same results (i.e., the same checkpoint memory segment). If this is the case, the checkpoint data are saved as last valid checkpoint and tagged as validated. If the data segments of the two checkpoints differ, the reference execution based on last valid checkpoint is migrated to the CPU, which continues to execute the application/kernel from the last validated checkpoint, in a slower but safer manner. We can also compare the two mismatching checkpoint segments with the checkpoint computed by the CPU execution after migration to identify the source of the failure, for fault isolation and analysis.

Listing 6. Checkpoint Verification Algorithm (pseudo-code)

```
while(system_active)
    // Wait for new checkpoints from both FPGAs
    ckpt_a = poll_checkpoint_buffer(fpga_a);
    ckpt_b = poll_checkpoint_buffer(fpga_b);
    if (ckpt_a.id == ckpt_b.id) // Compare checkpoint data
        if (memcmp(ckpt_a.data, ckpt_b.data, ckpt_size) != 0)
            // Disagreement detected
            trigger_cpu_migration(last_valid_checkpoint);
            flag_fpga_fault();
        else // Agreement - update last valid checkpoint
            backup_checkpoint(ckpt_a);
```

9 Evaluation

9.1 Experimental Setup

The evaluation of our proposal comprises two axes: 1) behavioral validation and 2) overhead assessment. Most experiments are carried out on the following platform: An Ubuntu 18.04 workstation with a quad-core Intel i5-6500 at 3.20GHz, 32GB of RAM, linked to a Xilinx Alveo U50 board [65] via PCIe gen3x16. The Xilinx board is configured with the platform version *xilinx_u50_gen3x16_xdma_201920_3*. We use Xilinx XRT driver version 2024.1. For the N-modular redundancy experiment, we use two FPGA instances (FPGA A and FPGA B), as required by the redundancy setup in Section 8.

We evaluate our proposal on the following applications:

- **Blur filter**, a floating point implementation of a Gaussian Blur filter adapted from Ref [8]. We perform only a single pass filter with a Gaussian Kernel 10×10. The input is a 460x690 pixels image with the standard 3-color channels.
- **Cholesky** (SPLASH2 Benchmark suite) [62], which computes a matrix decomposition of a sparse matrix. The input is a 512×512 matrix of double-precision floats. The time complexity of this algorithm is $O(n^3)$.
- **Kmeans** (Rodinia Benchmark suite) [48], a clustering algorithm frequently used in data mining. We run it on 1 638 400 points with 34 features and sort them into 5 clusters.
- **LUD** (Rodinia Benchmark suite) [49], which factors a dense matrix. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$). The block size has been fixed to the standard size of 16 for this evaluation. We use two different sizes for the input matrix: 1024×1024 (Lud 1024) and 3072×3072 (Lud 3072). Each is made up of random float variables.
- **MiniMap**[36] (MMap in the rest of the paper), which performs read mapping of short DNA sequences against a reference genome. The input is a subset of "chain" operation derived from a test case that consists of 100,000 query sequences of length 100 base pairs each, matched against a reference genome segment of 10 million base pairs.

- **MaxPool** (from finn-hlslib [60], abbreviated to MPool in the rest of the paper), a fundamental operation in convolutional neural networks that performs downsampling through maximum value selection. The input is a 112×112 pixel feature map with 64 channels. We use a pool size of 2×2 with a stride of 2.
- **FIR filter** (from finn-hlslib [60], abbreviated to FIR in the rest of the paper), a finite impulse response filter. The input is a 1 638 400 sequence of random float variables.

The input data for each application is sized to best fit the available resources of the adopted FPGA. Hence, *we do not use partial reconfiguration.* Table 2 shows the runtimes of the vanilla, uncheckpointed versions of the evaluated kernels. The runtime of these applications is highly dependant on the processed data size. Since these data-size-related runtime trends hold regardless of the data set, this paper uses small input data sets to keep runtimes short and the evaluation process tractable.

Table 2. Vanilla kernels' runtime in seconds on FPGA

Lud 1024	Lud 3072	Kmeans	Cholesky	Blur	MiniMap	MaxPool	FIR
0.79	16.6	0.38	0.69	2.67	134.28	23.6	17.9

Each application is assessed for different checkpoint rates/frequencies (High, Medium, Low, Very Low), though for some applications, the maximum feasible checkpoint rate is constrained by FPGA resource limitations on the size of the live execution state that can be stored. Table 3 shows the approximate average inter-checkpoint time ($AvgT_{ic}$) for each kernel, calculated from the number of checkpoints performed during the vanilla kernel runtime on FPGA.

Table 3. Average time between checkpoints ($AvgT_{ic}$) in ms for different rates and kernels

		High Rate	Medium Rate	Low Rate	Very Low Rate
Lud	$AvgT_{ic}$ (ms)	12.4	24.8	123.9	-
1024	Num Ckpt	64	32	6	-
Lud	$AvgT_{ic}$ (ms)	86.5	172.9	864.7	-
3072	Num Ckpt	192	96	19	-
kmeans	$AvgT_{ic}$ (ms)	1.88	3.76	18.8	188.1
	Num Ckpt	200	100	20	2
Cholesky	$AvgT_{ic}$ (ms)	1.34	2.68	13.4	134.2
	Num Ckpt	512	256	51	5
Blur	$AvgT_{ic}$ (ms)	5.9	11.85	59.2	592.4
	Num Ckpt	450	225	45	4
MiniMap	$AvgT_{ic}$ (ms)	-	83.9	167.3	891.0
	Num Ckpt	-	1604	802	151
MaxPool	$AvgT_{ic}$ (ms)	44.2	62.3	103.6	-
	Num Ckpt	534	380	227	-
FIR	$AvgT_{ic}$ (ms)	76.0	163.1	469.2	-
	Num Ckpt	233	116	38	-

9.2 Validation of the migration framework

To validate the proposed migration framework, each application is run twice: once on CPU (only) to compute a reference output, and then a second time on the FPGA, where the application is migrated to the CPU when an FPGA failure occurs. The outputs from both runs are then compared to check the validity of the computation that underwent live migration. *All applications* have been shown to produce valid results. Failures are emulated by interrupting the checkpoint memory segment update on the target. The failure time can thus be adjusted when kernels start.

Figure 11 shows the runtimes of the Kmean kernel (with migration) for the 4 different checkpoint rates. The blue bars show the execution time of the kernel on the FPGA, before migration. The red bars show the execution time of the kernel on the CPU, after migration. The migration time is of the order of microseconds and is therefore not visible in the figure.

All applications migrate when about 25% of the output computation on the FPGA has been consolidated by checkpoints. We observe that the runtimes required to achieve 25% of output computation increase with the checkpoint rate.

Indeed, performing checkpoints takes time and therefore slows down the kernel's execution. Section 9.3 investigates this behaviour. The CPU runtime is the same for each use case, since the CPU task completes the remaining 75% of computation for each test case.

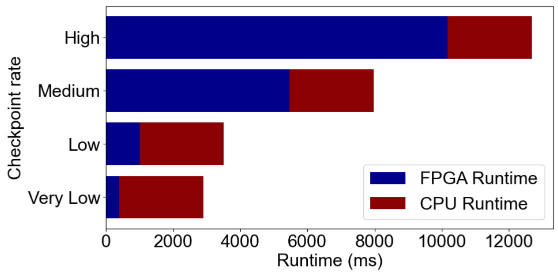


Fig. 11. Kmeans Kernel execution with a live heterogeneous migration after 25% of data computing completion.

9.3 Runtime overheads assessment

The insertion of additional checkpoints incurs two types of overhead: time and area (or hardware).

Figure 12 shows the runtime overheads incurred by checkpoints on the evaluated kernels for different checkpointing rates: High, Medium, Low, and Very Low. The approximate checkpointing frequencies for each of these rates for each kernel are shown in Table 3. This assessment considers both *incremental* and *non-incremental* checkpointing strategies. Slowdowns are normalized to the runtime of the Vanilla kernel version on FPGA.

In Figure 12, the dark blue bars show the runtimes of the kernels on the FPGA with *non-incremental checkpoints*; while the light blue bars correspond to the kernels running on the FPGA with *incremental checkpoints*. As expected, reliability comes at a cost, and all dependable kernels take longer to complete than the Vanilla version. There are two reasons for the observed slowdowns:

- The checkpointing process itself, as copying data from FPGA registers and internal RAMs to target memory is time-consuming. This phenomenon is apparent in the performance of non-incremental checkpointing cases, where the entire live data set is copied each time a checkpoint is processed. The higher the checkpoint rate, the higher the runtime overhead.
- Changes to the workflow resulting from the addition of checkpoints. The HLS tool may fail to fully optimize a kernel due to the addition of checkpoints, which changes the CFG of the task. This can be observed in the Kmeans kernel, where the initial perfect loop is impacted by the checkpointing process [67], which prevents the HLS tool from fully optimizing the loop. This explains the large slowdown despite the very low checkpoint rate (Figure 12c).

Inspecting the runtimes for incremental checkpoints, Figures 12a and 12c show less extensive slowdowns for all checkpoint rates compared to non-incremental checkpointing: only $\approx$ 1.1x slower for Blur and $\approx$ 19x slower for Kmeans. Such behavior highlights the benefits of incremental checkpointing. However, Cholesky (Fig 12d) and Lud (Fig 12e, 12f) kernels display opposite performance trends because they perform too many memory writes per checkpoint, which causes the overheads of our incremental checkpoints' memory tracking mechanism to exceed its runtime savings. For these kernels, it is more efficient to simply perform non-incremental checkpointing instead. We can also notice that incremental checkpointing overheads for Blur and Cholesky remain constant with decreasing checkpoint frequency, because here the runtime savings from sparse checkpointing are offset by the increase in the amount of execution context that must be incrementally checkpointed as the checkpoints are placed further apart. We conclude that runtime overheads are highly dependent on the specific kernel workloads (checkpoint size, code implementation, etc.). Indeed, if the rate of the checkpoints has an impact on performance, their location in the kernel is the primary factor affecting performance. Hence, certain criteria must be considered when selecting a suitable location for a checkpoint. While reducing the size of the active variable seems obvious, other criteria may be less straightforward, such as avoiding inserting a checkpoint in a section of code that could/should be aggressively optimised by the compiler's optimization passes (particularly by the hardware backend). As this can be complex for an unfamiliar user to figure out, an interesting future work could be to improve the proposed toolchain with automatic checkpoint selection. This extension could leverage the report generated by HLS and use a cost model to automatically insert checkpoints at the most appropriate location.

These experiments also show that, despite the migration-enabled FPGA kernels comes at a cost, with a well-chosen configuration, many kernels can still benefit from FPGA acceleration. The most noticeable example is FIR 12b that exposes significant speedups for the instrumented FPGA versions compared to CPU execution.

Table 4. Kernels' restore latency in milliseconds

	Lud 1024	Lud 3072	Kmeans	Cholesky	Blur	MiniMap	MaxPool	FIR
Restore latency (ms)	1.94	18.07	6.32	0.98	1.03	31.90	10.62	7.33
Checkpoint size (MB)	4.19	37.75	7.70	2.10	3.81	123.68	22.70	16.6

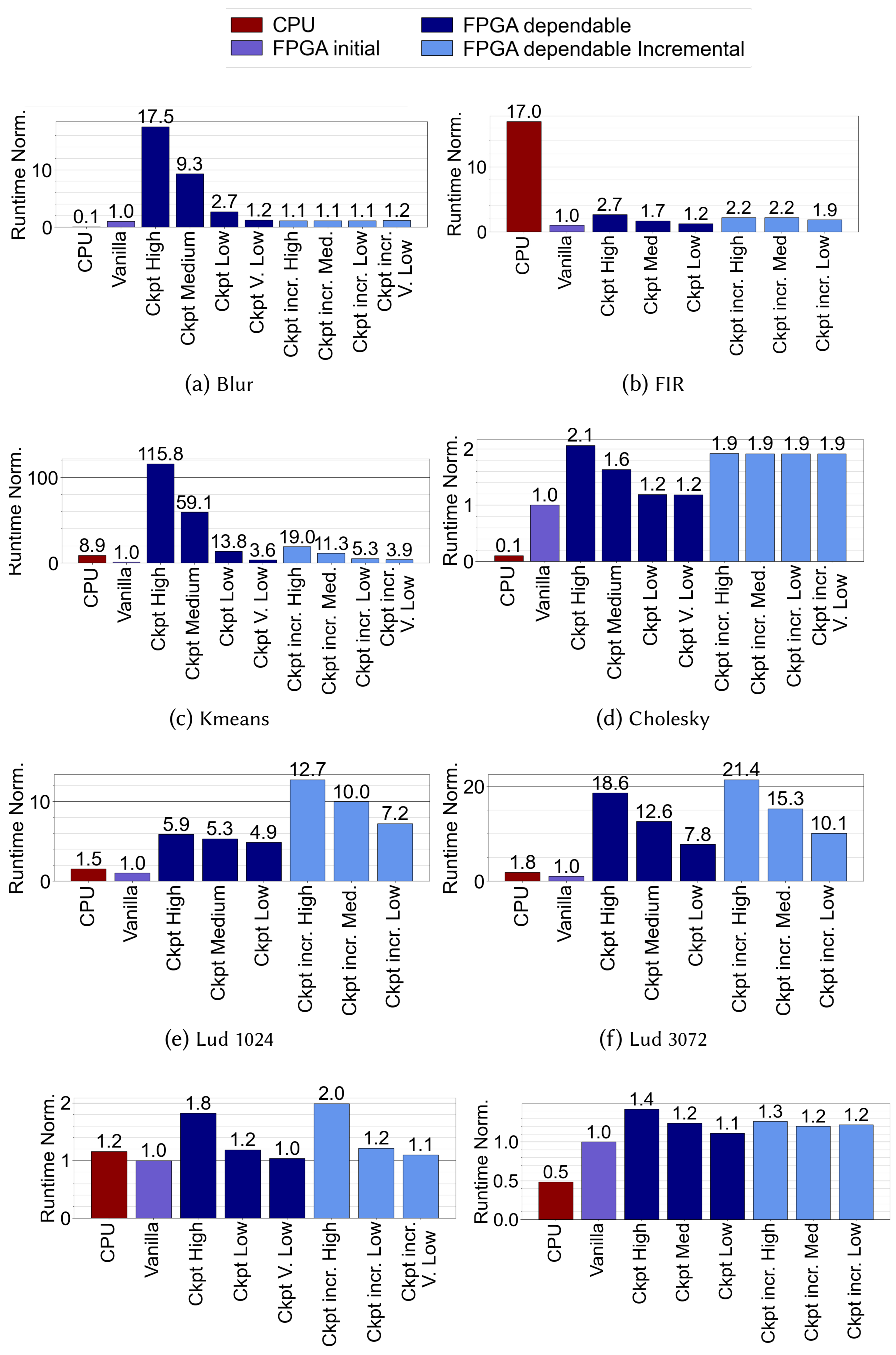
9.4 Kernel resume latency

Table 4 shows the time taken by our system to migrate the application from an FPGA and restore its execution on a CPU. This is competitive relative to the FPGA-FPGA migration proposed by related work, which incurs additional delays (at least a few seconds) to reconfigure the target FPGA. Our restore latency depends on the data checkpoint size, but there is no direct correlation between these two figures, as the nature of the data must be taken into consideration — copying n 32-bit variables takes more time than copying a single memory chunk of $n \times 32$ bits (e.g. an array).

9.5 Incremental checkpointing analysis

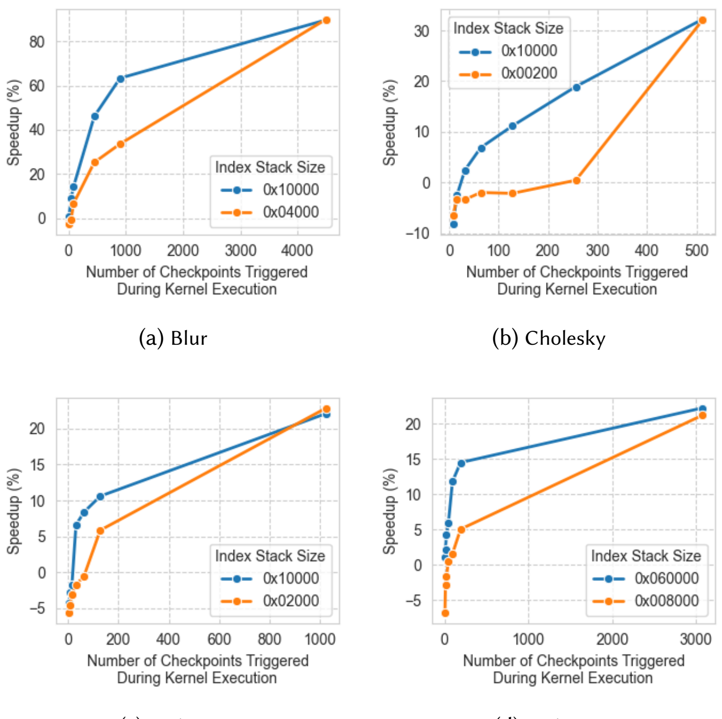
Figure 13 shows how the runtime speedup introduced by our incremental checkpointing scheme varies across checkpoint frequencies and index-stack sizes. Speedup is measured relative to a checkpointed kernel without incremental checkpointing. Overall, speedup is positively correlated to checkpoint frequency. Indeed, at high frequencies, fewer array elements are updated between successive checkpoints, so there are fewer elements to copy. We achieve up to 90% speedup for Blur, 30% for Cholesky, and 20% for LUD. At lower frequencies, successive checkpoints accumulate increasingly more updated elements between them that must be incrementally checkpointed. At very low frequencies, speedup can become negative, as the incremental checkpoint scheme becomes *saturated* with updated elements and must copy the entire array. At this point, the overheads introduced by additional subroutines used for array index tracking can exceed the incremental checkpointing scheme's runtime savings.

With respect to a baseline stack size (blue): reductions in stack size (orange) can cause early onset of saturation. If the number of updated elements to be incrementally checkpointed at once exceeds the stack size, i.e., at lower checkpoint frequencies, our scheme is forced to do a full array



(a) Blur
(b) FIR
(c) Kmeans
(d) Cholesky
(e) Lud 1024
(f) Lud 3072
(g) MiniMap
(h) MaxPool

Fig. 12. Kernel time overhead



(a) Blur (b) Cholesky

(c) Lud 1024 (d) Lud 3072

Fig. 13. Kernel (Optimised) Runtime Speedup

copy. Smaller stack sizes are exceeded more easily than larger ones. We thus advise users to use as large a stack size as needed for the arrays being checkpointed.

Nonetheless, our evaluation shows that of our incremental checkpointing scheme can afford significant runtime benefits if configured well. In practice, checkpoint frequency should be chosen in balance with these runtime characteristics to achieve the best runtime for their kernel.

9.6 Evaluation of FPGA resource utilization

Figure 14 shows the hardware overhead of our dependable system kernels (in blue) compared to the Vanilla version (in green). The maximum increase in LUT usage resulting from checkpoint insertion is 14.6% for the Lud kernel, which we consider to be acceptably small. Note that this is independent of the checkpoint rate, as the rate is controlled by a simple if statement.

9.7 Checkpoint State Size Assessment

Checkpoint state size is critical for large-scale deployment, as states must be stored in memory for rapid recovery. Figure 15 compares state-of-the-art circuit analysis-based FPGA checkpointing methods [3]. The state size presented was determined without incremental optimization.

From the figure, it is evident that our HW-SW migration consistently produces smaller checkpoint states than state-mover across all tested programs. This indicates that our approach is more memory-efficient, which is crucial for minimizing resource usage and optimizing performance in large-scale systems. The significant difference in state sizes, especially for more complex programs like "Blur" and "minimap," highlights our advantage in scenarios requiring extensive state management.

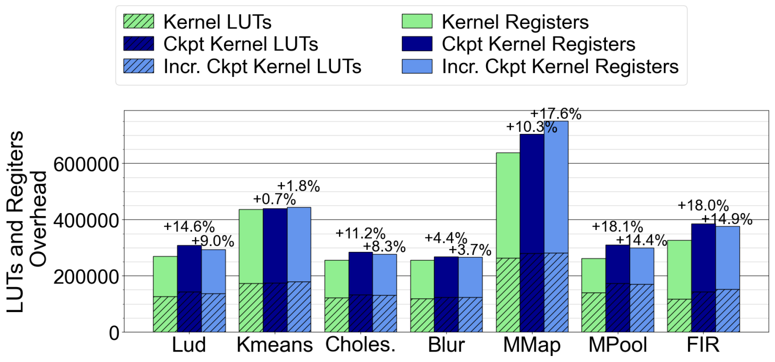


Fig. 14. Hardware overhead

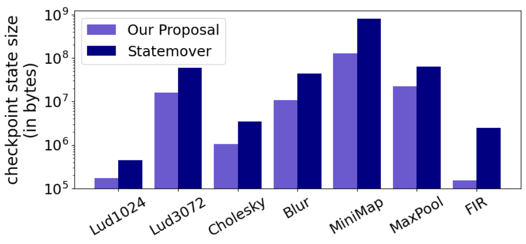


Fig. 15. Checkpoint size Our proposal vs Statemover[3]

9.8 Migration based preemption

Figure 16 evaluates the suitability of our proposal for task preemption in a multitasking system with tasks having different priority levels. Our scenario considers two tasks, e.g., Minimap, competing for the same FPGA resource. Task 1 is running with low priority, and Task 2 is running with high priority. Users initially want to run these tasks on FPGA for efficiency reasons. The high-priority Task 2 arrives at an arbitrary time, while the low-priority Task 1 is already running on the FPGA.

In the absence of preemption support, the tasks are executed sequentially on the FPGA. This means that Task 1 must be completed fully before Task 2 begins to run. This can result in extensive latency from Task 2. With preemption, Task 1 starts running on the FPGA but is preempted when Task 2 arrives at time 77638. The FPGA then switches to Task 2, while Task 1 continues to run on the CPU. Unsurprisingly, the case with preemption not only 1) serves and completes the execution of the high-priority task earlier, 2) but also reduces the total computation time.

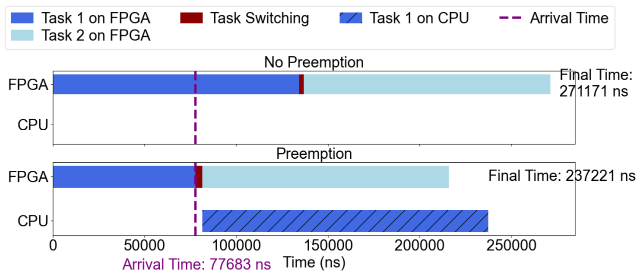


Fig. 16. Preemption Timeline

9.9 N-Modular Redundancy Evaluation

Figure 17 evaluates our checkpoint-based N-modular redundancy framework running on two FPGAs (A and B) as described in Section 8. The benchmark used is FIR with a "medium" checkpoint rate. Additionally, we modified the kernel to export an extra internal variable in AXI Lite interface as a backdoor to inject data corruption.

In our experiment, at 3847 ms, we inject corrupt data via the backdoor to simulate silent data corruption. With checkpointing enabled, the verification module then detects the error, i.e., checkpoint divergence, at 3891 ms, 44 ms after the injection. The CPU execution starts at 3987 ms after finishing the restoring process. The execution is restored from the last validated checkpoint, allowing a safe continuation without data loss.

For comparison, baseline dual-modular redundancy without checkpointing only detects corruption when comparing final outputs after the full 17.9 second execution on both FPGAs, a 3×longer detection latency. Also there is no way to tell which FPGA to blame without adding a third FPGA.

This demonstrates that our framework enables fine-grained error detection (milliseconds vs. seconds) and rapid fault localization, critical requirements for safety-critical systems.

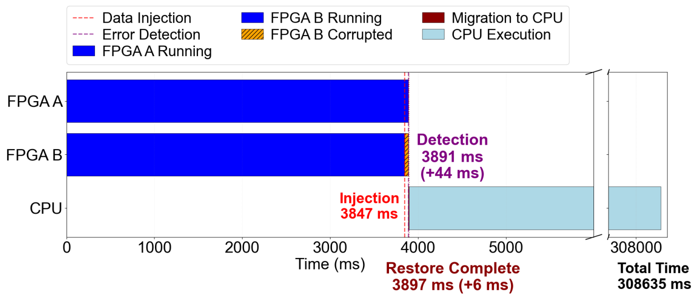


Fig. 17. N-Modular Redundancy Timeline

9.10 Compile-time Overhead Assessment

Our compilation passes incur a computation time ranging from a few seconds to a few minutes, depending on the number of live variables to process and checkpoints to insert. However, since the full HLS compilation and synthesis process often requires a few hours to complete, the additional computation time incurred by our passes is comparatively negligible.

10 Limitations

FPGA-CPU architectural differences create kernel design constraints. Checkpoints must be placed in the core function and cannot exist in data parallel regions, as this could cause nondeterministic behavior. Function call in kernel function is not supported, unlike CPUs that use stack-based function calls, FPGAs handle function execution differently, making it extremely complex to reliably restore call stack after migration. However, this can be alleviated by function inlining.

Restoring a checkpoint back onto the FPGA (i.e., CPU-to-FPGA or FPGA-to-FPGA migration) is not supported in our current implementation. Enabling this reverse migration would require non-trivial additions to the generated kernel circuit: the design must expose fine-grained control over the pipeline to (re)inject state into internal memories and resume execution at the exact stage where the kernel was suspended. Such control can be expensive, potentially increasing FPGA resource utilisation and/or degrading timing closure [11]. Therefore, designing an innovative restoration mechanism is an important direction for future work, as it could further improve the service quality of FPGAs.

11 Conclusion

This paper introduces a compiler-based toolchain enabling stateful HW-SW migration. Across different use cases – including fault-tolerant FPGA and FPGA preemption, we demonstrated how our toolchain, and more generally fine-grain Hardware to Software migration, can address the QoS issue of FPGAs-as-a-Service. This new type of migration is handled by custom compilation passes that create equivalence points between hardware and software kernels such that the task can be migrated from FPGA to CPU.

We show that preempting a low-priority task running on FPGA allows multitasking systems to better manage resource allocation, allowing high-priority tasks to be assigned to the FPGA earlier and reducing overall computation time. As for fault-tolerance, although increasing a system's dependability inevitably comes at a cost, our proposal achieves respectable performance, adding (in the best case) only a 10% and 20% slowdown for the Blur and Cholesky kernels, respectively. We also showed that the same checkpointing foundation supports N-modular redundancy by running redundant FPGA instances with checkpoint-level comparison, which enables significantly earlier fault detection and safer recovery. However, unlike previous works, our open-source solution[3] does not rely on specific programming frameworks and does not require code rewriting, making it directly applicable to legacy code.

Amongst others, future work could investigate potential security breaches that could arise with the proposed solution. Indeed, storing internal variables into memory (i.e., in the checkpoints) could reveal certain secrets and therefore be used by attackers to steal information, especially in multi-tenant systems. This could mean that users might be willing to encrypt checkpointed data or avoid checkpointing at critical execution points to avoid exposing confidential data.

Acknowledgements

We thank the anonymous reviewers for their insightful comments which helped greatly improve the paper. This work is supported by the US Office of Naval Research (ONR) under grants N00014-18-1-2022, N00014-19-1-2493, and N00014-211-2523.

References

[1] Sam Ainsworth and Timothy M. Jones. 2018. Parallel Error Detection Using Heterogeneous Cores. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 338–349. doi:10.1109/DSN.2018.00044

[2] Mohammad Al Hassan, Paul Britton, and Glen S. Hatfield. 2018. FPGA Reliability and Failure Rate Analysis for Launch and Space Vehicles Environments. https://ntrs.nasa.gov/citations/20180003490 NTRS Author Affiliations: NASA Marshall Space Flight Center, Bastion Technologies, Inc. NTRS Report/Patent Number: M17-6139 NTRS Document ID: 20180003490 NTRS Research Center: Marshall Space Flight Center (MSFC).

[3] Sameh Attia and Vaughn Betz. 2020. StateMover: Combining Simulation and Hardware Execution for Efficient FPGA Debugging. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 175–185. doi:10.1145/3373087.3375307

[4] Sameh Attia and Vaughn Betz. 2020. StateReveal: Enabling Checkpointing of FPGA Designs with Buried State. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 206–214. doi:10.1109/ICFPT51103.2020.00036

[5] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. 2020. Edge computing: the case for heterogeneous-ISA container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 73–87. doi:10.1145/3381052.3381321

[6] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 645–659. doi:10.1145/3037697.3037738

[3]https://github.com/systems-nuts/junco-compiler_assisted_checkpointing

[7] Berten. 2016. GPU vs FPGA Performance Comparison. https://www.bertendsp.com/gpu-vs-fpga-performance-comparison/

[8] bfraboni. 2023. C++ implementation of a fast Gaussian blur algorithm by Ivan Kutskir - Integer and Floating point version. https://gist.github.com/bfraboni/946d9456b15cac3170514307cf032a27

[9] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. 2022. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 34 (feb 2022), 42 pages. doi:10.1145/3506713

[10] Yasmina Bouizem, Nikos Parlavantzas, Djawida Dib, and Christine Morin. 2021. Active-Standby for High-Availability in FaaS. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing* (Delft, Netherlands) *(WoSC'20)*. Association for Computing Machinery, New York, NY, USA, 31–36. doi:10.1145/3429880.3430097

[11] Alban Bourge, Olivier Muller, and Frédéric Rousseau. 2016. Generating Efficient Context-Switch Capable Circuits through Autonomous Design Flow. *ACM Trans. Reconfigurable Technol. Syst.* 10, 1, Article 9 (dec 2016), 23 pages.

[12] Andrew Boutros, Mathew Hall, Nicolas Papernot, and Vaughn Betz. 2020. Neighbors From Hell: Voltage Attacks Against Deep Learning Accelerators on Multi-Tenant FPGAs. doi:10.48550/ARXIV.2012.07242

[13] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. doi:10.1109/MICRO.2016.7783710

[14] Jayeeta Chaudhuri, Hassan Nassar, Dennis R. E. Gnad, Jorg Henkel, Mehdi B. Tahoori, and Krishnendu Chakrabarty. 2025. FLARE: Fault Attack Leveraging Address Reconfiguration Exploits in Multi-Tenant FPGAs. arXiv:2502.15578 [cs.CR] https://arxiv.org/abs/2502.15578

[15] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (Cagliari, Italy) *(CF '14)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. doi:10.1145/2597917.2597929

[16] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. 2022. Compiler-Directed High-Performance Intermittent Computation with Power Failure Immunity. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 40–54. doi:10.1109/RTAS54340.2022.00012

[17] Intel Corporation. 2023. FPGA vs. GPU for Deep Learning Applications. https://www.intel.com/content/www/us/en/artificial-intelligence/programmable/fpga-gpu.html

[18] Intel Corporation. 2023. High-Level Synthesis Compiler - Intel® HLS Compiler. https://www.intel.com/content/www/uk/en/software/programmable/quartus-prime/hls-compiler.html

[19] Intel Corporation. 2025. 1.6. Failure Rates. https://www.intel.com/content/www/us/en/docs/programmable/683602/21-3/failure-rates.html

[20] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245* (2021).

[21] Alibaba Cloud ECS. 2023. Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. https://www.alibabacloud.com/blog/594057

[22] Dan Fay and Derek Chiou. 2018. *The Catapult Project - An FPGA view of the Data Center.* Microsoft Research. http://www.prime-project.org/wp-content/uploads/sites/206/2018/02/Talk-7-Dan-Fay-The-Catapult-Project-%E2%80%93-An-FPGA-view-of-the-Data-Center.pdf

[23] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1327–1330. doi:10.1109/DAC18074.2021.9586110

[24] Dennis R. E. Gnad, Fabian Oboril, and Mehdi B. Tahoori. 2017. Voltage drop-based fault attacks on FPGAs using valid bitstreams. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. doi:10.23919/FPL.2017.8056840

[25] Edson Horta, Ho-Ren Chuang, Naarayanan Rao VSathish, Cesar Philippidis, Antonio Barbalace, Pierre Olivier, and Binoy Ravindran. 2021. Xar-Trek: Run-Time Execution Migration among FPGAs and Heterogeneous-ISA CPUs. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 104–118. doi:10.1145/3464298.3493388

[26] Huawei. 2023. GaussDB active-standby Cluster Mechanism. https://forum.huawei.com/enterprise/en/gaussdb-active-standby-cluster-mechanism/thread/560767-893

[27] IEC. 2010. *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems* (2.0 ed.). International Standard. International Electrotechnical Commission, Geneva, Switzerland. Parts 1-7.

[28] ISO. 2018. *Road vehicles – Functional safety.* Standard ISO 26262:2018G. International Organization for Standardization, Geneva, CH.

[29] Chenglu Jin, Vasudev Gohil, Ramesh Karri, and Jeyavijayan Rajendran. 2020. Security of Cloud FPGAs: A Survey. doi:10.48550/ARXIV.2005.04867

[30] Muhammed Kawser Ahmed, Maximillian Panoff Kealoha, Joel Mandebi Mbongue, Sujan Kumar Saha, Erman Nghonda Tchinda, Peter Esenju Mbua, and Christophe Bobda. 2025. Multi-Tenant Cloud FPGA: A Survey on Security, Trust, and Privacy. *ACM Trans. Reconfigurable Technol. Syst.* 18, 2, Article 23 (April 2025), 44 pages. doi:10.1145/3713078

[31] Andrew M. Keller and Michael J. Wirthlin. 2019. Impact of Soft Errors on Large-Scale FPGA Cloud Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 272–281. doi:10.1145/3289602.3293911

[32] Eric P. Kim and Naresh R. Shanbhag. 2012. Soft N-Modular Redundancy. *IEEE Trans. Comput.* 61, 3 (2012), 323–336. doi:10.1109/TC.2010.253

[33] Dirk Koch, Christian Haubelt, and Jürgen Teich. 2007. Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '07)*. Association for Computing Machinery, New York, NY, USA, 188–196. doi:10.1145/1216919.1216950

[34] Tuan La, Khoa Pham, Joseph Powell, and Dirk Koch. 2021. Denial-of-Service on FPGA-based Cloud Infrastructures — Attack and Defense. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 3 (July 2021), 441–464. doi:10.46586/tches.v2021.i3.441-464

[35] Kyungmi Lee, Maitreyi Ashok, Saurav Maji, Rashmi Agrawal, Ajay Joshi, Mengjia Yan, Joel S. Emer, and Anantha P. Chandrakasan. 2025. Secure Machine Learning Hardware: Challenges and Progress [Feature]. *IEEE Circuits and Systems Magazine* 25, 1 (2025), 8–34. doi:10.1109/MCAS.2024.3509376

[36] Kisaru Liyanage, Hiruna Samarakoon, Sri Parameswaran, and Hasindu Gamaarachchi. 2023. minimap2-fpga: Integrating hardware-accelerated chaining for efficient end-to-end long-read sequence mapping. doi:10.1101/2023.05.30.542681

[37] Yukui Luo, Cheng Gongye, Shaolei Ren, Yunsi Fei, and Xiaolin Xu. 2020. Stealthy-Shutdown: Practical Remote Power Attacks in Multi - Tenant FPGAs. In *IEEE 38th International Conference on Computer Design (ICCD).* doi:10.1109/ICCD50377.2020.00097

[38] Dina G. Mahmoud, Vincent Lenders, and Mirjana Stojilović. 2022. Electrical-Level Attacks on CPUs, FPGAs, and GPUs: Survey and Implications in the Heterogeneous Era. *ACM Comput. Surv.* 55, 3, Article 58, 40 pages. doi:10.1145/3498337

[39] Nikolaos Mavrogeorgis, Chris Vasiladiotis, Pei Mu, Amir Khordadi, Björn Franke, and Antonio Barbalace. 2024. UNIFICO: Thread Migration in Heterogeneous-ISA CPUs without State Transformation. doi:10.1145/3640537.3641565

[40] Shayan Moini, Shanquan Tian, Daniel Holcomb, Jakub Szefer, and Russell Tessier. 2021. Power Side-Channel Attacks on BNN Accelerators in Remote FPGAs. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11, 2 (2021), 357–370. doi:10.1109/JETCAS.2021.3074608

[41] NVIDIA. 2023. GPU-Accelerated Google Cloud Platform | NVIDIA. https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/google-cloud-platform/

[42] Pierre Olivier, A K M Fazla Mehrab, Sandeep Errabelly, Stefan Lankes, Mohamed Lamine Karaoui, Robert Lyerly, Sang-Hoon Kim, Antonio Barbalace, and Binoy Ravindran. 2024. HEXO: Offloading Long-Running Compute- and Memory-Intensive Workloads on Low-Cost, Low-Power Embedded Systems. *IEEE Transactions on Cloud Computing* 12, 4 (2024), 1415–1432. doi:10.1109/TCC.2024.3482178

[43] Adrien Prost-Boucle, Olivier Muller, and Frédéric Rousseau. 2014. Fast and Standalone Design Space Exploration for High-Level Synthesis under Resource Constraints. *J. Syst. Archit.* 60, 1 (jan 2014), 79–93. doi:10.1016/j.sysarc.2013.10.002

[44] Dylan Rankin, Jeffrey Krupa, Philip Harris, Maria Acosta Flechas, Burt Holzman, Thomas Klijnsma, Kevin Pedro, Nhan Tran, Scott Hauck, Shih-Chieh Hsu, Matthew Trahms, Kelvin Lin, Yu Lou, Ta-Wei Ho, Javier Duarte, and Mia Liu. 2020. FPGAs-as-a-Service Toolkit (FaaST). In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 38–47. doi:10.1109/H2RC51942.2020.00010

[45] Gabriel Rodriguez-Canal, Nick Brown, Yuri Torres, and Arturo Gonzalez-Escribano. 2022. Programming abstractions for preemptive scheduling in FPGAs using partial reconfiguration. doi:10.48550/ARXIV.2209.04410

[46] Gabriel Rodriguez-Canal, Nick Brown, Yuri Torres, and Arturo Gonzalez-Escribano. 2023. Task-based preemptive scheduling on FPGAs leveraging partial reconfiguration. *Concurrency and Computation: Practice and Experience* 35, 25 (2023), e7867. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.7867 doi:10.1002/cpe.7867

[47] Amazon Web Services. 2023. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[48] SFU-HiAccel. 2023. rodinia-hls/Benchmarks at master · SFU-HiAccel/rodinia-hls. https://github.com/SFU-HiAccel/rodinia-hls/tree/master/Benchmarks/kmeans/kmeans_3_unroll

[49] SFU-HiAccel. 2023. rodinia-hls/Benchmarks at master · SFU-HiAccel/rodinia-hls. https://github.com/SFU-HiAccel/rodinia-hls/tree/master/Benchmarks/lud/lud_1_tiling

[50] H. Simmler, L. Levinson, and Reinhard Männer. 2000. Multitasking on FPGA Coprocessors. In *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications (FPL '00)*. Springer-Verlag, Berlin, Heidelberg, 121–130.

[51] Cisco Systems. 2023. Cisco Security Appliance Command Line Configuration Guide, Version 7.2. https://www.cisco.com/c/en/us/td/docs/security/asa/asa72/configuration/guide/conf_gd/failover.html

[52] She Tang, Jian Wang, Zhe Chen, and Shize Guo. 2025. A Covert and Efficient Attack on FPGA Cloud Based on Adaptive RON. *IEEE Transactions on Dependable and Secure Computing* 22, 5 (2025), 4641–4653. doi:10.1109/TDSC.2025.3550568

[53] TechRepublic. 2019. Feniks: Microsoft's cloud-scale FPGA operating system. https://www.techrepublic.com/article/feniks-microsofts-cloud-scale-fpga-operating-system/

[54] Ye Tian, Jean-Christophe Prevotet, and Fabienne Nouvel. 2019. Efficient OS Hardware Accelerators Preemption Management in FPGA. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 367–370. doi:10.1109/ICFPT47387.2019.00069

[55] Anuj Vaishnav, Khoa Pham, and Dirk Koch. 2018. Live Migration for OpenCL FPGA Accelerators. In *2018 International Conference on Field-Programmable Technology (FPT)*. 38–45. doi:10.1109/FPT.2018.00017

[56] D. G. Von Bank, C. M. Shub, and R. W. Sebesta. 1994. A Unified Model of Pointwise Equivalence of Procedural Computations. *ACM Transactions on Programming Languages and Systems* 16, 6 (Nov. 1994), 1842–1874. http://www.acm.org/pubs/toc/Abstracts/0164-0925/197402.html

[57] Hoang Gia Vu, Supasit Kajkamhaeng, Shinya Takamaeda-Yamazaki, and Yasuhiko Nakashima. 2016. CPRtree: A Tree-Based Checkpointing Architecture for Heterogeneous FPGA Computing. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. 57–66. doi:10.1109/CANDAR.2016.0024

[58] Devon Wanner, Hashim A. Hashim, Siddhant Srivastava, and Alex Steinhauer. 2024. UAV avionics safety, certification, accidents, redundancy, integrity, and reliability: a comprehensive review and future trends. *Drone Systems and Applications* 12 (2024), 1–23. doi:10.1139/dsa-2023-0091

[59] Arief Wicaksana, Alban Bourge, Olivier Muller, Arif Sasongko, and Frédéric Rousseau. 2017. Prototyping Dynamic Task Migration on Heterogeneous Reconfigurable Systems. In *International Symposium on Rapid System Prototyping*.

[60] Xilinx. 2022. Vitis HLS Library for FINN — github.com. https://github.com/Xilinx/finn-hlslib.

[61] Xilinx. 2023. FPGA_as_a_Service. https://github.com/Xilinx/FPGA_as_a_Service

[62] Xilinx. 2023. Vitis-Tutorials 06-cholesky-accel/03-Algorithm. https://github.com/Xilinx/Vitis-Tutorials/tree/2022.2/Hardware_Acceleration/Design_Tutorials/06-cholesky-accel/03-Algorithm_Acceleration/docs/module1_baseline

[63] Xilinx. 2023. Xilinx/hls-llvm-project. https://github.com/Xilinx/hls-llvm-project

[64] AMD Xilinx. 2022. Vitis HLS — Vitis™ Tutorials 2022.1 documentation. https://xilinx.github.io/Vitis-Tutorials/2022-1/build/html/docs/Getting_Started/Vitis_HLS/Getting_Started_Vitis_HLS.html

[65] AMD Xilinx. 2023. Alveo U50. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html

[66] AMD Xilinx. 2023. AMBA AXI4 Interface Protocol. https://www.xilinx.com/products/intellectual-property/axi.html

[67] AMD Xilinx. 2023. Working with Nested Loops • Vitis High-Level Synthesis User Guide (UG1399). https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Working-with-Nested-Loops

[68] AMD Xilinx. 2023. Xilinx Runtime Library (XRT). https://www.xilinx.com/products/design-tools/vitis/xrt.html

[69] Tong Xing, Antonio Barbalace, Pierre Olivier, Mohamed L. Karaoui, Wei Wang, and Binoy Ravindran. 2022. H-Container: Enabling Heterogeneous-ISA Container Migration in Edge Computing. *ACM Trans. Comput. Syst.* 39, 1–4, Article 5 (July 2022), 36 pages. doi:10.1145/3524452

[70] Fangkai Yang, Lu Wang, Zhenyu Xu, Jue Zhang, Liqun Li, Bo Qiao, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, Zhen Ma, Íñigo Goiri, Eli Cortez, Terry Yang, Victor Rühle, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2023. Snape: Reliable and Low-Cost Computing with Mixture of Spot and On-Demand VMs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 631–643. doi:10.1145/3582016.3582028

[71] Shaza Zeitouni, Ghada Dessouky, and Ahmad-Reza Sadeghi. 2020. SoK: On the Security Challenges and Risks of Multi-Tenant FPGAs in the Cloud. arXiv:2009.13914 [cs.CR] https://arxiv.org/abs/2009.13914

[72] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, and Yu Wang. 2023. DF-GAS: a Distributed FPGA-as-a-Service Architecture towards Billion-Scale Graph-based Approximate Nearest Neighbor Search. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 283–296. doi:10.1145/3613424.3614292

[73] Chuck Zhao, J. Gregory Steffan, Cristiana Amza, and Allan Kielstra. 2012. Compiler Support for Fine-Grain Software-Only Checkpointing. In *International Conference on Compiler Construction*.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009