Efficient hardware checkpointing: Concepts, overhead analysis, and implementation

Conference Paper · January 2007

DOI: 10.1145/1216919.1216950 · Source: DBLP

Dirk Koch
Friedrich-Alexander-University of Erlangen-Nürnberg
55 PUBLICATIONS 937 CITATIONS

Christian Haubelt
University of Rostock
192 PUBLICATIONS 1,775 CITATIONS

Efficient Hardware Checkpointing – Concepts, Overhead Analysis, and Implementation

Dirk Koch, Christian Haubelt and Jürgen Teich
University of Erlangen-Nuremberg
Am Weichselgarten 3
D-91058 Erlangen, Germany
{dirk.koch, haubelt, teich}@cs.fau.de

ABSTRACT

Progress in reconfigurable hardware technology allows the implementation of complete SoCs in today's FPGAs. In the context design for reliability, software checkpointing is an effective methodology to cope with faults. In this paper, we systematically extend the concept of checkpointing known from software systems to hardware tasks running on reconfigurable devices. We will classify different mechanisms for hardware checkpointing and present formulas for estimating the hardware overhead. Moreover, we will reveal a tool that takes over the burden of modifying hardware modules for checkpointing. Post-synthesis results of applying our methodology to different hardware accelerators will be presented and the results will be compared with the theoretical estimations.

Categories and Subject Descriptors

B.5.3 [**Hardware**]: Register-Transfer-Level Implementation-Reliability and Testing[Redundant design]

General Terms

Design, Reliability

Keywords

Checkpointing, State Access

1. INTRODUCTION

Distributed reconfigurable systems are becoming important for networked applications like in the automotive or body area network domain. The introduction of reconfigurability permits the implementation of dedicated modules in hardware, in order to meet objectives that cannot be reached by software-only systems (e. g., throughput, power).

In order to manage such reconfigurable systems, operating system concepts for dynamically reconfigurable systems have been proposed [3, 12] solving scheduling, placement, and system integration issues online. These approaches, however, do not focus on aspects of fault tolerant system design especially for distributed systems that allow to compensate faults at the network level. If a distributed reconfigurable embedded system demands high availability it must be designed to deal with faults that can occur anytime at runtime.

In this work, we investigate how *checkpointing* mechanisms that are well known for software systems [6], can be utilized in hardware modules running on reconfigurable devices. Checkpointing is a technique where the state of a task is stored during fault free operation. Most common, this state is stored on a remote computing node that will take over task execution in case of a fault. The process when a task is set to the last fault free state is called "rollback" and helps to avoid a massive loss of computation.

If we apply checkpointing schemes we have to define a certain fault model. In this work, we aim to compensate permanent faults of hardware modules that have internal states that have to be considered when the system recovers from a fault. We detect a module's fault if it produces no results or no alive messages within a certain time span specified at design time. An extensive overview about fault models and fault detection techniques can be found in [5]. Based on specific fault models, various approaches for fault tolerant reconfigurable systems have been proposed to compensate faults on FPGA devices. In [16], the possibility of reconfiguration is used to separate faulty resources from a hardware module. It must be mentioned that reconfigurability produces an enormous logic overhead as compared to ASIC implementations, and if the reconfiguration logic itself fails permanently, the system has no possibility to recover. For networked systems with nodes based on FPGAs, we aim to exploit both the network level and the FPGA device level to compensate faulty resources. Therefore, we need techniques to extract a module's state, typically given by all or a subset of the register and memory values of a hardware module. This state needs to be transferred to another device that will take over the execution of a faulty task starting from the last correct stored state (checkpoint).

One major aspect examined in this work is the question how the state of a hardware module can be extracted and re-

FPGA'07, February 18–20, 2007, Monterey, California, USA.

stored with a) low latency penalty and b) with low resource overhead. In addition, we want to estimate these penalties and resource overheads as soon as possible (e.g., when we have information about the state space and the data types used to implement a hardware module) in order to assist a design space exploration at the system level.

Beside i) hardware checkpointing, this analysis is relevant for ii) system test where hardware modules can be initialized with some test vectors or where the internal state of a hardware module can be verified and finally iii) for preemptive hardware task execution on dynamically reconfigurable FPGAs.

State extraction and restoration of hardware modules has been proposed in systems where FPGAs are used for a preemptive execution [11]. Hardware task preemption is one possibility allowing different modules to share the same resource over time. One main difference between hardware checkpointing and preemptive hardware task execution is that checkpoints are taken rapidly in order to decrease the loss of computation in case of a fault while a hardware task preemption will be triggered rarely because a hardware task preemption comes together with a time consuming reconfiguration process. As a consequence, the time overhead to access the state of a hardware module is more critical when the state extraction is used for hardware checkpointing. Another difference is that the extracted state data will be used on the same device with the same reconfiguration in case of hardware task preemption while in case of hardware checkpointing, the state data should even be transferable between different devices, i.e., FPGA types.

In the following, we will present related work where hardware task preemption has been investigated for FPGA devices. In [10, 13], multi-context FPGAs have been presented, where the execution of the FPGA can be switched between different configuration planes in a single cycle. As these planes should be read/writable, it would be possible to use such architectures for checkpointing. However, multi-context FPGAs are extremely costly. Another approach [4] based on the XC6200 FPGA proposes to separate the combinatorial logic from the register values in order to store just the fraction of the configuration data that contains the register values. This design style massively decreases the maximum operating frequency because of elongated routing paths. Moreover, no dedicated tools have been developed. The configuration readback capability was used in [11] for preemptive hardware execution and in [8] for a FPGA defragmentation of modules including their state. These approaches are straightforward to implement but the readback process is very slow and additional effort is required to restart a module with a specific state.

Beside these general approaches that are ineligible for devices found on the market today, hardware checkpointing on FPGAs has been addressed rarely. In [7], a checkpoint scheme for a fault tolerant LZ-compressor has been proposed. The approach is highly optimized but no automatic design for checkpointing was presented.

In this work, we present a formal model for hardware checkpointing for the first time. Based on this model, we propose different checkpointing mechanisms that by transformation of existing module specifications, allow to enhance the module with the capability of hardware checkpointing. Furthermore, we present a new tool named StateAccess that takes over the above transformations of a given module for supporting hardware checkpointing. This tool can be easily integrated in commonly used design-flows.

The paper is organized as follows: In the next section, we introduce a formal model for hardware checkpointable modules based on finite state machines. In Section 3, we will propose and classify different checkpointing alternatives to access the state of a hardware module. Section 4 presents an extended design-flow for automatically integrating hardware checkpoint mechanisms into hardware modules based on our tool StateAccess. Section 5 will present experimental results on hardware and latency overheads for hardware checkpointing.

2. TASK MODEL

Before we present the model assumed for hardware checkpointing, we define a checkpoint as follows:

> **Definition:** A *checkpoint* is a set of data items representing the image of the last error-free state of a module of computation from which in case of the occurrence of a fault may be restarted.

In the context of hardware modules, we consequently need techniques to extract a module's state as well as techniques to restore a module with such a state in case of a fault.

In the following, we assume that each hardware module can be modeled by a Finite State Machine (FSM). An FSM is a sextuple $(I, O, S, \delta, \omega, s_0)$ with a set of inputs I, a set of outputs O, a set of internal states S, a state transition function δ, an output function ω, and an initial state s_0 (see also Figure 1a)).

In order to allow to restore a module from a checkpoint, we have to restore the input values $i \in I$, and the output values $o \in O$ together with the internal state $s \in S$ captured during the checkpointing process. This feature has to be supplied by the interface of the hardware module. An FSM that has the capability to save and to restore checkpoints will be called a *checkpoint FSM* (CFSM).

The most general case of a CFSM is presented in Figure 1b). Depending on the utilized checkpointing protocol the CFSM may have different flavors. One interesting variant for fault tolerant system design is to use one CFSM as a main task that sends periodically a checkpoint to a redundant shadow-task (again a checkpoint-FSM) hosted on another node in a networked system. In case of a fault, e.g., because of a complete node failure, the shadow module can take over the execution starting from the last checkpoint. In this example, both checkpoint-FSMs require only one set of state-registers and the checkpoint may be stored inside the registers of the shadow module. As presented later in this section, our approach allows to save checkpoints only in userdefined states that are a subset of the original state $s \in S$, thus a state can be stored with less memory or can be transferred faster to a redundant node.

Given an FSM $m = (I, O, S, \delta, \omega, s_0)$ and a set of checkpoints $S_c \subseteq S$, we can derive the corresponding CFSM $m_c = (I', O', S', \delta', \omega', s_0')$ as follows:

$I' = I \times S_c \times I_{\text{save}} \times I_{\text{restore}}$; $I_{\text{save}} = I_{\text{restore}} = \{0, 1\}$,
$O' = O \times S_c$,
$S' = S \times S_c$.

In the concept of a CFSM, the original input set I is extended by two signals $I_{\text{save}} \in \{0, 1\}$ and $I_{\text{restore}} \in \{0, 1\}$ to control the operation mode of the state machine. In addition, a checkpoint i_c can be supplied through I' for

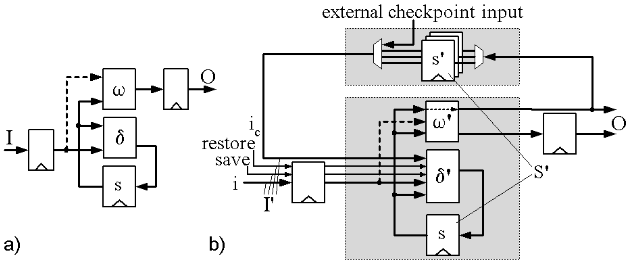


Figure 1: a) Finite State Machine (FSM) and b) the corresponding Checkpoint-FSM (CFSM). The original module will be extended, such that its state can be read or set from outside through i_c. The read and write process is controlled by the external signals I_save and $I_restore$. When the present state s of the module is accessed for saving or restoring a checkpoint, the values of the input and output registers are also accessed. This is treated as part of the external interface. Note that it is possible to swap the state between s and s'.

a rollback. The checkpoint is extracted through O'. The checkpoint is stored with the original state $s \in S$ of m.

In the following, it is assumed that the current state of our state machine m' is given by $(s, s') \in S'$, where s is the current state and s' the last saved checkpoint. The current input is denoted i'. The state transition function $\delta' : S' \times I' \to S'$ is then given as:

$$\delta' = \begin{cases} (\delta(s,i), s') & \text{if } i' = (i, -, 0, 0) \\ & \quad \text{run mode, } s' \text{ remains checkpoint} \\ (\delta(s,i), s) & \text{if } i' = (i, -, 1, 0) \wedge s \in S_\mathrm{c} \\ & \quad \text{save checkpoint} \\ (\delta(s,i), s') & \text{if } i' = (i, -, 1, 0) \wedge s \notin S_\mathrm{c} \\ & \quad \text{continue to save checkpoint} \\ (\delta(i_\mathrm{c},i), s') & \text{if } i' = (i, i_\mathrm{c}, 0, 1) \\ & \quad \text{restore checkpoint} \\ (\delta(i_\mathrm{c},i), s) & \text{if } i' = (i, i_\mathrm{c}, 1, 1) \wedge s \in S_\mathrm{c} \\ & \quad \text{swap state } s \text{ with checkpoint } s_c \\ (\delta(s,i), s') & \text{if } i' = (i, i_\mathrm{c}, 1, 1) \wedge s \notin S_\mathrm{c} \\ & \quad \text{continue to swap checkpoint} \end{cases}$$

The output function ω' is defined as: $\omega' = (\omega(s,i), s')$. And the initial state is $s'_0 = (s_0, s_0)$.

In general, it is possible to omit a register from the checkpoint if this register is written before it will be read under all possible state sequences started from a checkpoint state s_c. It is further possible to initialize saving a checkpoint and restoring another checkpoint at the same time. In this case, s is exchanged with s' at the next possible checkpoint.

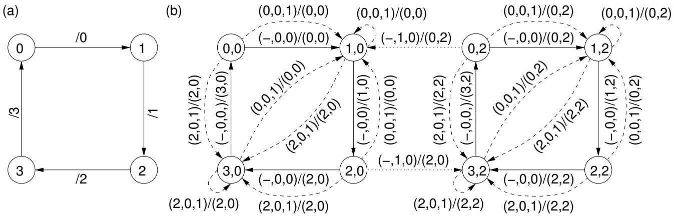


Figure 2: (a) FSM of a modulo-4-counter. (b) Corresponding CFSM for $S_\mathrm{c} = \{0, 2\}$, i.e., only in states 0 and 2 saving of a checkpoint is permitted. The state space is given by the actual state and the last saved checkpoint.

With this system model, we have full control over saving checkpoints or initiating a rollback by restoring the last saved checkpoint. If we request to store a checkpoint in case when $s \notin S_c$, the CFSM will continue its operation in the run mode until the next possible checkpoint has been reached. In case $S_c = S$, it is possible to save a checkpoint in each state. However, under some circumstances, it can be advantageous to omit some states from the set of possible checkpoints. If, for example, a hardware module requires a lot of intermediate register values during execution, it is possible to reduce the amount of memory required to store a checkpoint by taking checkpoints only when this module is in an idle state, thus, allowing to omit saving the intermediate register values. In this case only the idle states are in the set of checkpoints, S_c. Figure 2 gives an example of a modulo-4-counter module with the states $S = \{0, 1, 2, 3\}$ and the derived CFSM with the set of checkpoint states $S_c = \{0, 2\}$. In this example, we can decrease the number of bits to one required to recover the module in case of a fault.

We have not restricted yet how and where the checkpoint s' is stored. With respect to the fault model, this system model can be used on the device level as well as on board or even network level. In the latter case, s' needs to be stored on a different device of a network than on the one accommodating the CFSM. In case of a fault, a redundant CFSM has to take over the computation starting with the last saved checkpoint. This model fits ideally distributed systems that are based on FPGAs where the redundant CFSM can be instantiated on demand due to the flexibility provided by partial runtime reconfiguration. The model demands that a checkpoint access is executed atomically, and that the CFSM stores only consistent states for S'.

At this point, we will divide checkpointing schemes into two classes. In the first class, it is acceptable to lose input data arrived at the CFSM between the last saved checkpoint and the occurrence of a fault. In the second class, this is not acceptable and all input values have to be supplied to the module regardless when the fault occurs. For the first class, we do not demand any restrictions to the communication model while we demand FIFO semantics on channels between the hardware modules in the second case. For this purpose, we have to include the memory elements of the channel into our checkpoint because they represent a state that is required to restart and continue the operation after a fault. For this reason, we propose a *transaction-based FIFO* (T-FIFO). An example of this FIFO is shown in Fig. 3 where the T-FIFO is illustrated as a ring buffer with a write pointer WR, a read pointer RD and a checkpoint pointer RD_c. Whenever a checkpoint is saved in the attached CFSM, we will further store the value of RD_c together with s' from the CFSM. FIFO data between RD_c and RD is not allowed to be overwritten. In the rollback case, the read pointer (RD) is set to the position of the stored read pointer at the last taken checkpoint (RD_c). Consequently, the output values since the last checkpoint will be supplied again to the CFSM. Note that RD and WR play the role of typically FIFO pointers with a coresponding read and write port each, while RD_c is just a marker for the last stored Checkpoint.

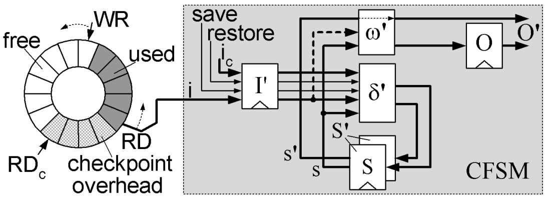


Figure 3: Transaction-based FIFO for fault tolerant storing communicated input and output data between HW-modules. Whenever a checkpoint is taken, the RD_c pointer is saved together with the state of the hardware module. When a rollback is performed, e.g., in case of a fault, the read pointer RD will be set back to the stored RD_c value. As a consequence, the module will get again all incoming data since the last stored checkpoint.

In our task model, the interprocess communication is separated from the CFSM. Therefore, the FIFO is not bounded to the same device as the CFSM. If the T-FIFO cannot be assumed to be robust to potential faults, it has to be realized redundantly. This T-FIFO concept has some relationship with message logging approaches [6] known from the software domain.

3. CONCEPTS FOR SAVING AND RESTORING HW-CHECKPOINTS

The state of a hardware module is represented by the value of all memory elements inside the module. In the following, we assume fully synchronous hardware modules without any latches. The presented system model allows many variations in the way how the flip-flops values, representing a current state s, can be accessed. Different methodologies are required because of the opposed objectives, e.g., time behavior and hardware overhead.

In the following, we will introduce and analyze three methods for hardware checkpointing i) *memory-mapped state access* (MM), ii) *scan chain based state access* (SC), and iii) *shadow based scan state access* (SHC) that can be applied to a hardware module, in order to automatically derive the Checkpoint FSM (CFSM) from a high level module specification. All these methodologies differ in terms of the latency L to extract a checkpoint and the additional hardware H required for saving and restoring the checkpoints.

3.1 Metrics for Accessing HW-Checkpoints

Before we examine different methodologies to access the current state of a hardware module, we define metrics for a quantitative classification of different methodologies.

We assume that the original hardware module has the following parameters: (1) a_{LUT}, a_{FF} are the number of look-up tables (LUTs) and flip-flops of the module. (2) F_{max} denotes the modules maximal operating frequency. The variables a^*_{LUT}, a^*_{FF}, and F^*_{max} describe the corresponding parameters of the checkpointable hardware module after modification. Then, the different methodologies can be compared using the following indicators:

•*Checkpoint Hardware Overhead* specifies the amount of additional resources required by a specific checkpoint mechanism. We distinguish between $H_{LUT} = a^*_{LUT} - a_{LUT}$ and $H_{FF} = a^*_{FF} - a_{FF}$ denoting the number of required extra look-up tables and flip-flops, respectively.

•*Checkpoint Performance Reduction R* specifies the reduction of the maximal achievable clock frequency due to the increased module complexity ($R = F^*_{max} - F_{max}$).

•*Checkpoint Overhead C* specifies the time, a module must be interrupted when saving a checkpoint. This interrupt leads to an increase in the execution time.

•*Checkpoint Latency L* specifies the amount of time required until the complete checkpoint data arrived at the device that will take over the task execution in case of a fault.

Based on the overhead C and the latency L, we can derive further indicators used for a quantitative comparison of the approaches presented later in this section. The relationship between C and L is depicted in Fig. 4. The example shows a task T sending a checkpoint to another task T' with the checkpoint period P specifying the time between two checkpoints. Due to communication, the latency L is larger than the overhead C. We can further define the *checkpoint efficiency* $\lambda = 1 - \frac{C}{P}$ denoting the relative amount of time a module spends on its execution. Fig. 4 shows an example where a fault of task T is recognized after a fault detection latency D. After fault detection, task T' takes over the execution starting with the last completely stored state (here at t_0). The maximal *loss of computation time* is then $\gamma = P + L + D$.

3.2 Memory-Mapped State Access

In the memory-mapped methodology (MM), a processor accesses the module's state by integrating the flip-flops storing the checkpoint into a read/writable memory space of a CPU. In order to keep the state consistent, the atomicity property for read/write operations must be met and the module must not operate until a state extraction or a state restore process has been completed. This is achieved by a multiplexer that feeds back the present value of a flip-flop while saving a checkpoint (Fig. 5). In order to reduce the checkpoint overhead C when storing a checkpoint, flip-

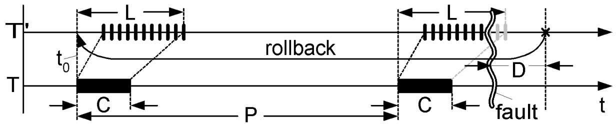
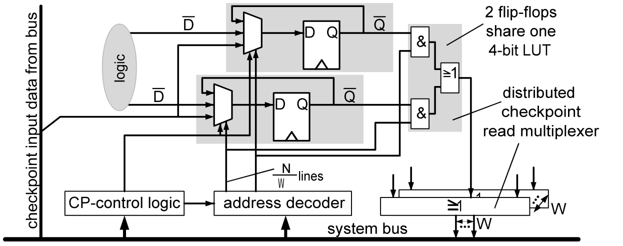


Figure 4: Relationship between the checkpoint overhead C and the checkpoint latency L. Checkpoints are strorred continuously with the period P. In order to read a consistent state, the hardware task has to stop its operation for the time C. L denotes the time required to save the checkpoint into a secure memory. After a fault occurs, it takes the time D to recognize this event. In this case the CFSM or a redundant one performs a rollback to the last completely stored checkpoint (in this example, the state stored at time t_0).



checkpoint input data from bus
logic
D
D
D Q
D Q
Q
Q
&
&
≥1
2 flip-flops share one 4-bit LUT
distributed checkpoint read multiplexer
N/W lines
CP-control logic
address decoder
≥1
W
W
system bus

Figure 5: Hardware checkpointing by memory-mapped state access. W Flip-flops are clustered together according to the word size of the bus, thus, reducing the complexity of the address decoder to $\frac{N}{W}$ select lines.

flops will be grouped together to *packages* according to the data-bus width W.

As we focus on FPGA architectures, we can examine the hardware overhead with respect to multiplexer based SoC-busses like the Avalon from Altera [1] or the OPB from Xilinx [15]. The extra amount of flip-flops H_{FF} is mainly related to the module interface. This stems from the fact that input and output values are additionally registered during state access. However, this value is marginal with respect to the overall amount of flip-flops found in the module.

Let N be the number of flip-flops of a module and W be the data bus width, then the amount of additional 4-bit LUTs can be estimated as:

$$H_{LUT}^{MM} \approx \underbrace{\frac{N}{W} + \sum_{n=0}^{\lceil log_8(\frac{N}{W})\rceil - 1} 8^n}_{\text{address decoder}} + \underbrace{\frac{N}{2} + W \left\lceil \frac{\frac{N}{2W} - 1}{3} \right\rceil}_{\text{read mux}} + \underbrace{N}_{\text{FF mux}} \quad (1)$$

It must be noted that an address decoder needs a LUT for each output select signal. In case of 4-bit LUTs, it is possible to evaluate three address bits in addition to the enable signal generated by the evaluation of the higher address lines. Therefore, the decoder can be modeled as a B-tree with 8 children (selected by three address bits) for each select output wire. It must be further mentioned that the read multiplexer consists of an OR-gate for every bit connected to the read data bus. If we map wide input OR-gates to 4-bit LUTs, we get a B-tree with 4 children for each input node.

Eq. (1) estimates the amount of logic elements required for the address decoder and the additional logic for the feedback and restore multiplexer (Fig. 5) required for each flip-flop of the module. Eq. (1) gives a worst case estimate. Current architectures and tools support to translate the feedback multiplexer to a flip-flop enable signal, and the restore multiplexer is mapped together with the original circuit sharing resources. Therefore, H_{LUT} will be less than N in practice.

In order to estimate the checkpoint overhead C, we can determine the bus read/write operations necessary to save or restore a checkpoint. Beside a control register access in the beginning and the end to set the read/restore mode, we need:

$$C_{MM} = \frac{N}{W} \cdot \tau_{CPU} \quad (\tau_{CPU} \text{ represents a CPU read cycle}) \quad (2)$$

bus cycles to transfer the state data plus the bus-cycles required to access the wrapper registers storing the interface signals of the module. Thus, C scales with $O(N)$. The checkpoint latency L depends mainly on the state data size and the communication performance and can be easily estimated for a specific target environment.

3.3 Scan Chain based State Access

Instead of overlaying the flip-flop values into the memory address space, the flip-flops can be chained together through a long shift register chain (SC). This is known from established chip test techniques [2]. A multiplexer in front of every flip-flop is used to switch between normal operation and the shift register operation where all flip-flops are connected to a long shift register. In general, it is not possible to read the complete chain atomically. Therefore, we propose to connect the register chain ends to form a ring shift register during time C. This allows us to keep a state consistent without the need of additional logic for each flip-flop of the module. We use an additional state machine that keeps track of the present shift position of the module's state data in the chain. This additional state machine is further used to automatically read and write selectively portions of the flip-flop values through an extra register. This mechanism allows to access the state with the full data bus width W.

The hardware overhead H_{FF} depends again on the module interface Z. In addition, some memory is required for the state machine controlling the scan chain access. Beside a constant value k (for the control logic), it needs $\lceil log_2 N\rceil$ flip-flops for a scan shift position counter and $\lceil log_2 N/W\rceil$ flip-flops for the word position register for a module with N flip-flops connected to a W-bit wide bus. Thus, the flip-flop overhead is:

$$H_{FF}^{SC} \approx k + Z + \lceil log_2 N\rceil + \lceil log_2 \frac{N}{W}\rceil + W \quad (3)$$

The additional amount of look-up tables H_{LUT} is in the worst case given by the logic for the scan chain interface plus an additional LUT for the scan multiplexer in front of each flip-flop of the module. Again, this overhead will be less than N in practice. If, for example, the value of a preceding flip-flop F_p of a scan chain flip-flop F is also evaluated to determine the value of F ($F = f(F_p)$) during normal operation, then just one additional control input signal is fed to the combinatorial logic in front of F. The advantage of the ring-shift is that state consistency can be guaranteed with a low resource overhead by the cost of an increased transfer time for the CPU subsystem. When a specific fraction of the ring shift register is addressed, it takes in average $N/2$ cycles until the state data has passed the extra register for the state access. $\frac{N}{W}$ times a value has to be transferred between the extra register and the CPU subsystem. Therefore, the checkpoint overhead C scales with $O(N^2)$ leading to a poor checkpoint efficiency λ for large checkpoints. Instead of using one ring-shift register of length N, it is alternatively possible to generate M parallel running shift-register chains each of length $\frac{N}{M}$. If in this case $N \, mod \, M \neq 0$, padding flip-flops have to be included to balance the chains. M must be chosen to be less or equal than W. Then, the checkpoint overhead C is about a factor of $\left(\frac{1}{2}\frac{N}{M}\right) \cdot \frac{N}{W}$ larger as compared to the memory-mapped approach.

3.4 Shadow Scan Chain based State Access

For systems where checkpoints are taken frequently, the checkpoint efficiency λ can decrease dramatically. In such situations, it is advantageous to duplicate all flip-flops of the original module in order to perform a state copy within a

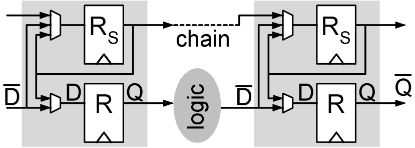


Figure 6: Hardware Checkpointing by shadow scan chain based state access. This technique allows taking a snapshot in parallel to the normal execution of a module.

single clock cycle. This is illustrated in Fig. 6 where registers R of the original circuit have been duplicated (R_S). The duplicated ones can be chained together by multiplexers. During normal execution, each flip-flop R is connected to the unmodified logic. With respect to our CFSM model, R stores the state representation of the present state s of the original FSM while R_S stores the checkpoint state s'.

A checkpoint is saved by connecting both flip-flops (the working register R and the shadow register R_S) to the current logic output for a single cycle. This is completely transparent for the operating module. Consequently, the checkpoint overhead is zero and the efficiency is $\lambda = 1$. However, we will use C_{SHC} to denote the time required to read the shadow chain in the background while the module keeps running. In case of a rollback, the state information is shifted into the shadow chain, then loaded into the working registers and operation will start immediately. If the checkpoint data is stored inside a redundant CFSM, the rollback latency can be reduced to a minimum.

If an FSM with N binary memory elements (i.e., flip-flops) is transferred to a CFSM, the overhead for the additional flip-flops and LUTs is obviously:

$$H_{FF}^{SHC} \approx k + Z + N \quad (4) \qquad H_{LUT}^{SHC} \approx k + Z + 2 \cdot N \quad (5)$$

3.5 Adapting Netlist Primitives

Beside the presented hardware extensions applied at the netlist level, we have to ensure that during state extraction and state restoration all set or clear signals of the flip-flops are disabled. Furthermore, if memories use enable signals, we have to force these enable signals to be active. This is achieved by adding an additional control signal to the logic that generates the set, clear, or enable signals for the flip-flops. We have not considered the logic overhead in the estimates presented above because the amount of additional logic is normally very small and the control signals of the flip-flops are shared for registers and register banks.

If a hardware module instantiates on-chip memory blocks, we build a wrapper similar to the memory-mapped approach around this memory block. This allows the CPU to randomly access the state stored in on-chip memories.

4. AUTOMATIC HARDWARE TRANSFORMATION

In the following we will present a tool flow how the above mentioned techniques can be integrated seamlessly into typical system design-flows.

For hardware design-flows we developed the StateAccess command line tool that is capable to modify hardware modules that may not be designed for hardware checkpointing. The tool works at the netlist level where all types of flip-flops of a given module are identified. In the following, all or a user defined subset of flip-flops will be replaced with different types of *checkpoint flip-flops* according to the approaches presented in Section 3. The checkpoint flip-flop replacements are illustrated in Figure 5 and 6 with a gray underlied box. Next, logic for interfacing the state extraction logic to the system bus will be created and all inserted components will be connected by automatically generated signals. The result of this process is again a netlist that will be synthesized together with the remaining system. Note that our modifications ensure that during checkpointing, a module stops its operation for the simple scan chain and the memory-mapped approach in order to keep the state represented by the flip-flops consistent to a specific operation cycle. This is necessary to ensure the property of the CFSM model that checkpoint read or write operations have to be performed atomically.

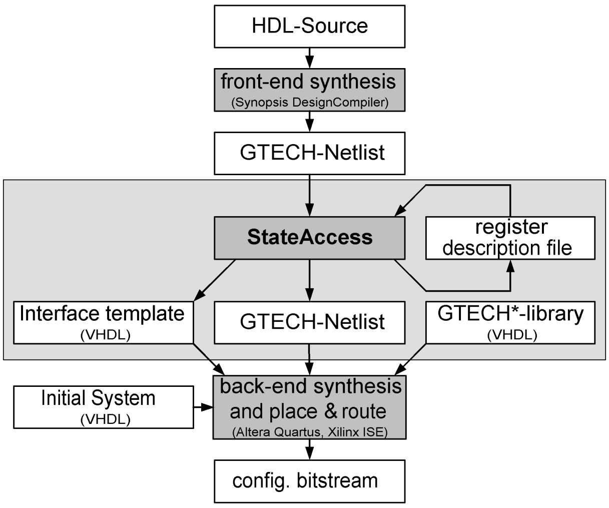


Figure 7: Modified design flow to enable HW-checkpointing with StateAccess.

The design-flow for the proposed hardware modifications is depicted in Fig. 7. It starts with a module implementation specified by a hardware description language (HDL). The module can be implemented manually or selected from a library. The HDL-specification can be simulated and verified on the functional layer regardless to the following design steps. Next, the HDL code is compiled into a netlist. For this purpose, we utilized the Synopsys Design Compiler generating an EDIF-netlist based on GTECH-primitives (Generic TECHnology).

Beside identifying all flip-flops in the netlist, StateAccess also tests if the netlist contains combinatorial feed back paths or unsupported memory elements like latches. The result of this process is stored in a register description file (RDF) listing all registers in a human readable form line by line. The order inside the RDF-file specifies the order in which the flip-flops will be accessed and stored during the state extraction phase.

The back-end synthesis step involves a library with all GTECH-primitives, the replaced *checkpoint flip-flops*, and some additional logic templates instantiated during the hardware modification process. The last step does not demand a special synthesis tool. Therefore, our approach allows to generate modules with checkpoints also in case of heterogeneous FPGA networks.

Alternatively, StateAccess can be used to save all identified registers into a file in a first step, and arranging the hardware modifications according to a user adapted register description file in a second step. With respect to our system model presented in Section 32, this possibility allows to generate hardware transformations where the checkpoint may not be saved in every state ($S' \subset S$). As a consequence of the reduced state set, it may be possible to store the state of a checkpoint (s') using less memory. In addition, the checkpoint latency L and the checkpoint overhead C can be reduced, because a smaller amount of data has to be transferred to the target CFSM.

5. EXPERIMENTAL RESULTS

This section is divided into three parts. In the following paragraph, we will present measured and estimated results according to the formulas presented in Section 3 to evaluate the presented estimations for some test cases where all flip-flops have been accessed. In the second paragraph, we will quantitatively compare these results with FPGA readback techniques. Finally, we will present an example where we examined the possibility of checkpoint optimizations. For example, we can reduce the amount of checkpoint states S_C to allow checkpointing without storing all flip-flop values.

5.1 Complete Checkpointing

The capability of checkpointing comes along with the cost of additional hardware resources and a decreased operating frequency. In the following, we will present quantitative trends on these overheads for each proposed checkpointing mechanism. In the following table, we present synthesis results achieved for a DES56 cryptographic IP core from www.opencores.org, a 16 tab FIR filter, and an FFT/IFFT coprocessor using 32 bit floating point arithmetic. We included all flip-flops found inside the unmodified module into the set of flip-flops accessed during the state extraction and restoration process. Therefore, this paragraph presents worst case results.

The first two results are based on two Altera Cyclone evaluation boards that are linked by a proprietary communication protocol allowing to send data in a rate of about 1 MB/sec. The results of the FFT coprocessor are based on a Cyclone-II device because of the huge amount of required logic. We took the original VHDL specifications and synthesized them according to our flow (Fig. 7). Note that we modify the hardware on a generic netlist level allowing to use the same hardware checkpoint for different FPGA device types.

The following table presents the synthesis results in three sections in order to indicate i) the flip-flop overhead H_{FF}, ii) the look-up table overhead H_{LUT}, and iii) the achieved clock frequencies indicating the performance reduction R. The first two sections state the synthesis results together with the overheads that were calculated by this synthesis results (H^*_{FF}, H^*_{LUT}). In addition, we present the overhead estimation according to the formulas presented in Section 3 (H_{FF}, H_{LUT}). These estimations are based on the number of flip-flops found in the unmodified module (e.g., for the DES56 module, $N = 862$). The data bus width is chosen to be $W = 32$ bits for all experiments. The values in brackets indicate the relative change in percent with respect to the unmodified module. The results point out that the estimates match the measured overheads well. In the worst case, the difference between the measured and the estimated logic overhead is 28% (found for the look-up table overhead of the DES56 module when applying the memory-mapped approach). The amount of additional LUTs for the FIR16 module is 1.44 times larger than the number of flip-flops found in the unmodified module when using a simple scan chain to access the state. This has two reasons: Firstly, the unmodified module has a high degree of fully used 4-bit LUTs (82%) preventing us to fit in a larger amount of additional logic among these existing LUTs. Secondly, the module has a very complex control structure with a huge amount of individual flip-flop enable signals.

The impact of inserting scan chains into FPGA designs on the netlist level for testing hardware modules was examined in [14]. This work focuses just on one approach similar to our simple scan chain. The results in [14] reported an average look-up table overhead of 130% that is more than our worst case result. The work does also not report on time overheads or the influence on the achievable clock frequency F_{max}.

We further measured the checkpoint overhead C in terms of clock cycles by the use of a counter that has been started when a checkpoint transfer starts and has been stopped when the transfer finishes. For the DES56 core we measured:

$$C_{MM} = 1306 \quad C_{SC} = 10354 \quad C_{SHC} = 1188$$

The test was done using the previously described system parameters. Read and write operations took about the same time and the given values represent the median of both. As expected, C_{MM} is in the range of C_{SHC}. We can use these values to determine the value τ_{CPU} from Eq. (2):

$$\tau_{CPU} = \frac{C*W}{N};\ \tau^{MM}_{CPU} = \frac{1306 \cdot 32}{928} = 45;\ \tau^{SHC}_{CPU} = \frac{1188 \cdot 32}{928} = 41.$$

Thus, the processor needs about 40 cycles for the transfer operations, pointer computations, and the loop evaluation per transferred 32-bit state value.

5.2 Comparison with FPGA Readback Techniques

We can quantitatively compare our results with an FPGA readback approach. We will consider the Xilinx VirtexII [15] architecture for this inspection. The time required to read back a configuration on present Xilinx FPGAs is related to the number of configuration columns utilized by a module and therefore its width. In the best case, when a module is constrained to a rectangle shape of minimal size and maximal height, the readback data contains 20 times more data than the data required to store just the flip-flop values if all flip-flops will be used, and selectively only the configuration columns containing register values are read back.

In the case of the DES56 core for example, this overhead ratio is in the best case 57, because not all flip-flops provided by the FPGA were used. This means that instead of reading just the 563 flip-flop values, we have to read back about 4 kilobytes of configuration data taking therefore 4000 clock cycles at the configuration interface. If we further assume that it takes just 20 additional CPU cycles to filter out the flip-flop states in order to store and transfer the checkpoint efficiently, we require at least 84000 cycles to extract the state. In this example, applying readback techniques

Table 1: Synthesis results on flip-flops (H_{FF}), look-up tables (H_{LUT}) overheads as well as the effect on maximal achievable clock rate (F_{max}) for the proposed hardware checkpointing methodologies.

		flip-flops			4-bit LUTs			F_{max} [MHz]
		synthesis	overhead		synthesis	overhead		
		results	measured	estimated	results	measured	estimated	
DES56	blank	862			1899			113
	MM	903	$H_{FF}^{MM^*}=41$ (5%)	negligible	2794	$H_{LUT}^{MM^*}=895$ (47%)	$H_{LUT}^{MM}\approx 1425$ (75%)	104 (-8%)
	SC	928	$H_{FF}^{SC^*}=66$ (8%)	$H_{FF}^{SC}\approx 47$ (5%)	2438	$H_{LUT}^{SC^*}=539$ (28%)	$H_{LUT}^{SC}\leq 862$	111 (-2%)
	SHC	1831	$H_{FF}^{SHC^*}=969$ (112%)	$H_{FF}^{SHC}\approx 894$ (103%)	3712	$H_{LUT}^{SHC^*}=1813$ (96%)	$H_{LUT}^{SHC}\approx 1756$ (93%)	94 (-17%)
FIR16	blank	563			1223			22.7
	MM	576	$H_{FF}^{MM^*}=13$ (2%)	negligible	2243	$H_{LUT}^{MM^*}=1020$ (83%)	$H_{LUT}^{MM}\approx 967$ (80%)	21.0 (-7%)
	SC	583	$H_{FF}^{SC^*}=20$ (4%)	$H_{FF}^{SC}\approx 47$ (8%)	2034	$H_{LUT}^{SC^*}=811$ (66%)	$H_{LUT}^{SC}\leq 563$	21.7 (-4%)
	SHC	1142	$H_{FF}^{SHC^*}=579$ (103%)	$H_{FF}^{SHC}\approx 595$ (106%)	2707	$H_{LUT}^{SHC^*}=1484$ (121%)	$H_{LUT}^{SHC}\approx 1158$ (95%)	21.5 (-5%)
	blank	5057			17626			27.7
FFT/	MM	5632	$H_{FF}^{MM^*}=575$ (11%)	negligible	23320	$H_{LUT}^{MM^*}=5694$ (32%)	$H_{LUT}^{MM}\approx 7945$ (45%)	25.6 (-8%)
(1024)	SC	5651	$H_{FF}^{SC^*}=594$ (12%)	$H_{FF}^{SC}\approx 53$ (1%)	18493	$H_{LUT}^{SC^*}=867$ (5%)	$H_{LUT}^{SC}\leq 17626$	26.6 (-4%)
	SHC	11242	$H_{FF}^{SHC^*}=6185$ (122%)	$H_{FF}^{SHC}\approx 5089$ (101%)	30322	$H_{LUT}^{SHC^*}=12696$ (72%)	$H_{LUT}^{SHC}\approx 10146$ (58%)	24.8 (-10%)

Table 2: Synthesis results of optimized checkpoint for the FIR16 module. Values in square brackets show the results according to a full flip-flop access (Table 1).

	flip-flops			4-bit LUTs			F_{max} [MHz]
	synthesis	overhead		synthesis	overhead		
	results	measured	estimated	results	measured	estimated	
blank	563			1223			22.7
MM	587	$H_{FF}^{MM^*}=24$ (4%) [2%]	negligible	1928	$H_{LUT}^{MM^*}=705$ (58%) [83%]	$H_{LUT}^{MM}\approx 625$ (51%)	23.2 (+2%)
SC	571	$H_{FF}^{SC^*}=8$ (1%) [4%]	$H_{FF}^{SC}\approx 45$ (8%)	1890	$H_{LUT}^{SC^*}=667$ (54%) [66%]	$H_{LUT}^{SC}\leq 360$	20.7 (-8%)
SHC	931	$H_{FF}^{SHC^*}=368$ (65%) [103%]	$H_{FF}^{SHC}\approx 392$ (70%)	2152	$H_{LUT}^{SHC^*}=929$ (76%) [121%]	$H_{LUT}^{SHC}\approx 752$ (61%)	21.3 (-6%)

will have at least a checkpoint overhead of 64 times the amount of clock cycles as required for our memory-mapped approach. Note that the readback configuration data needs to be manipulated before it can be used to reconfigure the device again, and that there is no trivial method on Xilinx FPGAs to preset just a submodule with an initial state during partial reconfiguration. Furthermore, the module demands global clock gating in order to keep the state consistent when a checkpoint rollback occurs. In [9] readback techniques are used to achive multitasking on a Virtex FPGA. It is reported that it takes 407 ms to read back the configuration (766 kilobytes), 465 ms to extract the state information and another 365 ms to confgure the device.

Altogether, we conclude that it is worth to spend some additional logic while reducing the latency overhead by at least one order of magnitude. In addition, our approach takes even more advantage when optimized checkpoint schemes are utilized as shown in the following paragraph.

5.3 Optimized Checkpointing

In order to examine the possibility of an optimized checkpoint scheme we analyzed the FIR16 module again in order to reduce the amount of flip-flops required for hardware checkpointing. We restricted the system to take checkpoints only when the module is in the `idle` state where the module is waiting for the next input value. With respect to the system model, we defined the `idle` state of the control FSM of the FIR16 module as the only state in the set of checkpoints S_c. In this case, only 360 instead of the 563 flip-flops of the unmodified FIR16 module have been found to be essential to recover from a fault. For the rest of the flip-flops, a write operation will take place before there state will be read.

It is important to determine the worst case runtime between two checkpoint states in order to guarantee that checkpoints can be stored within certain time spans and therefore limiting the maximum loss of computation. As a consequence, the system will have a predictable time behavior when a fault occurs. In the case of the FIR16 module it takes 41 clock cycles after starting a computation process before the module reaches the `idle` state again.

Table 2 displays that an optimized checkpoint scheme can massively reduce the cost to access the state of a hardware module. Furthermore, an optimized checkpoint scheme allows to reduces the checkpoint overhead time and the time to send a checkpoint to a remote node in the network.

6. CONCLUDING REMARKS

In this work, we have systematically demonstrated how concepts for checkpoints known from the software world can be applied to hardware systems modeled formally by state machines. Different approaches for accessing the flip-flop values of hardware modules have been presented. The shadow scan chain (SHC) approach allows us to take checkpoints without stopping the hardware module by the cost of a massive resource overhead. On the other side, we proposed the simple scan chain (SC) approach, where the hardware overhead was measured between 78% and 5% for the logic by an increased time to access the flip-flops in large modules. The memory-mapped (MM) approach ranks between the previous two. All this approaches have been classified and formulas estimating the hardware and time overheads have been presented. These estimates allow to evaluate the overheads stemmed from a specific hardware checkpoint technique before the final synthesis steps.

For further improvements, we proposed the concept of optimized checkpoints reducing the hardware overhead significantly. The results can be summarized into the following rules:

1 Whenever possible, try to use optimized checkpoints.

2 Use the SHC approach in systems with rapidly taken checkpoints (where P is small).

3 Use the SC method for small modules ($N < 1000$).

4 Use the MM variant in all other cases.

In order to support engineers in the development of checkpointable hardware, we presented the tool StateAccess that can be integrated into typically design-flows.

7. ACKNOWLEDGMENTS

This work is supported in part by the German Science Foundation (DFG) Priority Programm SPP 1148 “Rekonfigurierbare Rechensysteme” under grant Tel163/10-2.

8. REFERENCES

[1] Altera webpage: `http://www.altera.com`.

[2] `http://www.synopsys.com/products/test/`.

[3] A. Ahmadinia. *Optimization Algorithms for Dynamically Reconfigurable Embedded Systems*. PhD thesis, University Erlangen-Nuremberg, 2006.

[4] G. Brebner. The Swappable Logic Unit: A Paradigm for Virtual Hardware. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 77–86, Los Alamitos, CA, April 1997.

[5] A. Doumar and H. Ito. Detecting, Diagnosing, and Tolerating Faults in SRAM-Based Field Programmable Gate Arrays: a Survey. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(3):386–405, 2003.

[6] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Comput. Surv.*, 34(3), 2002.

[7] W.-J. Huang and E. J. McCluskey. Transient Errors and Rollback Recovery in LZ Compression. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, page 128, Washington, DC, USA, 2000.

[8] H. Kalte and M. Porrmann. Context Saving and Restoring for Multitasking in Reconfigurable Systems. In *15th International Conference on Field Programmable Logic and Applications*, pages 223–228, 24 - 28 Aug. 2005.

[9] W. J. Landaker, M. J. Wirthlin, and B. Hutchings. Multitasking Hardware on the SLAAC1-V Reconfigurable Computing System. In *FPL ’02: Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, pages 806–815, 2002.

[10] S. M. Scalera and J. R. Vzquez. The Design and Implementation of a Context Switching FPGA. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 78. IEEE Computer Society, 1998.

[11] H. Simmler, L. Levinson, and R. Manner. Multitasking on FPGA Coprocessors. In *Proceedings of the 10rd International Conference on Field Programmable Logic and Application (FPL’00)*, pages 121–130, 2000.

[12] C. Steiger, H. Walder, and M. Platzner. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-time Tasks. *IEEE Transactions on Computers*, 53(11):1392–1407, November 2004.

[13] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM ’97)*, page 22. IEEE Computer Society, 1997.

[14] T. Wheeler, P. Graham, B. E. Nelson, and B. Hutchings. Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification. In *Proceedings of the 11th International Conference on Field Programmable Logic and Application (FPL’01)*, pages 483–492, 2001.

[15] Xilinx webpage: `http://www.xilinx.com`.

[16] W. Xu, R. Ramanarayanan, and R. Tessier. Adaptive Fault Recovery for Networked Reconfigurable Systems. In *FCCM ’03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 143, Washington, DC, USA, 2003. IEEE Computer Society.