Verification of Coarse-Grained Reconfigurable Arrays through Random Test Programs

Bernhard Egger, Eunjin Song, Hochan Lee, and Daeyoung Shin
Dept. of Computer Science & Engineering, Seoul National University, Korea
bernhard@csap.snu.ac.kr

Abstract

We propose and evaluate a framework to test the functional correctness of coarse-grained reconfigurable array (CGRA) processors for pre-silicon verification and post-silicon validation. To reflect the reconfigurable nature of CGRAs, an architectural model of the system under test is built directly from the hardware description files. A guided place-and-routing algorithm is used to map operations and operands onto the heterogeneous processing elements (PE). Test coverage is maximized by favoring unexercised parts of the architecture. Requiring no explicit knowledge about the semantics of operations, the random test program generator (RTPG) framework seamlessly supports custom ISA extensions.

The proposed framework is applied to the Samsung Reconfigurable Processor, a modulo-scheduled CGRA integrated in smartphones, cameras, printers, and smart TVs. Experiments demonstrate that the RTPG is versatile, efficient, and quickly achieves a high coverage. In addition to detecting all randomly inserted faults, the generated test programs also exposed two yet unknown actual faults in the architecture.

CCS Concepts • **Hardware** → **Functional verification**; **Post-manufacture validation and debug**; • **Computer systems organization** → *Embedded systems*;

Keywords Coarse-grained reconfigurable array; random test program generation; verification; validation

ACM Reference Format:
Bernhard Egger, Eunjin Song, Hochan Lee, and Daeyoung Shin. 2018. Verification of Coarse-Grained Reconfigurable Arrays through Random Test Programs . In *Proceedings of 19th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3211332.3211342

1 Introduction

Coarse-grained reconfigurable array (CGRA) processors have recently been proposed and adopted in computation-intensive mobile systems such as smartphones, cameras, printers, or smart TVs [13, 17, 18, 27]. CGRAs are attractive because they offer programmatic flexibility while providing large raw computation capabilities at a low energy consumption and reduced development cost [21, 29]. CGRAs operate in data-flow mode and are fully programmable. They consist of an array of processing elements (PE) and register files (RF) connected by an interconnection network. In addition to the dynamic reconfigurability when routing data through the on-chip network, several CGRA implementations also offer *design-time reconfigurability* in the number and functionality of the PEs, number and size of the RFs, the style of the interconnection network, and even domain-specific instructions, so-called *intrinsics*, through a high-level textual hardware description [17, 21].

The reconfigurable nature of the hardware poses special challenges for functional verification and validation. Existing formal methods proving functional correctness [6, 35] are limited with respect to the wide range of reconfigurable options. We adopt a compiler-based approach in which the system under test (SUT) executes automatically generated random test programs. The test programs are generated by a random test program generator (RTPG) that produces a valid sequence of instructions while adhering to hardware and user-defined constraints [1]. Random test programs are preferred over compiled code because compiler-generated code tends to contain regular code patterns that do not exercise the full range of all possible valid execution combinations.

In contrast to RTPGs for conventional microprocessors where input operands are encoded directly in the instruction in the form of register identifiers or immediate values, an RTPG for CGRAs not only has to generate a valid instruction sequence for a large number of PEs but must also ensure that the input operands to operations are routed correctly through the interconnection network. In addition, the interconnection network with its many multiplexers and connections needs to be tested to ensure functional verification. We adapt an edge-centric modulo scheduling algorithm used in commercial CGRA compilers [26] to achieve this.

Another requirement for the proposed RTPG is seamless support for custom ISA extensions. Existing RTPGs [1, 12, 24] require the semantics of every instruction to be defined in

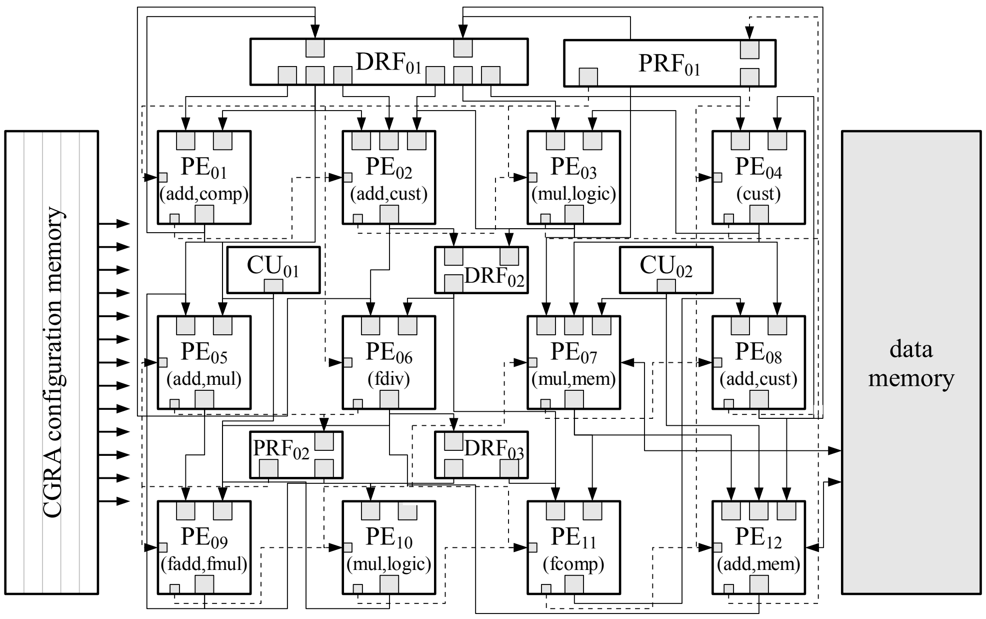


Figure 1. Example of a coarse-grained reconfigurable architecture.

an architecture description language (ADL). We take this abstraction one level higher by only requiring the *syntax* of the intrinsics but (with few exceptions) not their exact semantics.

The proposed RTPG is evaluated in terms of coverage of the generated test programs and by running the test programs on various CGRA instances of the Samsung Reconfigurable Processor (SRP) [34]. The experiments show that the proposed RTPG quickly achieves high coverage not only in traditional measures (instruction coverage and register file coverage) but especially also in exercising all possible connection in the interconnection network. In the real-world tests, the generated test programs detected all randomly inserted faults and uncovered two yet unknown errors in the Verilog implementation of the CGRA.

In summary, the contributions of this work are as follows:

- we propose an RTPG for modulo-scheduled CGRAs for pre-silicon verification and post-silicon validation.
- seamless support for custom ISA extensions by requiring only the syntax but not the semantics of intrinsics.
- high coverage through adopting a modified edge-centric scheduling algorithm.
- the effectiveness of the RTPG is demonstrated on the SRP, an commercial CGRA.

The paper is organized as follows: Section 2 introduces CGRAs. Section 3 discusses the design and implementation of the RTPG. The verification framework and test program generation are discussed in Section 4; an evaluation using the generated test programs is given in Section 5. Section 6 discusses related work, and Section 7 concludes the paper.

2 Coarse-Grained Reconfigurable Arrays

CGRAs are low-power, high-performance accelerators typically used in embedded systems to provide image, video, and audio processing [21]. This work considers CGRAs that operate in data-flow mode, i.e., no changes in control flow are possible during execution of a loop. To exploit the computing power of the many processing elements, a software-pipelining technique called modulo scheduling [30] is used. This section first provides a general introduction to CGRAs, then discusses the Samsung Reconfigurable Processor.

2.1 Architecture

The computational power of CGRAs is provided by a number of **processing elements** (PE) capable of executing word-level operations on scalar, vector, or floating point data. PEs are often heterogeneous in their architecture and functionality. Some PEs may only support operations with two input operands and thus have two input operand ports, others with three input operand ports can support three-operand operations. **Register files** (RF) provide temporary and fast storage of data values. Unlike traditional ISAs, immediate operand values are not directly encoded in an instruction. Instead, data values are routed from their producers to the inputs of a PE. Constant values are generated by **constant units** (CU). Figure 1 shows an example of a CGRA with twelve PEs, five register files (labeled DRF/PRF), and two CUs. Input operands and results of PEs are routed through an **interconnection network** comprising physical connections (wires), multiplexers, and latches. The interconnection network is typically sparse and irregular. Different networks exist to carry the different data bit widths through the CGRA. PEs often support **predicated execution**. A one-bit input predicate instructs the PE whether to execute the planned operation or perform a no-operation (nop). Predicates are generated by PEs and stored in predicate register files (PRF) and routed to the PEs through a dedicated predicate interconnection network shown with dotted lines in Figure 1.

The **configuration memory** holds the execution plan of the CGRA in *configuration lines*. A configuration line represents one cycle in the execution plan in decoded form.

2.2 Execution Model

CGRAs execute software-pipelineable loops kernels [16]. The kernel of a loop is encoded as a number of configuration lines and stored in the configuration memory. A special hardware entity, the so-called **loop stop** signal, causes the array to exit a loop when set to 1. Unlike conventional processors where the instruction encoding directly specifies the input/output operands in the form of register indices or immediate values, a CGRA instruction does not contain this information. Instead, an operation executed on a certain PE at time t will use whatever data is available at the PE's input ports and produce the result of the computation at the output port at time $t + lat$, where lat represents the latency of the operation. If no data is available at an input port, the value of the corresponding input operand and consequently the result of the computation are undefined.

All components of the CGRA operate in lock-step, and there is no control flow except loop control. A stall caused by, for example, a memory access, causes the entire array to stall. The hardware does not provide any hazard resolution or out-of-order execution. Similar to VLIW architectures it is the compiler's responsibility to generate code that does not cause any hazards. Since the timing is fixed, the exact timing of a loop's schedule can be computed at compile-time.

2.3 Reconfigurability and Programmability

CGRAs are both *reconfigurable* and *programmable*. By **reconfigurability** we refer to the *at-design time reconfigurability* of the array. Typically, CGRAs are defined by a very high-level description of the architecture [21]. This high-level definition allows, for example, to describe the number of PEs and, for each PE, the number of input and output ports and the supported operations. Register files can be defined and configured in terms of the number of registers, the bitwidth, and the number of read and write ports. The high-level description also defines the interconnection network of the CGRA by declaring multiplexers, latches, and connections between ports. Some CGRAs, including the SRP, support custom ISA extensions by specifying the instruction's syntax in textual form and its semantics by a piece of C code [21, 34].

The **programmability** of a CGRA, on the other hand, refers to the execution plan, i.e., the specification what operations to execute on which PEs in which cycle. Other than single-issue or VLIW processors, the routing of data values through the interconnection network is also programmable in a CGRA. This allows a compiler to forward data directly from the output of one PE to the input of another PE without saving the value in a register. CGRAs are typically so complex that even simple loops cannot be mapped by hand but require a modulo scheduler.

2.4 The Samsung Reconfigurable Processor

The SRP is a modulo-scheduled CGRA based on the ADRES design [21]. Its high-level organization is stored in XML format and allows the specification of the architecture, the instruction set encoding, and custom ISA extensions. The entire toolchain from the compiler, to functional and binary simulators, down to the HDL generator read this information and adapt to the SRP instance at hand. The SRP architecture file defines the number and types of PEs, the register files, constant units, and the interconnection network. Components are assigned a bitwidth, allowing for the specification of predicate, scalar, and vector data paths.

Figure 2 shows an excerpt of an SRP architecture file. The description is split into two parts: resource definitions (lines 2–34) and connections linking resources (lines 36–40). In the resource section, a register file named `DRF` is defined with 64 registers of width 32 bit. The RF has one write ports, `wp0`, and two read ports (`rp0/1`). A processing element is declared by defining its input/output ports (`in/out` for 32-bit data ports, `pred_in/out` for 1-bit predicated execution and predicate generation, respectively). The definition of a PE also lists all supported operation groups. Line 27 shows the declaration of a constant unit that is able to generate a 17-bit constant. Finally, in lines 31–33, the multiplexers are defined.

Three separate file defines syntax, semantics, and properties of operations. *Syntax* defines the number and types of input and output operands plus an operation's latency, whereas *semantics* defines the effect of an operation. For built-in general-purpose operations such as addition or subtraction, the semantics are implicitly given and do not need to be defined. The *properties* of operations define, for example, how memory addresses are computed from the input operands and the alignment constraints of memory accesses.

Figure 3 gives short examples the contents of the three files that define operations. Figure 3 (a) shows the syntax definitions of three general purpose and one user-defined operation. The `syntax` attribute specifies the type and number of input and output operands. The `class` attribute contains a list of property classes. Figure 3 (b) lists the semantics of the user-defined operations `CUSTOM_MIN` in C. The other operations are built-in operations recognized by the compiler/simulator toolchain and thus do not need to be defined. Figure 3 (c), finally, shows the definition of the property classes referred to in the operations' syntax definitions. The `MEMORY` property instructs the framework that a memory address is computed by adding input operands 1 and 2. The `ALIGN2` property defines a constraint that requires memory addresses to be a multiple of 2.

```
<srp name="audio_4x4" architecture_version="1.0">
  <resources>
    <!-- register file definitions -->
    <RF name="DRF" size="64" width="32">
      <in  name="wp0"/>
      <out name="rp0"/>
      <out name="rp1"/>
    </RF>
    ...

    <!-- processing element definitions -->
    <PE name="PE01">
      <in  name="pred_in" width="1"/>
      <out name="pred_out" width="1"/>
      <in  name="in0" width="32"/>
      <in  name="in1" width="32"/>
      <out name="out" width="32"/>
      <opgroups>
        <opgroup name="arith"/>
        <opgroup name="shift"/>
        ...
      </opgroups>
    <PE>
    ...

    <!-- constant unit definitions -->
    <CU name="CU01" width="17"/>
    ...

    <!-- interconnection network: muxes -->
    <MUX name="pe01_in0" delay="0" width="32"/>
    <MUX name="pe01_in1" delay="0" width="32"/>
    ...
  </resources>

  <connections>
    <CON src="DRF" src_port="rp0" dst="pe01_in0"/>
    <CON src="PE01" src_port="out" dst="pe02_in0"/>
    ...
  </connections>
</srp>
```

Figure 2. High-level architecture description.

```
<op name="ADD_SAT"  latency="1"
  class="ALU"
  syntax="(int:32)= (int:32,int:32,int:32)"
/>
<op name="I_TO_F"  latency="3"
  class="ALU"
  syntax="(float:32)=(int:32)"
/>
<op name="LD_U16"   latency="4"
  class="MEMORY,ALIGN2"
  syntax="(uint:16)=(int:32,int:32)"
/>
<op name="CUSTOM_MIN"    latency="2"
  class="ALU"
  syntax="(int:32)=(int:32,int:32,int:32)"
/>
...
```

(a) Syntax declaration

```
int custom_min(int a, int b, int c) {
  if (a < c) {
    if (a < b) return a; else return b;
  } else {
    if (b < c) return b; else return c;
  }
}
```

(b) Semantics of a custom operation

```
<class name="ALU"
/>
<class name="MEMORY"
  memory="src1+src2"
/>
<class name="ALIGN2"
  constrain="memory
/>
...
```

(c) Operation properties

Figure 3. Operation specification.

The SRP architecture definition is supported by the entire SRP toolchain (compiler, debuggers, simulators, and the proposed RTPG). To compile an application for another architecture, the programmer simply has select another architecture and recompile his applications to build the application binary code, debuggers and simulators capable of debugging/simulating the selected architecture. The proposed RTPG also respects the architecture definition and generates test programs for the selected architecture.

For the reminder of this work, we assume that the following parameters of the CGRA architecture are reconfigurable:

- the number of PEs, RFs, and CUs in the CGRA
- for each PE: number and data width for each input and output port and supported operations
- for each RF: number and data width of registers, number of read and write ports
- for each CU: the data width
- the interconnection network: muxes, latches, connections, and the data width of each component
- the ISA, including custom extensions
- the syntax of each operation

The following section describes the test program generation process in detail.

3 Test Program Generation

Traditional RTPGs [4, 11, 12, 23] are not well-suited to support CGRAs for a number of reasons: first, the vast majority of test generators aim at single-issue microprocessors. Test code can be structured as a arbitrarily long sequence of instructions. Modulo-scheduled CGRAs, on the other hand, are loop accelerators and not suited to execute long sequences of sequential code. Furthermore, scalar processors automatically resolve hardware hazards by delaying/reordering operations in the instruction stream; consequently, an RTPG for such architectures does not need to consider hazards. CGRAs do not provide hazard-resolution at the hardware level. Instead, the compiler (or the RTPG) is responsible to generate instruction sequences that are hazard-free.

Second, existing RTPGs require the semantics of every operation of the ISA to be known in order to pre-compute the outcome of the computation. Advanced features, such as Genesys-Pro's biased results [1], also necessitate the operation's semantics to be available. Existing frameworks for CGRAs and RTPGs that support custom ISA extensions require that the semantics of custom operations are provided in an ADL [8, 12]. The proposed framework only requires an operation's syntax and properties to be known (see Section 2.4). The semantics of the operation are not necessary to

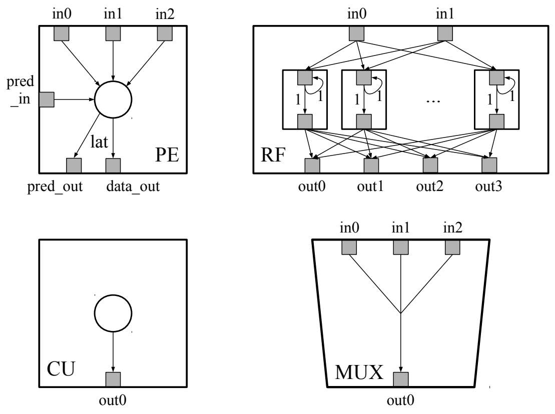


Figure 4. Graph models for hardware entities.

generate the random test programs. This has the advantage that architecture designers and application programmers can easily extend the ISA and test the effect of custom operations without requiring the knowledge of a hardware engineer.

Third, generating valid operation sequences for a CGRA requires a more complex scheduler than one for single-issue processors. One reason is the aforementioned lack of hardware hazard resolution. In addition, the scheduler must not only consider whether an operation can be scheduled on a PE at a given time, but also make sure that the input operands are correctly routed through the interconnection network and arrive at the PE's inputs at the correct time. An RTPG for CGRAs thus has to model the hardware at a much more fine-grained level than RTPGs for single-issue microprocessors. Standard approaches such as CSP- or SAT-based solutions cannot easily cope with the massive increase in complexity and suffer from long test program generation times.

Fourth, a CGRA requires additional coverage metrics to cover the interconnection network. Also, coverage metrics used for traditional architectures may not be applicable to CGRAs. Consider, for example, PE_{11} in Figure 1. PE_{11} is the only PE that can execute the `fcomp` operation group. However, PE_{11} is not directly connected to the central data register file DRF_{01}. Instead, input 0 is connected to the local register files DRF_{02} and DRF_{03}, and input 2 is connected to the output of PE_{07}. To achieve full instruction coverage, operations belonging to the `fcomp` group must be scheduled on PE_{11} with properly generated input operands in $DRF_{02,03}$ and PU_{07}.

To summarize, the requirements of an RTPG for CGRAs are as follows:

1. the generated test program is hazard-free
2. data operands are routed through the network
3. all possible operations are executed on each PE
4. all possible routes through the interconnection network are exercised
5. support for user-defined instructions
6. no knowledge of the semantics of operations is required, only their syntax and properties

Algorithm 1 Overview of test program generation

```
1:  for $t = 1$ to $time\_limit$ do
2:      for all $PEs$ do
3:          $op = GetCandidateOperation(PE, t)$
4:          if $CanPlace(PE, op, t)$ and
               $SearchRouteFromInputs(PE, op, t)$
            then
5:              Place $op$ on $PE$ at $t$
6:              Confirm routes for all inputs of $op$
7:          else
8:              Place $nop$ on $PE$ at $t$
9:          end if
10:     end for
11: end for
```

3.1 Test Generation Engine

At the core of the RTPG lies an edge-centric place-and-route scheduler [25] that operates on a graph representation of the architecture model [22]. The hardware entities are modeled as shown in Figure 4. Grey boxes represent input/output ports, edges represent connections. Ports for operation selection, register address selection and mux control ports are not shown. The latency of unlabeled connections is 0 cycles. The number of input/output ports, the number of registers in RFs, and the width of connections vary according to the textual hardware description. Entities are connected through edges representing the data and predicate networks.

Schedule generation is outlined in Algorithm 1. Operations are placed cycle-by-cycle, one PE at a time, until a schedule of the desired length $time\_limit$ has been generated (lines 1–2). For a candidate operation op on PE at a time t (line 3, Section 3.2), we check if placing the operation on PE at t does not cause any data hazards on the involved ports (line 4, $CanPlace()$), then try to find suitable data values to satisfy the required inputs of the operation (line 4, $SearchRouteFromInputs()$, Section 3.3). On success, the operation op is placed on the PE and the data routes confirmed (lines 5–6). Otherwise, a nop is scheduled (line 8). Figure 5 illustrates the placement of a one-cycle latency `ADD` operation onto a PE at time t. The data flow graph of the `ADD` operation spans two cycles representing the operation's latency. Inputs are routed backwards through the interconnection network until suitable data providers have been found.

3.2 Operation Selection

Operations selection is done for one PE at a time. Several constraints influence the selection of an operation. Hardware constraints include (a) the PE must support the selected operation and (b) the PE's input operands at time t and the output operands at time $t + latency(op)$ must be available. Additional constraints imposed by the verification engineer such as restricting selection to a certain type of operations or placement of a specific operation on a certain PE at a given time further guide the process. Operation selection

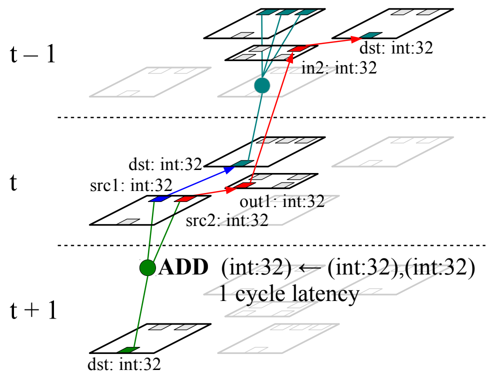


Figure 5. Placement of an operation and routing of its inputs.

has a large impact on test coverage of the interconnection network. If, for example, only operations with two operands are scheduled on a PE with three input operands, it is impossible to exercise the connections originating at the third input operand. In order to achieve a high coverage in the interconnection networks, the operation selection thus also considers whether some of its input operands (including paths originating at the input operands) have never been exercised and, if so, gives higher priority to operations that cover those input operands.

3.3 Routing Algorithm

Once an operation has been selected for a given PE, a routing algorithm connects the required inputs of the operation to existing data providers as follows. For a given input operand at time t, the *sink*, the task of the routing algorithm is to find a yet unoccupied path through the architecture's interconnection network and hardware components leading to a suitable data provider. Valid data providers must produce the requested data type at a time $t - \delta t$ for a route with a latency of δt, i.e., the route only goes backwards in time. Blindly following all possible connections leads to an exponential fanout and unacceptable runtimes. In order to keep the complexity low, we employ a priority queue and stop the search after a pre-defined number of candidate providers have been found. Ports form the nodes, connections between the nodes represent edges. The search algorithm is guided by a parameterizable *priority functions* which determine the priority of each connection and hardware port. In order to achieve a high coverage in the interconnection network, for example, yet untested connections and hardware entities are assigned a higher preference over already exercised ones. Since routing resources are limited, it is possible that no route to the output port of a PE or CU can be found. The routing algorithm thus also considers data flowing through nodes as part of a route as potential data sources.

Algorithm 2 shows, in simplified form, the operation of the $SearchRouteFromInputs()$ function called on line 4 in Algorithm 1. For all input operands of a given operation (line 1) the algorithm starts at the input port of the PE (line 3).

Algorithm 2 SearchRouteFromInputs(PE, op, t)

```
1: for all inputs $in$ of $op$ do
2:    $datatype = in.get\_datatype()$
3:    $queue = \{(in, t, 0)\}$
4:    $candidates = \emptyset$
5:    while ($queue$ not empty) and
        ($|candidates| < threshold$)
      do
6:       $(port, time, weight) = queue.pop\_head()$
7:       for all connections $c$ out of $port$ do
8:          $head = c.get\_head()$
9:          $htime = time - c.latency()$
10:         if ($head$ provides $datatype$ at time $htime$) then
11:            $candidates.insert(head, htime,$
                       $weight + Priority(c))$
12:         else if ($head$ free at $time$) then
13:            $weight += Priority(c) + Priority(head)$
14:            $queue.insert(head, htime - head.latency(),$
                       $weight)$
15:         end if
16:      end for
17:   end while
18:   if ($candidates$ not empty) then
19:      $(port, time, weight) = candidates.pop\_head()$
20:      $ApplyRoute(in, t, port, time)$
21:   else
22:      $UndoRouting()$
23:      return false
24:   end if
25: end for
26: return true
```

The priority queue holding current route ends as well as the candidate priority queue contain triples in the form of $(port, time, weight)$ where $port$ denotes the hardware entity, $time$ the current time at this entity, and $weight$ the accumulated weight (priority) of this (partial) route. In each step, the best candidate is extracted from the queue (line 6). For all connections out of the candidate port (line 7) the algorithm first checks if the port at the other end, the *head*, of the connection can provide the requested data type (line 10). If yes, the algorithm has found a viable route to a data provider and adds this route to the candidate list (line 11). If the requested data type is not provided, we check if the *head* is free (line 12) and, if yes, compute the priority of exercising this connection and finally insert the connection point to the priority queue (lines 13–14). If suitable candidates have been found (line 18), we pick the one with the highest priority and apply the route from the operations input port to the data provider (lines 19–20). If no candidate provider has been found, it is not possible to place the operation op at time t on the given PE. In that case, already routed input operands are undone and the function returns $false$ (lines 22–23).

3.4 Data Type Tracking

A distinctive feature of the proposed RTPG is that the semantics of operations need not be known. Operation selection and routing are solely based on the syntax of an operation. Part of an operation's syntax is the specification of the *data types* of its input and output operands as outlined in Section 2. Every route of a value from a source to a sink is annotated by a data type. The RTPG supports a total of six different data types. `Integer` and `float` types denote integer and floating point numbers of unknown values. Values of these types are used as input operands for operations that have no special constraints, such as an integer addition. `Known integer` and `known float` types represent known integer and floating point numbers. Whenever the result of a computation must be predictable, values of these two types are used. This is, for example, the case for values involved in loop boundary checks. To support memory operations, a `memory base address` type represents a pointer to a valid memory region and its size. Together with a value of `known integer` type, the RTPG can then generate valid memory accesses that satisfy the constraints given by the operation's syntax definition. Since the value of a predicate in predicated operations is not known, it is not possible to predict whether a predicated operation will be executed and generate a valid result or not. A sixth data type, `unknown`, is thus used for nodes that represent outputs of predicated operations (see next section). The routing algorithm never selects values of type `unknown` as sources, ensuring adherence to the operations' syntax in the generated test program.

3.5 Predicate Network Testing

The predicate network is independent from the data interconnection network and enables predicated execution. The data scheduler used to find routes from PE's input operands to potential data providers (Algorithms 1 and 2) can be used in unmodified form to perform predicate network testing since the routing algorithm is already aware of the bit-width of connections and can thus easily cope with the 1-bit predicate network. For PEs with a routed predicate of unknown value at test generation time (because the predicate is, for example, generated by a comparison instruction) it is impossible to know whether the scheduled operation will be executed at runtime or not. As a consequence it is unclear whether the generated output value is valid or not. We assign the `unknown` data types to such values which prevents the scheduler from using the result as inputs to other PEs.

4 Verification Framework

This section describes the verification framework to verify the architecture pre- and post-silicon. For pre-silicon verification, observability of internal registers and states of the architecture is provided by a Verilog-based simulator. For post-silicon validation only data values in registers and memory are available without special provisions of the hardware.

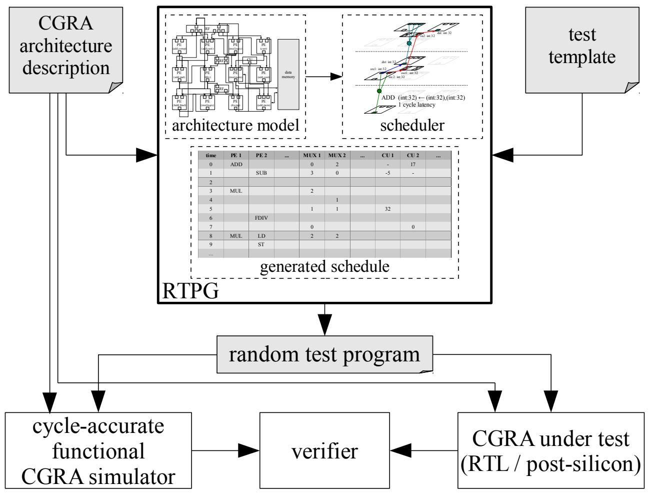

CGRA architecture description
architecture model
scheduler
test template
generated schedule
RTPG
random test program
cycle-accurate functional CGRA simulator
verifier
CGRA under test (RTL / post-silicon)

Figure 6. Operation of the verification framework.

During pre-silicon verification, we employ the RTL checker of the testing framework as outlined in Cho [5]. The RTL checker is a modified RTL simulator that compares the values in data ports with reference values obtained by a cycle-accurate functional simulator. Values can be compared after every n cycles, $n \geq 1$. Greater values for n result in a faster verification speed. If a fault is detected, the segment between the last (correct) checkpoint and the one that failed verification can be verified in one-cycle steps to locate the exact position of the fault.

Unlike the RTPG, both the RTL model and the cycle-accurate functional simulator need to be aware of instructions' semantics. To allow rapid architecture exploration, the framework allows a developer to extend an architecture instance with custom instructions simply by providing the instruction's syntax and a C-code implementation. The compiler does not select custom instructions automatically, instead, they can be called from the application code like normal functions. The toolchain automatically links the provided C-code segment to the various simulators which then execute this code whenever the corresponding custom instruction is executed on a PE.

For post-silicon validation, the RTPG inserts special code at the end of the actual random test to write-back the values of all registers back to memory. Even though the RTPG guarantees that all computed values are either used as an input in a successive instruction or written back to a register or memory, in the absence of an instruction's semantics we cannot guarantee that a wrong value on one of the inputs of an instruction will affect the computed value of the instruction. This is a limitation of the RTPG presented in this work which can only be overcome if the RTPG framework is made aware of the semantics of every operation.

Figure 6 shows the operation of the major components in the verification framework. The CGRA instance under test is given the architecture description files (Section 2.4), and a test template configures the generated test in terms of the number of cycles to be generated, what hardware units (PEs, RFs, CUs) to exercise and which operations to compose the generated test program of. The RTPG first builds an architecture model and then uses the adapted edge-centric scheduler to generate the random test program in binary form. The CGRA toolchain uses the same architecture description to automatically build the cycle-accurate functional CGRA simulator. The RTL model of the architecture is also generated automatically based on templates modules for the specific hardware entities. For pre-silicon functional verification, the RTL model is run on an RTL simulator. The automatically-built, cycle-accurate functional CGRA simulator runs in parallel with the RTL simulator. A verifier controls the progress and step-size in both components and compares the values of selected data ports such as register files, PE input/output ports, latches in the interconnection network, CUs, and so on. In post-silicon validation, random test programs are used foremost to test the integrity of the interconnection network. The functional simulator executes the test program and stores the computed values in global memory. The test program is run again on each manufactured chip, and its output compared to the reference data.

5 Evaluation

The proposed RTPG is evaluated in terms of (1) the quality of the generated test programs and (2) the fault detection achieved when running the test programs on several instances of the Samsung Reconfigurable Processor [34]. The SUT is a CGRA with 4x4 PEs, 6 RFs, 8 CUs and data/predicate interconnection networks comprising over 5000 point-to-point connections and 400 multiplexers. The PEs support a total of 250 integer, floating point, and custom operations.

5.1 Test Program Generation

The time required to generate the random test programs used in the tests below (1000 cycles each) is in the order of a few seconds on a modern desktop computer (Intel Core i7); test generation time is thus not an issue. During test generation the routing algorithm allocates and deallocates hundreds of thousands of small memory objects. A custom allocator using a pool to store and reused freed objects may speed up memory allocation performance, however, since the overall test generation time is not a concern we have chosen not to optimize memory allocation.

5.2 Test Program Coverage

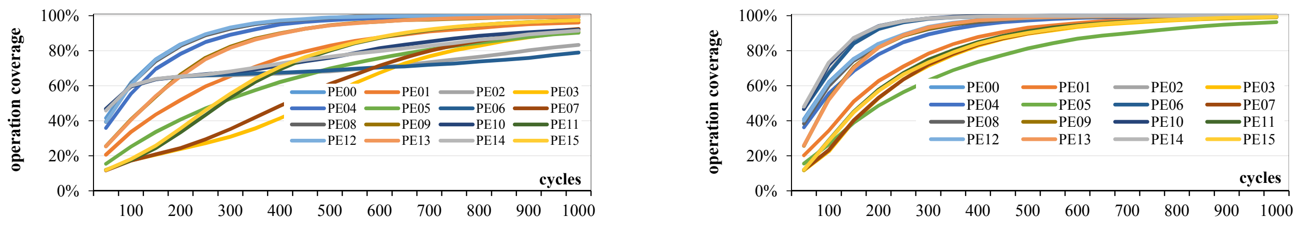
The quality of random test programs is determined by the *coverage* of a number of criteria. Our main focus here lies on coverage in the data and predicate interconnection network as this criteria is unique to CGRAs. Coverage for the interconnection network is defined as the ratio of exercised connections over the total number of connections. It is safe to assume that each input port of a multiplexer has an incoming connection; the interconnection network coverage thus also includes coverage of the different input ports of the routing elements of the network. We compute separate coverage ratios for the data and the predicate network. Coverage for register files is defined as the number of registers read from or written to over the total number of registers. We compute separate values for read coverage and write coverage. Operation coverage, typically the main goal of RTPGs in related work, is the ratio of executed distinct operations over the total number of supported operations.

Figure 7 demonstrates the effect of a prioritized routing algorithm when generating test programs for CGRAs by comparing operation, register file and interconnection network coverage for purely random test programs (left-hand side) with a test program for which priority is given to unexercised connections in the interconnection network (right-hand side of the figure). With pure random selection, only RF read coverage achieves 100% within the first 1000 cycles. Without prioritizing yet-untested connections and hardware components, the routing algorithm selects the closest (both in space and time) data sources leading to short but frequently exercised routes. In the right-hand side figures, unexercised routes are given priority over exercised ones. Looking at subfigures (c) and (d), we observe that full coverage in the interconnection networks is reached within less that 250 test cycles. Giving preference to unexercised connections has also a strong effect on register file coverage which is to be expected since register files are modeled using latches and connections. CU coverage (not shown) achieves 100% after less than 50 cycles. Operation coverage improves as a side-effect since the operation generator gives preference to operations that allow testing of yet unexercised connections.

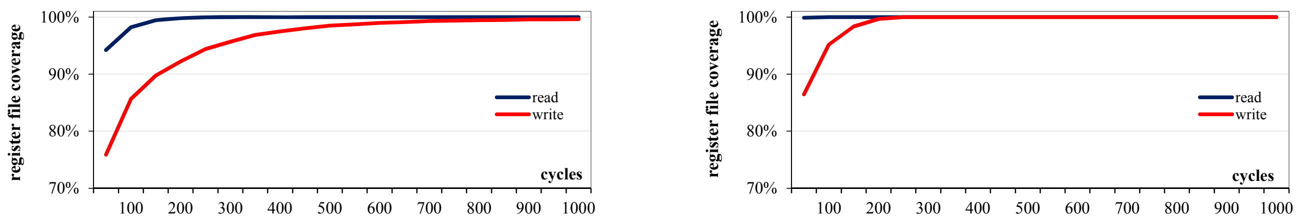
5.3 Fault Detection

To put the RTPG to a practical test, the test programs were executed on the RTL implementation of an actual SRP processor. The verification framework executes the binary test program simultaneously on an RTL simulator and a cycle-accurate functional CGRA simulator. For the test, all values on every data/predicate port and in registers in the system under test are compared with the values in the simulator.

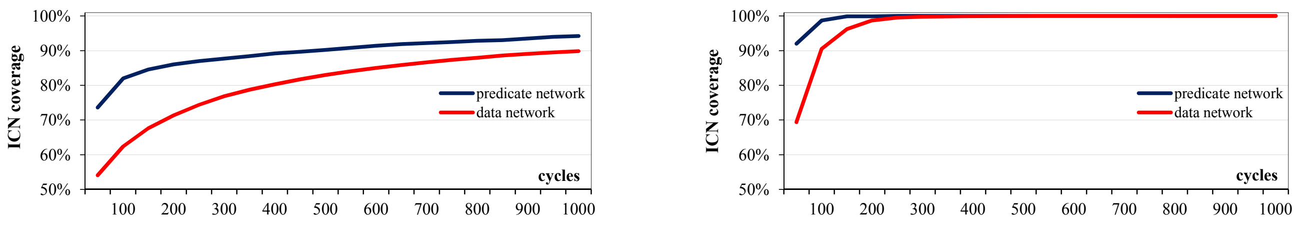
A total of 1000 distinct faulty Verilog implementations of the CGRA were generated. The randomly generated and inserted faults cover all aspects of signal routing such as routing of data and predicate values in the interconnection networks, input selection bits for the multiplexers and operation selection on the PEs. Into each of the faulty architectures, the fault generator inserted at least one and up to eight random faults from one of the following classes: selection errors in the multiplexers, missing write-enable signals, wrong register address decoding, faulty registers in the RFs, stuck-at-0/1 errors, and floating connection errors where the entire or a certain number of bits of a physical



(a) PE operation coverage



(b) RF read/write coverage



(c) Interconnection network coverage

Figure 7. Coverage of random vs. guided testing.

connection remain undefined. On average, each of the 1000 test architectures contains three faults.

Running the pre-silicon verification tests with a 1000-cycle long test program resulted in a 100% fault detection rate. To our surprise the proposed RTPG also discovered two yet unknown errors in the original (and assumed-to-be error-free) implementation of the CGRA. Both faults were caused by copy-pasting code and resulted in wrongly wired connections where one end of the connection was connected to the wrong input of a multiplexer.

5.4 Stuck-at Fault Testing

An important use case of the RTPG is post-silicon chip validation. As part of this validation, stuck-at fault testing aims at detecting signals and pins that are stuck at 0 or 1. The interconnection networks are tested by flooding them with 0 and 1 values, respectively. A verification engineer generates such tests by initializing registers and memories with 0/1 and defining an operation generator that only emits `move` operations. Similar tests can be applied to on-chip memories, initializing them by only 0s or only 1s, then reading back each value. Such a test, however, can be executed in VLIW mode of the SRP and is thus not in the scope of this work.

5.5 Discussion

The experiments confirm that the proposed RTPG for CGRA is versatile and is able to produce test programs that achieve full coverage withing a few hundred cycles and are thus excellent candidates for pre-silicon verification where simulation speed may be an issue. The fact that the proposed RTPG can generate valid test programs without any knowledge about the operations' semantics allows the RTPG to seamlessly support custom ISA extensions. While this is an important feature for a reconfigurable processor it also comes with a disadvantage: without knowing the semantics the RTPG cannot pre-compute the expected result of a sequence of operations. In pre-silicon verification where observability is no problem this poses no difficulty. For post-silicon validation with its limited observability the RTPG ensures that all computed values are either written back to memory or used as input operands in successive operations. These values can then be compared against a set of reference values; however, there is no guarantee that an invalid input operand also produces an invalid output value. This not a problem in our post-silicon validation process because the functionality of PEs is tested separately, but it may be a limiting factor if the test programs generated by the proposed RTPG are the sole post-silicon validation step.

6 Related Work

Motivated by the significant financial and time-wise overhead of verification, both academia and industry have proposed a large number of verification methods over the past few decades. Many different methods from low-level formal verification to instruction-level functional verification and from pre-silicon verification to post-silicon validation have been proposed.

For pre-silicon verification, (random) test programs and formal verification are the two prevalent methods. Approaches for instruction-level functional verification are mainly concerned with the generation of directed and/or (pseudo-) random test programs. The methods for automatic test program generation include simple random instruction selection, finite state machines (FSM), linear programming, boolean satisfiability problems (SAT), constraint satisfaction problems (CSP), or graph-based test program generation. Bin [4] and Adir [1] model the test program generation problem as a CSP. Their framework, Genesys-Pro, combines architecture-specific knowledge and testing knowledge and uses a CSP solver to generate efficient test programs. The test template language of Genesys-Pro is quite complex and allows, for example, biased result constraints. Fine [9] uses machine learning techniques to improve the initial stimuli for CSP-based RTPGs. Qin [28] combines the CSP solver and simulation techniques to analyze the real hardware design,this enable to support dynamic array references. Corno [7] and Mishra [23, 24] use graph-based algorithms to generate test programs. While Corno uses a predefined library of instructions, Mishra's work extracts the structure of the the pipelined processor directly from the architecture description language. This model is then fed to a symbolic model verifier. Di Guglielmo [11] proposes a pseudo-deterministic automatic test pattern generator based on extended FSMs. The test vectors are generated using a constraint or SAT solver. Koo [15] also uses an FSM combined with reduction techniques to achieve high coverage with a small number of directed tests. In Sanches' work [33], an automatic feedback-based approach that generates assembly instruction sequences for timing verification or speed binning is presented. Their approach is fully automatic and does not require any information about the processor's microarchitecture. The recent work of Foutris [10] analyzes the four major ISAs (ARM, MIPS, PowerPC, and x86) and finds that three quarters of the instructions can be replaced with equivalent instructions. Based on this analysis, random tests are executed that detect bugs by comparing results of equivalent instructions. Filho [8], Kamkin [12], and Rullmann [32] all propose augmented architecture description languages (ADL) to specify reconfigurable designs, however, neither work can be adapted to CGRAs with reconfigurable interconnection networks. Velev's work [35] is the first to propose a formal verification framework for CGRAs. Functional units and memories are abstracted while the control of the CGRA is modeled. The framework is applied to the ADRES architecture [20]. In contrast to our work, their method formally verifies parts of the chip but cannot be applied to test the final product, i.e., the chip itself.

In post-silicon validation, the correct operation of a processor architecture is tested by executing sequences of instructions and validating the results. These tests often produce a large amount of data which limits the speed and/or the scope of the tests. Ko [14] and Liu [19] tackle this problem by storing only a small set of trace signals that represent a larger number of states. Adir [3] propose to execute the post-silicon test program on a pre-silicon validator to obtain exact coverage measurements for later comparison. Ray [31] and Adir [2] combine pre-silicon and post-silicon verification. The former work partitions pre-silicon checkers with full observability into limited-observability checkers with the same accuracy for post-silicon validation. The latter extends coverage-driven verification methodology to the post-silicon verification domain by using test-generation languages and coverage models.

While reconfigurability is a goal in many of the of previous works, none of the presented approaches tackles the problem of routing data values in an irregular interconnection network as found in CGRAs. Unaware of instructions' semantics it is impossible to pre-compute the correct result. Instead, testing is performed by comparing the results of the SUT to a reference implementation. In contrast to [3], the proposed RTPG operates on a detailed model of the architecture and can thus compute the exact coverage metrics during test generation. The work presented here aims at achieving maximal coverage in the interconnection network; existing techniques that improve other metrics can be integrated as needed.

7 Conclusion

This paper discusses the design and implementation of an RTPG for modulo-scheduled CGRAs. To support the reconfigurable nature of CGRAs, a graph representation is built directly from the architecture description. Seamless support for custom ISA extensions is achieved by requiring only the syntax of operations but not their semantics to be known. The RTPG is able to correctly schedule operations by tracking the types of the data values.

Experimental results show that the test programs achieve maximal coverage within a few hundred cycles. Applied to the Samsung Reconfigurable Processor, a commercial CGRA shipped in the millions in smartphones and smart TVs, the test programs correctly classified 100% of the tested architectures and even exposed several to-date unknown bugs in the architecture.

A Artifact Description

A.1 Abstract

The RTPG framework described in this paper can be used for pre-silicon verification/post-silicon validation of a reconfigurable CGRA processor, the Samsung Reconfigurable Processor. The framework is integrated in the proprietary SRP development toolchain. The toolchain and the actual SRP architecture are not publicly available; it is thus not possible to run the framework on an actual SRP instance. Nevertheless, this AE package provides the tools to allow verification of the key contribution of this paper, namely the generation of test programs and the computation of the different coverage metrics.

The AE package contains an SRP architecture instance defined as outlined in Section 2.4, the source code of the test program generation as described in Section 3, the SRP SDK including the RTPG in binary form, and the test program configuration required to reproduce the results given in Section 5.2 and Figure 7.

All of the above is provided in a VirtualBox image. By following the instruction contained in the AE package itself, interested readers are able to examine the source code of presented CGRA routing algorithms, reproduce the coverage results given in the paper, or generate different test programs.

A.2 Artifact check-list (meta-information)

- **Algorithm:** all algorithms outlined in the paper are provided. This includes, in particular, the algorithm for coverage-aware routing in a modulo scheduler.
- **Program:** algorithms are implemented in C++, scripts to generate the test programs and evaluate the coverage thereof are provided as Bash/Python scripts.
- **Compilation:** automatic.
- **Data set:** the included data comprises a textual description of the SRP architecture and the configurations for the RTPG in C++..
- **Hardware:** the high-level definition of an SRP architecture instance is included.
- **Execution:** by following the `README.html` and executing the scripts provided in the AE package. Execution of the scripts can take 1-2 hours to completion.
- **Output:** recreates results of Section 5.2 and Figure 7. Coverage data of the individual and averaged over 20 runs is extracted and output in CSV format.
- **Experiments:** Download the VirtualBox AE machine, run experiments in VM, compare results.
- **Workflow frameworks used?:** VirtualBox
- **Publicly available?:** yes, upon request. Contact bernhard@csap.snu.ac.kr.

- **Artifacts publicly available?:** no
- **Artifacts functional?:** partially
- **Artifacts reusable?:** yes
- **Results validated?:** yes

A.3 Description

A.3.1 How Delivered

The artifact is provided in form of a VirtualBox VM image that includes all the necessary software to recreate the coverage results. Users interested in obtaining access to the artifact contact bernhard@csap.snu.ac.kr.

A.3.2 Hardware Dependencies

There are no particular dependencies to generate and evaluation test programs. Access to the commercial SRP architecture, required to run the fault detection test described in Section 5.3, is only possible with an NDA.

A.3.3 Software Dependencies

There are no particular software dependencies. The RTPG is implemented in C++, and the driver scripts are Bash or Python scripts. A binary version of the SRP SDK is provided in the AE package.

A.3.4 Data sets

All datasets necessary to re-create the experiments and reproduce the results are included. Detailed information about the format of the input/output data and the generated results is provided in the documentation in the AE package.

A.4 Installation

The provided AE package first needs to be imported into VirtualBox (https://www.virtualbox.org/). The virtual machine contains all the necessary software and tools as described above, no further installation steps are necessary.

A.5 Experiment Workflow

The AE package includes a series of shell scripts to perform and recreate the experiments described in the paper.

- Obtain a link to the AE VM image (contact bernhard@csap.snu.ac.kr).
- Download and import the VM image into VirtualBox.
- Start the VM, you are automatically logged in. The Desktop contains a copy of the paper (this file), and a `README.html`. Open and follow the instructions in the `README.html` file.
- The results are generated in separate subdirectories for the different techniques. Detailed explanations regarding the format of the generated results are provided inside the AE VM.

A.6 Evaluation and Expected Result

The generated results allow evaluation of the coverage of the presented method, in particular Section 5.2 and Figure 7.

A.7 Experiment customization

Detailed information on how to customize the experiments are provided by the RTPG manual available in the AE VM.

Acknowledgments

We thank the RP Core Group at Samsung Electronics for their support and the access to the necessary tools for this study. This work was supported in part by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by the National Research Foundation (NRF) of Korea (Grant 21A20151113068), the Basic Science Research Program through NRF funded by the Ministry of Science, ICT & Future Planning (Grants NRF-2015K1A3A1A14021288 and 2016R1A2B4009193), and by the Promising-Pioneering Researcher Program through Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

References

[1] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. 2004. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *IEEE Des. Test* 21, 2 (March 2004), 84–93. https://doi.org/10.1109/MDT.2004.1277900

[2] Allon Adir, Shady Copty, Shimon Landa, Amir Nahir, Gil Shurek, Avi Ziv, Charles Meissner, and John Schumann. 2011. A unified methodology for pre-silicon verification and post-silicon validation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '11)*. 1–6. https://doi.org/10.1109/DATE.2011.5763252

[3] Allon Adir, Amir Nahir, Avi Ziv, Charles Meissner, and John Schumann. 2011. Reaching Coverage Closure in Post-silicon Validation. In *Hardware and Software: Verification and Testing*, Sharon Barner, Ian Harris, Daniel Kroening, and Orna Raz (Eds.). Lecture Notes in Computer Science, Vol. 6504. Springer Berlin Heidelberg, 60–75. http://dx.doi.org/10.1007/978-3-642-19583-9_10

[4] Eyal Bin, Roy Emek, Gil Shurek, and Avi Ziv. 2002. Using a constraint satisfaction formulation and solution techniques for random test program generation. *IBM Systems Journal* 41, 3 (July 2002), 386–402. https://doi.org/10.1147/sj.413.0386

[5] Youngchul Cho, Seonghun Jeong, J. Jeong, H. Shim, Yenjo Han, Soojung Ryu, and Jeongwook Kim. 2012. Case Study: Verification Framework of Samsung Reconfigurable Processor. In *13th International Workshop on Microprocessor Test and Verification (MTV), 2012*. 19–23. https://doi.org/10.1109/MTV.2012.11

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. The MIT Press.

[7] Fulvio Corno, Gianluca Cumani, Matteo Sonza Reorda, and Giovanni Squillero. 2003. Fully Automatic Test Program Generation for Microprocessor Cores. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*. IEEE Computer Society, Washington, DC, USA, 11006–. http://dl.acm.org/citation.cfm?id=789083.1022854

[8] Julio Oliveira Filho, Stephan Masekowsky, Thomas Schweizer, and Wolfgang Rosenstiel. 2009. CGADL: An Architecture Description Language for Coarse-Grained Reconfigurable Arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 9 (2009), 1247–1259. https://doi.org/10.1109/TVLSI.2008.2002429

[9] Shai Fine, A. Freund, I. Jaeger, Y. Mansour, Yehuda Naveh, and A. Ziv. 2006. Harnessing Machine Learning to Improve the Success Rate of Stimuli Generation. *IEEE Trans. Comput.* 55, 11 (2006), 1344–1355. https://doi.org/10.1109/TC.2006.183

[10] Nikos Foutris, Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera, and Antonio Gonzalez. 2011. Accelerating microprocessor silicon validation by exposing ISA diversity. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 44)*. 386–397. https://doi.org/10.1145/2155620.2155666

[11] Giuseppe Di Guglielmo, Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli. 2011. Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs. *J. Electron. Test.* 27, 2 (April 2011), 137–162. https://doi.org/10.1007/s10836-011-5209-8

[12] Alexander Kamkin, Eugene Kornykhin, and Dmitry Vorobyev. 2011. Reconfigurable Model-Based Test Program Generator for Microprocessors. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011*. 47–54. https://doi.org/10.1109/ICSTW.2011.35

[13] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. 2012. ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical applications. In *Field-Programmable Technology (FPT), 2012 International Conference on*. IEEE, 329–334. https://doi.org/10.1109/FPT.2012.6412157

[14] Ho Fai Ko and Nicola Nicolici. 2008. Automated Trace Signals Identification and State Restoration for Improving Observability in Post-Silicon Validation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*. 1298–1303. https://doi.org/10.1109/DATE.2008.4484858

[15] Heon-Mo Koo and Prabhat Mishra. 2008. Specification-based compaction of directed tests for functional validation of pipelined processors. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (CODES+ISSS '08)*. 137–142. https://doi.org/10.1145/1450135.1450167

[16] Monica Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 318–328. https://doi.org/10.1145/53990.54022

[17] Jaedon Lee, Youngsam Shin, Won-Jong Lee, Soojung Ryu, and Kim Jeongwook. 2013. Real-time ray tracing on coarse-grained reconfigurable processor. In *International Conference on Field-Programmable Technology (FPT)*. 192–197. https://doi.org/10.1109/FPT.2013.6718352

[18] Won-Jong Lee, Shi-Hwa Lee, Jae-Ho Nah, Jin-Woo Kim, Youngsam Shin, Jaedon Lee, and Seok-Yoon Jung. 2012. SGRT: a scalable mobile GPU architecture based on ray tracing. In *ACM SIGGRAPH 2012 Posters*. ACM, 44.

[19] Xiao Liu and Qiang Xu. 2009. Trace signal selection for visibility enhancement in post-silicon validation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*. European Design and Automation Association, Leuven, Belgium, 1338–1343. http://dl.acm.org/citation.cfm?id=1874620.1874942

[20] B. Mei, B. Sutter, T. Aa, M. Wouters, A. Kanstein, and S. Dupont. 2008. Implementation of a Coarse-Grained Reconfigurable Media Processor for AVC Decoder. *Journal of Signal Processing Systems* 51, 3 (2008), 225–243. https://doi.org/10.1007/s11265-007-0152-8

[21] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo Man, and Rudy Lauwereins. 2003. *Field Programmable Logic and Application: 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix, 61–70. https://doi.org/10.1007/978-3-540-45234-8_7

[22] Bingfeng Mei, Serge Vernalde, Diedrik Verkest, Hugo De Man, and Rudy Lauwereins. 2002. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. In *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings*. 166–173. https://doi.org/10.1109/FPT.2002.1188678

[23] Prabhat Mishra and Nikil Dutt. 2005. Functional Coverage Driven Test Generation for Validation of Pipelined Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*. IEEE Computer Society, Washington, DC, USA, 678–683. https://doi.org/10.1109/DATE.2005.162

[24] Prabhat Mishra and Nikil Dutt. 2008. Specification-driven directed test generation for validation of pipelined processors. *ACM Transactions on Design Automation of Electronic Systems* 13, 3, Article 42 (July 2008), 36 pages. https://doi.org/10.1145/1367045.1367051

[25] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. 2009. Recurrence Cycle Aware Modulo Scheduling for Coarse-grained Reconfigurable Architectures. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '09)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/1542452.1542456

[26] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08)*. ACM, New York, NY, USA, 166–176. https://doi.org/10.1145/1454115.1454140

[27] Yongjun Park, Hyunchul Park, and Scott Mahlke. 2009. CGRA Express: Accelerating Execution Using Dynamic Operation Fusion. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '09)*. ACM, New York, NY, USA, 271–280. https://doi.org/10.1145/1629395.1629433

[28] Xiaoke Qin and Prabhat Mishra. 2014. Scalable Test Generation by Interleaving Concrete and Symbolic Execution. In *Proceedings of the 27th International Conference on VLSI Design (VLSID '14)*. IEEE Computer Society, Washington, DC, USA, 104–109. https://doi.org/10.1109/VLSID.2014.25

[29] Marc Quax, Jos Huisken, and Jef van Meerbergen. 2004. A Scalable Implementation of a Reconfigurable WCDMA Rake Receiver. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '04)*. IEEE Computer Society, Washington, DC, USA, 30230–. http://dl.acm.org/citation.cfm?id=968880.969243

[30] B. Ramakrishna Rau. 1994. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO 27)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/192724.192731

[31] Sandy Ray and Warren A. Hunt. 2009. Connecting pre-silicon and post-silicon verification. In *Formal Methods in Computer-Aided Design (FMCAD)*. 160–163. https://doi.org/10.1109/FMCAD.2009.5351128

[32] Markus Rullmann, Sebastian Siegel, Renate Merker, Julio A. Oliveira Filho, Thomas Schweizer, Tobias Oppold, and Wolfgang; Rosenstiel. 2007. Efficient Mapping and Functional Verification of Parallel Algorithms on a MultiüContext Reconfigurable Architecture. In *20th International Conference on Architecture of Computing Systems (ARCS)*.

[33] Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. 2011. Automatic Generation of Software-based Functional Failing Test for Speed Debug and On-silicon Timing Verification. In *12th International Workshop on Microprocessor Test and Verification (MTV)*. https://doi.org/10.1109/MTV.2011.19

[34] Dongkwan Suh, Kiseok Kwon, Sukjin Kim, Soojung Ryu, and Jeongwook Kim. 2012. Design space exploration and implementation of a high performance and low area Coarse Grained Reconfigurable Processor. In *International Conference on Field-Programmable Technology (FPT)*. 67–70. https://doi.org/10.1109/FPT.2012.6412114

[35] Miroslav N. Velev and Ping Gao. 2011. Exploiting Abstraction for Efficient Formal Verification of DSPs with Arrays of Reconfigurable Functional Units. In *Formal Methods and Software Engineering*, Shengchao Qin and Zongyan Qiu (Eds.). Lecture Notes in Computer Science, Vol. 6991. Springer Berlin Heidelberg, 307–322. https://doi.org/10.1007/978-3-642-24559-6_22