Compiler Support for Software Checkpoint-Restore for C/C++ Kernel Code

For Increased QoS in FPGA-Accelerated Systems in HPC Contexts

Huang Zihan

MInf Project (Part 1) Report
Electronics and Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

As modern computing applications move increasingly towards big-data computing, many high-performance computing (HPC) systems are adopting heterogeneous accelerated computing methods for greater performance and efficiency. This includes the growing popularity of FPGA acceleration in cloud datacentre servers, where it is used to offload and accelerate tasks such as deep learning and high frequency trading.

However, one reason preventing the widespread adoption of FPGAs in HPC is the limited flexibility in stateful dynamic resource allocation, contributed to by the lack of fine-grained CPU-FPGA task migration capability. In the event of FPGA component failure, current FPGA to FPGA migration schemes allow tasks to be migrated from the failed FPGA to a working FPGA where execution is continued. However, in systems where FPGA resources are in high demand, it can be time-consuming and expensive to find and allocate a suitable FPGA migration target. This can potentially disrupt the execution of FPGA-accelerated computing tasks, which impacts the dependability of the FPGA-accelerated system. To avoid this, it is thus desirable in this case to allow tasks to be statefully migrated from the failed FPGA to the CPU host instead.

This dissertation therefore proposes a custom compiler support for software checkpoint insertion into the C/C++ source code of FPGA-accelerated kernels, for the purpose of fine-grained CPU-FPGA task migration. This compiler support is a crucial component of a broader task migration framework leveraging the open-source Xilinx Vitis HLS toolchain. Our compiler support is shown to successfully achieve execution state extraction and restoration on CPU test benches. Upon integration into the broader task migration system, it is shown to successfully achieve safe FPGA to CPU task migration. However, the checkpointing it performs can incur sizeable runtime overheads depending on kernel configuration, especially on FPGAs. We evaluate this along with our compiler support's other limitations, and propose further improvements and expansions to be pursued in the MInf Part 2 project.

Research Ethics Approval

Declaration

(*Huang Zihan*)

Acknowledgements

I would like to thank Dr. Maxime France-Pillois and Dr. Antonio Barbalace for their patience, guidance and support, and for the opportunity to work with them on this fascinating subject.

Table of Contents

Chapter 1

Introduction

As modern computing applications move increasingly towards big-data computing, many high-performance computing (HPC) systems are adopting heterogeneous accelerated computing methods for greater performance and efficiency [23]. While these systems had mainly employed CPUs with GPU accelerators [39], their high power-consumption-to-compute ratio have prompted systems architects to look towards FPGAs for powerful, energy-efficient acceleration [23].

Indeed, FPGAs offer excellent performance and high computational efficiency [18]. FPGAs integrate a programmable logic region that allows for the implementation of custom hardware. Thus, users can specify custom hardware implementations (data path, memory layouts, etc.) that are designed to precisely and efficiently meet the accelerated task's performance requirements while minimising excess power consumption. This allows FPGA performance to exceed that of GPUs in certain use cases [14], whilst consuming as little as 28% of the GPU power on average [12].

Hence unsurprisingly, FPGA utilisation has become increasingly popular in cloud datacentre servers, where they have been used in smart NICs [18], smart SSDs [49], and in FPGA-accelerated computing to offload and accelerate tasks such as deep learning and high-frequency trading [4][19], e.g. in Amazon's AWS EC2 F1 [43] and Microsoft's Project Catapult [11]. However, FPGA adoption in HPC has been slowed by the challenges of programmability and flexibility in dynamic scheduling [39]. While the former is being addressed by High Level Synthesis (HLS) tooling [39], the latter is still an active area of research [8], and motivates the focus of this work.

1.1 Motivation

In HPC and datacentre contexts, flexibility in resource allocation — the ability to dynamically re-allocate resources depending on workload and hardware availability — forms an important aspect of a system's Quality of Service (QoS) [3]. This includes the ability to pre-empt and migrate tasks at runtime (dynamically) from one execution platform to another, allowing the system to re-distribute compute tasks efficiently (e.g. in load-balancing) for higher throughput and efficiency. Migrations that enable this fine-grained control are stateful: they require the transfer of the task's execution state — live variables, intermediate computations, etc — from the source to the destination machine such that execution can be resumed after migration [48]. This flexibility also provides the basis for dependable accelerated computing in HPC and datacentres by enabling the building of resilience against component-level failures, which helps them avoid undue service disruptions.

In the past, this flexibility in resource allocation was limited among FPGAs due to the lack of dynamic FPGA task pre-emption and scheduling capabilities [48]. Recent progress has changed that: currently-executing FPGA tasks can now be interrupted and resumed [39], or migrated to a different FPGA [48]. However, these works only target the homogeneous "FPGA-only" space. This flexibility is still limited in heterogeneous FPGA-accelerated systems/servers. At present, tasks can only be migrated in a non-stateful manner between CPUs and FPGAs [18][46]; there is not yet the ability to pre-empt and statefully migrate currently-executing tasks between hardware and software.

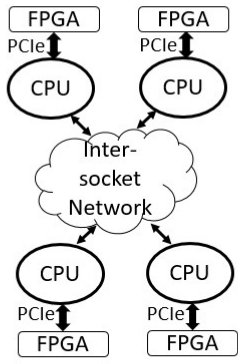


Figure 1.1: One CPU to One FPGA Configuration

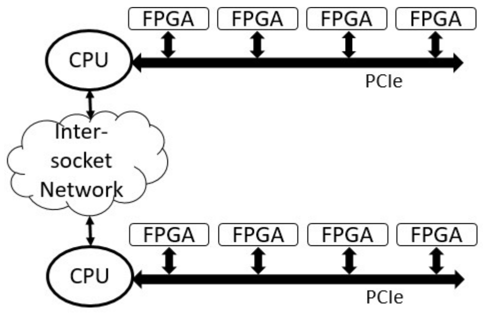


Figure 1.2: One CPU to Many FPGAs Configuration

The lack of such flexibility is especially problematic for FPGA-accelerated systems adopting the Single Node Accelerator Model (e.g. in AWS, Nimbix, Huawei) [8], where each node consists of a single host CPU connected one-to-one to a single FPGA (Fig1.1), or one-to-many to multiple FPGAs via PCIe buses (Fig1.2). In the one-to-one configuration, should an FPGA component fail, the currently-executing task would need to be migrated onto an operable FPGA on a different node, and is subject to infrastructural bottlenecks between nodes. The one-to-many configuration could suffer less from bottlenecks because the task could ideally be migrated to an operable FPGA on the same node. However in both configurations, finding a suitable FPGA to migrate to can be a challenge: many stateful context-saving FPGA-FPGA migration methods offered by FPGA providers require the source and destination FPGAs to have similar characteristics or be identical [48]. Since FPGA resources are often in contention in FPGA-accelerated systems, it can therefore be difficult to find a destination FPGA that is **1)** suitable and **2)** available, be it on the same node or on a different node. Hence, FPGA-FPGA migration in heterogeneous-compute environments can incur significant time costs — to re-allocate, re-schedule, re-configure (the new FPGA), migrate, and resume a task — which limits the ability of these systems to flexibly perform fine-grained resource (re)allocation. This can potentially disrupt the execution of FPGA-accelerated computing tasks, which impacts the dependability of the FPGA-accelerated system.

1.2 Report Focus and Contributions

This dissertation thus focuses on improving the flexibility in resource allocation in heterogeneous FPGA-accelerated compute systems, through *enabling stateful software-hardware task migration* between FPGAs and CPUs. This migration capability increases the granularity of dynamic resource allocation that the system can achieve, which in turn provides the basis to enhancing the dependability of the FPGA-accelerated computing it performs — e.g. time-consuming FPGA-accelerated tasks can be safely migrated from the FPGA to the host CPU on the same node in the event of FPGA component failure, without loss of intermediate progress/results.

This dissertation proposes custom compiler middle-end support, designed and implemented under the LLVM compiler framework, that inserts system-level checkpoint-restore mechanisms into the C/C++ source code of the user's acceleration task (kernel) at LLVM IR level. These mechanisms save the kernel's execution state at a given checkpoint, and restore execution at said checkpoint in the event of kernel pre-emption and migration. The proposed compiler support comprises various compiler passes that work on the kernel code at LLVM IR level to:

1. Pre-process the source code's IR (via selective splitting of Basic Blocks) to facilitate safe and efficient checkpoint insertion.
2. Perform liveness analysis to identify variables to track for each Basic Block in the source code.
3. Identify checkpoint locations via automatic selection methods or via programmer specification using a domain-specific directive.
4. Perform checkpoint insertion: i.e. insert new subroutines to save and restore execution state, and subroutines to manage the save/restore processes.

1.3 Scientific Contribution

Our compiler support is an integral component of a broader software-hardware task migration framework, built around open-source components of the Xilnx Vitis HLS toolchain for improved process automation. This migration framework is the subject of the paper titled “Datacenter: Dependable FPGA-based Acceleration” led by Dr. Maxime France-Pillois (Associate Researcher, ICSA, University of Edinburgh), and pursued in collaboration with Dr. Antonio Barbalace (Senior Lecturer in Operating Systems, ICSA, University of Edinburgh) and Huang Zihan (the author). This dissertation only focuses on the compiler support aspect of the paper; the design and implementation of the broader task-migration system is beyond this dissertation's scope.

1.4 Project Effort and Output

Approximately 900-1000 hours was dedicated to this dissertation over the 2022/23 Academic Year. This project and its proposed compiler support is open-source, and can be found at https://github.com/systems-nuts/junco-compiler_assisted_checkpointing.git.

Chapter 2

Background and Related Works

2.1 Background

2.1.1 FPGA Acceleration Model

In hardware acceleration, hardware-accelerated tasks (kernels) can be initially described in high level languages such as C/C++ that contain HLS pragmas specific to the accelerator hardware. The HLS toolchain will first lower kernel descriptions into Register Transistor Level (RTL) designs in a hardware description language (HDL) like Verilog. It then uses a synthesis tool to synthesize the RTL designs into logic gate level descriptions as bitstreams that are eventually mapped onto the FPGA fabric [18].

To manage the execution of these kernels, FPGA accelerators are paired with a host CPU. In OpenCL-esque accelerator models [20][45], the host transfers data from its memory to the accelerator's memory as inputs for processing by the hardware kernel. During execution, the kernel's outputs are written to buffers within the accelerator's local memory. After the acceleration has completed, the final output data is transferred from the accelerator's memory back to the CPU host.

Under these models, the kernel's output data is only copied to the CPU host if the FPGA does not fail during execution. If it does, the CPU host considers it as defective and no longer interacts with it; any computational progress it made is thus lost. For FPGA-accelerated systems that can detect such failures, the kernel can later be re-started on a new FPGA using the original input data [46]. Repeated kernel restarts due to FPGA failures thus do not affect the CPU host's state. Furthermore, kernel computations run in isolation from the rest of the platform; they do not induce any side-effects on the state of the FPGA, nor any other program executing concurrently on the FPGA [45].

2.1.2 State Representation in CPU vs FPGAs

The software (CPU) task context is broadly held as local storage data, user-space stack variables, and user-space-visible states of the CPU, i.e. a set of well-defined variables and pointers stored in memory segments (heap, stack) and CPU registers [6], as per

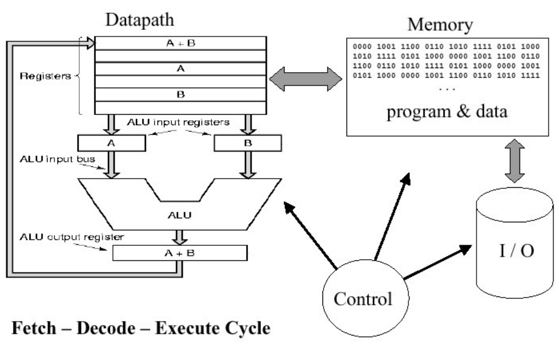


Figure 2.1: CPU Execution Model [42]

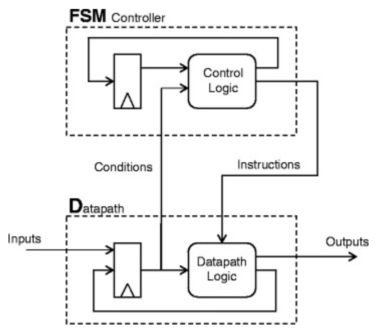


Figure 2.2: FPGA Execution Mode [41]

Fig2.1. In contrast, the hardware (FPGA) task context is held in set memories — registers, BRAMS, and DSP registers [46]. When synthesised into hardware configuration bitstreams, high level source code (e.g. in C/C++) will be "flattened" — as per Fig2.2, successions of function calls are converted into a dataflow format, and implicit variables are implemented through FSM states controlled by hardware. Therefore, equivalence points between these two representations will need to be established in order for execution context to be cross-compatible between CPUs and FPGAs [5]. Without this cross-compatibility, CPU-FPGA migration cannot be achieved.

2.1.3 Inside an LLVM Pass

LLVM is a compiler framework that allows a program to be compiled into the LLVM Intermediate Representation (IR) language to be analysed and transformed over multiple compiler passes [32]. A program's IR can be interpreted as a control flow graph (CFG) of basic blocks. When being processed by an LLVM pass, all values/variables/registers, instructions, basic blocks, functions, modules (etc.) in the input IR are represented within the pass' internal runtime memory, and can be accessed via pointers of the *Value*, *Instruction*, *BasicBlock*, *Function*, *Module* (etc.) classes respectively [31]. Instances of these classes store metadata associated with the IR unit they represent, e.g. name, memory allocation information, etc. We use the names of these classes interchangeably with the names of these IR units. All analysis and transformations performed on the input IR must be done via these pointers. Existing IR units can be modified by modifying their corresponding pointers' attributes; new IR units can be created by instantiating new objects, pointed to by new pointers. We term these pointers 'LLVM pointers'. Refer to Appendices B.2 and B.3 for further details on LLVM and LLVM IR.

2.1.4 Further Definitions and Concepts

This dissertation establishes and follows a specific set of definitions for Checkpointing, Migration and Execution Context. These definitions can be found in Appendix A. Further concepts relating to compilers, LLVM, and variable liveness analysis can be found in Appendix B.

2.2 Related Work

To the best of our knowledge, there are no existing works dealing with stateful CPU-FPGA task migration. Existing literature mainly address FPGA-FPGA task migration and non-stateful CPU-FPGA task migration. Nevertheless, the development of compiler support for stateful CPU-FPGA task migration benefits from the study of existing methods for **1)** hardware and software checkpointing, **2)** stateful migration of tasks between FPGAs, and **3)** non-stateful migration of tasks between CPUs and FPGAs.

This section therefore covers the related works concerning such task pre-emptions and migrations. These works enable a currently-executing FPGA task to be **1)** interrupted with the intention of being resumed at a later point in time, **2)** migrated statefully from one FPGA to another, and **3)** migrated non-statefully between CPUs and FPGAs. Since these are achieved via various methods of checkpointing and context extraction, this section will also cover existing works on hardware checkpointing, and compiler-based software checkpointing methods for state extraction and restoration.

2.2.1 Hardware Checkpointing

The ability to extract and restore execution state is fundamental for achieving task migration — when a task is migrated, its execution context must first be saved, and later restored on the destination machine where it resumes execution. This subsection studies the previous works on hardware checkpointing that enable FPGA-FPGA task migration, which help inform this dissertation's approach to checkpointing for state extraction and restoration in stateful CPU-FPGA task migration.

2.2.1.1 Formal Framework for Hardware Checkpointing

In [21], Koch et. al. extends the checkpointing concepts used in software systems [15] to hardware tasks running on FPGAs by proposing a formal hardware checkpointing framework, and various checkpointing mechanisms based on this framework. Their framework sets out to address situations where FPGAs encounter permanent faults during execution (i.e. component failure).

The proposed framework models hardware systems as Finite State Machines (FSM), and defines checkpointing as a technique that extracts and stores the execution context of a task during fault-free operation. These checkpoints are later used to prevent significant losses in computational progress in the event of FPGA faults via a "rollback" sequence where the task is restored to the last fault-free state, e.g. on a separate device.

Checkpointing in hardware involves extracting a subset or all of a hardware module's register and memory values. Koch et. al. proposes a variety of hardware checkpointing methods that can be applied directly to an FPGA module to automatically generate the Checkpoint-enabled FSM for the checkpointed hardware task. These methods are integrated into system design flows via an accompanying command line tool called StateAccess that works at the Netlist level.

Koch et. al. [21] is one of the first to formalise hardware checkpointing methods, and informs many later works on hardware state-extraction and restoration, for example

[9]. However, their work is designed to work solely at the hardware level, and the checkpointing mechanisms are not cross-compatible with CPU versions of the tasks. Hence, the methods proposed here are insufficient to realise the migration of acceleration tasks between CPUs and FPGAs.

2.2.1.2 HLS Design Flow for Context-Switch Capable Circuits

Bourge et. al. [9] proposes a HLS design flow to generate efficient context-switch capable circuits in FPGAs that can support multi-tenant usage. Their work inserts a hardware scan-chain — a serial link between each flip-flop — into the initial hardware circuit at chosen checkpointing locations to extract the contents of flip-flops or memory.

This design flow interprets hardware tasks as classical FSMs comprising states and a datapath, where each state has a set of live variables that represent the current task execution context at that state. The values of these live variables will change unpredictably during task execution; they cannot be tracked from an external host CPU, and must thus be stored explicitly by checkpoints within the task. These live variables are generated by performing liveness analysis on a Hierarchical Task Graph (HTG) obtained from the HLS tool.

To minimise the size of memory required to store the checkpointed data, this design flow selects the computing point with the fewest live variables as checkpoints. Each time a checkpoint is reached during execution, the application context is saved from the flip-flops/memory. This execution context can be restored back to the previously-saved checkpoint state by the checkpointing infrastructure.

This approach by [9] is hardware-centric — the checkpointing infrastructure is inserted directly as hardware elements into the FPGA fabric. While this is applicable to purely hardware-oriented checkpointing and context-switching, this method only enables context-switching in FPGAs. This method is not sufficient for CPU-FPGA application cases because the saved application context is in a hardware-specific form that is not suitable for interpretation and usage on a CPU.

2.2.1.3 CPRtree

CPRtree [47] is a checkpointing architecture for FPGAs that performs systems-level checkpointing at the Hardware Description Language (HDL) level.

CPRtree's checkpointing mechanisms use a reduced set of "state-holding elements" to represent the entire execution state of the hardware-accelerated task. These elements comprise of the registers, RAMs and wires defined in the HDL source code; correct restoration and resumption of task execution requires that all of these elements be restored. A reduced set is generated by removing elements in the set that are interpolated from other elements in the set. CPRtree's checkpointing infrastructure comprises context-capturing/restoring hardware for registers and RAM; CPRtree uses a python-based tool to insert checkpointing infrastructure into the user's HDL code.

Since CPRtree works at the HDL level, it positions itself before the hardware synthesis stage of the hardware design flow, and can thus be ported across different hardware platforms. However, CPRtree is hardware-specific, meaning that its checkpointing infrastructure cannot be used in a CPU-FPGA migration context, where both sides need to have equivalence points in terms of state representation.

2.2.2 Hardware-Hardware Migration

This subsection studies the various related works that apply checkpoint-based methods to perform state extraction and restoration for the purposes of FPGA task pre-emption and FPGA-FPGA task migration. These works help inform this dissertation's approach to using checkpoint-based methods for CPU-FPGA task migration. At the end of this subsection, we access the overarching limitations of the FPGA-FPGA task migration methods when applied to a CPU-FPGA migration context.

2.2.2.1 FPGA Task Pre-emption

In the FPGA-only space, [39] presented a domain-specific language for dynamic and pre-emptive scheduling of hardware tasks for reconfigurable SoCs in embedded devices. These tasks are OpenCL kernels managed using the Controller model [37].

This language (and programming abstraction) provides programmers with a set of pre-defined checkpointing macro-functions to specify the locations of the checkpoints, and the variables to be saved. One such macro is `checkpoint`, which stores a user-provided set of variables at a given point in program execution. Another is `for_save`, which saves the state of a for-loop at each iteration so that the kernel can be resumed from within the loop without losing the computational progress made by any previous iterations. These software abstractions work together with an accompanying on-chip hardware infrastructure to help ensure that the kernels are resumed from a consistent state.

However, this abstraction relies on the user to explicitly indicate the variables that are to be saved at each checkpoint. Hence, the correctness of the pre-empt/resume mechanism depends on the user's ability to correctly calculate/identify the sets of variables needed for accurate state representation at each checkpoint. This adds complexity to the workflow and introduces potential sources of human error.

2.2.2.2 FPGA-FPGA Task Migration

Wicaksana et. al. [48] recognized the lack of native task-migration support on FPGAs, and proposed a framework to enable dynamic FPGA to FPGA migration — a task running on an FPGA can be suspended and resumed on another, by extracting and restoring FPGA registers and memory values using task-specific state-extraction mechanisms implanted into the task. This framework manages FPGA configuration and the extracted contexts using an embedded processor tightly-coupled to the FPGA.

Wicaksana et. al. [48] leverages on previous work done by [9] to assist in the insertion of context-extraction mechanisms into the hardware tasks as part of the HLS flow. These mechanisms include elements such as state machines, connections, and partial scan-chains shared between checkpoints with the same execution context.

One limitation of this migration method is speed: the hardware scan chains on which this approach relies for state saving and restoration are slow. Scan chains also prevent certain low-level hardware optimisations such as re-timing and register duplication, and can affect certain place-and-route steps and the usage of vendor-specific Intellectual Properties [9].

2.2.2.3 Overall Limitations of Hardware-Hardware Migration

Overall, the checkpoint-based FPGA-FPGA task pre-emption and migration methods studied here are insufficient for CPU-FPGA migration, because the latter poses a different set of challenges from the former: CPU-FPGA migration raises equivalence issues with respect to how execution state is represented and stored in a CPU compared to on an FPGA (Section 2.1.2), and "raw" execution state extracted from FPGAs is incompatible with the CPU's model of state representation. CPU-FPGA migration thus requires establishing equivalence points between these two different representations of execution context in order for tasks to be migrate-able between FPGA and CPU in a stateful manner. Unfortunately, such equivalence points are beyond of the scopes of the hardware-to-hardware migration schemes studied here.

2.2.3 Hardware-Software Migration

Research has been done on non-stateful CPU-FPGA task migration for the primary purposes of improving resource management. By virtue of their CPU-FPGA use case, these works relate more closely with this dissertation compared to the FPGA-FPGA migration works. This subsection studies works that achieve non-stateful CPU-FPGA task migration by either targeting OpenCL-specified FPGA accelerators, or providing a more generic migration capability.

2.2.3.1 Xar-Trek: Coarse-grained CPU-FPGA Migration

Xar-Trek [18] is a compiler and runtime framework that targets the dynamic provisioning of FPGAs in multi-tenant datacentres as a way to alleviate dynamic workload spikes on servers. This framework allows run-time migration of application functions from host CPUs to FPGAs, primarily for the purposes of coarse-grained resource allocation.

Xar-Trek compiles an application into multi-ISA binaries, including FPGA implementations of a select set of application functions meant for acceleration on an FPGA. These functions are mapped to FPGA logic resources using hardware description tools such as Xilinx's Vitis, and then executed on the FPGA for their entire lifetime. After execution, the results of the acceleration are transferred back to the host CPU.

Xar-Trek's CPU-FPGA migration works by converting entire functions into hardware implementations, and can only achieve software-hardware migration at function boundaries. While this is sufficient for the purposes of coarse-grained resource-allocation, it is insufficient for more fine-grained migrations that require pre-emption from within an application function, because that would require execution state to be extracted and restored regardless of execution progress, i.e. including within function boundaries.

2.2.3.2 Live Migration: Non-stateful CPU-FPGA Migration for OpenCL-based Acceleration

Vaishnav et. al. [46] proposes a general migration approach that targets OpenCL-specified accelerators that run OpenCL work-groups. Their approach achieves coarse hardware-software migration by saving and restoring the execution context of FPGA accelerators only at points in execution where there is little to no internal state propagation from the current execution step to the next execution step. In the OpenCL execution model, these "consistency points" correspond to the end of a work-group execution, where no inter-work-group synchronisations nor global memory write-backs occur.

However, this proposed approach is an OpenCL-specific solution that cannot be used on kernels for which the OpenCL work group model is unsuitable — these include legacy kernels that cannot benefit from this model unless they are modified. Additionally, while [46] claims to handle FPGA to CPU migration, they only demonstrated FPGA to FPGA migration.

2.2.4 Compiler-based Software Checkpointing

This section studies existing works on compiler-based methods used to insert software checkpoints for state extraction and restoration. These works relate closely to the proposed compiler support that this dissertation develops. Note that the type and purpose of the checkpoints developed in these works differ from that used in this dissertation — the checkpoints in this section cover a region of code, and are designed to be used mainly for execution rollback instead of task migration. Nevertheless, the methods used by these works help inform this dissertation's design of the proposed compiler support for checkpoint insertion for fine-grained CPU-FPGA task migration.

2.2.4.1 Compiler-based Software Checkpoint Regions Targeting `store` Operations

Zhao et. al. [52] proposed a software checkpointing framework implemented through transformation and optimisation passes in the LLVM compiler infrastructure. This framework provides programmers with an API to specify sections of the program code as checkpoint regions, and uses its compiler component to insert checkpointing infrastructure into the programmer's source code via a series of compiler-driven transformations. The checkpoints achieve a granularity of individual variables (by tracking individual `store` operations) instead of entire objects or ranges of memory, which helps increase efficiency and reduce checkpointing overheads.

For each specified checkpoint region in the given program code, the compiler instruments all relevant write/store operations with function calls to back up the values of the stored variables to memory, and inserts 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. The compiler ensures that checkpointing infrastructure is inserted for every user-defined routine that is potentially reachable from within the checkpoint region, by traversing and transforming each node in the sub-call-graph that originates from the checkpoint region. The compiler then performs a series of optimisations to remove redundancies and improve efficiency.

This framework allows unlimited rewinds of program execution to desired locations within the checkpointed region, which has been shown to be helpful for debugging arbitrarily large sections of application code. This framework has also been shown to help simplify the process of implementing backtracking capabilities for the VPR [7] place-and-route algorithm.

However, while this approach allows for checkpoint-rewind and backtracking operations, it only allows them within the checkpoint region. Moreover, it only tracks variables that are part of `store` operations within the checkpoints, and does not track live variables. Even though there can be overlaps between these two sets of variables, the former does not precisely represent the data needed to fully restore execution from the checkpointed location (as required in migration). In fact, variables that are live-out of the checkpoint location (i.e. which are required in later parts of execution and thus would need to be restored during migration) might not be involved in store operations within the checkpoint region, and would thus be omitted. Hence, this framework is insufficient for the type of execution restoration that is required for CPU-FPGA task migration.

2.2.4.2 Bolt: Compiler-directed Checkpointing Targeting Idempotent Regions of Program Execution

Bolt [26] is a compiler-based soft error recovery scheme that enables error recovery on transient faults via re-execution of errored code regions. It is designed to carry out idempotence-based recovery, where checkpointing granularity is at the level of individual idempotent regions in code.

Bolt defines idempotent regions as regions which are single-entry multiple-exit (SEME) subgraphs of the program code's CFG, that generate the same output whenever the program execution jumps from any point within the region back to the region's entry point. In other words, the inputs for these regions do not change when execution jumps back to the start of the region. Bolt partitions and transforms the program code into idempotent regions, and checkpoints each region at the region boundaries. When a soft error is detected during the execution of a checkpointed region, Bolt restores all the input values for the (faulted) region, redirects program control flow back to the region's entry point, and re-starts execution.

Bolt and other idempotence-based checkpoint-recovery schemes can only achieve state restoration at the boundaries of idempotent regions of code. While this is sufficient for execution rollbacks for the purposes of error recovery, this method is unsuitable for stateful CPU-FPGA task migration, because FPGA tasks might not be able to be partitioned into idempotent regions of code. Even if they are, the region boundaries where checkpointing occurs might not be suitable migration points for the given task/kernel. The programmer thus has limited flexibility in deciding the checkpointing location, and cannot do so with a high degree of granularity, which limits the flexibility of migration that can be achieved with this method.

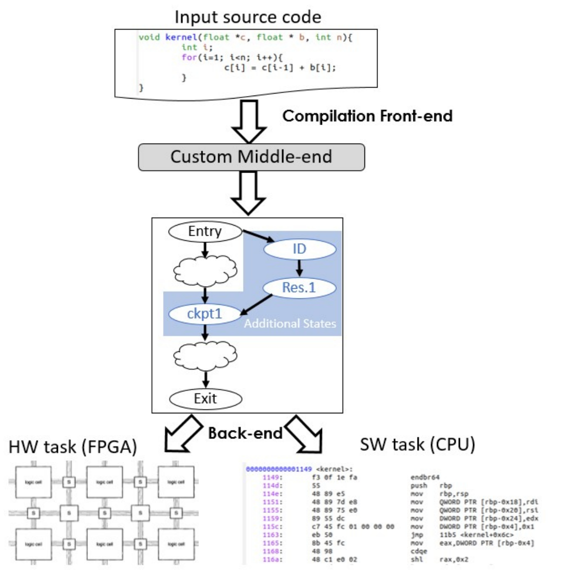
Chapter 3

Overview and Approach

3.1 High-level Approach

Heterogeneous task migration is non-trivial due to how execution state is represented differently between different architectures. Even amongst CPUs, processors of different instruction set architectures (ISA) represent execution contexts in different formats under varying address space layouts; dedicated support is needed to establish equivalence points between them before native applications can be migrated without emulation [6]. Migrations between CPUs and FPGAs pose an even greater challenge because the differences in state representations are greater (Section 2.1.2).

We therefore propose dedicated support to establish equivalence points in kernel representation between CPUs and FPGAs, so that checkpoints — and the data saved — are cross-compatible. Our approach takes advantage of the Xilinx Vitis High-Level Synthesis (HLS) flow to specify a common C/C++ kernel source code for both CPUs and FPGAs, which we instrument with checkpoints (as shown in Fig3.1). This checkpointed C/C++ code is later compiled into the CPU version of the kernel; the FPGA version is obtained via HLS synthesis into hardware implementation bitstreams. The checkpoints inserted are thus common to both versions, and load/store execution context data in a common format. This allows the context extracted from one version to be restored into the other. In the Vitis HLS toolchain, LLVM IR is the last common representation of the kernel code before it is separately processed into CPU binaries and FPGA bitstreams. Thus, we perform our source code analysis and transformations at the LLVM IR level.



```
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
```

Figure 3.1: Common Source Representation

We thus enable stateful software-hardware task migration via a custom LLVM middle-end compiler support that inserts checkpoints into kernel code at IR level. This insertion can be driven either by explicit user-direction, or an automated checkpoint selection scheme. Checkpoint insertion is fully automated — the compiler support automatically determines the variables to save and restore, and inserts custom subroutines into the kernel to do so. These subroutines (checkpoints) save/restore checkpointed data to/from an externally-allocated memory segment '*ckpt_mem_seg*'. Our compiler support is accompanied by a lightweight domain-specific programming model and annotations framework

3.2 Our Approach vs Related Works

Our compiler support is designed to work as part of the HLS flow, and uses live variables to represent execution state. Unlike in [39], liveness analysis is automated — thereby minimising human error — and is done on the kernel code's Control Flow Graph (CFG). We overcome the limitations of coarse-grained migration as seen in [18][26][46] by performing checkpoint insertion within function boundaries, which enables more fine-grained migration. Since our design does not rely on the OpenCL Execution model, it is compatible with a broader range of kernel codes than in [46]. Crucially, unlike existing hardware-oriented checkpointing and migration works, we insert checkpoints into programmer code at LLVM IR level via compiler passes instead of into hardware further down the HLS flow. However, unlike in [26][52], the checkpoints that our passes insert are designed for the explicit purpose of task migration, not for backtracking/rewinding or error recovery.

3.3 Checkpoint Design

We define a checkpoint as a point in program execution at which execution state/context is extracted and later restored, and from which execution can resume. We use 'checkpoint' interchangeably with 'checkpoint infrastructure' to refer to the additional subroutines used to implement checkpointing operations. We term a checkpoint's save and restore operations collectively as 'checkpoint-restore'. A Basic Block (BB) selected for checkpointing is termed a 'checkpointed BB'; checkpoints are inserted immediately after their corresponding checkpointed BBs. We use the set of 'live-out' variables from each BB to represent the kernel's execution context. Broadly, the live-out variables of a BB q are those used in BBs reachable from q (Appendix B.1). If taken from well-chosen points in the kernel, these live-out sets can include intermediate results from the kernel's computations. Each checkpoint is thus designed to save and restore the live-out data — also referred to as 'tracked' data — of its corresponding checkpointed BB. Refer to Appendices A.1 for detailed definitions on checkpoints and A.3 for execution context.

3.4 Checkpoint Memory Segment Design

Designing a storage solution for checkpointed data is a challenge. Each *Module* can have multiple checkpoints that must each store its (possibly unique) set of tracked variables into memory, and later correctly locate and retrieve it during restoration. A naive design might allocate separate memory segments for each checkpoint. However, this causes heavily-checkpointed kernels to incur large memory overheads. Moreover, since our migration model (Appendix A.2) restores execution state to the last executed checkpoint *ckpt* before migration, only the context at *ckpt* is needed to restore kernel execution; the contexts stored by previous checkpoints are not used.

Our compiler support thus uses only a single memory segment '*ckpt_mem_seg*' to store the checkpointed data for all checkpoints in a *Module*. Since our compiler passes do not modify the program logic within each BB, each checkpoint's tracked variables set is static for a given kernel code, meaning that its total size (in bytes) remains constant. We therefore configure *ckpt_mem_seg* as a statically-sized, contiguous region of memory (as shown in Fig3.2) sized to accommodate the largest checkpointed set of tracked variables (in bytes) in the entire *Module*. It is of the datatype *ckpt_mem_type* corresponding to the widest, highest-precision type used in the kernel code; values not of this datatype are type-converted. Since *ckpt_mem_seg* must be accessible by both the FPGA and host CPU for context-transfer during migration, the broader migration framework allocates it as a global shared buffer outside the scope of the kernel.

Memory Segment	
0	ckptID
1	isComplete
2	trackedVal_2
3+0	trackedArr[0]
...	...
3+(n-1)	trackedArr[n-1]
3+n	trackedVal_1

Figure 3.2: Checkpoint memory Segment

Arrays are stored in *ckpt_mem_seg* as unbroken contiguous blocks, with individual values interspersed between them, as shown in Fig3.2. Since the tracked variables set is static for each checkpoint, checkpoints are designed to "remember" the index positions in *ckpt_mem_seg* where each tracked variable is stored. These values and arrays can thus be stored in any deterministic and consistent ordering within *ckpt_mem_seg*. Refer to Section 4.3.2.4 for more details.

ckpt_mem_seg is overridden each time a new checkpoint is executed, and thus always contains the most recently-checkpointed execution context. Each checkpoint in the *Module* is assigned a unique ID; when performing context-saving operations, a checkpoint will write its ID to a dedicated `ckptID` field (shown in Fig3.2) in the *ckpt_mem_seg*. During context-restoration, this stored ID ensures that the tracked variables stored in *ckpt_mem_seg* are correctly retrieved by the correct checkpoint. `ckptID` also serves as a "lock" variable to synchronise the separate read/write operations to *ckpt_mem_seg* performed by the FPGA and the CPU host. Refer to Sections 3.6 and 4.3.2 for more details. *ckpt_mem_seg* also includes an `isComplete` metadata field for testing and accountability; more information on this can be found in Appendix D.2.

3.5 Checkpoint Selection Methods

Our compiler support provides a user-directed checkpoint selection scheme, where the user specifies checkpoint locations by placing our domain-specific `checkpoint()` directive (Section 3.7) at one or more locations in the kernel source code, e.g. within a for-loop as shown in FigA.3. Our compiler support identifies these directives and inserts checkpoints at their locations. This is further detailed in Section 4.3.1. We also implemented an automatic checkpoint-selection algorithm that, due to time constraints, was not integrated into the current version of our compiler support. Refer to Appendix C for more details on this algorithm.

3.6 Checkpoint Infrastructure

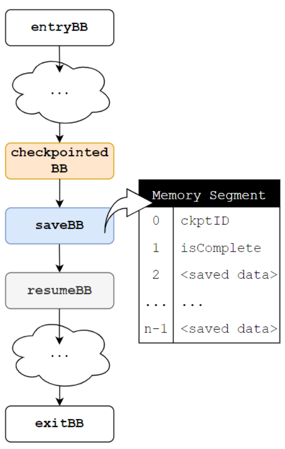


Figure 3.3: saveBB

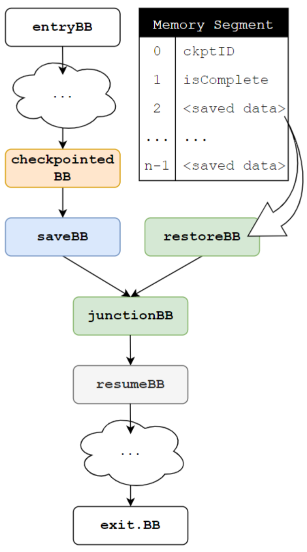


Figure 3.4: restoreBB & junctionBB

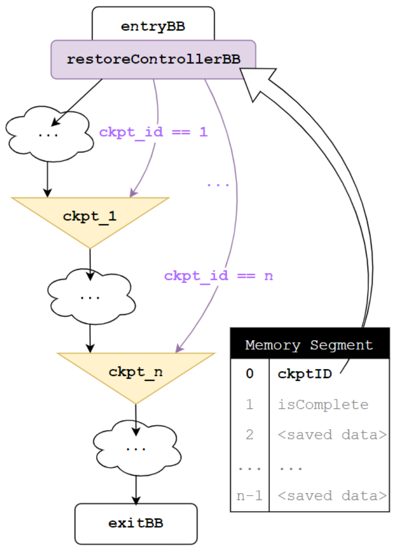


Figure 3.5: restoreControllerBB

Checkpoints are implemented by inserting new subroutines — new instructions and BBs — into the kernel IR at the chosen checkpoint locations. The "save" and "restore" subroutines are inserted at each checkpoint to perform checkpoint-restore. Each checkpoint has their own set of "save" and "restore" subroutines. The "restoreManager" subroutine is inserted within the function's entry block (entry BB) to control which checkpoint to restore execution to.

3.6.1 Save Subroutines: saveBB

The save subroutines are implemented via our saveBB BB type. As described in Appendix A.1 and shown in Fig3.3, we insert the saveBB (blue) between the checkpointed BB (orange) and its successor (resumeBB) further down in the CFG; the saveBB stores the contents of each checkpointed BB's tracked variables into *ckpt_mem_seg*. It also stores metadata such as the checkpoint ID and the `isComplete` flag.

3.6.2 Restore Subroutines: restoreBB & junctionBB

The restore subroutines are implemented via our restoreBB and junctionBB BB types. During migration, execution context is restored by the restoreBB by loading previously-saved checkpointed variables from *ckpt_mem_seg* (shown in Fig3.4). Since LLVM IR follows SSA form, existing variable registers cannot be re-defined; the restoreBB must thus restore certain tracked variables into new registers. The 2 resultant versions of these tracked variables — 'saved' (original) and 'restored' — are resolved via PHI instructions in the junctionBB (Fig3.4). If program flow comes from the saveBB branch, the PHI instructions' results use the contents of the original registers. Else, they use the contents of the new registers. These resolved value versions are then propagated across the CFG via a propagation algorithm (Section 4.3.3).

3.6.3 RestoreManager Subroutine: restoreControllerBB

A function instrumented with multiple checkpoints will have multiple restoreBBs. We thus design a restoreControllerBB to decide which restoreBB to use during migration. This BB is embedded "within" the entry BB of the checkpointed function's CFG as shown in Fig3.5. At the start of the restoration process, the function re-executes the entry BB and enters the restoreControllerBB, which reads the checkpoint ID field from *ckpt_mem_seg*. This ID corresponds to the most recently-executed checkpoint in the function (as per Section 3.4), i.e. the checkpoint to restore execution from (Appendix A.2). The restoreControllerBB uses this ID to decide which checkpoint's restoreBB to jump to to perform this restoration. Refer to Section 4.3.2.2 for more details.

3.7 Domain-specific Programming Model & Annotations

We design a lightweight domain-specific programming model and annotation framework to assist our compiler support during checkpoint insertion, and to provide it with important metadata that cannot be obtained directly from within the kernel IR.

3.7.1 Domain-specific Programming Model

Our model enforces the following standards:

1. **`ckpt_mem` Pointer as Input Parameter:**
   Since *ckpt_mem_seg* is allocated outside the scope of the kernel, the compiler support cannot retrieve its base address from the kernel IR. Each function in the kernel must therefore include an input parameter `ckpt_mem` (Fig3.6) that points to *ckpt_mem_seg*. Details on how this pointer is used by our compiler support can be found in Section 4.3.2.4. In the final migration framework, this parameter will be added automatically by a separate pre-processing script.

2. **`checkpoint()` Directive:**
   For user-directed checkpointing (Section 3.5), the user inserts `checkpoint()` function calls at desired checkpoint locations in the kernel, as seen in FigA.3. The compiler support locates these `checkpoint()` calls and performs checkpoint insertion at these locations using auto-calculated variable liveness data.

```
/*#FUNCTION_DEF#*/
/* FUNC cholesky_kernel : ARGS diagSize{const}[], matrixA{}[262144] */
void cholesky_kernel(const int diagSize, dataType* matrixA, dataType* ckpt_mem) {
```

Figure 3.6: Domain-specific Annotations

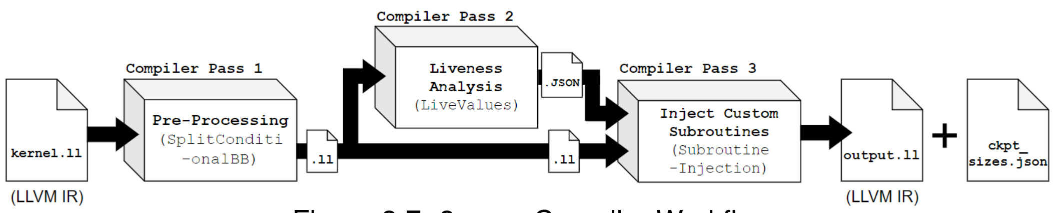


Figure 3.7: 3-pass Compiler Workflow

3.7.2 Domain-specific Annotations

As per Section 3.4, *ckpt_mem_seg* stores both single values and arrays of values, accessed during checkpoint-restore via their index positions. Fig3.2 shows how the index position of `trackedVal_1` depends on the length *n* of the array `trackedArr`. Correct indexing and data access into *ckpt_mem_seg* thus requires knowing the length of each array stored within it. However, kernel functions often take in pointers to arrays allocated outside kernel scope as input arguments, because they contain input data for kernel computations (e.g. `dataType* matrixA` in Fig3.6). Unlike with locally-allocated arrays, the kernel IR contains no metadata on the lengths of these externally-allocated arrays, which makes evaluating their lengths a challenge.

We thus use domain-specific annotations to provide auxiliary information on such input arguments. As demonstrated in Fig3.6, users must preface each function definition with the `/*#FUNCTION_DEF#*/` tag, followed by a comment with the "`<arg_name>{<is_const>} [<arr_len>]`" substring format for each argument. `arr_len` details the array length of an input array pointer, and must be specified as an integer string. If this argument is checkpointed, this field helps calculate its size and the offsets used to index *ckpt_mem_seg* (Sections 4.2.3 and 4.3.2.3), and the total memory to allocate for *ckpt_mem_seg*. The `is_const` field details whether an input argument has the '`const`' attribute (e.g. `diagSize` in Fig3.6). Since `const` input arguments are not modified within the function, this annotation tells our compiler support to exclude it from checkpointing to reduce memory usage. These fields can be left empty if not applicable for a given argument. `ckpt_mem` is sized and handled later by the broader migration system, so no annotation for it is needed.

3.8 3-pass Compiler Workflow

We design our compiler support as a custom 3-pass compiler flow for use in the middle-end of a compiler framework. It is designed to only work with kernels adhering to our domain-specific programming model and annotations format. The kernel code is first compiled by clang++ into LLVM IR. The 3-pass flow consumes this IR and 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 Fig3.7, and are as follows:

1. **Pass 1: SplitConditionalBB (IR pre-processing)**
   This pass performs per-*Function* pre-processing transformations (BB-splitting) on an unmodified kernel IR file `kernel.ll` to facilitate subsequent subroutine injection. It emits the pre-processed IR `split_kernel.ll` as output.
2. **Pass 2: LiveValues (liveness analysis)**
   This pass performs per-*Function* liveness analysis for each BB for a given *Function* in the pre-processed `split_kernel.ll` IR *Module*. Analysis results are outputted in JSON files `live_values.json` and `tracked_values.json`.
3. **Pass 3: SubroutineInjection (checkpoint insertion)**
   This pass consumes the analysis data from LiveValues, and performs *Module*-level transformations on the pre-processed `split_kernel.ll`. Checkpoint locations are identified either via automatic selection or by locating user-directed `checkpoint()` directives in the kernel. Checkpoints are inserted in the form of additional BBs and edges in the IR CFG. This pass outputs the final checkpointed IR `split_kernel_out.ll` together with a JSON `ckpt_sizes.json` of checkpoint sizes (bytes). The latter is used by the broader migration system to allocate *ckpt_mem_seg* memory.

3.9 Integration into Broader Migration Toolchain

Fig3.8 shows an overview the compiler toolchain used by the broader CPU-FPGA task migration system. This toolchain takes both the kernel's and the CPU host's source codes as input, and outputs binaries for each platform. The Kernel Code is the source code for the FPGA-accelerated task; the Host Code (outside this dissertation's scope) is responsible for transferring checkpointed data between the FPGA and the CPU host, and for performing the actual migrations. The right-hand-side flow shows the HLS workflow that lowers the kernel source code (C/C++) into hardware bitstreams. This dissertation's proposed custom compiler support (highlighted in orange) is integrated as an enhancement within the HLS tool's compilation stage. It is so placed to perform modifications on the last "common-representation" form of the kernel code before it is separately lowered into CPU binaries and FPGA bitstreams. Our proposed domain-specific programming model and annotations framework (highlighted in green) is applied at the C++ source code level. Toolchain stages outside the highlighted regions are beyond the scope of this dissertation

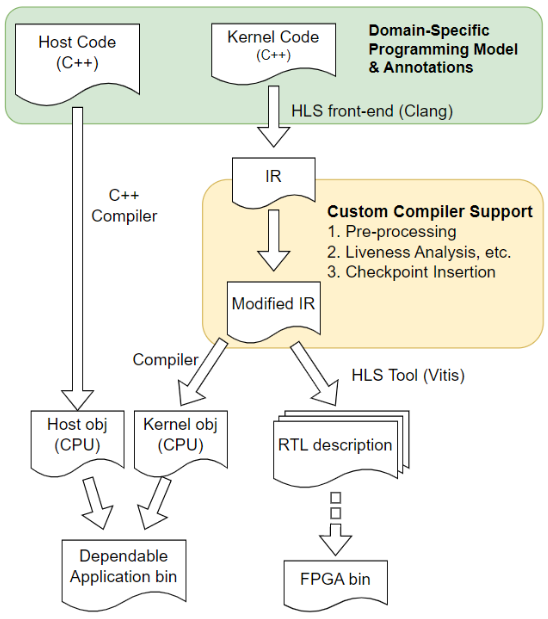


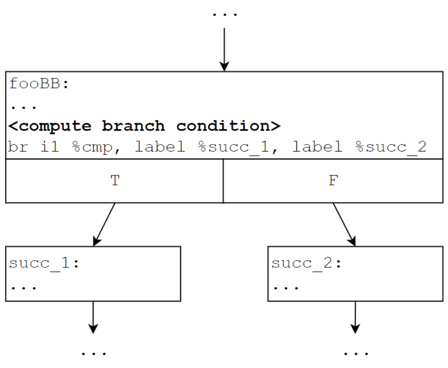
Figure 3.8: Toolchain Overview

Chapter 4

Compiler Pass Design and Implementation

4.1 The SplitConditionalBB Pass

The SplitConditionalBB pass is a transformation *FunctionPass* (Appendix B.3) that performs pre-processing operations on a given *Function* in the kernel IR, and is the 1st stage of our 3-pass workflow. It splits each *Function*'s entry BB, and any BBs that terminate with conditional branch instructions. This pass is run for each *Function* in the *Module* before moving on to the next compiler workflow stage.



...
fooBB:
...
<compute branch condition>
br i1 %cmp, label %succ_1, label %succ_2
T
F
succ_1:
...
succ_2:
...
...
...

Figure 4.1: BB with multiple successors

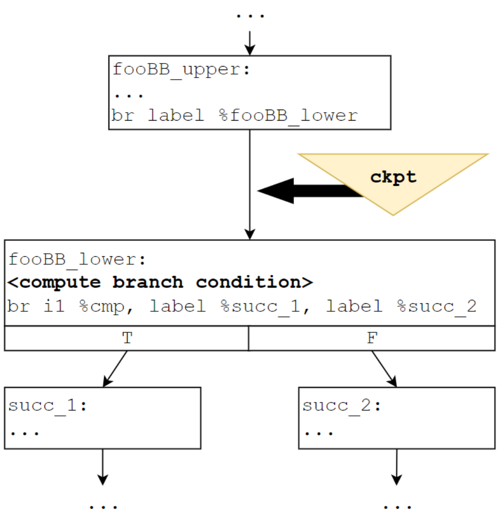


Figure 4.2: Splitting and checkpointing

4.1.1 Splitting Conditional-Branched BBs

As per our checkpoint definition in Appendix A.1 and shown in Section 3.6, checkpoints are inserted between checkpointed BBs and their successors. However, handling checkpointed BBs with multiple successors (e.g. in Fig4.1) is a challenge. A naive approach might insert one checkpoint on each of `fooBB`'s exit edges. However, this approach doubles the number of checkpoints inserted for each checkpointed BB, which can translate (in the worst case) to a x2 increase in the number of checkpoints inserted into a program, and lead to bloat in the final checkpointed IR.

We overcome this challenge by pre-splitting all BBs with conditional branching. If a BB in a given *Function* terminates with a conditional branch instruction, the pass locates the instruction that computes the branch condition via instruction back-tracking. This instruction is often located immediately before the conditional branch instruction. The pass then splits the checkpointed BB at this instruction into upper and lower BBs via LLVM's `splitBasicBlock()` API . As per Fig4.2, the checkpointed BB `fooBB` is split into `fooBB_upper` containing all of `fooBB`'s instructions up to before the branch condition instruction, and `fooBB_lower` containing only the condition and branching instructions. We thus need only insert a single checkpoint `ckpt` between the upper and lower BBs. The variables used to compute the branch condition will be live-out of `fooBB_upper` and saved by `ckpt` as part of the execution context. After migration, they are restored and used in `fooBB_lower` to compute the correct branch to take.

However, the branch condition can sometimes be calculated separately in one of the checkpointed BB's predecessors, and be live-in to the checkpointed BB. In this case, we split the BB at just the (conditional) branch instruction. Since the condition variable will be live-out of the upper BB, it will be saved by the checkpoint and thus available to the lower BB upon state restoration. Note that this approach of splitting at the conditional branching instruction can also be applied to the case demonstrated previously via Fig4.2 to yield a more generic/unified solution. This optimisation was discovered late in the implementation stage, and will be included in future versions of this compiler support.

This pass ignores *Functions* with only a single BB because it is likely more efficient to simply re-run such *Function*s upon migration than to restore it via checkpoints.

4.1.2 Splitting entryBB

This pass also implements mandatory splitting of entry BBs in the manner detailed in Section 4.1.1 — regardless of whether they terminate with conditional branching — so that the restoreControllerBB can later be embedded "within" the entry BB, i.e. between `entryBB_upper` and `entryBB_lower` in Fig4.5. This design executes all variable declarations in the entry BB before diverting execution flow to `entryBB_lower` or an appropriate restoreBB. This ensures that all uses of these variables are correctly dominated by their definitions, and is required even during restoration because checkpointed arrays are restored back into these pointers later in the restoreBBs (Section 4.3.2.4).

4.2 The LiveValues Pass

The LiveValues pass is an analysis *FunctionPass* that performs liveness analysis and variable size analysis on a given *Function* in the inputted IR, and is the 2nd stage of our 3-pass workflow. This pass is run for each *Function* in the *Module*. Complete analysis results of all *Function*s in the *Module* are aggregated in JOSN files to be consumed by the final SubroutineInjection pass.

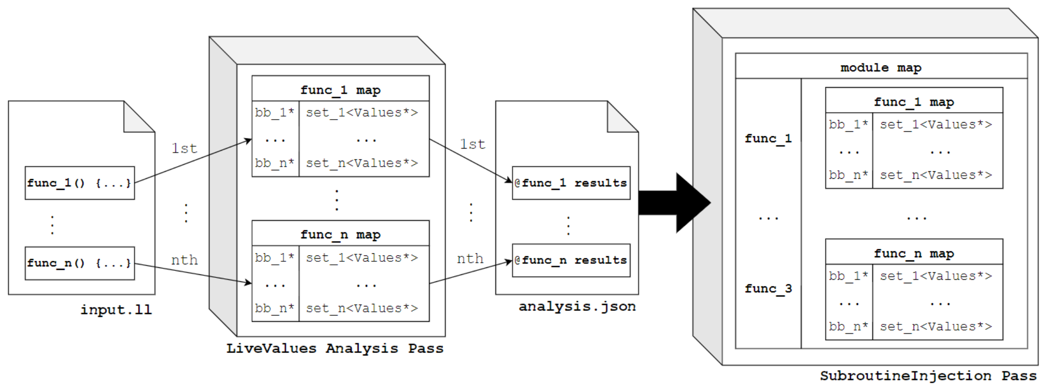


Figure 4.3: JSON Workaround

4.2.1 Performing Liveness Analysis

The liveness analysis component of LiveValues is adapted from the the open-source Popcorn Linux Compiler toolchain [36], which performs liveness-analysis based on the non-iterative dataflow algorithm for reducible CFGs from [10]. This implementation first computes partial liveness sets via postorder traversal of the CFG, then refines them by traversing the loop-nesting forest to propagate liveness within loop bodies. It outputs the *Function*'s liveness data as a C++ map of LLVM pointers containing a set of live-in and live-out *Value*s for each BB in the *Function*. The design and implementation of this liveness-analysis algorithm is beyond the scope of this dissertation.

Additionally, LiveValues collates the *Function*'s live-out data into a separate (but similarly-structured) map of *Value* pointers representing the set of tracked variables for each BB in the *Function*. This map is produced for each *Function* in the *Module* to facilitate checkpoint selection and insertion later by the SubroutineInjection pass (Section 4.3). These maps (e.g. `func_1 map`) are shown within LiveValues in Fig4.3.

4.2.2 The JSON Workaround

A major challenge in this dissertation is designing the data transfer 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 LLVM7 framework that does not allow such data transfer between *FunctionPass*es and *ModulePass*es (unless in special situations). Although LLVM7 does allow 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 (Section 4.3).

Since it is unfeasible for this dissertation to modify all the relevant dependencies, we instead perform this data transfer using JSON files. Fig4.3 shows the incremental aggregation and final collation framework for tracked variables data; an equivalent framework is used for transferring liveness data. Each time LiveValues runs and generates analysis data for a *Function* in `input.ll`, it writes it incrementally to `live_values.json` and `tracked_values.json`. After LiveValues is run over all *Functions* in the *Module*, these files would have aggregated *Module*-level analysis data that can be consumed by the SubroutineInjection pass, where they are used to re-construct the C++ maps of LLVM pointers produced by each LiveValues execution. These maps are then collated into a single nested *Module*-level map of analysis results.

Since LLVM pointers hold runtime-specific addresses, they cannot be used to re-construct the LLVM pointer maps in a different runtime instance. Our JSONs thus store the names of the IR units they represent instead. With these, the SubroutineInjection pass (re)constructs the nested map by finding LLVM pointers in its own runtime that matches these names. This is only possible because each LLVM pointer resolves to a unique name that is derived directly from the IR. These names remain constant unless the IR is modified. Thus, to ensure consistency and correctness, SubroutineInjection performs this map re-construction step before performing any IR transformations. Refer to Appendix E for details on the JSON file formats.

We implement this workaround via a custom JSON library built upon the open-source JsonCpp library [24]. LiveValues initialises and partially populates the JSONs when running over the first *Function* in the *Module*. It then performs the aforementioned incremental writing by:

1. Reading the analysis data of previous runs from the JSON files into an in-memory map of LLVM pointers' names.
2. Updating this in-memory map with analysis results from the current *Function* being processed.
3. Writing this updated in-memory map as a JSON object back into the JSON files.

This library is also used in SubroutineInjection when re-constructing the now *Module*-scoped LLVM pointer maps of analysis data.

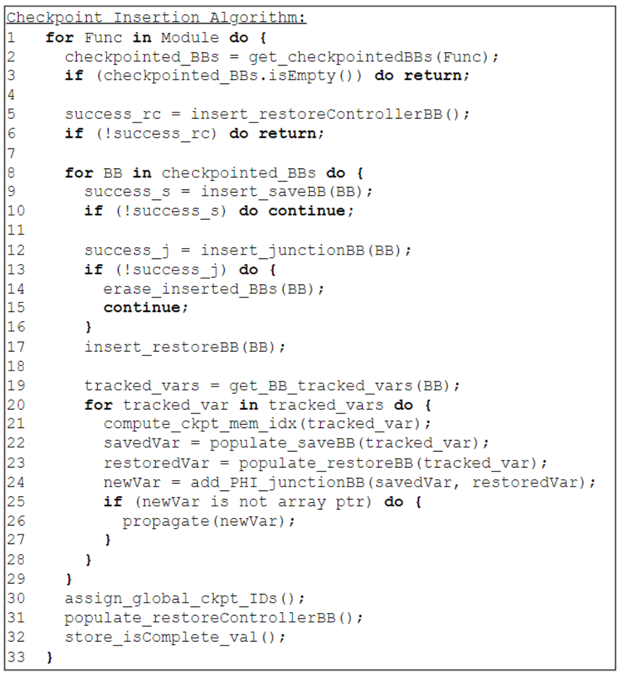
4.2.3 Determining Variable Sizes

Our compiler support must obtain information on the size of each tracked variable for each checkpointed BB to allow correct access to *ckpt_mem_seg*'s contents via indexing, as per Sections 3.4 and 4.3.2.4. For pointers, we need the size of the variables that they point to (i.e. their "pointees"). Pointees can either be arrays or single values. Variable sizes (in bytes) are also needed to facilitate memory allocation for *ckpt_mem_seg*.

To determine the pointee sizes of pointers declared locally within the kernel's scope, we find their corresponding memory-allocating `alloca` instructions in the IR, and directly compute their allocated sizes using LLVM's data layout information [28]. However, it is more difficult to determine the pointee sizes of externally-allocated pointers. As per Section 3.7.2, the kernel's function parameters often contain such pointers; they are often used in kernel computations, and can thus be part of many BBs' tracked variables sets. Since the kernel IR does not contain memory allocation metadata for these pointers, LiveValues relies on the domain-specific annotations covered in Section 3.7.2 to calculate their pointee sizes. For a given *Function*, LiveValues first scans through the kernel's C/C++ source code to find the annotation that matches the *Function*'s name. Then, for each of the *Function*'s pointer arguments, LiveValues **1)** locates the relevant section of the annotation, **2)** retrieves the pointee size (length) from within the '[' and ']' delimiters, **3)** calculates the pointee size in bytes according to its datatype, and **4)** pairs it with the corresponding LLVM pointer to that argument. These pairs are stored in a C++ map, and used by our custom JSON library to produce the `live_values.json`. Non-pointer variables are temporarily evaluated to a 1 byte size; SubroutineInjection later "pads" this size to that of *ckpt_mem_type*. Section 4.3.2.3 elaborates on how these sizes are used for calculating the indexes.

4.3 The SubroutineInjection Pass

The SubroutineInjection pass is a transformation *ModulePass* (Appendix B.3) that performs checkpoint selection and insertion, and is the final pass in our 3-pass workflow. It outputs the final checkpointed IR code, and a JSON file with the sizes (bytes) of each checkpoint in the *Module*. The broader migration system uses the latter to allocate memory for *ckpt_mem_seg*. Future versions of SubroutineInjection could include more advanced algorithms for automatically evaluating tracked variables and/or selecting checkpoints. These algorithms could require *Module*-scoped traversal of a program's call multigraph — a CFG representing calling relationships between subroutines in the program [22]. To accommodate this, we thus design SubroutineInjection as a *ModulePass*; it processes the *Module* by iterating over and processing each *Function* in turn, while maintaining and updating *Module*-scoped metadata.



```
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);
      restoredVar = populate_restoreBB(tracked_var);
      newVar = add_PHI_junctionBB(savedVar, restoredVar);
      if (newVar is not array ptr) do {
        propagate(newVar);
      }
    }
  }
  assign_global_ckpt_IDs();
  populate_restoreControllerBB();
  store_isComplete_val();
}
```

Figure 4.4: Checkpoint Insertion Algorithm

4.3.1 User-Directed Checkpoint Selection

Following Section 4.1, SubroutineInjection only considers BBs with 1 immediate successor as potential checkpointed BBs, and ignores BBs with no tracked variables. For these candidate BBs, SubroutineInjection ignores tracked variables that were marked as `const` in the domain-specific annotation. It also ignores any pointers to *ckpt_mem_seg* as checkpointing them is recursive and unnecessary.

SubroutineInjection implements a user-directed checkpointing scheme as per Section 3.5 and 3.7.1. As shown later in Chapter 5, checkpoint performance is kernel-specific. Moreover, certain kernels require well-chosen checkpoint placements for correct checkpointing. These kernel-dependent considerations do not generalise easily, and thus cannot be easily met by a generic selection algorithm. We therefore give users the option to specify checkpoint locations manually. This scheme only considers *Functions* with the domain-specific `ckpt_mem` input parameter. For each of these *Functions*, SubroutineInjection finds all BBs containing the `checkpoint()` function call, retrieves their tracked variables information from the *Module*-scoped nested LLVM pointers map (Section 4.2.2), erases the `checkpoint()` calls, and performs checkpoint insertion via the checkpoint insertion algorithm (Section 4.3.2).

4.3.2 Checkpoint Insertion Algorithm

4.3.2.1 Overview

SubroutineInjection implements a multi-stage *Module*-wide checkpoint insertion algorithm that processes each *Function* in turn. Its pseudocode is shown in Fig4.4. It first handles checkpoint selection via `get_checkpointedBBs()`, and skips to the next *Function* if no checkpointed BBs are selected/found. It then attempts to insert a restoreControllerBB, skipping to the next *Function* if it fails. If successful, it proceeds to process each checkpointed BB in turn, where it tries to insert a saveBB, restoreBB and junctionBB trio, skipping to the next checkpointed BB if unsuccessful. 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.

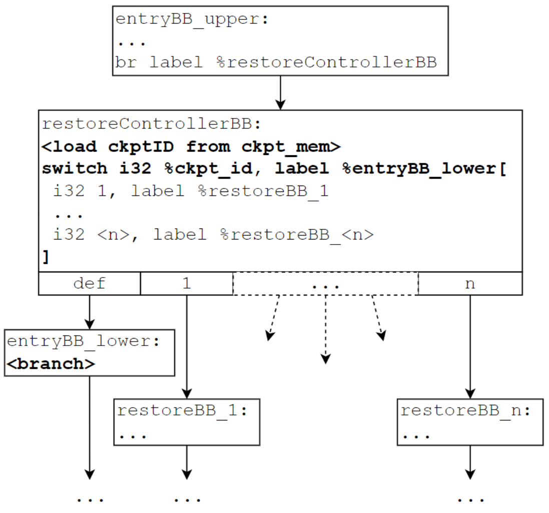

```
entryBB_upper:
...
br label %restoreControllerBB
```
```
restoreControllerBB:
<load ckptID from ckpt_mem>
switch i32 %ckpt_id, label %entryBB_lower[
 i32 1, label %restoreBB_1
 ...
 i32 <n>, label %restoreBB_<n>
]
```
```
entryBB_lower:
<branch>
```
```
restoreBB_1:
...
```
```
restoreBB_n:
...
```

Figure 4.5: restoreControllerBB Implementation

We adopt this staged approach because junctionBBs and saveBBs are inserted via LLVM's `splitEdge()` APIs, which do not guarantee successful insertion, e.g. if the edge's destination corresponds to the catch section of a try-catch sequence [29]. For simplicity, this dissertation interprets `splitEdge()` failures as being caused by unsuitable placements of `checkpoint()`, and thus ignores checkpointed BBs for which insertion fails. This section explains the algorithm for the case where all checkpointed variables are of *ckpt_mem_type*; type-conversion features for mixed datatype kernels are detailed in Section 4.3.2.6. For details on the `store_isComplete_val()` step in Fig4.4, refer to Appendix D.2. SubroutineInjection also enables selective partial insertion of checkpointing infrastructure to improve performance and reduce the final bitstream/binary sizes; refer to Appendix D.4 for more details.

4.3.2.2 restoreControllerBB

We insert the restoreControllerBB between `entryBB_upper` and `entryBB_lower` (Fig4.5) via `insert_restoreControllerBB()` (Fig4.4). After all the *Function*'s restoreBBs have been inserted, `assign_global_ckpt_IDs()` assigns each checkpoint in the *Function* with an ID unique within the *Module* using a global ID counter. These IDs start from 1; if `ckptID`'s value is 0, it means that *ckpt_mem_seg* does not contain any checkpointed data. We then create exit edges from the restoreControllerBB to each restoreBB in the *Function* (Fig4.5) — we insert **1)** instructions to load the ID value from *ckpt_mem_seg*, and **2)** a switch instruction that directs program flow to `entryBB_lower` when `ckptID`=0, and to the respective restoreBBs for all other valid ID values.

4.3.2.3 Calculating Index Offsets for Accessing *ckpt_mem_seg*

In `compute_ckpt_mem_indx()` in Fig4.4, we calculate the index offsets for a given tracked variable in *ckpt_mem_seg* based on its size in bytes — these index positions ("slots") are where the instructions in saveBB and restoreBB store and load data from. While LiveValues provides most of the size data required, it evaluates all non-pointer variables to 1 byte. For the internal IR pointers responsible for handling function arguments, LiveValues only evaluates their sizes to that of the pointers themselves, not of the arguments they point to. We therefore perform additional steps to obtain the padded size of each variable, and the number of slots they need in *ckpt_mem_seg*.

For single pointer/non-pointer variables (e.g. `i32*` or `i32`), we assign their padded size to that of *ckpt_mem_type* — since we only support pointers to primitive datatypes, these variables will have the same size. For pointers to locally-allocated arrays (e.g. `[3 x i32]*`), we set the number of slots used as the array's length *len*, and the padded size to $len \times$*ckpt_mem_type*. For nested pointers to externally-allocated arrays (e.g. `i32**`), we dereference them to get their contained non-nested pointers (i.e. the function arguments). The number of slots used is thus the length *len* of the contained pointer's pointee array; the padded size is $len \times$*ckpt_mem_type*. Refer to Appendix D.1 for more details.

Having obtained the padded size and number of *ckpt_mem_seg* slots n_{slots} required for each tracked variable, we compute their checkpoint-specific index offsets using a per-checkpoint index counter that is incremented by each variable's corresponding n_{slots}, starting from the slot after the last checkpoint metadata field (e.g. index 2 in Fig3.2). The total size of each checkpoint is the sum of the padded sizes of all tracked variables in the checkpoint (excludes checkpoint metadata); this size is written to an in-memory map in SubroutineInjection and outputted in `ckpt_sizes.json`.

4.3.2.4 saveBB and restoreBB

saveBBs are inserted via LLVM's `splitEdge()` APIs; restoreBBs are inserted via LLVM's `BasicBlock::Create()` API. We use LLVM's `BranchInst::Create()` to create the branch instructions that link the restoreBBs' exit edges to the junctionBBs. The `populate_saveBB()` and `populate_restoreBB()` steps use the checkpoint-specific indexes calculated in `compute_ckpt_mem_indx()` to instrument the empty saveBB and restoreBB with symmetric context saving and restoring instructions. We compute the addresses of the slots in *ckpt_mem_seg* that each tracked variable is located at via `getelementptr` instructions, with reference to the *ckpt_mem_seg* base address pointed to by our domain-specific `ckpt_mem` input parameter. If a tracked variable references an array, its computed slot address will be the "base address" starting from which the array's contents are stored. We then insert the following save/restore subroutines for each of the 4 supported IR datatypes:

1. **Single non-pointer Variables (e.g. `i32`):**
   These are are usually internal variables generated in the IR to store intermediate values between instructions. They are directly stored to their slot addresses via `store` instructions in saveBB, and directly loaded from their slot addresses into new variable registers via `load` instructions in restoreBB as per SSA form.

2. **Single Pointer Variables (e.g. `i32*`):**
   These often reference single user-defined variables in the kernel, e.g. `int i`. When saving, we first dereference them via a `load` instruction, and store the dereferenced values to their slot addresses via `store` instructions. When restoring, we **1)** allocate new pointers via `alloca`, **2)** `load` the dereferenced values from the slot addresses, and **3)** `store` the values into these new pointers.

3. **Locally-allocated Array pointers (e.g. `[3 x i32]*`):**
   These reference user-defined arrays in the kernel. We perform save and restore operations via `memcpy` instructions to directly copy the arrays' contents from their pointers to their slot addresses during save. During restore, the array contents are `memcpy`-ed back into their original pointers; no new pointers are allocated.

4. **Nested-pointers to Externally-allocated Arrays (e.g. `i32**`):**
   These are often used by the IR to handle externally-allocated arrays passed into the kernel as input arguments. We first dereference them via `load` to obtain their contained pointers, which hold the array base addresses. Array contents are then `memcpy`-ed between these pointers and their slot addresses during save and restore. No new pointers are allocated.

Fig4.6 shows the psuedo-instructions for restoreBB. As can be seen, we adopt different restoration policies for single-value variables and array variables. Single non-pointer internal IR variables often form part of a checkpointed BB's execution context and are thus involved in checkpoint-restore. However, they cannot be re-assigned under LLVM IR's SSA form, so new variable versions (*Values*) must be created and propagated during restoration. Although new values can be repeatedly stored into pointers without breaking SSA form, creating new single-value pointers does not incur significant memory loads. Thus for clarity and consistency, we create new variable registers during restore for both single non-pointer and single pointer variables (red instructions in Fig4.6).

```
restoreBB_n:
...
%idx_valA_r = getelementptr ...
%valA_r = <load i32 from i32* %idx_valA_r in
ckpt_mem>
...
%idx_ptrB_r = getelementptr ...
%ptrB_r = <alloca i32>
%deref_ptrB_r = <load i32 from i32* %idx_ptrB_r
in ckpt_mem>
<store i32 %deref_ptrB_r to i32* %ptrB_r>
...
%idx_ptrptrC_r = getelementptr ...
%deref_ptrptrC_r = i32* <dereference %ptrptrC>
<memcpy i32*%idx_ptrptrC_r in ckpt_mem to
i32* %deref_ptrptrC_r>
...
%idx_arrPtrD_r = getelementptr ...
<memcpyi32* %idx_arrPtrD_r in ckpt_mem to i32*
%arrPtrD>
...
br label %junctionBB_n
```

Figure 4.6: restoreBB pseudo-Instructions

Pointers to externally-allocated arrays are mainly passed by CPU hosts into kernel functions as “data containers” — the kernel stores its computed results into them for retrieval by the CPU host at the end of computation (Section 2.1.1). If new arrays are created during restoration, they would only be “visible” within the kernel’s scope, and thus cannot be accessed from the outside via pointers available to the CPU host. We cannot modify the function to return them as outputs due to incompatibility with kernels designed to return specific exit codes. Furthermore, allocating new memory for each array causes memory-overload issues if multiple large arrays are checkpointed. Thus, we restore the contents of externally-allocated arrays back into their original pointers. For clarity and consistency, we choose to adopt the same restoration policy for all array type variables including locally-allocated arrays (blue instructions in Fig4.6).

`assign_global_ckpt_IDs()` instruments each checkpoint’s saveBB with instructions to store its assigned checkpoint ID into the `ckptID` field in *ckpt_mem_seg* (Fig3.2). To prevent the CPU host from asynchronously reading data from *ckpt_mem_seg* while the FPGA kernel is still executing context-saving (write) operations, we “invalidate” *ckpt_mem_seg* during context-saving operation by writing -1 to the `ckpt_ID` field at the start of saveBB (FigD.9), and only write the correct `ckpt_ID` at the end of saveBB. `ckpt_ID` = -1 indicates to the migration system that context-saving (write) operations are still underway, and that no read operations to *ckpt_mem_seg* are allowed. Refer to Appendix D.3 for more details on the instructions used in saveBB and restoreBB.

4.3.2.5 junctionBB

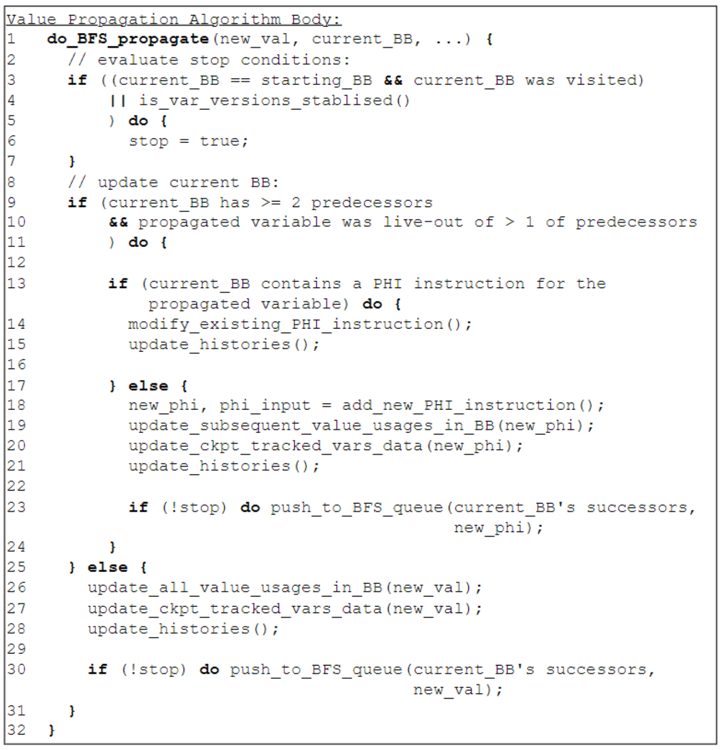
junctionBBs are created via LLVM’s `splitEdge()` APIs. To maintain SSA form, new *Value*s (registers) created in the restoreBB must be resolved against their original *Value* versions from the saveBB via PHI instructions. The `add_PHI_junctionBB()` step (Fig4.4) therefore inserts a PHI instruction into the junctionBB for each single-value tracked variable via the `PHINode::Create()` API, and configures them to choose the original *Value* if execution flow originates from the saveBB, and the newly-created *Value* if from the restoreBB, as shown in `junctionBB_n` in FigD.9. The outputted PHI results must then be propagated throughout the CFG, as detailed in Section 4.3.3.

4.3.2.6 Type Conversions for Mixed Datatype Checkpoints

For cases where checkpointed variables are not all of *ckpt_mem_type*, the we insert type conversion instructions (e.g. `sitofp`, `fptosi`) before storing into and after loading from *ckpt_mem_seg*. This type conversion feature currently supports conversions between `i32` and `float`/`double` types (and their pointers), and allows all these types to be saved/restored by the same checkpoint. However, this feature only supports non-array variables because array values are saved/restored via `memcpy` instructions that do not support array-level/element-wise type conversions. A future extension of this feature could be a custom `memcpy` library function that performs element-wise type conversion.

4.3.3 Propagation Algorithm

This section details the `propagate()` algorithm used to update all uses of the original (outdated) tracked variable versions in the CFG with new versions from the junctionBBs’



```
Value Propagation Algorithm Body:
1   do_BFS_propagate(new_val, current_BB, ...) {
2     // evaluate stop conditions:
3     if ((current_BB == starting_BB && current_BB was visited)
4         || is_var_versions_stablised()
5         ) do {
6           stop = true;
7     }
8     // update current BB:
9     if (current_BB has >= 2 predecessors
10        && propagated variable was live-out of > 1 of predecessors
11        ) do {
12
13        if (current_BB contains a PHI instruction for the
              propagated variable) do {
14          modify_existing_PHI_instruction();
15          update_histories();
16
17        } else {
18          new_phi, phi_input = add_new_PHI_instruction();
19          update_subsequent_value_usages_in_BB(new_phi);
20          update_ckpt_tracked_vars_data(new_phi);
21          update_histories();
22
23          if (!stop) do push_to_BFS_queue(current_BB's successors,
                                          new_phi);
24        }
25    } else {
26      update_all_value_usages_in_BB(new_val);
27      update_ckpt_tracked_vars_data(new_val);
28      update_histories();
29
30      if (!stop) do push_to_BFS_queue(current_BB's successors,
                                      new_val);
31    }
32  }
```

Figure 4.7: Propagation Algorithm (Body)

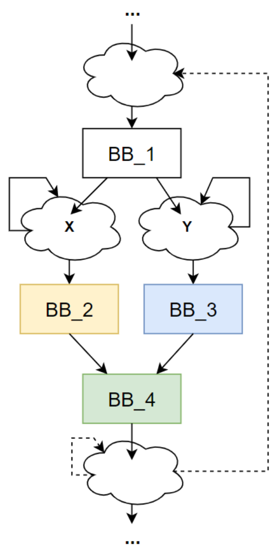


Figure 4.8: BB Revisiting

PHI instructions. ‘Version’ refers to the different *Value*s/registers that represent the same given variable within different subgraphs of the CFG (Appendix B.2). This algorithm uses modified Breadth-First Search (BFS) to visit and update BBs *repeatedly*, because different versions of the propagated variable can be generated when propagating through different CFG subgraphs. The algorithm is run for each new PHI result in the junctionBB. Propagation begins from the checkpoint’s resumeBB, and separates into different “trains of propagation” when splitting down branches in the CFG. This algorithm was challenging to develop as it needed to be simple and generic; pseudocode for its body is shown in Fig4.7.

Taking Fig4.8 as an example, suppose a variable *var* is propagated from BB_1 into two subgraphs *X* and *Y* that generate — via loops and branchings — new variable versions var_1 (yellow) and var_2 (blue) at BB_2 and BB_3 respectively. These subgraphs merge at BB_4, which must use a PHI instruction to resolve var_1 and var_2 into a new version var_{phi} (green). Suppose BB_4 is first visited by the train of propagation from BB_2. Here, BB_4 only receives information about var_1, which is insufficient to construct the correct PHI instruction; BB_4 only receives var_2’s information and constructs the correct PHI instruction when visited later by the train of propagation from BB_3. var_{phi} is then propagated to the rest of the CFG, where this process recurses/repeats.

We evaluate the following stop conditions each time a train of propagation visits a BB:

1. *Stop propagating if the current BB was previously visited before, and is the BB from which the propagation first started* (Fig4.7 line 3). This prevents propagation from overshooting the starting BB in looped CFGs.

2. *Stop propagating if the current BB’s variable versions have stabilised* (Fig4.7 line

4). Satisfying this condition indicates that the algorithm has finished propagating across the CFG's loops.

If any stop conditions are satisfied, the train of propagation processes only the currently-visited BB and not its successors. If the current BB needs to merge the variable versions carried by multiple incoming trains of propagation from multiple predecessors, we modify/update any relevant existing PHI instructions in the BB to use these carried versions. Since no new PHIs are created in this case, propagation stops at this BB. However, if no relevant PHI instructions exist, we create new PHI instructions and propagate their results as new variable versions. New PHI instructions use placeholders to represent variable versions from different predecessors; these are updated incrementally to the actual versions when the BB is (re)visited by the different trains of propagation that modify this PHI instruction (e.g. in the Fig4.8 example). Whenever a BB is updated, we also update its tracked variables set so that subsequent checkpoints will propagate the updated variable versions. We also update the version histories (Section 4.3.3.1) of both the BB and the train of propagation with the new variable versions.

4.3.3.1 Determining If Variable Versions have Stabilised

Each train of propagation maintains a history of variable versions it used for updating BBs. Each BB maintains a history of variable versions that is the aggregate of the version histories of all the trains of propagation that had visited it. These histories are updated whenever a BB is visited. When propagating across a loop, these histories will not match if there are still active trains carrying variable versions that have yet to reach the currently-visited BB. These two histories match only if all previously-branched trains of propagation for this variable have traversed the loop-ed CFG, and reached back to the current BB without generating any new variable versions that must be re-propagated. Matched variable histories thus indicates that the algorithm has finished propagating across the loops in the CFG.

4.3.3.2 The Alternative Converting-All-Pointers Method

Since LLVM IR pointers can be stored to repeatedly without breaking SSA form, an alternative restoration framework that does not require propagation would be to convert all tracked non-pointer variables into single pointer variables by storing their values into pre-allocated pointers during variable declaration. With this, all currently-supported tracked variables would be of pointer types. We can thus adopt a unified restoration policy in the restoreBBs that restores checkpointed variables directly into these pointers — similar to the array restoration policy in Section 4.3.2.4 — without needing to create new *Values*. This approach requires these pointers to be be declared in the entry BB so that their definitions dominate all their uses in saveBB and restoreBB, as per SSA form.

This method would require a possibly non-trivial CFG traversal algorithm that **1)** finds tracked single non-pointer variables in the IR, **2)** allocates pointers for them in the entryBB, and **3)** modifies/augments all affected instructions across the IR to work with these pointers. This method was discovered late into this dissertation's implementation phase after the propagation algorithm had been developed, so we elect to leave it to be explored in later iterations of the compiler support.

Chapter 5

Testing and Evaluation

5.1 Testing Methodology

We evaluate our compiler support via a combination of qualitative validations and quantitative performance assessments on three common benchmarking kernels:

1. `lud.cpp`, a Lower-Upper Decomposition (LUD) algorithm implementation [40].
2. `Cholesky.cpp`, a Cholesky Decomposition algorithm implementation [50].
3. `blur.cpp`, an implementation of Gaussian Blur based on [16].

Each kernel implements its algorithm as a single function f, thus making f synonymous with the kernel. Input data and computed results are transferred between the kernel and the test bench via buffers passed to the kernels as function arguments. We test each kernel for CPU-only, FPGA-only and CPU-FPGA migration cases. CPU-only tests are run on an Ubuntu 22.04.1 LTS Virtual Machine (VMware Workstation 17) with 2 CPU cores and 8GB memory. The remaining cases are tested on a Xilinx Alveo U50 FPGA.

5.1.1 Qualitative Evaluation of Validity

Since mathematical and/or mechanised proofs of our propagation algorithm's correctness is beyond this dissertation's scope, we judge its validity via **1)** LLVM's in-built CFG verifier, **2)** manual calculation and **3)** state restoration tests integrated within the quantitative test setups (incorrect propagation will cause incorrect restoration). Restoration tests also test the validity of the entire checkpoint-restore system. For these, we first run the checkpointed kernel to completion, then attempt to restore execution from the most recent checkpoint. Restoration only occurred correctly if their computed results match. Our propagation algorithm has successfully passed validation for the LUD, Cholesky and Blur kernels, with multiple checkpoints inserted per function.

5.1.2 Quantitative Evaluation of Performance and Overheads

Here we evaluate the runtime overheads introduced by our checkpoints for varying checkpoint sizes and frequencies, and weigh the benefits of frequent checkpointing against consequent runtime slowdowns. We also evaluate the runtime overheads incurred by the IR pre-processing stage (SplitConditionalBB), and observe how our checkpoints impact the size of the final binaries produced. Finally, we observe the performance of FPGA to CPU migration achieved using our checkpointing framework. All analyses of scaling and performance incorporate a margin of experimental error. No post-processing optimisations are applied to the kernels unless otherwise stated.

5.2 (CPU) Performance Overheads

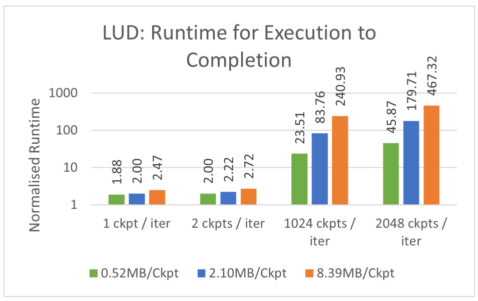


Figure 5.1: LUD CPU Runtime Overheads

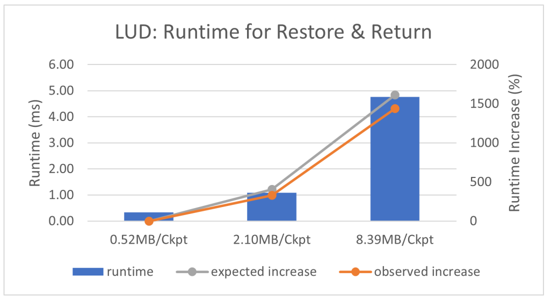


Figure 5.4: LUD CPU Restoration Overheads

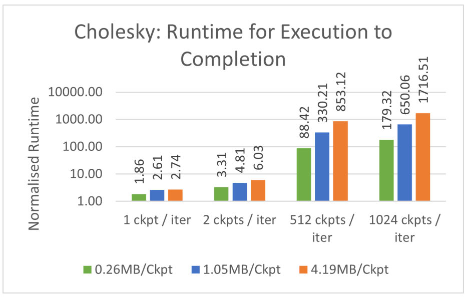


Figure 5.2: Cholesky CPU Runtime Overheads

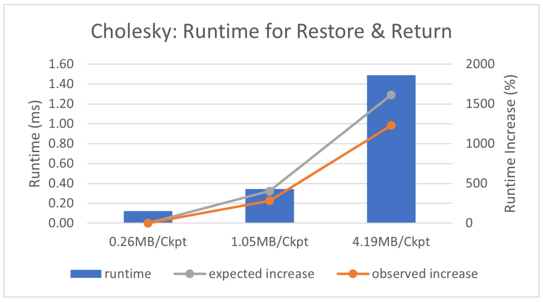


Figure 5.5: Choleksy CPU Restoration Overheads

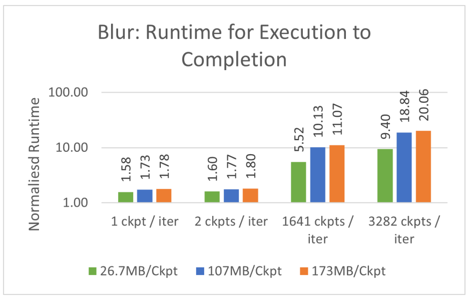


Figure 5.3: Blur CPU Runtime Overheads

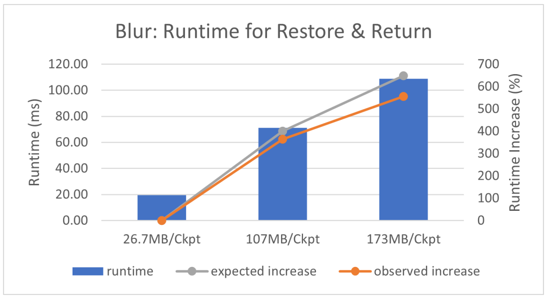


Figure 5.6: Blur CPU Restoration Overheads

Here we analyse (on CPUs) how the runtime overheads introduced by our checkpoints vary with checkpoint size and frequency. Checkpoint frequency (C_{freq}) is the rate at which checkpoints are triggered during kernel execution. Each kernel contains nested for-loops; here we alter C_{freq} by placing checkpoints at varying loop levels. Checkpoint size (C_{size}) is the amount of data tracked by a checkpoint. Here we alter

C_{size} by varying the size of the kernel's input data. Figures 5.1, 5.2, 5.3 show the normalised execution-to-completion runtimes ($\hat{T_{full}}$) on a logarithmic scale for different C_{size} and C_{freq}. C_{freq} is represented as the number of checkpoint executions per iteration of the outermost loop level in each kernel ($N_{ckpts/iter}$), where $N_{ckpts/iter} \propto C_{freq}$; C_{size} is represented as MegaBytes per checkpoint. $\hat{T_{full}}$ is calculated relative to the runtime of the un-checkpointed version of each kernel configuration (which have $\hat{T_{full}} = 1$). C_{freq} can also be interpreted via the normalised time [$\hat{s}$] between 2 consecutive checkpoints, given by $\hat{C_{\Delta T}} \approx \frac{\hat{T_{full}}}{N_{ckpts/iter} \times N_{total\ iters}}$. E.g. for 0.52MB/Ckpt LUD: $\hat{C_{\Delta T}} \approx \{7.34m\hat{s}, 3.91m\hat{s}, 89.7\mu\hat{s}, 87.5\mu\hat{s}\}$ for $N_{ckpts/iter} = \{1, 2, 1024, 2048\}$, $N_{total\ iters} = 256$. Since checkpoints are executed based on the kernel's execution progress and not its runtime, $\hat{C_{\Delta T}}$ serves mainly as a helpful approximation of inter-checkpoint time intervals.

5.2.1 Checkpoint Frequency

Checkpoints introduce additional instructions into the IR, which increases a program flow's instruction count and thus its runtime. Higher C_{freq} also triggers more frequent memory accesses to *ckpt_mem_seg*, which further increases overheads. Indeed, $\hat{T_{full}} > 1$ for all test cases in Figures 5.1, 5.2, 5.3. $\hat{T_{full}}$ varies positively with C_{freq} — higher C_{freq} generally corresponds with longer $\hat{T_{full}}$. However, $\hat{T_{full}}$ appears to be independent of C_{freq} for the low-frequency cases in Blur (1 to 2 ckpts/iter). This is likely because unlike LUD and Cholesky, Blur only runs a small number of outermost-loop iterations (3 iters), which is insufficient to manifest the expected runtime penalties at low C_{freq}.

Interestingly, $\hat{T_{full}}$ only scales proportionately to C_{freq} for high-frequency checkpointing cases (100s to 1000s of ckpts/iter). Low-frequency cases for LUD show a less than proportionate increase in $\hat{T_{full}}$ (at most $\approx$1.1x) when C_{freq} is doubled from 1 to 2 ckpts/iter. This could be because LUD's program characteristics also do not manifest the expected runtime penalties at low C_{freq}. When transitioning from low-frequency to high-frequency checkpointing, the runtime increase is also less than proportionate. This could be due to data pre-fetching or architecture-specific optimisations performed further downstream by compilers; the benefits of these are likely more apparent in the high-frequency cases. Regardless, checkpointing every loop iteration is generally inefficient because the changes in context and intermediate results are likely marginal. A more efficient approach is sparse checkpointing, as seen in Section 5.4.

5.2.2 Checkpoint Size

Larger checkpoints transfer more data to/from *ckpt_mem_seg* and thus incur larger memory-access-related runtime overheads. Indeed, Figures 5.1, 5.2, 5.3 show that larger C_{size} generally corresponds to longer $\hat{T_{full}}$ for almost all test cases. The only exception is Blur with low-frequency checkpointing (1 to 2 ckpts/iter), where runtime appears to be independent of size. As alluded to in Section 5.2.1, this is likely because Blur's program characteristics do not manifest the expected runtime overheads at low checkpoint frequencies.

$\hat{T_{full}}$ scales proportionately to C_{size} only for high-frequency checkpointing test cases where C_{size} is small ($C_{size} \lesssim$ 3MB/Ckpt); larger C_{size} values incur a less than proportional

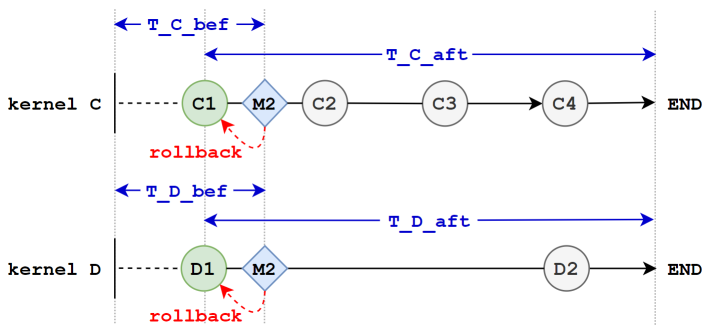


Figure 5.7: Shorter Rollback Distance

increase in $\hat{T_{full}}$. This is likely again due to data pre-fetching and downstream compiler optimisations that are more noticeable at larger C_{size}. On the other hand, C_{size} only has a marginal effect on $\hat{T_{full}}$ for the low-frequency checkpointing test cases. Again, this is likely because our test kernels' characteristics do not manifest the expected runtime penalties at low C_{freq}. In any case, we recommend that users keep checkpoint sizes as small as possible to minimise the absolute runtime overheads incurred.

5.3 (CPU) Checkpoint Size vs Restoration Time

Here we analyse the impact of checkpoint size (C_{size}) on the speed of context restoration on CPUs. During restoration, the saved context is loaded from *ckpt_mem_seg* via memory-access instructions. Larger C_{size} thus corresponds with longer memory-access and restoration times. We test this by analysing the time taken for a kernel to restore to its final checkpoint and return (T_{RR}). We place this final checkpoint at the end of the last main for-loop body in each kernel (`ckpt_3` in FigA.2); after restoring context from *ckpt_mem_seg*, the kernel immediately exits the loop and shortly returns. Checkpoint frequency is held constant.

Figures 5.4, 5.5, 5.6 show T_{RR} runtime (in ms) across different C_{size} values. They also show the expected percentage runtime increase ($P_{expected}$) against the observed increase ($P_{observed}$). These are computed with respect to the T_{RR} corresponding to the smallest C_{size}. $P_{expected}$ is a "reference" profile proportional to C_{size}. As expected, larger C_{size} corresponds to longer T_{RR}. However, $P_{observed}$ is always slightly lower than $P_{expected}$, with the difference increasing with C_{size}. E.g. in Cholesky: $P_{observed}$ is ≈ 120 percentage points lower than $P_{expected}$ when C_{size} increases to 1.05MB/Ckpt, and ≈ 380 points lower when C_{size} increases to 4.19MB/Ckpt. This could be due to data pre-fetching and architecture-specific optimisations applied further downstream by compilers; benefits from these would grow more apparent as checkpoint sizes increase, which is consistent with the behaviour seen here.

5.4 (CPU) Trade-off Analysis of Sparse Checkpointing

Here we analyse (on CPUs) the trade-offs between checkpoint frequency (C_{freq}) and execution rollback during restoration. Fig 5.7 shows the progress of checkpointed task execution and rollback distance for 2 otherwise identical kernels with different C_{freq},

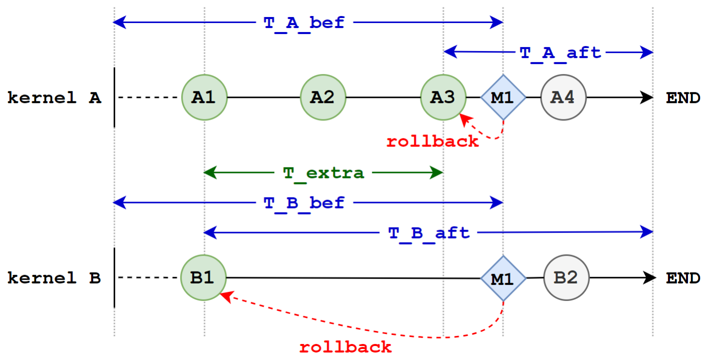


Figure 5.8: Longer Rollback Distance

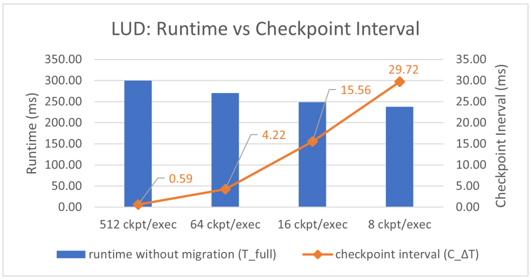


Figure 5.9: LUD CPU Runtime vs Checkpoint Interval

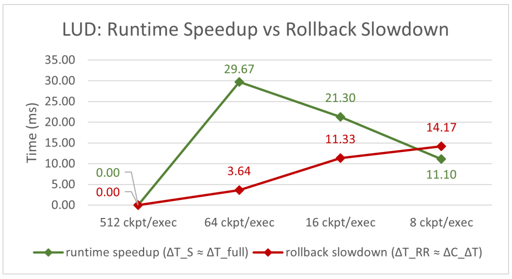


Figure 5.12: LUD CPU Runtime Speedup vs Rollback Slowdown

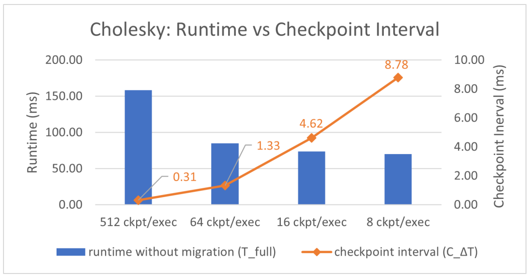


Figure 5.10: Cholesky CPU Runtime vs Checkpoint Interval

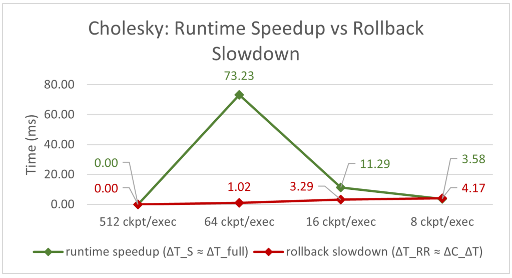


Figure 5.13: Choleksy CPU Runtime Speedup vs Rollback Slowdown

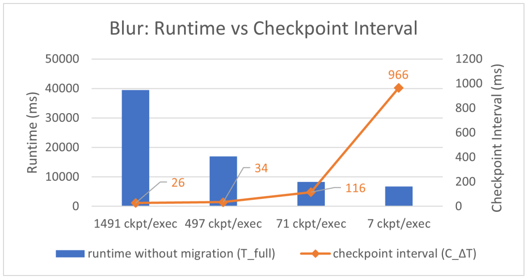


Figure 5.11: Blur CPU Runtime vs Checkpoint Interval

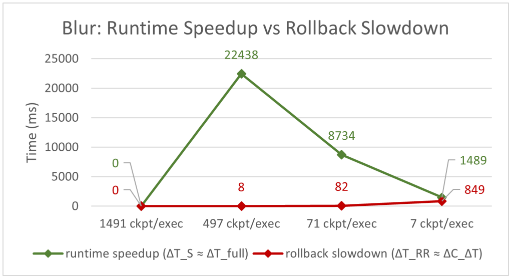


Figure 5.14: Blur CPU Runtime Speedup vs Rollback Slowdown

relative to each kernel's runtime. Here, the migration point `M2` incurs the same short rollback for both kernels. However, kernel D's lower C_{freq} incurs shorter runtimes during normal context-saving execution both before and after migration (T_S), resulting in a T_S speedup of $\Delta T_S = T_{S_C} - T_{S_D}$, where $T_{S_C} = \texttt{T\_C\_bef} + \texttt{T\_C\_aft}$, $T_{S_D} = \texttt{T\_D\_bef} + \texttt{T\_D\_aft}$. In Fig5.8, a more "expensive" migration point `M1` causes the more sparsely-checkpointed kernel B to roll execution back a further distance to checkpoint `B1` compared to checkpoint `A3` for kernel A. Kernel B thus performs more re-computation to finish its execution, resulting in a longer post-migration restore-and-return runtime (T_{RR}) than kernel A, and a worst-case T_{RR} slowdown of $\Delta T_{RR} \approx$ `T_extra` $\approx$ `T_B_aft` $-$ `T_A_aft`. In such cases, for sparsely-checkpointed kernels such as kernel B, $\Delta T_S > \Delta T_{RR}$ must hold for the runtime benefits of low C_{freq} to outweigh the larger rollback-time penalties. Since fault/migration points are unpredictable, we focus our analysis on the worst case migration scenario with maximum rollback corresponding to the kernel's checkpoint interval ($C_{\Delta T}$), i.e the time between 2 consecutive checkpoint executions.

Figures 5.9, 5.10, 5.11 show the full no-migration runtimes (T_{full}) for each kernel and the estimated $C_{\Delta T}$ for various C_{count}, where C_{count} is the number of checkpoints executed per full no-migration execution, $C_{freq} \propto C_{count}$ and $C_{\Delta T} \approx \frac{T_{full}}{C_{count}}$. We observe that although T_{full} decreases with decreasing C_{freq}, the worst case rollback (i.e. $C_{\Delta T}$) increases. We compare the associated runtime speedups against the rollback slowdowns in Figures 5.12, 5.13, 5.14, which approximates $\Delta T_S \approx \Delta T_{full}$ and $\Delta T_{RR} \approx \Delta C_{\Delta T}$. Overall, we observe that although $\Delta T_S > 0$ when sparser checkpointing is used, ΔT_S decreases with decreasing C_{count}, whereas ΔT_{RR} consistently increases. For LUD and Cholesky, $\Delta T_S > \Delta T_{RR}$ only for $C_{count} \lesssim 16$ ckpt/exec; $\Delta T_S < \Delta T_{RR}$ for $C_{count} = 8$ ckpt/exec. This indicates that while sparse checkpointing can improve checkpoint-restore and migration efficiency, its benefits diminish and disappear after C_{freq} decreases beyond a certain threshold. Conversely for Blur, $\Delta T_S > \Delta T_{RR}$ for all C_{count} values tested, even for 7 ckpt/exec. This is likely because Blur's checkpoint sizes are much larger than in LUD and Cholesky. Therefore for Blur, each reduction in C_{freq} causes a much greater reduction in overall memory-access penalties in T_S that outweighs the increases in T_{RR}. This indicates that the C_{freq} threshold — and the overall speedup-slowdown trade-off — depends on the nature of the kernel.

5.5 (CPU) BB Pre-Processing Overheads

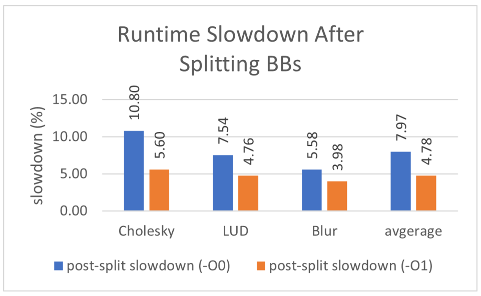


Figure 5.15: CPU Runtime Slowdown After Splitting BBs

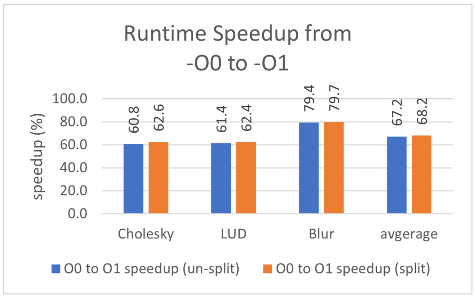


Figure 5.16: CPU Runtime Speedup from `-O0` to `-O1`

Here, we analyse how the IR pre-processing stage (SplitConditionalBB) impacts kernel runtime on CPUs. Splitting BBs increases the number of branch instructions in the IR, which could incur greater runtime overheads. Since pre-processing is a prerequisite to checkpoint insertion, we test the before-pre-processing and after-pre-processing runtimes on un-checkpointed kernels. Clang++ includes the option to post-process the modified kernel IR according to different optimisation levels (e.g. `-O0`, `-O1`, `O2`, etc.); higher optimisation levels perform increasingly advanced optimisations [27]. The broader migration system post-processes the modified IRs using `-O1` optimisation. We thus test each kernel setup for `-O0` (un-optimised) and `-O1` optimisations.

As shown in Fig5.15, BB-splitting only causes a marginal runtime slowdown compared to the un-split kernel — an average 7.97% slowdown for `-O0` and 4.78% for `-O1` across all kernels. Slowdowns are generally greater for `-O0` cases than `-O1` cases. Conversely, moving from `-O0` to `-O1` introduces significant speedups across all kernels — an overall average 67.2% speedup for non-split kernels and 68.2% for split kernels (Fig5.16). The percentage speedup is generally similar between split and un-split versions of the kernel. Thus overall, it is beneficial to apply a higher optimisation level to the IR during systems integration to reduce the runtime overheads caused by BB splitting.

5.6 (CPU) Impacts on Binary Size

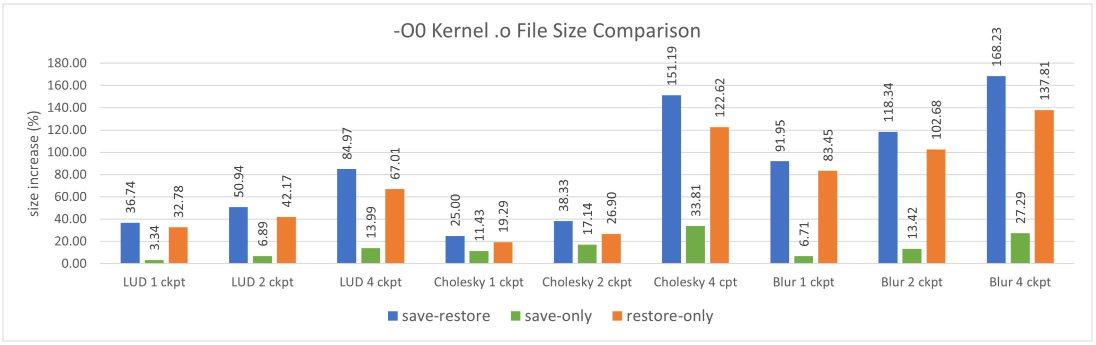


Figure 5.17: Kernel Object Files Size Comparison

Since FPGAs have limited on-chip memory [38], checkpoint insertion should ideally not introduce excessive bloat in a kernel's final binaries and bitstreams. We thus examine our checkpoints' impacts on the size of a kernel's object file (`kernel.o`) on CPU. FPGA utilisation is analysed separately in Section 5.7.

Fig5.17 shows the percentage increases in `kernel.o` size of checkpointed kernels relative to un-modified kernels; each kernel is instrumented with 1, 2, 4 checkpoints in the save-restore (*SR*), save-only (*S*) and restore-only (*R*) modes via selective partial insertion of checkpoint infrastructure (Appendix D.4). We observe that *SR* and *R* modes consistently cause the most significant size increases, while *S* mode only causes a comparatively small increase. Size increase for *SR* mode marginally exceeds that for *R* mode. This behaviour is consistent with our checkpoint design — *S* mode only inserts context-saving infrastructure, which is much smaller than the context-restoring infrastructure inserted by *R* mode. *SR* mode combines both context saving and restoring infrastructures, and thus naturally causes the largest size increase. We also observe that `kernel.o` sizes increase when more checkpoints are added. This is again consistent with our design — more checkpoints means more instructions, which means larger files. Size increases can be significant for high checkpoint counts, but is attenuated by selective partial insertion. E.g. 4-checkpoint Blur incurs a ≈170% increase in `kernel.o` size in SR mode, but ≈140% in R mode and ≈30% in S mode. In any case, size increases are a natural outcome of such checkpointing, and are thus inevitable.

5.7 (FPGA) Performance Overheads

Here, we inspect the FPGA runtime performance for checkpointed LUD and Cholesky. These kernels were synthesized into FPGA bitstreams by the broader migration system and run to completion (without migration) on an FPGA. Figures 5.18, 5.19 show the normalised runtimes ($\hat{T_{full}}$) of a CPU reference, and of FPGA kernels of varying checkpoint frequency (C_{freq}). $\hat{T_{full}}$ is calculated relative to the runtimes of the un-checkpointed FPGA kernels (which have $\hat{T_{full}} = 1$). These test cases insert a single checkpoint into each kernel's outermost for-loop.

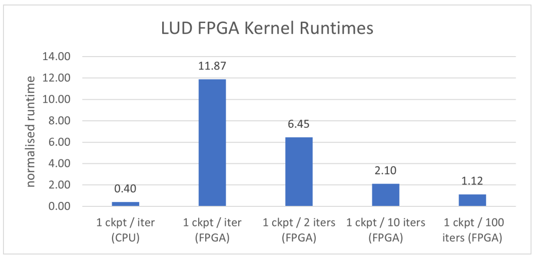


Figure 5.18: LUD FPGA Runtimes

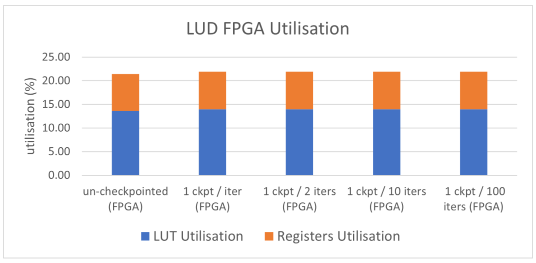


Figure 5.20: LUD FPGA Utilisation

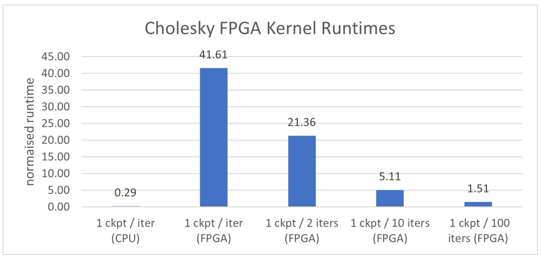


Figure 5.19: Cholesky FPGA Runtimes

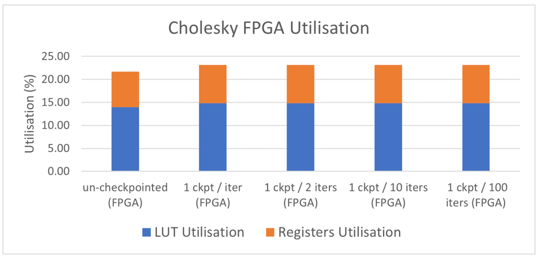


Figure 5.21: Cholesky FPGA Utilisation

We observe that FPGA runtimes for checkpointed kernels greatly exceed that of un-checkpointed kernels — for 1-checkpoint-per-loop-iteration cases, $\hat{T_{full}} \approx 10$ for LUD and ≈ 40 for Cholesky. This is expected due to the frequent accesses to the FPGA DRAM (where *ckpt_mem_seg* is stored) during checkpoint-restore. Additionally, checkpointing disables various dataflow optimisations further down the HLS workflow, which decreases the efficiency of the synthesised hardware [48]. Fortunately, runtime increases are significantly attenuated by sparse checkpointing — $\hat{T_{full}}$ decreases with decreasing C_{freq} to as low as ≈ 1 in LUD and ≈ 1.5 in Cholesky. Consistent with Section 5.4, the rate of decrease in $\hat{T_{full}}$ diminishes with decreasing C_{freq}.

We also observe that kernel execution on CPU is generally faster than that on FPGA — for the same checkpoint frequency, FPGA runtime is up to $\approx$ 30x that of CPU for LUD, and up to $\approx$ 140x for Cholesky. Even though FPGA runtimes decrease significantly with sparse checkpointing, the fastest FPGA runtime is still $\approx$ 3x that of CPU for LUD, and $\approx$ 5x for Cholesky. This could again be due to disabled HLS optimisations — even though no checkpoints are added, the compiler support still performs IR pre-processing, which could prevent certain optimisations further downstream.

The observed FPGA utilisation overheads are small. As seen from Figures 5.20, 5.21, FPGA utilisation (LUT and Registers) by the checkpointed kernels is only marginally higher than that for un-checkpointed kernels, and remains within the 20% to 25% range regardless of checkpoint frequency. This is likely because only a single checkpoint is inserted within the loop, and frequency is controlled via only a single control block variable. Utilisation overheads will likely increase when more checkpoints are inserted.

5.8 FPGA to CPU Migration

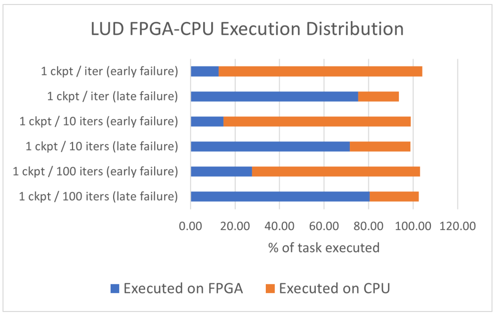


Figure 5.22: LUD FPGA-CPU Execution Distribution

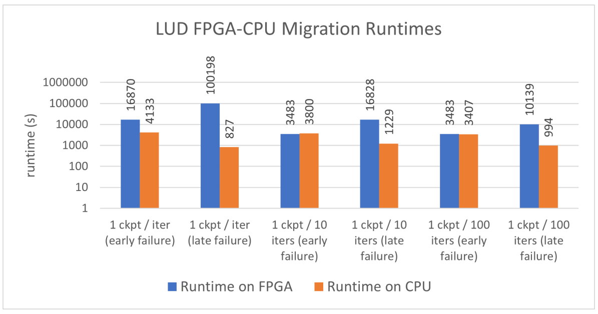


Figure 5.23: LUD FPGA-CPU Migration Runtimes

In this section, we integrate our compiler support into the broader migration system, and successfully perform safe checkpoint-based FPGA to CPU task migration for LUD at various checkpoint frequencies. The migration is triggered by a simulated FPGA failure, after which the task is migrated from the FPGA and resumed on the CPU. For each test case, the simulated FPGA failure is injected either near the beginning of kernel execution, or near the end. Here we inspect the performance of these migration cases.

Fig5.23 shows CPU and FPGA runtimes on a logarithmic scale. Consistent with Section 5.7, runtimes are consistently longer on the FPGA than the CPU for late-failure cases, as most of the computation occurs on the FPGA. Interestingly, FPGA runtime is only marginally shorter than CPU for most early-failure cases, even when the majority of execution takes place on the CPU. FPGA runtime even exceeds CPU for the 1 ckpt/iter early failure case. This is again because checkpointing disables HLS optimisations [48], which can cause FPGA kernels to be less efficient than their CPU counterparts. Fig5.22 shows the estimated percentage distribution of task execution between the FPGA (source) and CPU (destination), given by the sum of the percentages of execution progress achieved on the FPGA before migration (P_{FPGA}) and on the CPU after migration (P_{CPU}). $P_{FPGA} \approx \frac{T_{pre-migration}}{T_{FPGA-only}} \times 100\%$, where $T_{pre-migration}$ is the FPGA runtime from program start until migration, and $T_{FPGA-only}$ is the kernel's total runtime when run to completion on an FPGA without migration. $P_{CPU} \approx \frac{T_{post-migration}}{T_{CPU-only}} \times 100\%$, where $T_{post-migration}$ is the CPU runtime from migration until program end, and $T_{CPU-only}$ is the kernel's total runtime when run to completion on a CPU without migration. Since the migration testbench triggers the simulated FPGA failures via a separate asynchronous helper kernel, it is difficult to determine the precise failure and migration point relative to the main compute kernel's execution progress. Additionally, task restoration on CPU is subject to execution rollback as per Appendix A.2 and Section 5.4. Therefore, the P_{FPGA} and P_{CPU} values in Fig5.22 are approximations and may not sum up to 100%. Nevertheless, more crucially, these help verify that the migration system — compiler support included — operates as intended.

Chapter 6

Conclusions

6.1 Limitations

Our `checkpoint()` directive relies on the user to select well-chosen checkpoint locations within kernel code. It does not guarantee successful checkpoint insertion (e.g. within a try-catch sequence), nor does it guarantee that intermediate results are always checkpointed (e.g. if these intermediate results are not live-out of the checkpointed BB). It also does not guarantee the consistency of checkpointed data in *ckpt_mem_seg* if multiple kernel threads perform checkpointing in parallel — users can currently only checkpoint sequential regions of the kernel. Kernels must also declare all local variables at the start of the function (in the entry BB) so that their declarations correctly dominate their uses in the restoreBBs. All arrays used in the kernel must be 1-D pseudo-multidimensional arrays of type *ckpt_mem_type*, because the `memcpy` instructions used cannot integrate array serialisation nor per-element type conversion. `memcpy` also copies entire arrays instead of just the elements that changed since the previous checkpoint, resulting in inefficient memory accesses during checkpoint-restore. Moreover, our checkpointing causes certain downstream HLS optimisations to be disabled, resulting in more inefficient hardware implementations and significantly longer FPGA runtimes.

Crucially, we can currently only achieve FPGA to CPU migration, because Vitis HLS cannot consistently handle the "disruptions" to execution flow — caused by the restoreControllerBB's check-and-jump scheme — required for context restoration on an FPGA. We observed during systems integration that Vitis can only (partially) handle such disruptions under specific conditions which are difficult to enforce in practice. We therefore only instrument the FPGA kernel version with context-saving infrastructure.

6.2 Conclusion

We presented a custom LLVM compiler support that establishes equivalence points between software and hardware representations of FPGA-accelerated kernels, and performs software checkpoint insertion into C/C++ kernel code for *stateful* checkpoint-based CPU-FPGA task migration. This migration capability can be leveraged to enhance the flexibility and dependability of FPGA-accelerated systems. Although it incurs some runtime slowdowns, our compiler support has successfully been used for kernel checkpoint-restore on CPU. It has also been integrated into the broader task migration system, where it has been used to perform successful and safe FPGA to CPU task migration. Furthermore, the slowdowns from checkpointing are greatly attenuated by sparse checkpointing; the increases in the final size of kernel binaries can be attenuated via selective partial insertion of checkpoint infrastructure. Since all checkpointing frameworks inevitably incur some overheads, we conclude that our proposed compiler support — and the sparse-checkpoint-based migration (with reduced overheads) that it enables — is a viable solution for our flexibility and dependability goals in Section 1.2.

6.3 Plans for MInf Part 2

For MInf part 2, we plan to extend our compiler support to enable stateful CPU to FPGA migration. To address the Vitis HLS tool's inability to handle disrupted program flows, we aim to avoid disruptions altogether by offloading restoration control to the broader migration system. We will devise a system to generate unique context-restoring versions (V_R) of the kernel for *individual* checkpoints. We will explore a "pre-compiled" approach where a unique V_R is generated and synthesised for each checkpoint at compile time, and the suitable V_R is chosen and "flashed" onto the FPGA at migration time. We will also explore whether this approach (and others) is sufficient to allow the re-enabling of downstream HLS optimisations. FPGA re-configuration can be time-consuming in practice; we will evaluate how this approach affects migration time, and how this re-configuration time impacts the runtime of the broader hardware-accelerated system.

We also plan to design custom `memcpy` functions as part of a domain-specific library. These `memcpy`s will implement "context-aware" copying — given an array *arr* and a reference arr_{ref}, they will only `memcpy` elements from *arr* that are different from those in arr_{ref}. For checkpoint-restore, arr_{ref} will be the region of *ckpt_mem_seg* where *arr* is stored. Each element in *arr* will be read during each difference-check; if the differences in intermediate results are small between checkpoints, the total amortised number of write operations will be small. Since read operations are generally faster than write operations for flash memory [25], this could manifest in an overall runtime reduction (to be evaluated). Our custom `memcpy` will also incorporate element-wise type conversion and n-dimensional array serialisation functionality.

Finally, we plan to expand our checkpoint-restore capability to incorporate fault-tolerance against transient FPGA faults (e.g. single event upsets [2]). Fault tolerance is a crucial challenge for HPCs; the increase in transistor density in modern CMOS circuitry significantly increases the rate of transient system faults [1]. One method of fault detection is to incorporate redundant FPGA logic to detect when a logic function is generating an incorrect output [44]. Another is multi-platform redundancy, where FPGA execution is periodically checked against redundant threads running on CPUs. We will investigate the incorporation of / integration with such fault-detection infrastructures at compiler level by expanding our current compiler support, and also the incorporation of an auto-rollback functionality to restore execution at the previous (un-faulted) checkpoint if a transient fault is detected.

Bibliography

[1] Hartwig Anzt, Jack Dongarra, and Enrique S. Quintana-Ortí. “Fine-grained bit-flip protection for relaxation methods”. In: *Journal of Computational Science* 36 (Sept. 2019), p. 100583. ISSN: 18777503. DOI: 10.1016/j.jocs.2016.11.013.

[2] Luis Alberto Aranda, Pedro Reviriego, and Juan Antonio Maestro. “A Comparison of Dual Modular Redundancy and Concurrent Error Detection in Finite Impulse Response Filters Implemented in SRAM-Based FPGAs Through Fault Injection”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65 (3 Mar. 2018), pp. 376–380. ISSN: 1549-7747. DOI: 10.1109/TCSII.2017.2717490.

[3] Danilo Ardagna et al. “Quality-of-service in cloud computing: modeling techniques and their applications”. In: *Journal of Internet Services and Applications* 5 (1 Dec. 2014), p. 11. ISSN: 1867-4828. DOI: 10.1186/s13174-014-0011-3.

[4] Marco Bacis, Rolando Brondolin, and Marco D. Santambrogio. “BlastFunction: an FPGA-as-a-Service system for Accelerated Serverless Computing”. In: IEEE, Mar. 2020, pp. 852–857. ISBN: 978-3-9819263-4-7. DOI: 10.23919/DATE48585.2020.9116333.

[5] David G. von Bank, Charles M. Shub, and Robert W. Sebesta. “A unified model of pointwise equivalence of procedural computations”. In: *ACM Transactions on Programming Languages and Systems* 16 (6 Nov. 1994), pp. 1842–1874. ISSN: 0164-0925. DOI: 10.1145/197320.197402.

[6] Antonio Barbalace et al. “Breaking the Boundaries in Heterogeneous-ISA Datacenters”. In: ACM, Apr. 2017, pp. 645–659. ISBN: 9781450344654. DOI: 10.1145/3037697.3037738.

[7] Vaughn Betz and Jonathan Rose. *VPR: a new packing, placement and routing tool for FPGA research*. 1997, pp. 213–222. DOI: 10.1007/3-540-63465-7_226.

[8] Christophe Bobda et al. “The Future of FPGA Acceleration in Datacenters and the Cloud”. In: *ACM Transactions on Reconfigurable Technology and Systems* 15 (3 Sept. 2022), pp. 1–42. ISSN: 1936-7406. DOI: 10.1145/3506713.

[9] Alban Bourge, Olivier Muller, and Frédéric Rousseau. “Generating Efficient Context-Switch Capable Circuits through Autonomous Design Flow”. In: *ACM Transactions on Reconfigurable Technology and Systems* 10 (1 Mar. 2017), pp. 1–23. ISSN: 1936-7406. DOI: 10.1145/2996199.

[10] Florian Brandner et al. “Computing Liveness Sets for SSA-Form Programs”. In: *HAL Open Science* (2011). URL: https://hal.inria.fr/inria-00558509v2/document.

[11] Adrian M. Caulfield et al. “A cloud-scale acceleration architecture”. In: IEEE, Oct. 2016, pp. 1–13. ISBN: 978-1-5090-3508-3. DOI: 10.1109/MICRO.2016.7783710.

[12] Jason Cong et al. “Understanding Performance Differences of FPGAs and GPUs”. In: IEEE, Apr. 2018, pp. 93–96. ISBN: 978-1-5386-5522-1. DOI: 10.1109/FCCM.2018.00023.

[13] Keith D. Cooper and Linda Torczon. “Chapter 4 - Intermediate Representations”. In: *Engineering a Compiler (Third Edition)*. Ed. by Keith D. Cooper and Linda Torczon. Third Edition. Philadelphia: Morgan Kaufmann, 2023, pp. 159–207. ISBN: 978-0-12-815412-0. DOI: https://doi.org/10.1016/B978-0-12-815412-0.00010-3. URL: https://www.sciencedirect.com/science/article/pii/B9780128154120000103.

[14] Ben Cope et al. “Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study”. In: *IEEE Transactions on Computers* 59 (4 Apr. 2010), pp. 433–448. ISSN: 0018-9340. DOI: 10.1109/TC.2009.179.

[15] E. N. (Mootaz) Elnozahy et al. “A survey of rollback-recovery protocols in message-passing systems”. In: *ACM Computing Surveys* 34 (3 Sept. 2002), pp. 375–408. ISSN: 0360-0300. DOI: 10.1145/568522.568525.

[16] Basile Fraboni and Ivan Kutskir. *bfraboni/blur_float.cpp*. 2023. URL: https://gist.github.com/bfraboni/946d9456b15cac3170514307cf032a27.

[17] LLVM Developer Group. *LLVM Value Class Reference*. Feb. 2023. URL: https://llvm.org/doxygen/classllvm_1_1Value.html.

[18] Edson Horta et al. “Xar-Trek: Run-time Execution Migration among FPGAs and Heterogeneous-ISA CPUs”. In: (Oct. 2021). DOI: 10.1145/3464298.3493388.

[19] IBM. *cloudFPGA: Field programmable gate arrays for the cloud*. Feb. 2023. URL: https://www.zurich.ibm.com/cci/cloudFPGA/#:~:text=Field%5C%20programmable%5C%20gate%5C%20arrays%5C%20(FPGAs,conversion%5C%20and%5C%20high%5C%2Dfrequency%5C%20trading..

[20] David Kaeli et al. *Heterogeneous Computing with OpenCL 2.0: Third Edition*. 2015. DOI: 10.1016/C2013-0-15490-6.

[21] Dirk Koch, Christian Haubelt, and Jürgen Teich. “Efficient hardware checkpointing”. In: ACM, Feb. 2007, pp. 188–196. ISBN: 9781595936004. DOI: 10.1145/1216919.1216950.

[22] Arun Lakhotia. “Constructing call multigraphs using dependence graphs”. In: ACM Press, 1993, pp. 273–284. ISBN: 0897915607. DOI: 10.1145/158511.158647.

[23] Joshua Lant et al. “Toward FPGA-Based HPC: Advancing interconnect technologies”. In: *IEEE Micro* 40 (1 2020). ISSN: 19374143. DOI: 10.1109/MM.2019.2950655.

[24] Baptiste Lepilleur. *JsonCpp*. 2010. URL: https://github.com/open-source-parsers/jsoncpp.

[25] Qiao Li et al. “Access Characteristic Guided Read and Write Regulation on Flash Based Storage Systems”. In: *IEEE Transactions on Computers* 67 (12 Dec. 2018), pp. 1663–1676. ISSN: 0018-9340. DOI: 10.1109/TC.2018.2839671.

[26] Qingrui Liu et al. "Compiler-Directed Lightweight Checkpointing for Fine-Grained Guaranteed Soft Error Recovery". In: IEEE, Nov. 2016, pp. 228–239. ISBN: 978-1-4673-8815-3. DOI: 10.1109/SC.2016.19.

[27] LLVM. *Code Generation Options*. 2023. URL: https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options.

[28] LLVM. *Data Layout Modelling*. Feb. 2023. URL: https://mlir.llvm.org/docs/DataLayout/#:~:text=Data%5C%20layout%5C%20information%5C%20allows%5C%20the,or%5C%20its%5C%20address%5C%20alignment%5C%20requirements..

[29] LLVM. *Exception Handling*. Mar. 2023. URL: https://llvm.org/docs/ExceptionHandling.html#overview.

[30] LLVM. *LLVM Concepts*. Feb. 2023. URL: https://www.llvmpy.org/llvmpy-doc/dev/doc/llvm_concepts.html.

[31] LLVM. *LLVM Doxygen Documentation*. 2023. URL: https://llvm.org/doxygen/.

[32] LLVM. *LLVM Language Reference Manual*. 2023. URL: https://llvm.org/docs/LangRef.html.

[33] LLVM. *LLVM's Analysis and Transform Passes*. Feb. 2023. URL: https://llvm.org/docs/Passes.html.

[34] LLVM. *The LLVM Compiler Infrastructure Project*. 2023. URL: https://llvm.org/.

[35] LLVM. *Writing an LLVM Pass*. Feb. 2023. URL: https://llvm.org/docs/WritingAnLLVMPass.html.

[36] Rob Lyerly. *Popcorn Compiler*. 2017. URL: https://github.com/ssrg-vt/popcorn-compiler.

[37] Ana Moreton–Fernandez, Hector Ortega–Arranz, and Arturo Gonzalez–Escribano. "Controllers: An abstraction to ease the use of hardware accelerators". In: *The International Journal of High Performance Computing Applications* 32 (6 Nov. 2018), pp. 838–853. ISSN: 1094-3420. DOI: 10.1177/1094342017702962.

[38] Zhiqiang Que et al. "Mapping Large LSTMs to FPGAs with Weight Reuse". In: *Journal of Signal Processing Systems* 92 (9 Sept. 2020), pp. 965–979. ISSN: 1939-8018. DOI: 10.1007/s11265-020-01549-8.

[39] Gabriel Rodriguez-Canal et al. "Programming abstractions for preemptive scheduling in FPGAs using partial reconfiguration". In: (Aug. 2022). URL: http://arxiv.org/abs/2209.04410.

[40] Sable. *Sable/lud-benchmark*. 2023. URL: https://github.com/Sable/lud-benchmark/blob/master/README.md.

[41] Patrick R. Schaumont. *Finite State Machine with Datapath*. Springer US, 2013, pp. 113–156. DOI: 10.1007/978-1-4614-3737-6_5. URL: https://link.springer.com/10.1007/978-1-4614-3737-6_5.

[42] M.S. Schmalz. *Organisation of Computer Systems: ISA, Machine Language and Numbering Systems*. Feb. 2023. URL: https://www.cise.ufl.edu/~mssz/CompOrg/CDA-lang.html.

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

[44] Edward Stott, Pete Sedcole, and Peter Y. K. Cheung. “Fault tolerant methods for reliability in FPGAs”. In: IEEE, 2008, pp. 415–420. ISBN: 978-1-4244-1960-9. DOI: `10.1109/FPL.2008.4629973`.

[45] Jonathan Tompson and Kristofer Schlachter. *An Introduction to the OpenCL Programming Model*. NYU: Media Research Lab, 2012. URL: `https://cims.nyu.edu/~schlacht/OpenCLModel.pdf`.

[46] Anuj Vaishnav, Khoa Pham, and Dirk Koch. “Live Migration for OpenCL FPGA Accelerators”. In: IEEE, Dec. 2018, pp. 38–45. ISBN: 978-1-7281-0214-6. DOI: `10.1109/FPT.2018.00017`.

[47] Hoang-Gia VU et al. “A Tree-Based Checkpointing Architecture for the Dependability of FPGA Computing”. In: *IEICE Transactions on Information and Systems* E101.D (2 2018), pp. 288–302. ISSN: 0916-8532. DOI: `10.1587/transinf.2017RCP0010`.

[48] Arief Wicaksana et al. “Prototyping dynamic task migration on heterogeneous reconfigurable systems”. In: ACM, Oct. 2017, pp. 16–22. ISBN: 9781450354189. DOI: `10.1145/3130265.3130316`.

[49] Xilinx. *SmartSSD Computational Storage Drive (Big Acceleration for Big Data)*. Feb. 2023. URL: `https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html`.

[50] Xilinx. *Xilix Vitis-Tutorials*. 2023. URL: `https://github.com/Xilinx/Vitis-Tutorials/tree/2022.2/Hardware%5C_Acceleration/Design%5C_Tutorials/06-cholesky-accel`.

[51] Javad Yousefi, Yasser Sedaghat, and Mohammadreza Rezaee. “Masking wrong-successor Control Flow Errors employing data redundancy”. In: IEEE, Oct. 2015, pp. 201–205. ISBN: 978-1-4673-9280-8. DOI: `10.1109/ICCKE.2015.7365827`.

[52] Chuck Zhao et al. *Compiler Support for Fine-Grain Software-Only Checkpointing*. 2012, pp. 200–219. DOI: `10.1007/978-3-642-28652-0_11`.

Appendix A

Definitions

A.1 Checkpointing

Considering our objective of enabling task migration between CPUs and FPGAs, we define a checkpoint as a point in program execution *at* which execution context is extracted and later restored, and *from* which execution can resume. We use the term ‘checkpoint’ interchangeably with the term ‘checkpoint infrastructure’ to simultaneously refer to the additional subroutines needed to implement the checkpointing operation, and collectively refer to the save and restore operations as ‘checkpoint-restore’.

With respect to a checkpointed subgraph of a program’s CFG, we refer to a BB that has been selected for checkpointing as a ‘checkpointed BB’; a checkpointed BB’s corresponding checkpoint infrastructure is thus the additional BBs required to:

1. Save the execution state at a checkpointed BB.
2. Resume execution at this checkpointed BB’s immediate successor

We define the checkpoint infrastructure as being inserted immediately after the checkpointed BB on the edge between the checkpointed BB and its successor, such that the checkpoint performs checkpoint-restore for the current checkpointed BB and not for its successor. For convenience, we refer to this insertion as inserting checkpoint infrastructure *at* the checkpointed BB.

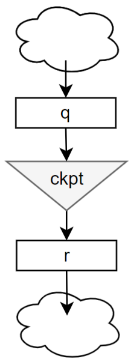


Figure A.1: Toy CFG

FigA.1 shows a checkpointed subgraph of a program’s CFG. Here, node *q* is the checkpointed BB and the checkpoint infrastructure *ckpt* is the set of additional BBs inserted immediately after *q* (i.e. at *q*) that implements checkpoint-restore for node *q*. *ckpt* allows execution to be resumed at *q*’s single immediate successor *r* in the event of migration.

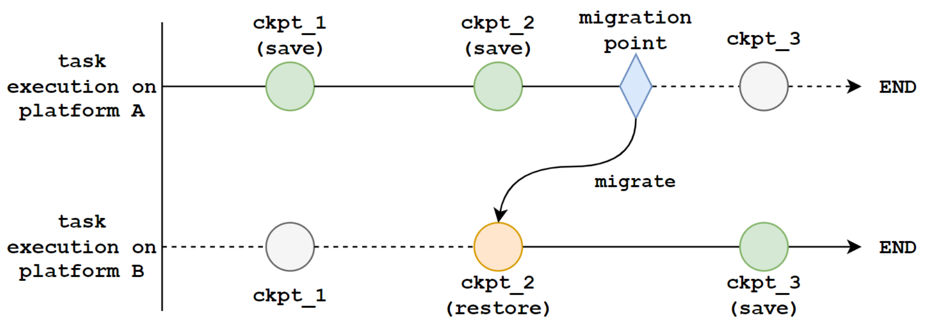


Figure A.2: Migration Model

A.2 Migration Model

Even though the actual task pre-emption and migration system is outside of the scope of this dissertation, the migration model that it employs still greatly influences the design of the checkpoint-restore mechanism that our compiler support inserts.

This dissertation is designed to support a checkpoint-based migration model. This model migrates the kernel at an arbitrary point in its execution, and resumes it at its most-recently executed checkpoint at the migration destination. This migration model is oriented towards improving data-centre dependability by allowing an FPGA task to resume on a CPU host in the event of FPGA component failure without loss of intermediate progress/results.

As shown in FigA.2, the execution of a task initially running on platform A progresses past checkpoints `ckpt_1` and `ckpt_2`, which perform state-extraction and saving operations. At the migration point between `ckpt_2` and `ckpt_3`, the task is migrated from platform A to run on platform B. On platform B, the task resumes at the checkpoint which it had executed most recently when on platform A, i.e. `ckpt_2`. However, this time, `ckpt_2` instead performs state-restoration operations by loading the checkpointed data it had previously saved on platform A into the execution context on platform B such that the task can resume execution on platform B.

This model performs migration at the granularity of individual checkpoints within the kernel code; if the kernel code is heavily checkpointed, then even finer-grained migration can be achieved, at the expense of higher checkpointing overheads.

A.3 Execution Context

In line with our definition of a checkpoint (Appendix A.1), the execution context that a checkpoint saves and restores must therefore be the variables that together provide a “snapshot” of the current execution context of the kernel at the checkpointed BB, that is sufficient to allow the checkpoint to restore kernel execution at the checkpoint BB’s successor.

For each BB in a kernel function’s CFG, these variables correspond to the set of variables that is live-out of the BB. Considering Appendix B.1, a BB q’s live-out set contains the variables that are used in BBs that are reachable from q. With reference to FigA.1 (which depicts a subgraph of a kernel function's CFG), consider the case where a checkpoint *ckpt* is inserted between a node q in a CFG and its immediate successor r for the purposes of task migration. The set of variables that are live-out from q are all the variables which are subsequently used in r, and/or used in other nodes reachable from r. This set of live-out variables thus provides a "snapshot" of the current execution context of the kernel function at node q that is sufficient to allow us to restore kernel execution from node r.

We therefore define the execution context at a particular BB in a kernel function's CFG to be the set of values which is live-out from this BB, hereon referred to as 'tracked variables'. These tracked variables represent the state of kernel execution and — if taken from well-chosen points in the kernel code — the intermediate results of the kernel's computations at each BB in the kernel. We allow our checkpoints to include these intermediate results because of the dependability objective that our migration model aims to achieve (refer to Appendix A.2). This therefore allows intermediate results computed before the migration point to be saved such that they will not be lost during the migration process.

```
int foo() {
  int k = ...;
  int arr[n] = {...};
  for(int i = 0; i < n; i++){
    ...
    arr[i] = arr[i] + i*2;
    k = min(k, arr[i]);
    ...
    checkpoint();
    ...
  }
  return k;
}
```

Figure A.3: `foo()` For-loop Example

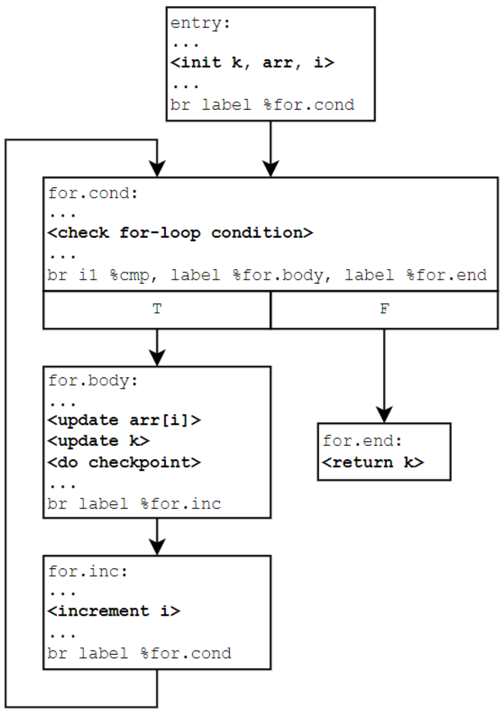


Figure A.4: `foo()` For-loop Example CFG

Consider a simple for-loop function `foo()` as shown in FigA.3: In each iteration, the for-loop modifies an element in the array `arr` and re-assigns a variable `k` to either itself or `arr[i]`, whichever is smaller. Suppose we checkpoint each iteration of the for-loop. This involves extracting the execution context at the point in the loop immediately after `arr[i]` and `k` have been updated with their new values. In the CFG of the LLVM IR representation of `foo()`, this corresponds to the checkpointing operation being performed in the BB `for.body` after `arr[i]` and `k` have been updated. Inspecting the liveness information of `foo()`, the live-out set of `for.body` comprises the variables `k`, `i` and `arr`. This is because these variables are used in BBs that are reachable from `for.body`, which in this case includes `for.inc` and `for.body` itself. These are thus the tracked variables for BB `for.body`.

This is consistent with our intuitive understanding of migration-oriented checkpoint-restore: if the for-loop is migrated in the middle of execution, the checkpoint must restore the current value of the loop iterator `i`, the contents of `arr` and the value of `k`. This is because `arr` and `i` are needed to obtain the correct value of `arr[i]`, and `k` is needed to obtain the correct result from the minimum function `min(k, arr[i])`. If any of these variables are omitted, the execution of `foo()` would not be able to correctly resume from the point it left off before migration, and the output will differ between the un-migrated `foo()` and the migrated `foo()`.

Appendix B

Compiler Concepts & LLVM

B.1 Live Variables and Liveness Analysis

Given a set of all variables in the program, liveness analysis is the process of determining the points in the program where each of these variables' values are eventually used by subsequent operations in the program control flow. A variable is live at a program point p, if p belongs to a path of the CFG leading from a definition of p to a usage of p without any re-definition of p [10]. In other words, a variable is live at p when its *current* value is used in the future (some point reachable from p) by any dynamic execution.

For any node in a CFG, this liveness information can be used to determine the node's set of live-in and live-out variables. The liveness analysis pass used in this dissertation adheres to the following definitions from [10] for live-in and live-out:

Definition B.1.1 (Live-In)**.** A variable a is live-in at a CFG node q if there exists a directed path from q to a node u where a is used, and that path does not contain the (re)definition of a.

Definition B.1.2 (Live-Out)**.** A variable a is live-out at a node q if it is live-in at least one successor of q.

More intuitively, a node q's live-in set contains the set of variables that are used either in q itself, and/or in nodes reachable from q, while its live-out set contains the set of variables used in nodes that are reachable from q. Hence from a task migration standpoint, given a checkpoint *ckpt* inserted between node q and its single successor r (FigA.1): for *ckpt* to restore execution at r, it needs to restore q's live-out variables, i.e. variables that are used in r and all nodes reachable from r. We therefore define the execution context at a particular BB in a kernel function's CFG to be the set of values which is live-out from this BB, hereon referred to as 'tracked variables'. Refer to Appendix A.3 for more details.

B.2 LLVM and LLVM IR

LLVM is a set of compiler and toolchain technologies designed around a strongly-typed language-independent intermediate representation (IR) that serves as a portable, high-level reduced instruction set computer (RISC) assembly language that can be analysed and transformed over multiple compiler passes [32][34].

The LLVM IR representation of a C/C++ program can be obtained by emitting it via clang/clang++. To expand on Section 2.1.3: In IR form, the overall program is represented as a *Module*, which contains information on the libraries, global variables, and functions in the program. Each *Module* is made up of *Function*s corresponding to the functions in the C/C++ program. Each *Function* can be visualised as a control flow graph (CFG) comprising Basic Blocks and edges that shows all the paths that might be traversed by the program during its execution [13]. A Basic Block (BB) is a sequence of instructions with only one entry point and one exit point, and no branching in-between. These instructions operate on values — data computed by a program that may be used as operands to compute other values [17]; a value is represented in the LLVM compiler framework by the *Value* class. Execution of a BB must begin from the entry point and end at the terminator instruction (exit point); executing the first instruction in a BB triggers the in-order execution of all subsequent instructions in the BB [51]. Usually, *Function* execution starts at a single entry BB with no predecessor BBs, and ends at exit BBs with no successor BBs. Information on how a value of a given type is stored in memory — e.g. value size, address alignment — is stored as data layout information that can be retrieved by a compiler pass [28].

LLVM IR uses an infinite set of temporary registers and follows a Static Single Assignment (SSA) form, where each variable (also known as a typed/virtual register or 'variable register') can only be defined/assigned to once [30]. In this dissertation, the term 'variable' is synonymous with '*Value*'. In SSA form, the definition of a variable must always dominate all its uses. We follow the following definition of Domination as per [10]:

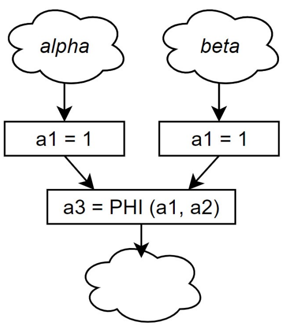


Figure B.1: PHI node usage in a CFG

Definition B.2.1 (Domination)**.** Given a node p with no incoming edges, if a node x in a CFG dominates a node y, then all paths from node p to node y will contain node x .

If a variable is updated when following a path of control flow, the new value is assigned to a new variable register. Different versions of values for the same variable originating from different parts of the control flow are resolved using PHI nodes placed at joins in control-flow [10]. PHI nodes are required in the SSA form because the values of variables that go through reassignment as part of the control flow cannot be determined statically — a PHI node is necessary to help choose which version of the variable's value to use further downstream, depending on the path taken by the control flow to reach the PHI node [30]. In FigB.1, control flow from regions $alpha$ and $beta$ in the

CFG will re-assign different versions of the variable a (i.e. a_1 and a_2) to either 1 or 2. The BB that merges/resolves the control flow from those two branches thus includes a PHI node to choose 1 if the flow comes from $alpha$, and 2 if the flow comes from $beta$. The result of this resolution is assigned to a new register a_3, and is thus a new version of the variable a.

B.3 LLVM Compiler Passes

B.3.1 Analysis Passes and Transformation Passes

Analysis passes are passes which compute information about a unit of IR (e.g. *Function* / *Module*) — which can be used by other passes — without mutating it [33]. Transformation passes are passes that mutate the program's IR, e.g. by inserting new instructions or basic blocks. These passes can use the analysis results obtained by analysis passes; their modifications to the IR can often also invalidate analysis pass' previously-computed results [33].

B.3.2 Function Passes and Module Passes

A function pass (*FunctionPass*) is a pass that performs analysis/transformations on a per-*Function* basis independently of all other *Function*s in the program [35]. *FunctionPass*es operate only within the scope of the single *Function* under inspection, and must be run multiple times if to process multiple *Function*s in a program. When run on a new *Function*, the *FunctionPass* does not retain any analysis data from previous runs on previous *Function*s.

In contrast, a module pass (*ModulePass*) is a pass that operates on a per-*Module* (i.e. per-program) basis and can perform analysis/transformations on a per-*Module* scope across all *Function*s within the program *Module* [35]. A *ModulePass* need only be run once over the *Module* to process all the *Function*s within it, meaning that unlike a *FunctionPass*, it can retain all analysis data on each *Function* in the *Module*, and on global *Module* data. Under certain conditions, *ModulePass*es can invoke *FunctionPass*es that do not themselves require/invoke any *ModulePass* or *ImmutablePass* [35].

Appendix C

Checkpoint Auto-Selection

C.1 Overview

Our compiler support includes the additional capability of automatically determining a set of suitable checkpoint locations where it later attempts to insert checkpoints at. Here, the entire select-and-insert process is fully automatic and does not require any user direction.

C.2 Automatic Selection Algorithm

The SubroutineInjection pass implements a checkpoint auto-selection algorithm as mentioned in Section 3.5 that chooses candidate BBs with the least number of tracked variables $N_{Tracked}$ as checkpointed BBs. This algorithm ignores entry BBs — since entry BBs are re-executed during state restoration (Section 4.1.2), checkpointing them is redundant.

Given a threshold *minValsCount* (initialised to 1), the algorithm attempts to find candidate BBs with an $N_{Tracked}$ of at least *minValsCount* to minimise the number of variables stored in *ckpt_mem_seg*. If successful, the algorithm attempts checkpoint insertion via the checkpoint insertion algorithm. If no candidate BBs meet the threshold requirement or if insertion fails, the algorithm tries again with a larger *minValsCount*, calculated by inspecting the smallest $N_{Tracked}$ amongst the candidate BBs. This algorithm terminates if no checkpoints are inserted by the time the threshold exceeds the largest $N_{Tracked}$ amongst all candidate BBs.

This algorithm can be easily modified to instead minimise the size of *ckpt_mem_seg* by using the size (bytes) of the tracked variables set as the threshold. This involves calculating the total size of each candidate BB's tracked variables via the size calculation techniques used in Sections 4.2.3 and 4.3.2.3, and further detailed in Appendix D.1.

Appendix D

Further Details on Checkpoint Infrastructure and Insertion

D.1 Detailed Variable Size Calculation

We employ the following technique for the 4 following IR datatypes supported by the SubroutineInjection pass to calculate the padded size of each variable, and the number of slots it needs in *ckpt_mem_seg*:

1. **Single Non-pointer Variables (e.g. `i32`):**
   We assign these variables 1 slot in *ckpt_mem_seg*. Since *ckpt_mem_type* is the widest datatype used in the kernel, a single slot can fit any single variable from the kernel. We pad this variable's recorded size to that of *ckpt_mem_type*.

2. **Single Pointer Variables (e.g. `i32*`):**
   We only support single pointer variables to primitive datatypes. These pointers and their pointees thus have the same size. We pad this variable's recorded size to that of *ckpt_mem_type*.

3. **Locally-allocated Array Pointers (e.g. `[3 x i32]*`):**
   The LiveValues data already correctly records the size of these pointers as that of their pointee arrays, so no dereferencing is required. We set the number of slots used as the array's length len, and pad the recorded size to $len \times$*ckpt_mem_type*.

4. **Nested Pointers to Externally-allocated Arrays (e.g. `i32**`):**
   We only support nested pointers of primitive datatypes. These point to the base-addresses (pointers) of externally-allocated arrays passed into kernel functions as arguments. LiveValues data only represents sizes of these pointers, not their pointee arrays. We thus "dereference" these nested pointers by locating their corresponding `alloca`-and-`store` instruction pairs in the IR and retrieving the contained non-nested pointer variable from their operands. We then retrieve the sizing information of this operand and compute its padded size and the number of slots used.

   Note: although LiveValues retrieves the sizes of the input arguments, the JSON

```
%idx_valA_s = getelementptr ...
<store i32 %valA to i32* %idx_valA_s in ckpt_mem>
```

Figure D.1: IR Pseudocode: Saving Single Non-pointer Variables

```
%idx_valA_r = getelementptr ...
%valA_r = <load i32 from i32* %idx_valA_r in ckpt_mem>
```

Figure D.2: IR Pseudocode: Restoring Single Non-pointer Variables

files only transfer data for live-variables, which might not include the dereferenced value. SubroutineInjection thus re-calculates the sizing data before performing any index calculations. A future workflow optimisation would be to instead calculate sizing data only once, within the SubroutineInjection pass.

Since parts of this analysis (and others done by SubroutineInjection) needs to check the names of each kernel function's arguments, the IR version of the kernel must retain human-readable value names. We thus specify `-fno-discard-value-names` to clang++ when first converting the `kernel.cpp` source code into IR.

D.2 The `isComplete` Field

In addition to the `ckptID` metadata field, *ckpt_mem_seg* also includes an `isComplete` field (Fig3.2) which indicates whether a kernel function has returned; if the kernel returns with an exit code, this code will be stored in this `isComplete` field. This field is primarily used for testing and accountability.

To populate the `isComplete` field in *ckpt_mem_seg* as per Section 3.4, the `store_isComplete_val()` step in Fig4.4 visits all exit BBs in the function (i.e. those terminating with a `ret` instruction) and instruments them with a `getelementptr` instruction to obtain the `isComplete` field's address, and a `store` instruction to store either **1)** the function's returned exit code or **2)** the integer value 1 if the function returns `void`.

D.3 Detailed saveBB & restoreBB Instructions

We first compute the address of the slot in *ckpt_mem_seg* that the tracked variable is located via `getelementptr` instructions — which takes as inputs **1)** the base address of *ckpt_mem_seg* given by the domain specific `ckpt_mem` argument, and **2)** the index offset of the slot — into both the saveBB and restoreBB. We then intrument the saveBB and restoreBB pair with symmetric instructions to save/restore the tracked variables. The overall saveBB, restoreBB and junctionBB pseudo-instructions are shown in FigD.9. To maintain the IR's SSA form, saveBB and restoreBB do `getelementptr` separately;

```
%idx_ptrB_s = getelementptr ...
%deref_ptrB_s = <dereference %ptrB>
<store i32 %deref_ptrB_s to i32* %idx_ptrB_s in ckpt_mem>
```

Figure D.3: IR Pseudocode: Saving Single Pointer Variables

```
%idx_ptrB_r = getelementptr ...
%ptrB_r = <alloca i32>
%deref_ptrB_r = <load i32 from i32* %idx_ptrB_r in ckpt_mem>
<store i32 %deref_ptrB_r to i32* %ptrB_r>
```

Figure D.4: IR Pseudocode: Restoring Single Pointer Variables

```
%idx_arrPtrD_s = getelementptr ...
<memcpy i32* %arrPtrD to i32* %idx_arrPtrD_s in ckpt_mem>
```

Figure D.5: IR Pseudocode: Saving Local Array Pointer Variables

variables symmetric across saveBB and restoreBB are differentiated by the '_s' suffix in saveBB, and '_r' in restoreBB. The 4 supported variable types each follow a different save and restore policy with different subroutines, as shown in FigD.9 and detailed below:

1. **Single Non-pointer Variables (e.g. `i32`):**
   With respect to FigD.1, the saveBB is instrumented with a `store` instruction to save the variable `%valA` directly to the slot address `%idx_valA_s`. This method is also used to store the checkpoint's ID value or the "dirty bit" -1 to `ckpt_ID` (Section 4.3.2.4). With respect to FigD.2, the restoreBB is instrumented with a `load` instruction to load the saved value from `%idx_valA_r` into a new variable register `%valA_r` in accordance with LLVM IR's SSA form.

2. **Single Pointer Variables (e.g. `i32*`):**
   With respect to FigD.3, the saveBB is first instrumented with a `load` instruction to "dereference" the pointer variable `%ptrB` into `%deref_ptrB`, then a `store` instruction to save this dereferenced value to the slot address `%idx_ptrB_s`. With respect to FigD.4, the restoreBB is instrumented first with an `alloca` instruction to allocate a new single-value pointer `%ptrB_r`, then a `load` instruction to load the saved value from `%indx_ptrB_r` into `%deref_ptrB`, and finally a `store` instruction to finally store this value into the new `%ptrB_r`.

3. **Locally-allocated Array pointers (e.g. `[3 x i32]*`):**
   With respect to FigD.5, the saveBB is instrumented with a `memcpy` instruction to copy the array's contents from `%arrPtrD` to *ckpt_mem_seg* at `%idx_arrPtrD_s`; the restoreBB is also instrumented with a similar `memcpy` instruction to copy the array from `%indx_arrPtrD_r` back into `%arrPtrD`, as shown in FigD.6. These `memcpy` instructions take as input the number of bytes to copy starting from the source address location, corresponding to the padded size of `%arrPtrD` previously calculated in `compute_ckpt_mem_indx()`.

   We restore the contents of locally-allocated arrays back to their original pointers because of consistency and the following: LLVM IR references such arrays directly via non-nested array pointers; creating new array pointers requires copying

```
%idx_arrPtrD_r = getelementptr ...
<memcpyi32* %idx_arrPtrD_r in ckpt_mem to i32* %arrPtrD>
```

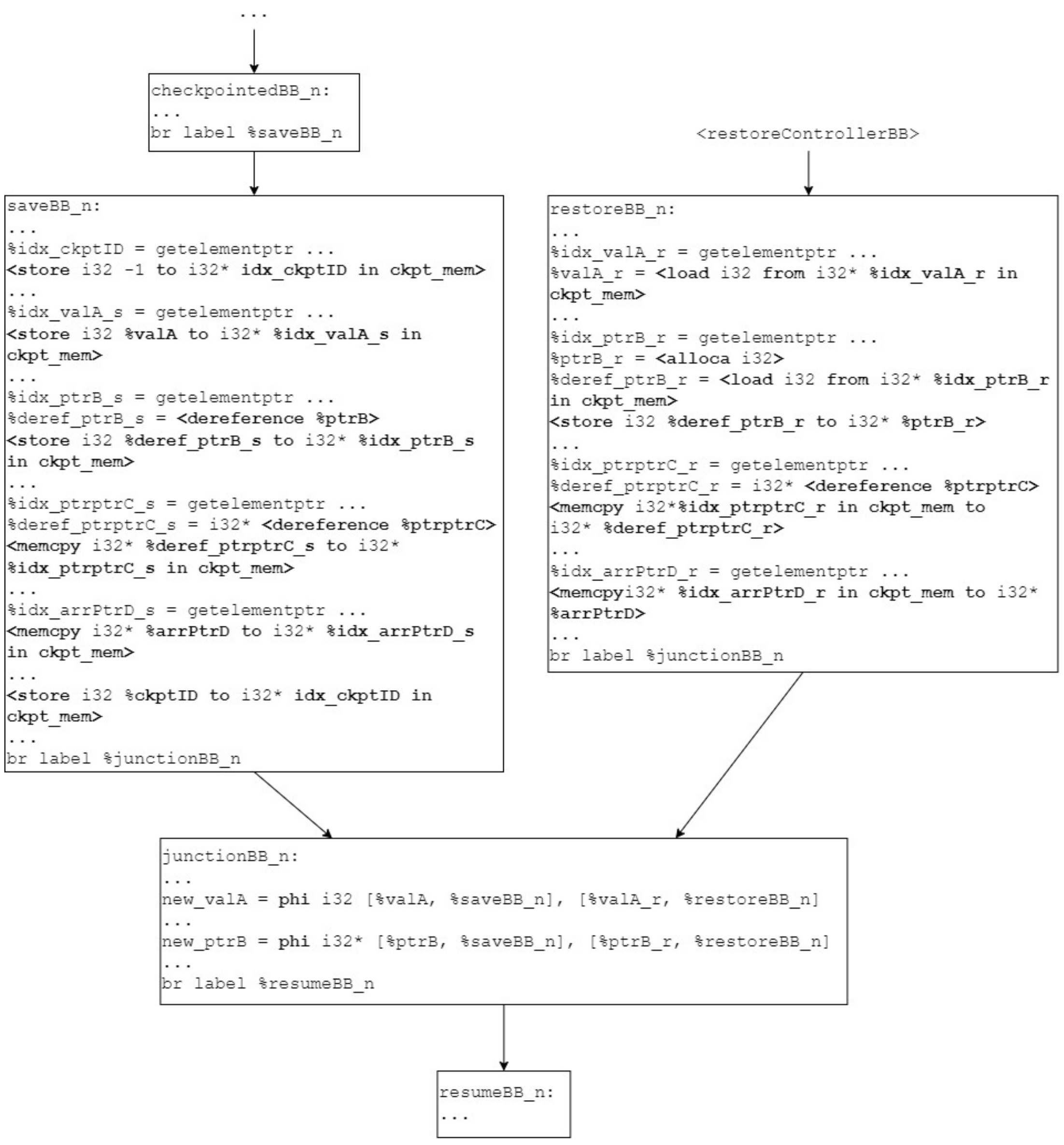
Figure D.6: IR Pseudocode: Restoring Local Array Pointer Variables

```
%idx_ptrptrC_s = getelementptr ...
%deref_ptrptrC_s = i32* <dereference %ptrptrC>
<memcpy i32* %deref_ptrptrC_s to i32* %idx_ptrptrC_s in ckpt_mem>
```

Figure D.7: IR Pseudocode: Saving Nested Pointer Variables

```
%idx_ptrptrC_r = getelementptr ...
%deref_ptrptrC_r = i32* <dereference %ptrptrC>
<memcpy i32*%idx_ptrptrC_r in ckpt_mem to i32* %deref_ptrptrC_r>
```

Figure D.8: IR Pseudocode: Restoring Nested Pointer Variables



```
...

checkpointedBB_n:
...
br label %saveBB_n

saveBB_n:
...
%idx_ckptID = getelementptr ...
<store i32 -1 to i32* idx_ckptID in ckpt_mem>
...
%idx_valA_s = getelementptr ...
<store i32 %valA to i32* %idx_valA_s in
ckpt_mem>
...
%idx_ptrB_s = getelementptr ...
%deref_ptrB_s = <dereference %ptrB>
<store i32 %deref_ptrB_s to i32* %idx_ptrB_s
in ckpt_mem>
...
%idx_ptrptrC_s = getelementptr ...
%deref_ptrptrC_s = i32* <dereference %ptrptrC>
<memcpy i32* %deref_ptrptrC_s to i32*
%idx_ptrptrC_s in ckpt_mem>
...
%idx_arrPtrD_s = getelementptr ...
<memcpy i32* %arrPtrD to i32* %idx_arrPtrD_s
in ckpt_mem>
...
<store i32 %ckptID to i32* idx_ckptID in
ckpt_mem>
...
br label %junctionBB_n

<restoreControllerBB>

restoreBB_n:
...
%idx_valA_r = getelementptr ...
%valA_r = <load i32 from i32* %idx_valA_r in
ckpt_mem>
...
%idx_ptrB_r = getelementptr ...
%ptrB_r = <alloca i32>
%deref_ptrB_r = <load i32 from i32* %idx_ptrB_r
in ckpt_mem>
<store i32 %deref_ptrB_r to i32* %ptrB_r>
...
%idx_ptrptrC_r = getelementptr ...
%deref_ptrptrC_r = i32* <dereference %ptrptrC>
<memcpy i32*%idx_ptrptrC_r in ckpt_mem to
i32* %deref_ptrptrC_r>
...
%idx_arrPtrD_r = getelementptr ...
<memcpyi32* %idx_arrPtrD_r in ckpt_mem to i32*
%arrPtrD>
...
br label %junctionBB_n

junctionBB_n:
...
new_valA = phi i32 [%valA, %saveBB_n], [%valA_r, %restoreBB_n]
...
new_ptrB = phi i32* [%ptrB, %saveBB_n], [%ptrB_r, %restoreBB_n]
...
br label %resumeBB_n

resumeBB_n:
...
```

Figure D.9: Detailed Checkpoint CFG

the address value from the original pointer. However, LLVM IR does not have 'move' or 'copy' instructions; while such semantics can be replicated via `alloca` and `load`/`store` instructions, we choose to adopt the aforementioned unified array restoration policy for consistency.

4. **Nested-pointers to Externally-allocated Arrays (e.g. `i32**`):**
   As per FigD.7, the saveBB is first instrumented with a `load` instruction to "dereference" the nested pointer to retrieve the address `%deref_ptrptrC` of the externally-allocated array, then a `memcpy` instruction to copy the array's contents from `%deref_ptrptrC` into *ckpt_mem_seg* at `%idx_ptrptrC_s`, taking the padded size of the array (bytes) as input. The restoreBB is instrumented with a symmetric set of load and `memcpy` instructions to copy the contents from *ckpt_mem_seg* back to the array address pointer, as shown in FigD.8.

If the tracked variable is not *ckpt_mem_type*, we insert type conversion instructions to convert them to *ckpt_mem_type* before storing to *ckpt_mem_seg*, and back to their original datatype after loading from *ckpt_mem_seg*. E.g. if *ckpt_mem_type* is a `float` and the tracked variable is a single non-array pointer of type `i32*`, the algorithm will insert an `sitofp` instruction in the saveBB to convert the dereferenced value into a `float` for storage, and an `fptosi` instruction in the restoreBB to convert the value back into an `i32` before restoring it into a new `i32*` *Value*.

D.4 Selective Partial Insertion: Save-Only, Restore-Only and Save-Restore Modes

SubroutineInjection also supports selective partial insertion of checkpointing infrastructure, specified via the following command line options under the `-inject` tag:

1. **`save_restore`:**
   Specifies full save-restore mode; SubroutineInjection inserts the full suite of context-saving and context-restoring infrastructure.

2. **`save`:**
   Specifies save-only mode; SubroutineInjection only inserts the context-saving elements of the checkpointing infrastructure, i.e. the saveBBs.

3. **`restore`:**
   Specifies the restore-only mode; SubroutineInjection only inserts the context-restoring elements of the checkpointing infrastructure, i.e. the restoreBBs and junctionBBs, and the restoreControllerBB.

This allows the user to configure the IR modifications to better suit their setup. E.g. if the user only wishes to migrate tasks from FPGA to CPU and not vice versa, they can specify `save` for the IR to be synthesised into FPGA bitstreams, and `restore` for the IR to be compiled into CPU executables. These IRs would only contain the infrastructure strictly required for FPGA to CPU migration, which improves performance and reduces the size of the bitstreams and binaries produced.

Appendix E

The JSON Workaround File Formats

```
{
  ...
  "@func_n" : {
    ...
    "%BB_n" : {
      "live-in" : [
        ...
        "%val_i_n", <size_i_n>
        ...
      ],
      "live-out" : [
        ...
        "%val_o_n", <size_o_n>
        ...
      ]
    },
    ...
  },
  ...
}
```

Figure E.1: `live_values.json` format

```
{
  ...
  "@func_n" : {
    ...
    "%BB_n" : {
      "tracked values" : [
        ...
        "%val_i_n",
        ...
      ]
    },
    ...
  },
  ...
}
```

Figure E.2: `tracked_values.json` format

Both `live_values.json` and `tracked_values.json` follow the format of the nested map that is re-constructed in SubroutineInjection (Section 4.2.2), with the *Module* as the top level. In `live_values.json` (FigE.1), the *Module* consists of *Function* entries (e.g. `%func_n`) comprising BB entries (e.g. `%BB_n`) that each contain the lists `live-in` and `live-out`, which are each formed of *Value*-size pairs, e.g. `%val_i_n`, `size_i_n` and `%val_o_n`, `size_o_n` respectively. These value sizes are used for variable size calculations in Appendix D.1 and index offset calculations in Section 4.3.2.3. The format of `tracked_values.json` (FigE.2) is largely similar, except each BB entry only comprises a single list `tracked variables` formed of *Values*.