Generating Eff cient Context-Switch Capable Circuits Through Autonomous Design Flow

ALBAN BOURGE, Univ. Grenoble Alpes, CNRS, TIMA F-38000
OLIVIER MULLER, Univ. Grenoble Alpes, CNRS, TIMA F-38000
FREDERIC ROUSSEAU, Univ. Grenoble Alpes, CNRS, TIMA F-38000

Commercial Off-the-Shelf (COTS) FPGAs are becoming increasingly powerful. In addition to their huge hardware resources, they are also integrated into complete systems on chips (SOCs), e.g. in the latest Xilinx Zynq or Altera Stratix platforms. However, cooperation between FPGAs and their surroundings, and the f exibility of hardware task management could still be improved. For instance, mechanisms have yet to be automated to allow multi-user approaches. A reconf gurable resource can be shared between applications or users only if it has a context-switch ability allowing applications to be paused and resumed in response to system demands. Here, we present a High-Level Synthesis (HLS) design f ow producing a context-switch-capable circuit. The design f ow manipulates the intermediate representation of a HLS tool to build the context extraction mechanism and to optimize performance for the circuit produced. The method is based on eff cient checkpoint selection and insertion of a powerful scan-chain into the initial circuit. This scan-chain can extract f ip-f ops or memory content. Experiments with the system produced show that it has a low hardware overhead for many benchmark applications, and that the hardware added has a negligible impact on application performance. Comparison with current standard methods highlights the eff ciency of our contributions.

CCS Concepts: •**Computer systems organization** → **Reconf gurable computing;** •**Hardware** → High-level and register-transfer level synthesis; Hardware accelerators;

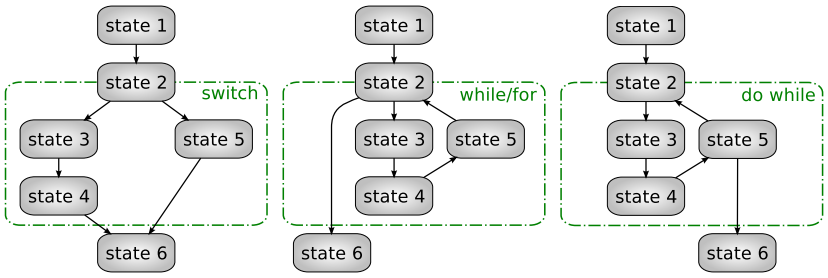
Additional Key Words and Phrases: Context-switch on FPGA, Checkpointing, Partial scan-chain

ACM Reference Format:
Alban Bourge, Olivier Muller and Frédéric Rousseau 2015. Generating Eff cient Context-Switch Capable Circuits Through Autonomous Design Flow. ACM Trans. Reconf g. Technol. Syst. 10, 1, Article 9 (December 2016), 23 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Over the past decade, FPGAs have been increasingly frequently viewed as a robust solution for hardware acceleration and the f eld of reconf gurable computing has undergone many technical changes and improvements [Hauck and DeHon 2010], but the path to even higher-performance FPGA applications is broad. In this paper, we describe a system to build toward more f exible management of hardware tasks running on FPGAs by providing a context-switch capacity or allowing for a preemptive scheduling paradigm in hardware applications. The potential benef ts of this technique have previously been described [Trimberger et al. 1997; Scalera and Vazquez 1998; Guan et al. 2008]. Context-switch-capable hardware could be more readily integrated into a complete system and would provide powerful features if combined with other techniques such as reconf guration [Papadimitriou et al. 2011] or task relocation and FPGA

contact: alban.bourge@imag.fr, olivier.muller@imag.fr, frederic.rousseau@imag.fr
Univ. Grenoble Alpes, TIMA F-38000 CNRS, TIMA F-38000 46, avenue Felix Viallet Grenoble, France



(a) Switch (b) For or while loop (c) Do-while loop

Fig. 1: Typical Finite State Machines

defragmentation [Fekete et al. 2012]. More generally, a context-switch-capable circuit would be compatible with multi-user approaches. An ideal hardware context-switch mechanism fulfills the following requirements:

— ensures a preemption within a given latency;
— is FPGA-technology independent;
— its implementation and usage is effortless for designers and users;
— the hardware overhead is low;
— the hardware overhead has negligible impact on the circuits performance;
— the impact on the whole system is low.

This paper presents a High-Level Synthesis (HLS) flow for the design of a context-switch-capable circuit, taking the previous requirements into consideration. The paper is organized as follows: Sections 2 and 3 present the background, definitions and work related to the solution presented; Section 4 describes the proposed method; Sections 5 and 6 explain the two steps in our method in more detail; multiple experiments to determine the advantages and disadvantages of the solution are presented in Section 7. This paper ends with a discussion section and a conclusion summarizing the contributions made.

2. BACKGROUND AND DEFINITIONS

2.1. System

Throughout the paper, we will consider a generic system made up of two main parts: 1) a control part responsible for managing the whole system. It could be as simple as a CPU running an operating system. 2) the reconfigurable resource, connected to a communication bus and behaving as a slave to the control part. The reconfigurable resource is considered to be a Commercial Off-the-Shelf (COTS) FPGA. This resource will receive context-switch requests from the CPU. In addition to these two central elements, other peripherals can also be present, such as memories.

2.2. Hardware Task

It is assumed that a hardware task can be represented by a Finite State Machine (FSM) comprised of states, and a datapath. In this classical (FSM/datapath) model, the hardware task is controlled by the FSM. Figure 1 shows various types of small FSM. Each state performs actions carried out in the datapath. This can be, for example reading a variable, writing a variable, or computing. For instance, on Figure 1a, state 2 could represent an *if* instruction, i.e., the test of a condition.

2.3. Hardware Context

One of our hypotheses is that the initial bitstream is not considered to be part of the task context. Indeed, the controlling system knows which bitstream in memory corresponds to the current task running. This initial configuration is assumed to be present in memory and therefore it is unnecessary to save it as part of context-switching. Another hypotheses concern the data that are currently being transferred to or from the hardware task. These data are not part of the context and the controlling system has to manage them. On the other hand, the content of the memory elements of the running circuit will change during task execution. These changes are not predictable and for this reason, the memory elements that have changed must be stored. More precisely, each state has a set of live variables, i.e., variables accessed (read or written) during the state actions or in subsequent states. When a variable is not yet or no longer used in a state, it can be removed from the context. Considering these hypotheses, the context of a task consists in a set of live variables related to the current state of the task.

2.4. Hardware Checkpoint Definition

The notion of checkpointing is classically used in fault-tolerant applications [Elnozahy et al. 2002; Egwutuoha et al. 2013]. Indeed, in the field of reliability, numerous articles on checkpointing have been published. For the purposes of this paper, checkpoints applied to hardware designs are most relevant. [Koch et al. 2007] is a good example of how software checkpointing techniques can be applied to hardware circuits. Some uses of checkpointing have been developed to create simple backup images of hardware tasks [Landaker et al. 2002]. Another possibility is to use checkpoints to add soft error mitigation, as in [Asadi and Tahoori 2005]. A more specific use is presented by Reorda et al. [2009], who specifically target soft cores by performing periodic checkpointing operations. Schmidt et al. [2011] present a complete monitoring infrastructure, dealing with multiple FPGAs.

In all these cases, the application context is saved each time a hardware checkpoint is reached. Whenever a fault occurs, checkpoint-based protocols can restore the systems state from the previously saved hardware checkpoint. From this process we can imagine how context-switch operations can be made possible. For example, the context of a hardware task could be saved when the next hardware checkpoint for the application is reached. Context restoration would then restart the task where it was stopped by using the previously saved hardware checkpoint. We define a hardware checkpoint as follows: a task state where context-switch operations are allowed.

2.5. Hardware Context-Switch

Figure 2 presents an example of a hardware context-switching process. Two preemption demands are shown: the first one switches task 1 for task 2 and the second switches back to task 1. Task 1 preemption is divided into two steps. The time needed for the task to reach a checkpoint is the first step. It is represented by the *cp* stage (total time t_c). When a checkpoint is reached, the context of the task is saved during the *save* step (duration: t_s); t_s varies depending on the size of the context to be extracted. The save step is followed by configuration of the chip to launch task 2, which is preempted in its turn. Finally, when the system switches back to task 1, a restoration step is necessary (*rest*) so as to re-write the previous context before re-launching the first task.

3. RELATED WORK

Context extraction methods for hardware tasks already exist but were mainly developed in a fault-tolerant perspective. For example, in 1999 [Fujii et al. 1999] used a specific hardware resource, i.e., not a typical FPGA chip. Specific FPGA based architectures

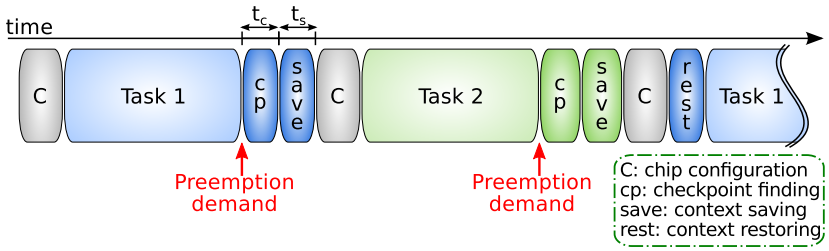


Fig. 2: Timeline for hardware context-switching

such as virtual CGRAs [Heyse et al. 2013], overlays [Brant and Lemieux 2012] or reconfigurable processors may offers facility for content extraction. Given that the ideal mechanism described in Section 1 aims low overheads, the usage of such architectures is not chosen. The proposed mechanism must hence use generic FPGA content extraction methods. Nowadays, two main families stand out to address this issue.

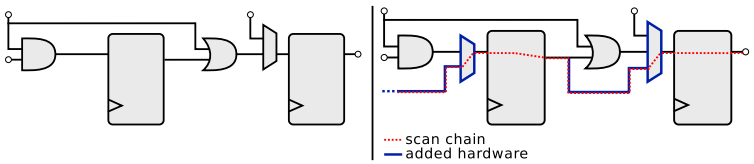
3.1. FPGA-Specific Extraction Technique

The first method consists in extracting the condition of the FPGA components (LUT, FF, routing) through the configuration mechanism. This is called a *readback* method as the configuration, hence the flip-flop values, are literally read and transferred back to the controlling system. This configuration is commonly handled by the Internal Configuration Access Port (ICAP) in Virtex technologies [Xilinx 2010]. The ICAP allows configuration (and readback) from within the FPGA. Blodget et al. [2003] and Ullmann et al. [2004] used that port for their research. Similarly, Sedcole et al. [2006] compared two reconfiguration methods and presented the use of the ICAP port to read back the FPGA configuration.

Readback is the most commonly used technique. It is difficult to set up because it is based on non released information of FPGA family bitstream encoding. It has the major drawback of extracting a configuration bitstream from the chip, making its data efficiency very poor. Indeed, much of the extracted data is not required as it is already present in the initial bitstream. Hence, the data footprint of the method, i.e., the amount of memory needed to store a tasks context, is considered high. This fact is highlighted by Kalte and Porrmann [2005], who indicate that less than 8% of the data in the readback stream represent hardware context. A system to filter out unnecessary data was therefore proposed by Simmler et al. [2000] and Levinson et al. [2000]. With these methods, the bitstream is filtered offline to store only information relating to registers and RAM. In an alternative approach, Kalte and Porrmann [2005] reduced the amount of readback data by only reading back portions of the FPGA that are used.

3.2. Task-Specific Extraction Technique

The principle of task-specific extraction techniques is to directly add some structures to the circuit to allow context extraction. In other words, the circuit embeds the mechanisms that allow context reading and writing. The most common mechanism is a *scan-chain*: a serial link one or more bits wide between each flip-flop of the circuit. A multiplexor is added at the input of each flip-flop (or another input is added to an existing multiplexor) to route each flip-flop signal to the next. The last flip-flop of the chain acts as an output. An example of a 1-bit scan-chain insertion is shown in Figure 3. This method has a smaller memory footprint than the readback method because only FF values are extracted. The extraction time is also reduced compared to the readback method. On the other hand, the extra design efforts required to add such structures



(a) Original datapath (b) Datapath and scan-chain

Fig. 3: Insertion of scan-chain into a simple circuit

Table I: Comparison of readback and full scan-chain methods

	Readback	Full scan-chain
Data footprint	Large	Moderate
Extra design efforts	No	Yes
Extra resource consumption	No	Yes
Technology independent	No	Possibly

can be costly [Koch et al. 2007]. Finally, the area overhead when such a structure is integrated is not negligible.

Other related works include Scalera and Vazquez [1998], describing a specific architecture extensively modifying the memory elements and adding an extraction structure. This architecture requires considerable design effort, time and increases the area overhead. Wheeler et al. [2001] present a typical approach to insert scan-chains, while Koch et al. [2007] describe three access methods (scan-chain, shadow scan-chain and memory mapped methods) and present a tool named StateAccess allowing automatic state extraction. The introduction of such a tool is clearly of interest to reduce the additional design efforts. The interesting notion of *switching point* was introduced by Mignolet et al. [2003] to limit the number of authorized context-switch states, but no selection method was proposed. Eventually, the specific problem of memories (cf. Section 6.1) has never been addressed.

3.3. Comparison of readback and scan-chain methods

Table I summaries the main characteristics of both methods. It can be pointed out that they have complementary advantages and drawbacks. Thus, the readback method cannot extract a completely relevant context and therefore has a large data footprint. By adding a filtering step, the data footprint can be reduced, becoming close to that of task-specific techniques. On the contrary, a scan-chain of all memory elements (full scan-chain) reduces the amount of data to be extracted. The readback method also relies on a feature of the FPGA fabric, and thus developing an application requires no particular design efforts. Similarly, no extra resources are consumed within the FPGA. In contrast, with a full scan-chain approach, the developer must modify the circuit and use otherwise free hardware resources. It should be noted that with a scan-chain method, the data footprint and the extra resources consumed are almost directly linked. Indeed, if the number of memory elements that must be extracted increases or decreases, hardware must be added or removed in parallel. Scan-chains also prevent some low level optimizations such as retiming and register duplication. They can also have an impact on the usage of vendor-specific IPs (e.g. they prevent merging register in DSPs due to multiplexor addition) and on the place and route steps. Finally, a readback approach is technology dependent (the bitstream format is not the same for different FPGAs). This may not be the case for a scan-chain if written in platform-independent HDL or RTL.

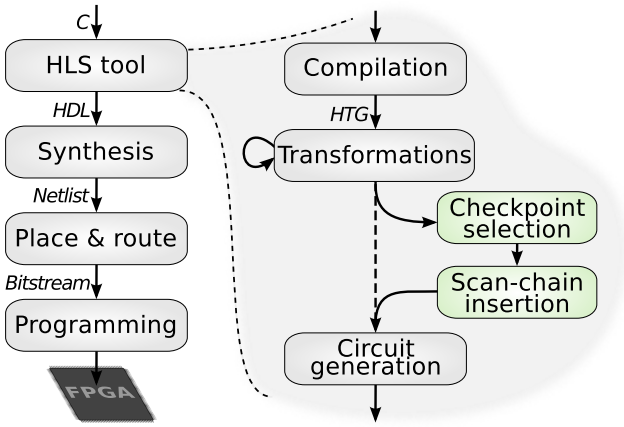


Fig. 4: Proposed design flow

Considering the ideal mechanism described in Section 1, a scan-chain based approach was chosen for this contribution.

4. METHOD OVERVIEW

In this paragraph, the two main contributions of this paper are first presented as two steps of a design flow. The consequences and the associated optimization objectives are then highlighted.

4.1. Proposed Design Flow

HLS allows to design hardware circuits using a high-level language such as C or C++. These languages enable a more concise description of algorithms compared to HDL and aim to reduce development time without neglecting performance [Liang et al. 2012].

The proposed design flow consists of two main steps corresponding to our two contributions. These two steps are clearly separate but do influence each other. The steps were added to an existing HLS flow to produce a circuit capable of context-switching while respecting design- and user-defined constraints. Figure 4 illustrates their positions within the design flow. Inside the HLS tool, the typical operations are a compilation step, providing an intermediate representation of the circuit (Hierarchical Task Graph, or HTG [Girkar and Polychronopoulos 1994]), followed by one or more transformations. The aim of these transformations is to improve the circuits performance by adapting it to the hardware targeted (e.g. unrolling a loop if sufficient resources are available). Finally, the HDL representation of the circuit is generated. The operations added to this flow were a checkpoint selection step and insertion of a scan-chain. By combining these steps, the circuit became capable of context-switching. These two contributions will be described in more detail below.

4.2. Impact on HLS

Introducing a context-switch capacity to a circuit at a high-level of abstraction is not without consequences. For instance, intermediate representation and optimization routines of HLS tools have a significant impact on the applications final architecture.

One drawback of our flow is a lack of control over the circuit that will ultimately be produced. Indeed, the HLS tool has no control over the following steps, which are performed by back-end tools (logic synthesis and place and route). These tools handle the addition mechanism with varying degrees of success.

On the positive side, the HTG obtained after compilation contains useful information thanks to the modern parsers and compilers. For instance, these tools can analyze the liveness of variables in the circuit, a task which is difficult to perform at a hardware description level. This analysis is central to the checkpoint selection step. It also allows abstraction of the memory elements whatever the underlying implementations (flip-flop, BRAM or LUTRAM). As proposed by Wheeler et al. [2001], it is possible to link each memory element in a scan-chain while allowing the addition of specific mechanism for RAMs. Furthermore, HLS removes the need to build the mechanism by hand. A typical scan-chain insertion occurs at the HDL or netlist level and if no automation tool is used it requires the intervention of the application developer. In this proposition, the method adopted requires no manual efforts. Finally, as the mechanism is written in HDL (by the HLS tool), the developer can choose any back-end tools and type of FPGA. The code produced will be portable, and thus a stored context can be used on any context-switching-capable hardware kernel of the same type (i.e., a context extracted from the same application), no matter which FPGA or back-end tools were used.

4.3. Global Optimization Objectives

Adding the context-switch capacity to a hardware task comes at a cost. In the first place, the preemption scheme requires the task context to be saved, which can represent a considerable amount of data. Transfer of these data has a triple impact on the surrounding system: the systems main memory must have an adequate capacity, bus congestion can occur depending on how many peripherals are present, and finally, the availability of the hardware task varies directly with the time needed to extract the whole context. In addition, the extraction capacity will naturally also impact the task itself. Adding the mechanism inside the HDL application results in a hardware overhead and may alter overall performance.

In line with the ideal mechanism described in the introduction, our proposal will have to deal with three constraints and three optimization objectives. Each of these points will be addressed by a particular step of the method (checkpointing or scan-chain insertion) or by the HLS flow itself. These points are summarized in Table II. The first constraint, namely staying within a certain latency when context-switching, is very important in a preemptive multitasking system. Switching can require a considerable number of cycles and the system schedulers adaptability will be improved with a bounded extraction time [Buttazzo et al. 2013]. The checkpointing algorithm must ensure that the switching process does not violate this latency constraint. The HLS-based design flow ensures the codes portability and removes any design efforts, as indicated in Section 4.2. Our optimization objectives were actually minimization objectives. Indeed, the method should ideally produce a context-switching circuit with no additional impact on the overall system, no hardware overhead and no impact on application performance (frequency, latency). Obviously, it is not possible to have it both ways especially with a scan-chain based mechanism. The two steps in our method – checkpointing and scan-chain insertion – tries to cover the three optimization objectives.

5. CHECKPOINT SELECTION

As mentioned above, checkpoint selection is the first step in our method. The main ambition of this step is to ensure the context-switch process will be bounded in time. This issue is addressed in this section through a mathematical definition which is resolved algorithmically.

5.1. General Idea

Context-switching requires extraction of the context for the currently running hardware task. This context consists in the set of *live* variables for the current FSM state. For

Table II: Constraints and optimization objectives of the problem

	Type	Addressed by
Constraints	respect system context-switch latency	checkpointing
	code portability	HLS flow
	no design efforts/automatic method	HLS flow
Optimization objectives	impact on whole system (mainly communication overhead)	checkpointing + scan-chains
	hardware overhead	checkpointing + scan-chains
	impact on application performances	checkpointing + scan-chains

a particular state, a live variable is a variable that has been written and will be read later as the task proceeds. A variable which has never been written is not live as its content is stored in the initial bitstream. A variable which is no longer read after a certain state is not live either as its value is no longer useful. Establishing the context only with live variables creates a smaller context than one containing all the variables. Another selection step makes it possible to further reduce the context size: checkpoint selection. During task execution, the volume of live variables varies. The objective is to select moments, or states of the hardware task, to allow extraction of the hardware context within the given context-switch latency. The task will continue running until reaching such state when a context extraction is wanted. The smaller the live variable volume is in a checkpoint, the more the task can run until reaching this checkpoint as the extraction of this state is faster than the others. The checkpoint selection algorithm ensures that in any state, the context can be extracted in a given latency and tries to minimize the previously mentioned objectives.

5.2. Mathematical Definition

When a preemption demand arises in a random state, a checkpoint has to be reached and the hardware context extracted. The corresponding timings are t_c and t_s, respectively, as stated in Section 2.5. The total timing must respect the system-wide context-switch latency, called t_{lat}: $t_c + t_s < t_{lat}$. In this case, it is said that the checkpoint covers the states crossed during the checkpoint search.

Let n be the number of states of a hardware task. A Boolean n-vector can represent a set of states for the task by assigning one state per vector component. The state is considered in the set only if its corresponding component equals 1.

Let x be a Boolean n-vector describing a set of hardware checkpoints. Let $A = (a_{ij})$ be a Boolean $n \times n$ matrix representing the covering matrix. The Boolean coefficient a_{ij} equals 1 when state j covers state i. Note that Ax is an integer n-vector giving the number of checkpoints defined in x that cover each state. x must ensure that for each state of the task, the context-switch will be possible, i.e., each state is covered by at least one checkpoint. This constraint can be represented by the equation:

$$Ax \geq \mathbf{1}$$

where $\mathbf{1}$ is the integer n-vector with all components equal to 1.

The optimization objective associated with the checkpoint selection problem is the minimization of the area overhead for the mechanism. Let us assume that the area overhead for a scan-chain is proportional to its length in bits. As the live variables of a hardware checkpoint are mapped in memory elements, their cumulative length determines the size of the scan-chain. Let c_i be the cost of state i as a checkpoint (the length in bits of the associated scan-chain). A simple estimation of the area overhead for the whole mechanism is the sum of the cost of all the checkpoints. Thus, the minimization equation is:

$$\min \sum_{j=0}^{n-1} c_j x_j$$

This type of "set covering problem" is a well known mathematical problem. It is an NP-complete problem which was first described by Cormen et al. [2001].

A closer look at this case reveals the minimization equation to be more complex than the usual one. This complexity is due to the fact that some memory elements can be part of several scan-chains. With the simple estimation presented above, these memory elements will be counted several times, whereas they should only be counted once since the scan-chain insertion step benefits from this overlapping. To obtain a more reliable estimation of the area overhead, a polynomial rather than a linear minimization equation should be applied. A more precise estimation of the optimization objective can be written as:

$$\min \sum_{j=0}^{2^n-1} c'_j x'_j$$

where j is the n-bit index of a set of states following the natural order (e.g. for $j = 9 = 2^0 + 2^3$ the set gathers the states 0 and 3), c'_j is the cost of set j (i.e., the length in bits of only the memory elements that are live in the states of set j), x'_j is a Boolean variable set if the intersection between the sets of states indexed by j and depicted by x is not null.

It should be noted that the number of terms in this formula grows exponentially with n. To cope with this complexity, we will use a heuristic. This checkpointing problem is divided into the two parts addressed in this article. First, a static analysis of the FSM is performed to identify the A matrix of the input hardware task. Secondly, heuristics are used to rapidly resolve the optimization problem using the A matrix.

5.3. Static Analysis

The first step consists in analysis of the FSM of the application. This analysis is described in Algorithm 1 and corresponds to the computation of the A matrix.

The coverage of state j, named S_j, corresponds to the set of states that state j can cover. S_j is the subset of $\{0, 1..., n-1\}$ such that $S_j = \{i | a_{ij} = 1\}$. Mathematically, S_j corresponds to the j^{th} column of A. Practically, this means that state j can be reached and its context extracted in less than t_{lat} from anywhere within S_j. The set of S_j or the A matrix, is computed with a recursive algorithm based on t_{lat} and every state time, t_s, needed to extract its associated context.

An iteration is performed on every state to compute its corresponding coverage. If the context of state j can be extracted within t_{lat}, then state js coverage will be computed. Otherwise, j is removed from the potential checkpoints (line 5). Checking coverage for state j is therefore done recursively using the *check_coverage* procedure defined on line 16. Each state potentially leading to state j is analyzed to check if coverage is possible (line 20). This analysis will be further described below. If coverage is possible, Dijsktra-style variables are set and/or checked. If the previous state, k, has already been visited, the search continues only if the path taken is shorter than the previous one, i.e., t_l is longer. Otherwise, $k_{visited}$ is set to **true**, t_{best} is set to t_l and state k is added to S_j.

The following textual description of the instruction on line 20 is provided since its pseudo-code description is broad and difficult to read. The main problem at this stage is to solve complex paths existing between states. Figure 5 illustrates how a switch is covered, considering that each state lasts one clock cycle, $t_{lat} = 12$ cycles and state 7 has 8 bits which must be extracted (i.e., $ts = 8$ cycles with a 1-bit scan-chain). If the $t_c + t_s < t_{lat}$ constraint also has to be met, the time left to reach state 7 is $t_l = t_{lat} - ts$, in the example this gives $t_l = 4$ clock cycles. The recursion for computing S_7 hence starts

Algorithm 1 Static FSM analysis

```
1: for j ∈ {0, 1 ..., n-1} do                          ▷ Iterate through all FSM states
2:     is_cp ← true                                    ▷ initially, state j is a checkpoint
3:     COMPUTE(t_s)                                    ▷ time to save state j
4:     if t_s > t_lat then
5:         is_cp ← false        ▷ context of state j too large to be extracted within t_lat
6:         S_j ← ∅                                     ▷ no states covered by state j
7:         continue
8:     else
9:         t_l ← t_lat − t_s                ▷ time left for coverage i.e., max value of t_c
10:        S_j ← {j}                                   ▷ state j covers itself
11:    end if
12:    if t_l > 0 then
13:        CHECK_COVERAGE(j,t_l)                       ▷ recursive procedure filling S_j
14:    end if
15: end for
16: procedure CHECK_COVERAGE(j,t_l)
17:    static k_visited                                ▷ Dijkstra-style implementation
18:    static t_best                                   ▷ Dijkstra-style implementation
19:    for all k, previous state of state j do
20:        if j is able to cover k then      ▷ depending on t_l and graph relationships
21:            t_l = t_l − 1                           ▷ decrementing time left
22:            if k_visited = false then
23:                k_visited = true
24:                t_best = t_l
25:                S_j ← S_j ∪ {k}
26:                if t_l > 0 then
27:                    CHECK_COVERAGE(k,t_l)           ▷ time is left to explore the graph
28:                end if
29:            else if k_visited = true and t_l > t_best then
30:                t_best = t_l                ▷ k already visited but current t_l is greater
31:                CHECK_COVERAGE(k,t_l)
32:            end if
33:        end if
34:    end for
35: end procedure
```

from state 7 with 4 cycles left. Step ① tests state 6 with the time remaining (4 cycles). The result of the test is positive (i.e., 7 covers 6) because the remaining latency is not null. Step ②, the first recursive call to *check_coverage* on line 28, tests the previous states of 6, starting with state 5. As the latency left is positive, state 5 is also covered. Step ③ is not completed because state 2 cannot be covered if all the switch branches starting from it are not covered. Hence, the recursion starts back at state 6 with step ④. The second branch of the switch is then covered in its turn, step ⑥ is completed and state 2 is covered. State 1 is not covered because no latency is left for step ⑦. The result of the coverage computed gives $S_7 = \{2, 3, 4, 5, 6, 7\}$.

The coverage analysis handles other complex situations in the same fashion. For example, coverage limitations of a switch branch state (e.g. state 4 cannot cover state 2 or state 1 in Figure 1a), coverage of a loop by a subsequent outer state (e.g. does

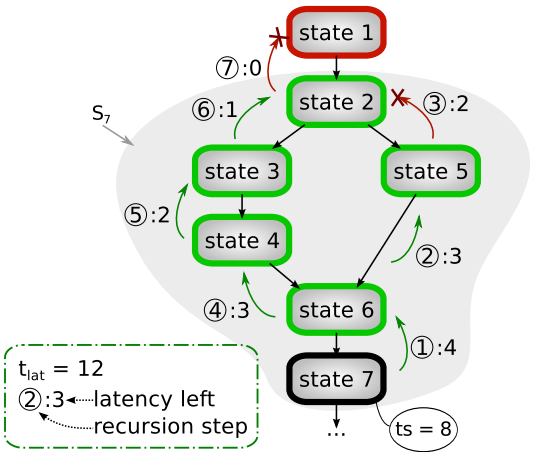


Fig. 5: Example of coverage computation for state 7

state 6 cover the entire loop in Figures 1b and 1c?), coverage of nested loops and mixed situations.

Note that to obtain coverage of a loop, the static analysis depends on information stored in the HTG. More precisely, the loop duration – derived from the loop size and the loop iteration number – must be known. An outer state cannot cover any states contained within a loop with a dynamic iteration number. Thus, loops with dynamic iteration numbers are not covered by any outer state. This is a worst-case scenario.

5.4. Greedy Heuristic

Given the complexity of the minimization problem (NP-complete) a greedy heuristic was chosen to search the set of checkpoints. Chvatal [1979] proposed a greedy algorithm to solve the set covering problem with a linear minimization objective. This algorithm has an $O(n^2)$ complexity. It also ensures that the solution obtained is close to the minimal solution in an *a priori* defined factor. This algorithm relies on the mean cost of a variable x_j defined as $c_j/|S_j|$. In this case, $|S_j|$ is the number of states covered by state j, and c_j is the cost of this state (as defined in Section 5.2). Iteratively, this algorithm sets the variable x_j to 1 with the minimal mean cost until the problem has been completely covered. At each iteration, the mean costs of variables are updated since some states are already covered by previous iterations (i.e., the $|S_j|$ terms change).

The heuristic presented here proposes a small modification to adapt to the polynomial optimization objective. The updates of the mean costs also take into account that the c_j are reduced. Indeed, variables set at previous iterations share some costs (c'_j in the complex formulation) with unset variables. Those shared costs need not be counted in subsequent iterations.

The greedy heuristic presented in Algorithm 2 begins with an initialization of different variables (R, M and x^*). Then the state k giving the minimal mean cost is chosen provided all the states are not covered (line 4) and all the variables are updated. Note that the cost is computed with the γ function giving the number of bits for a set of memory elements. M_k represents the set of memory elements associated with state k. In the algorithm, M_k is a constant which is provided by the HLS tool after its mapping step. At the first iteration, $\gamma(M_k \backslash M)$ gives c_k, like in the linear problem, as $M = \emptyset$. The mean cost is also updated with the states which have already been covered (R being removed from S_i, which corresponds to the $|S_i \backslash R|$ part of the equation).

Algorithm 2 Greedy heuristic for the set covering problem

1: $R \leftarrow \emptyset$ ▷ set of states covered
2: $M \leftarrow \emptyset$ ▷ set of memory elements covered
3: $x_j^* \leftarrow 0$ for $j = 0, ..., n-1$ ▷ vector describing the set of checkpoints
4: **while** $|R| < n$ **do**
5: Choose k giving $\frac{\gamma(M_k \backslash M)}{|S_k \backslash R|} = \min_{i=\{0...n-1\}} \frac{\gamma(M_i \backslash M)}{|S_i \backslash R|}$
6: $x_k^* \leftarrow 1$
7: $R \leftarrow R \cup S_k$
8: $M \leftarrow M \cup M_k$
9: **end while**

The solution is x^*, the n-vector representing the set of checkpoints. At the end of the heuristic a set of checkpoints is obtained reducing the area overhead of the partial extraction mechanism and ensuring the task can reach a checkpoint within time t_{lat}. Though non-optimal by nature, experimental results showed that the heuristic gives good results.

6. SCAN-CHAIN INSERTION

After the checkpoint selection, a scan-chain based mechanism is inserted to enable data extraction from the circuit. This paragraph describes our approach more precisely.

6.1. Partial Extraction Mechanism

Scan-chain insertion consists of adding hardware resources (mainly multiplexors and routing) to a circuit. Our objective was to reduce the potential impact of this addition mainly by adopting techniques from the test field. Assumptions in this field are quite different, but some ideas are nevertheless applicable. For instance, Touba [2006] inspired the idea of using partial scan-chains derived from scan trees.

The data to be extracted are given by the checkpoint selection described above. Each checkpoint has a precise set of live variables. Instead of creating a large scan-chain concatenating every variable, one *partial* scan-chain (PSC) will be created for each checkpoint with the same context. In other words, checkpoints with the same set of live variables trivially share the same mechanism. Such checkpoints with the same context are grouped as *scan families*. The number of scan families represents the actual number of PSCs in the circuit.

At this point, we have to oppose two technologically different types of live variables. Indeed, a variable which is accessed through an address (i.e., inside a BRAM or a LUTRAM) cannot be put in the same scan-chain as a variable in a register. This means the memories will not be inserted as is into a PSC but into a specific scan-chain. The mechanism is quite simple and consists in introducing memories directly into a dedicated scan-chain, called a *memory* scan-chain (MSC). We must therefore differentiate between the register-related context and the memory-related context. Note that no analysis of the liveness of memory elements is considered in this study, although a basic routine was run to remove ROMs from the memory-related context. Otherwise, every memory element will be part of the context.

The whole mechanism consisting of PSCs and MSCs on the datapath side of the circuit, and a scan control system on the FSM side, will later be called the partial extraction mechanism. The following sections describe the scan control system and the datapath modifications made to the circuit.

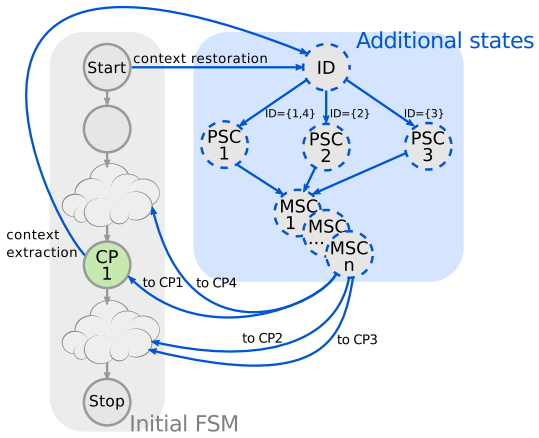


Fig. 6: Adding the scan control system

6.2. Scan Control System

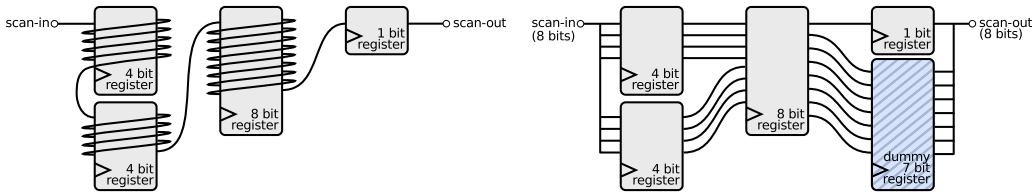
To make context extraction or restoration possible, the FSM needs some additional states. The scan control system is composed of three main parts: the identification state and the PSC and MSC activation states. These additional states allow the circuit to enter a context-switching mode when it reaches a checkpoint. They are depicted in Figure 6 (right-hand panel).

The identification state (`ID`) allows identification of the checkpoint considered (`CP`) whenever a context is to be restored or saved. It reads the value of a particular register in the datapath containing the checkpoints identification. This register is set to the identification of the current checkpoint when context extraction is required. Additionally, the register is updated before a restoration so that the correct checkpoint context will be restored and the correct restarting state attained.

A PSC, i.e., a scan-chain associated with a particular scan family, is activated during a state represented by the `PSC` prefix. In this state, all the multiplexors linking the different elements of the PSC are driven in order to extract the context from a checkpoint for a given scan family. Then, the set of MSCs are activated individually to extract the memory content. The `PSC` and `MSC` states are used both for extraction and restoration. Hence, the states following an extraction or a restoration will be the same, namely `ID` then `PSC#` then the set of `MSC`s, `#` being the scan family number for the checkpoint. In this example, checkpoints 1 and 4 are both part of scan family 1, for which the PSC is `PSC1`. Eventually, the circuit will jump to the restored checkpoint.

6.3. Scanning Registers

There are different ways of embedding registers in scan-chains, these methods are more or less resource intensive and vary in terms of power. Our proposal includes two types of links between registers. Both techniques have advantages, and their respective results are presented in the experiment section. The two schemes are illustrated in Figure 7. Only the link created is represented in the diagram, i.e., any additional multiplexors are omitted. The first method is the naive scan-chain, which is a serial one-bit-wide link between each register (Figure 7a). The main advantage of this method is its simplicity, and the minimal hardware it requires. However, it requires a large number of clock cycles to extract the whole context. A method based on parallelization of the scan-chain can be used to improve the extraction latency. Hamzaoglu and Patel [1999] define a parallel serial scan technique, which is similar to the solution proposed here, although



(a) 1-bit-wide scan-chain (b) 8-bit-wide scan-chain

Fig. 7: Scan-chain types

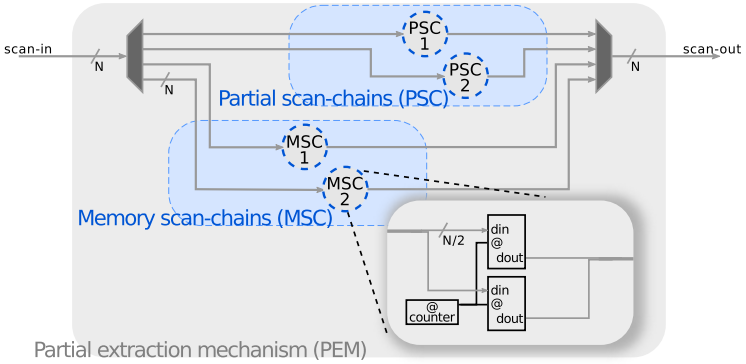


Fig. 8: Partial Extraction Mechanism (datapath), focusing on memories

it is more complex. The technique serializes whole registers, i.e., the first bit of register 1 is linked to the first bit of register 2, bit n of register 1 is linked to bit n of register 2, etc. This eventually gives multiple parallel one-bit-wide scan-chains, as illustrated in Figure 7b for an eight-bit-wide scan-chain. This type of chain is briefly but partially described and no anlysis is given in [Koch et al. 2007]. Note that two 4-bit registers can be parallelized or stacked to further improve latency. On the downside, when a register does not fill the entire bus, dummy registers must be added (final register Figure 7b). This is a major cause of overhead compared with the one-bit-wide scan-chain.

6.4. Scanning Memories

The final datapath modification consists in extracting memory content. To do this, MSCs are introduced into the circuit. Figure 8 describes the architecture of the partial extraction mechanism from the datapath side and focuses on a particular MSC.

MSCs are unrelated to PSCs. When a PSC corresponding to a checkpoint is used (to extract or restore register-related context), all the MSCs are activated one by one to extract memory-related context. Parallelization reduces the number of MSCs. For instance, in MSC 2, two memories are stacked to fit the bus width (e.g. two 32-bit word-length memories can fit in a 64-bit-wide scan-chain). Otherwise, an isolated memory will be present on its scan-chain. This isolation will cause fragmentation of the extracted data if a memory has a shorter word-length than the scan-chains width and if it cannot be stacked with another memory to fill the entire width. In order to route the memories input and output to the scanning interface, i.e., *scan-in* and *scan-out*, multiplexors are added in the same fashion as for registers (see Figure 3b). This may lead to a reduced operational frequency if the multiplexors are added in the critical path.

7. EXPERIMENTS

This section describes the different results obtained by implementing our proposal. The implementation is presented followed by the results of the two steps presented above (checkpointing and insertion of a partial extraction mechanism). Two experiments were performed on the register and memory extraction mechanisms, respectively. Both highlight the necessary trade-offs that had to be made with respect to the internal fragmentation of the scan-chain method. Finally, the hardware overhead and the impact on application performance are presented.

7.1. Implementation: CP3 Tool

An implementation of the method described was done in a tool named CP3 [Bourge et al. 2015], standing for CheckPoint PinPoint. This tool is written in C as a plugin for the existing HLS tool AUGH. AUGH is a free and open-source HLS tool developed by Prost-Boucle et al. [2014], it is designed to create hardware accelerators meeting resource constraints. It takes a subset of ANSI C as input to produce VHDL, regardless of the final target. Its specificity is that it runs an autonomous design space exploration (DSE) so that application developers need not focus on hardware.

As a plugin for AUGH, CP3 is also free and open-source. It is distributed as a branch in the current code repository. CP3 is not a standalone tool as it uses extensively the AUGH intermediate representation of circuits. The input of CP3, as stated in Section 4.1, is the HTG of the input application. More parameters are given to the AUGH command line, such as the authorized latency (t_{lat}) and the scan-chain type and width. The tool behavior has been verified through simulations and tests with a complete system-on-chip. A set of eight applications was used for benchmarking, seven were taken from the CHStone benchmark suite (adpcm, aes, blowfish, gsm, mjpeg, mpeg2 and sha) [Hara et al. 2009], and the remaining application is a simple idct algorithm. This set is representative as it goes from data-flow to control-flow oriented applications with a range of characteristics and sizes.

7.2. Checkpoints

The first step in our proposal consists in selecting checkpoints or states where context-switching is permitted. One should note that AUGH can generate multicycles states but they are kept from being selected as checkpoints by CP3 to ease the mechanism complexity. The main constraint respect, keeping within a bounded extraction time as stated in 4.3, was checked experimentally. Quantitative results for the algorithm proposed are presented in Table III and are commented on here. The data were obtained with $t_{lat} = 15,000$ cycles. In [Bourge et al. 2015], we showed that overestimating the latency value does not affect the best latency achievable by the circuit, which will be less than the selected latency. Hence, a constant value high enough to fit with every application was chosen.

The first result to point out is the small number of checkpoints effectively selected in the end, with a ratio of checkpoints to total states of 6.4%. The scan family selection was also notably more efficient with applications where a large number of checkpoints were selected (mjpeg and mpeg2). Many checkpoints share the same context in this case.

The fourth column shows the size of a full scan-chain, i.e., if all the registers of the circuits were selected for extraction. This value is used for further comparisons. The next column represents the size of the complete set of registers that will be present in one or more PSCs, it could be compared with the full scan-chain to illustrate the initial reduction of the set of extractable variables, but no actual data are available at this point. Nonetheless, a smaller number of registers to extract could result in a lower hardware overhead (fewer multiplexors and routing resources). In terms of

Table III: Results of the proposed method on a common application set

	# of PSC /checkpoints /states	checkpoint ratio	full scan-chain (bit)	partial extraction mech. (bit)	max PSC (bit)	mean PSC (bit)	**gain (full/ mean)**
adpcm	6/6/131	4.6 %	6304	4384	4288	2778	**2,3X**
aes	3/5/1053	0.5 %	1616	200	168	163	**9,9X**
blowfish	5/6/112	5.4 %	568	392	224	124	**4,6X**
gsm	12/14/1712	0.8 %	864	448	160	59	**14,6X**
idct	2/2/233	0.9 %	1000	40	40	36	**27,8X**
mjpeg	43/104/887	11.7 %	5762	2114	770	444	**13,0X**
mpeg2	12/19/94	20.2 %	1384	1216	1088	774	**1,8X**
sha	7/10/134	7.5 %	3168	352	256	176	**18,0X**
Mean		6.4%					**11.5X**

reducing hardware overhead and impact on the circuit's characteristics, the benefits are observable after implementation of the partial extraction mechanism.

A final remark can be made about the idct. This simple application can execute in less than t_{lat} $(15,000$ cycles). Thus, it could be integrated effortlessly in a "run to completion" model, i.e., where a task started by the system finishes without interruption. The experiment shows that CP3 automatically handles such cases, since only 2 checkpoints are selected: input and output of handshake loops.

7.3. Partial Scan-Chains

Partial scan-chains were implemented after the scan families had been selected. The three last columns of Table III, showing the gain obtained for the register-related context thanks to the PSCs, will be analyzed in this paragraph.

The fifth column (max PSC) is the size of the biggest PSC in the circuit. This value can sometimes be almost as great as the size of the partial extraction mechanism. However, having one large PSC is not necessarily harmful as it can correspond to a checkpoint with a low matching probability (i.e., a state rarely occurring). The next column – the mean PSC size – is the most interesting result in the table as it represents the mean register-related context that must be extracted when a checkpoint is reached. Again, this is not wholly representative of the mean size of the context as more probable checkpoints cannot be assessed. Nevertheless, when compared with a full scan-chain, where the footprint is set to a certain value, the gains are substantial. Extracting only the context-necessary data is thus apparently very beneficial, with a geometric mean reduction of 11.5 times the size of the register-related context. This equates to a reduction of impact on the surrounding system and a diminished context-switch latency compared to a full scan-chain method. Even if this last point is a firm constraint for the circuit (as the t_{lat} parameter of the checkpoint selection), it remains a worst-case scenario. Indeed, if a checkpoint is nearby when a context-switch is required, and if this checkpoint has a small context size, the context-switching process can be completed in a very small number of cycles.

7.4. Partial Scan-Chain Fragmentation

PSCs can be implemented in two ways (see Section6.3). The first way, with a naive one-bit scan-chain, comes at the minimum theoretical cost but also provides the slowest extraction time, as bits must be serially extracted. The other way involves multiple parallel one-bit scan-chains to build a parallel link between registers. This method divides the extraction time by a factor corresponding to the width of the scan-chain. CP3 can produce both types of chains. The overhead introduced by scan-chain fragmentation in the second method is discussed here.

Table IV: Dummy registers (DR) quantification

	total DR (bit)		mean DR/PSC (bit)		size wrt mean PSC size		ratio w/o id-related DR	
	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit
adpcm	n/a	157	n/a	77	n/a	2.8%	n/a	0.6%
aes	53	149	37	90	22.7%	55.2%	4.9%	17.8%
blowfish	53	213	33	91	26.6%	73.4%	3.2%	24.2%
gsm	124	412	36	89	61.0%	150.8%	13.6%	49.2%
idct	54	118	42	90	116.7%	250.0%	33.3%	77.8%
mjpeg	875	1611	44	93	9.9%	20.9%	4.3%	8.1%
mpeg2	27	187	27	69	3.5%	8.9%	0%	1.3%
sha	28	92	28	64	15.9%	36.4%	0%	2.3%
Mean					37%	75%	8.5%	23%

Table IV shows the results of the circuits built with the parallel scan-chain scheme in CP3. The PSC implementation was run both with a width of 32 bits and a width of 64 bits. The 32-bit-wide scan-chain could not be used with adpcm because this application uses some $64\,\mathrm{bit}$ registers and the current implementation cannot handle registers that need to be split to fit the available width. The first two columns present the number of dummy registers (DR) inserted to allow scan-chain parallelization. There can be at most one DR per scan-chain word (see Figure 7b). Some DRs may be shared between PSCs. In particular, the id of the corresponding checkpoint was added to every PSC header. In each implementation, this id is isolated on its PSC word, thus for a $32\,\mathrm{bit}$ wide scan-chain, if this id is 4 bits wide (e.g. for the sha application with 10 checkpoints), a shared DR of at least $32 - 4 = 28\,\mathrm{bit}$ will be present in every PSC. The same example with a $64\,\mathrm{bit}$ wide scan-chain gives $64 - 4 = 60\,\mathrm{bit}$ which more than double the dummy register width. This specific DR will later be called the id-related DR. For the sha application, the table shows this id-related DR to be the only DR added, and that it is shared by all 7 PSCs. For each application, an id-related DR will be created.

Because the primary contributor to the mean DR per PSC is the id-related DR, this mean is very stable even when comparing different applications. Indeed, the mean DR ranges from 1 bit (idct) to 7 bits (mjpeg), corresponding to an id-related DR ranging from 63 to 57 bits for 64-bit-wide PSCs.

The next two columns show the fragmentation problem faced (i.e., is it possible to stack the PSC registers to avoid introducing a fragmentation overhead?). We see that when the width of the PSCs increases, the fragmentation overhead also increases. In general, when an application is 32-bit-wide compatible (all except adpcm), the results are not improved by extending the width to 64 bits, as the overhead is more than doubled. Finally, some applications are very sensitive to fragmentation, and it is not possible to find a good way to build the PSC for these cases in our current implementation. For instance with a $64\,\mathrm{bit}$-wide scan-chain, the volume of extracted data for gsm and idct are more than doubled ($+150.8\%$ and $+250\%$, respectively). On the other hand, adpcm undergoes almost no data fragmentation, with only a 2.8% addition. The last two columns show the same ratio as the previous ones excluding the id-related DR. These figures highlight the contribution of the id-related DR to the cost of fragmentation.

Despite these results showing a considerable overhead when parallelizing scan-chains paths, an advantage remains: broadening the scan-chain makes it possible to introduce MSCs more easily.

7.5. Memory Scan-Chain

This paragraph describes the overhead associated with memory extraction as above for DR. Table V compiles the data gathered for the memory-related context size (second column), the extraction time for this context with fragmentation (third and fourth

Table V: Size of memories to extract and associated fragmentation

		Extraction time (cycles)		fragmentation overhead		fragmentation ratio	
	size (bits)	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit
adpcm	9600	n/a	200	n/a	3200	n/a	25%
aes	18432	576	544	0	16384	0%	47%
blowfish	34048	1082	1082	576	35200	2%	51%
gsm	3264	178	169	2432	7552	43%	70%
idct	2560	128	64	1536	1536	38%	38%
mjpeg	288808	13350	6803	138392	146584	32%	34%
mpeg2	16768	2060	2052	49152	114560	75%	87%
sha	672	21	16	0	352	0%	34%
Mean						27%	48%

columns) and the evaluation of this fragmentation. The latter is presented as a raw overhead in bits and as a ratio of the total data extracted.

Memory-related context size varies considerably between applications (sha has $672\,\mathrm{bit}$ whereas mjpeg has more than $288\,\mathrm{kbit}$). The current implementation, as for registers, can only produce a minimum scan-chain width of the same size as the largest memory word in the circuit. Indeed, MSCs cannot divide memory words into smaller parts. On the other hand, several memories can be parallelized if their combined word lengths fit into the scan-chain width (6.4). More efficient mechanisms could be designed in order to reduce or remove fragmentation. This paper presents the simplest one in order to avoid overheads. The adpcm has memory words wider than 32 bits, making a 32-bit-wide scan-chain inappropriate. However, for the other applications, it is not necessary to take a 64-bit-wide scan-chain. The experiment was performed to see if this could have a positive impact and to assess the drawbacks. Our conclusion is that broadening the data channel more than necessary is not effective, just as for the PSCs in the previous paragraph. A final observation is that extracting all the memories for the switching application is quite expensive in terms of latency, bandwidth, data storage needs and fragmentation costs.

7.6. Hardware Overhead

Here, we will discuss the hardware overhead of the method in LUT and FF, applied with different parameters. Firstly, a distinction was made between a bare circuit and three circuits differently equipped for context-switching. The results presented above with regard to the fragmentation cost of the different steps of the method were underlined. Then, two circuits with the same context-switch capability were compared to illustrate the advantages of the proposed method over a full scan-chain method.

The various means of embedding the partial extraction mechanism have a changing impact on the circuit size (Figure 9). The number of LUT and flip-flop were normalized with respect to the quantity of LUT for a bare circuit. This normalization makes it possible to compare the overhead between applications and to illustrate the LUT/FF ratio. All the results were taken after logic synthesis with ISE 14.7 software (Xilinx) with target xc7v585t. This target was oversized to avoid potential routing shortages and to obtain the most realistic method-related cost. In terms of the parameters for CP3, three circuits were built in addition to a bare one. The first circuit had a naive 1-bit-wide partial extraction mechanism and did not include memory extraction. The second circuit also lacked a memory scan-out mechanism; it was based on a 32-bit-wide partial extraction mechanism. The last circuit was built with a full 32-bit-wide partial extraction mechanism. Currently, PSCs and MSCs share the same input and output pins (see Figure 8). Hence, a 1-bit-wide partial extraction mechanism could

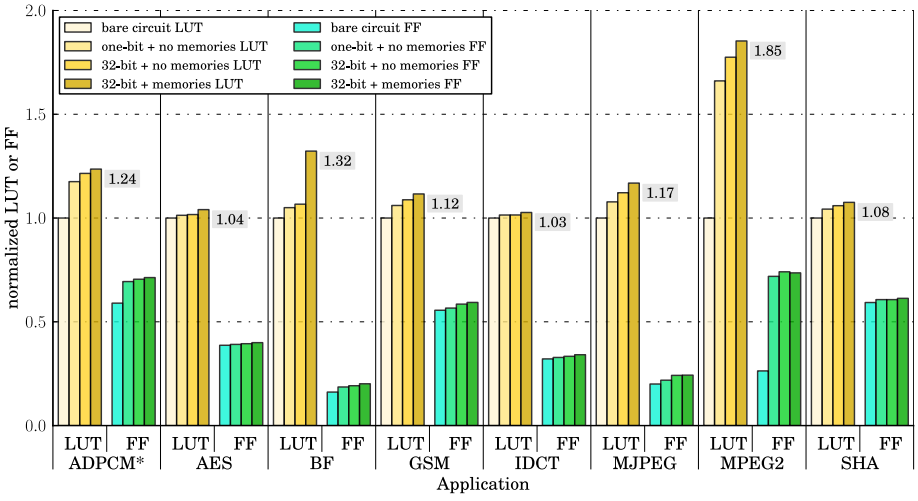


Fig. 9: Post logic synthesis area results expressed in LUT and FF (ISE 14.7)
*: adpcm has a 64-bit-wide mechanism

never contain memory words; similarly, as the adpcm is not 32 bit compliant, it was built with a 64 bit-wide mechanism.

Overall, the number of FF was less than twice the number of LUT (which is the resource distribution ratio in Virtex 7 technology [Xilinx 2014]). Because of this ratio, we focused on the LUT results. The applications responding best to the mechanism added were aes, idct and sha. These three applications have a LUT overhead of less than 8% even with a full mechanism. The FF overhead was also small (6%, 10% and 5%, respectively). These three circuits had the smallest partial extraction mechanisms (cf. Table III). Three other applications (adpcm, gsm and mjpeg) gave intermediate results. The LUT overheads were 24%, 12% and 17%, respectively. These overhead results are comparable with [Koch et al. 2007], even though in our solution an extra memory extraction mechanism is proposed.

One particular case, blowfish (abbreviated ”bf” in the Figure), has decent results for the mechanisms not including memory extraction. However, a hardware overhead peak was observed when the MSCs were added. This is because ISE can no longer optimize some memory mapping. The mpeg2 application had the highest overhead (85%). This is probably because this application has the highest checkpoint ratio (20%) in addition to a weak register-related context reduction (1.8-fold). A comparable application in terms of checkpoint ratio is the mjpeg decoder which had a lower overhead (17%). Indeed, both applications undergo approximately the same LUT growth but the bare circuits are not the same size initially. The decoder is a larger application at the outset, hence the LUT addition is less visible.

Table VI presents a comparison of two scan-chain types. Four syntheses (ISE 14.7 on xc7v585t) are shown: two are 1-bit-wide mechanisms and the other two are 32-bit-wide mechanisms. For each width, one scan-chain insertion is run with the method presented and another is added using a full scan-chain. The full scan-chain corresponds to the ”full” column of the table whereas the ”PSC” column corresponds to the circuit built using our method. The full scan-chain insertion is a link made between each register of the circuit. The addition is done with the CP3 software as if it was a circuit with a single PSC containing all the registers. Thus, the same CP3 software function is used

Table VI: Comparison between full scan-chain and partial extraction mechanism
*: adpcm has a 64-bit-wide mechanism

	1-bit-wide scan-chain						32-bit-wide scan-chain					
	LUT			FF			LUT			FF		
	full	PSC	%	full	PSC	%	full	PSC	%	full	PSC	%
adpcm*	12602	10399	$-17,5\%$	6566	6078	$-7,4\%$	12600	10773	$-14,5\%$	6600	6175	$-6,4\%$
aes	7833	7172	$-8,4\%$	2970	2769	$-6,8\%$	7889	7198	$-8,8\%$	2977	2793	$-6,2\%$
blowfish	4145	4156	$0,3\%$	739	736	$-0,4\%$	4145	4222	$1,9\%$	787	760	$-3,4\%$
gsm	5462	5294	$-3,1\%$	2868	2826	$-1,5\%$	5462	5431	$-0,6\%$	2868	2921	$1,8\%$
idct	4799	4481	$-6,6\%$	1478	1450	$-1,9\%$	4799	4482	$-6,6\%$	1502	1474	$-1,9\%$
mjpeg	36236	34629	$-4,4\%$	7038	7025	$-0,2\%$	36230	36044	$-0,5\%$	7069	7773	$10,0\%$
mpeg2	3388	3481	$2,7\%$	1535	1507	$-1,8\%$	3388	3720	$9,8\%$	1559	1553	$-0,4\%$
sha	8440	5803	$-31,2\%$	3351	3377	$0,8\%$	8415	5895	$-29,9\%$	3353	3377	$0,7\%$
Mean			**–8.5%**			**–2.4%**			**–6.2%**			**–0.7%**

to build the PSCs and the full scan-chain. The latter will suffer the same drawbacks due to DR addition when building a 32-bit-wide chain.

Again, the LUT results were more relevant as flip-flop numbers were two-fold lower for all the applications tested. The partial extraction mechanism gave better results overall on the LUT number than the simpler mechanism of the full scan-chain (which implies addition of fewer multiplexors) except for the mpeg2 application where the relatively large and numerous PSCs cause the circuit to be overweight. Register selection cannot compensate for this overhead. For all the applications, the mitigated (-0.8%) flip-flop results with the 64-bit-wide partial extraction mechanism were mostly due to the multiplication of DRs. These registers are not shared when building PSCs in the current implementation. As a whole, this table shows the average positive effect (-6.9% LUT) on the area covered by the method when using a $32\,\mathrm{bit}$-wide scan-chain. The two circuits compared had similar capacities, but the one built with our method tended to be smaller.

7.7. Application Performance

The ultimate optimization objective of the proposal, how it affects application performance is the focus of this paragraph. The maximum frequency achievable is used as the metric to compare how the circuits perform. After place and route with ISE 14.7, this particular value is reported in Table VII. The table presents the difference between the critical path and frequency of a bare circuit and a circuit with a full mechanism (i.e., a $32\,\mathrm{bit}$-wide mechanism with memory extraction). The mpeg2 suffers from the mechanism insertion as it experiences a 10% frequency drop. This can be explained in part by the high proportion of circuit registers present in the partial extraction mechanism (1216 on 1384 registers cf. Table III). The operating frequency of the majority of the circuits is reduced by less than 1% and is reduced by 2.6% on average.

8. DISCUSSION

With no design efforts, the CP3 plugin automatically produces a circuit in HDL with context-switch capacity from a C file. The circuit produced respects an extraction latency indicated by the user. The applications produced by CP3 are portable as they are written in HDL. The experiments performed demonstrated the advantages offered and trade-offs required by the method.

First, we worked on the impact on the system as a whole. The gain can be illustrated both in terms of context size and extraction latency as described by [Jozwik et al. 2012]. The two methods (readback and memory-mapped, see Section 3) presented by these authors allow a fair comparison of gsm, idct and sha applications. Table VIII summarizes the data gathered. With our solution, the context size was calculated by

Table VII: Post mapping critical path and frequency results (ISE 14.7)
*: adpcm has a 64-bit-wide mechanism

	Critical path (ns)		frequency (MHz)		
	bare	full	bare	full	%
adpcm*	9.654	9.709	103.581	102.997	0.56%
aes	5.173	5.178	193.298	193.141	0.08%
blowfish	5.87	5.874	170.364	170.242	0.07%
gsm	12.963	13.462	77.141	74.281	3.71%
idct	12.628	12.628	79.188	79.188	0.00%
mjpeg	41.102	44.006	24.33	22.724	6.60%
mpeg2	13.038	14.476	76.696	69.082	9.93%
sha	8.969	8.973	111.492	111.44	0.05%
Mean					**2.6%**

Table VIII: State of the art comparison

		[Jozwik et al. 2012] readback (CPA)	[Jozwik et al. 2012] memory map (TSAS)	Checkpointing + partial scan-chains
Context size	gsm	99.7 kbit	5.37 kbit	5.90 kbit
	idct	49.6 kbit	1.25 kbit	4.20 kbit
	sha	47.5 kbit	1.65 kbit	0.83 kbit
Extract time	gsm	299 µs	n/a	2.6 µs (190 cycles @ 74 MHz)
	idct	151 µs	266 µs	1.6 µs (130 cycles @ 79 MHz)
	sha	144 µs	351 µs	0.23 µs (26 cycles @ 111 MHz)

applying the following formula: C_{size} = scan-chain width $\times$ (extraction_time$_{memories}$ + extraction_time$_{registers}$) with scan-chain width = 32 bit, extraction_time$_{memories}$ taken from Table V as an exact value, and extraction_time$_{registers}$ given by CP3, but as an estimated mean value. The figures presented highlight the methods efficiency. Indeed, the context size is comparable for TSAS and checkpointing, but extraction time is much shorter in our case. CPA is profitable neither for context size nor for extraction time. A final comparison can be made with software context-switch. In general, the latter can span from several microseconds to more than one thousand microseconds as stated in [Li et al. 2007].

We then focused on the hardware overhead and the necessary trade-offs to be considered with the partial scan-chain method. The fragmentation problem induced by parallelization of the scan-chains and straightforward extraction of the memories is complex. The main idea is that the scan-chain width should be adapted to the application considered, as we showed that applications fitting a 32-bit scan-chain have poorer results when built with a 64-bit-wide scan-chain. In compensation however, extraction was almost always faster with a 64-bit-wide scan-chain. When compared to a full scan-chain, our method presented a smaller overhead in addition to the previous advantages. Finally, the application performances were consistent and post place and route results produced by ISE 14.7 did not show substantial variations in terms of operational frequency for the applications tested.

9. CONCLUSION AND FUTURE WORK

Efficient hardware context-switching on FPGA allows for numerous practical applications. This paper presents a method to enable context extraction from a circuit running on a reconfigurable fabric. The addition of this mechanism comes at a cost which is variable depending on the application. However, it has a low impact on application performance and consequently tends to reduce the context size compared with readback methods and even full scan-chain methods. Context extraction is also much faster with

our mechanism than with a readback method. Finally, the circuits built with CP3 are portable, which makes them potential candidates for widespread flexible usage.

This work is highly compatible with much of the existing literature: dynamic (or not) partial reconfiguration, scheduling of hardware tasks, etc. Future work should include liveness flags for addressable variables in order to reduce the memory-related context size. Another option could be to improve PSC construction by including scan-chain reordering [Zaourar et al. 2012] and/or cost-free scans [Lin et al. 1995]. Eventually, further research on how to limit the number of PSCs (to limit hardware overhead from multiplexors and dummy registers) while keeping a fast context extraction could prove interesting.

REFERENCES

Ghazanfar Asadi and Mehdi B Tahoori. 2005. Soft error rate estimation and mitigation for SRAM-based FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. ACM, 149–160.

Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. 2003. *A Self-reconfiguring Platform*. Springer Berlin Heidelberg, Berlin, Heidelberg, 565–574. DOI:http://dx.doi.org/10.1007/978-3-540-45234-8_55

Alban Bourge, Olivier Muller, and Frédéric Rousseau. 2015. Automatic High-Level Hardware Checkpoint Selection for Reconfigurable Systems. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. 100–103.

Alexander Brant and Guy GF Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 93–96.

G.C. Buttazzo, M. Bertogna, and Gang Yao. 2013. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *Industrial Informatics, IEEE Transactions on* 9, 1 (Feb 2013), 3–15.

Vasek Chvatal. 1979. A greedy heuristic for the set-covering problem. *Mathematics of operations research* 4, 3 (1979), 233–235.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, and others. 2001. *Introduction to Algorithms*. Vol. 2. MIT Press and McGraw-Hill.

IfeanyiP. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing* 65, 3 (2013), 1302–1326. DOI:http://dx.doi.org/10.1007/s11227-013-0884-0

Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)* 34, 3 (2002), 375–408.

Sándor P Fekete, Tom Kamphans, Nils Schweer, Christopher Tessars, Jan C van der Veen, Josef Angermeier, Dirk Koch, and Jürgen Teich. 2012. Dynamic defragmentation of reconfigurable devices. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 5, 2 (2012), 8.

Taro Fujii, K-i Furuta, and others. 1999. A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture. In *ISSCC, 1999*. IEEE.

Milind Girkar and Constantine D Polychronopoulos. 1994. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming* 22, 5 (1994), 519–551.

Nan Guan, Qingxu Deng, Zonghua Gu, Wenyao Xu, and Ge Yu. 2008. Schedulability Analysis of Preemptive and Nonpreemptive EDF on Partial Runtime-reconfigurable FPGAs. *ACM Trans. Des. Autom. Electron. Syst.* 13, 4, Article 56 (Oct. 2008), 43 pages. DOI:http://dx.doi.org/10.1145/1391962.1391964

Ilker Hamzaoglu and Janak H Patel. 1999. Reducing test application time for full scan embedded cores. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*. IEEE, 260–267.

Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. 2009. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Information and Media Technologies* 4, 4 (2009), 740–752.

Scott Hauck and Andre DeHon. 2010. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Karel Heyse, Timothy N Davidson, Elias Vansteenkiste, Karel Bruneel, and Dirk Stroobandt. 2013. Efficient implementation of virtual coarse grained reconfigurable arrays on FPGAs. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 1–8.

Krzysztof Jozwik, Hiroyuki Tomiyama, Masato Edahiro, Shinya Honda, and Hiroaki Takada. 2012. Comparison of preemption schemes for partially reconfigurable FPGAs. *Embedded Systems Letters, IEEE* 4, 2 (2012), 45–48.

Heiko Kalte and Mario Porrmann. 2005. Context saving and restoring for multitasking in reconfigurable systems. In *FPL, 2005*. IEEE.

Dirk Koch, Christian Haubelt, and Jürgen Teich. 2007. Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. ACM.

Wesley J. Landaker, Michael J. Wirthlin, and Brad L. Hutchings. 2002. *Multitasking Hardware on the SLAAC1-V Reconfigurable Computing System*. Springer Berlin Heidelberg, Berlin, Heidelberg, 806–815. DOI:http://dx.doi.org/10.1007/3-540-46117-5_83

L Levinson, Reinhard Männer, M Sessler, and Harald Simmler. 2000. Preemptive Multitasking on FPGAs.. In *FCCM, 2000*. Citeseer, 301–302.

Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *ExpCS 2007*. ACM, 2.

Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N Do, and Deming Chen. 2012. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering* 2012 (2012), 1.

Chih-Chang Lin, Mike Tien-Chien Lee, Malgorzata Marek-Sadowska, and Kuang-Chien Chen. 1995. Cost-free scan: a low-overhead scan path design methodology. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society, 528–533.

J-Y Mignolet, Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. 2003. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *DATE, 2003*. IEEE, 986–991.

Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. 2011. Performance of partial reconfiguration in FPGA systems: A survey and a cost model. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4, 4 (2011), 36.

Adrien Prost-Boucle, Olivier Muller, and Frédéric Rousseau. 2014. Fast and standalone Design Space Exploration for High-Level Synthesis under resource constraints. *Journal of Systems Architecture* 60 (2014), 79–93.

M Sonza Reorda, Massimo Violante, Cristina Meinhardt, and Ricardo Reis. 2009. A low-cost SEE mitigation solution for soft-processors embedded in systems on programmable chips. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 352–357.

Stephen M Scalera and Jóse R Vazquez. 1998. The design and implementation of a context switching FPGA. In *FCCM, 1998*. IEEE, 78–85.

Andrew G Schmidt, Bin Huang, Ron Sass, and Matthew French. 2011. Checkpoint/restart and beyond: resilient high performance computing with FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 162–169.

Pete Sedcole, Brandon Blodget, Tobias Becker, James Anderson, and Patrick Lysaght. 2006. Modular dynamic reconfiguration in Virtex FPGAs. In *Computers and Digital Techniques, IEE Proceedings*. IET.

H. Simmler, L. Levinson, and R. Männer. 2000. *Multitasking on FPGA Coprocessors*. Springer Berlin Heidelberg, Berlin, Heidelberg, 121–130. DOI:http://dx.doi.org/10.1007/3-540-44614-1_13

Nur A Touba. 2006. Survey of test vector compression techniques. *Design & Test of Computers, IEEE* 23, 4 (2006), 294–303.

Steven Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. 1997. A time-multiplexed FPGA. In *FCCM, 1997*. IEEE, 22–28.

Michael Ullmann, Michael Hübner, Björn Grimm, and Jürgen Becker. 2004. An FPGA run-time system for dynamical on-demand reconfiguration. In *IPDPS, 2004*. IEEE, 135.

Timothy Wheeler, Paul Graham, Brent Nelson, and Brad Hutchings. 2001. Using design-level scan to improve FPGA design observability and controllability for functional verification. In *FPL, 2001*. Springer, 483–492.

Xilinx. 2010. Partial Reconfiguration User Guide. (2010). UG702.

Xilinx. 2014. 7 Series FPGAs Configurable Logic Block. (2014). UG474.

Lilia Zaourar, Yann Kieffer, and Chouki Aktouf. 2012. A graph-based approach to optimal scan chain stitching using RTL design descriptions. *VLSI Design* 2012 (2012), 3.