Random Test Program Generation for Verification and Validation of the Samsung Reconfigurable Processor

Bernhard Egger, Eunjin Song, Hochan Lee, Daeyong Shin

Dept. of Computer Science and Engineering
Seoul National University, Seoul, Korea

Abstract

This paper describes a framework to verify the functional correctness of the Samsung Reconfigurable Processor, a dual-mode very-long instruction word (VLIW) and coarse-grained reconfigurable array (CGRA) processor integrated in smartphones, cameras, printers, and SmartTVs. Reflecting the reconfigurable nature of the processor, the test generator constructs an architectural model directly from the processor’s high-level hardware description files. Test programs are specified in a custom Constraint Specification Language that allows full coverage tests as well as verification of corner cases. Operations and operands are mapped to the heterogeneous processing elements of the CGRA using a guided place-and-routing algorithm. Requiring no explicit knowledge about the semantics of operations, the presented framework seamlessly supports custom ISA extensions. Experiments demonstrate that the framework 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.

Keywords: Coarse-grained reconfigurable array processors, random test program generation, verification, validation

Extension of Conference Paper: this article is an extension of a paper presented at LCTES ’18 [1]. Additional material includes a more detailed description of the test generation framework, the test generation language, and additional results with different architectures.

Email address: {bernhard,eunjin,hochan,daeyoung}@csap.snu.ac.kr (Bernhard Egger, Eunjin Song, Hochan Lee, Daeyong Shin)
1. Introduction

Over the past decade, Samsung Electronics has developed its own reconfigurable general-purpose accelerator called Samsung Reconfigurable Processor (SRP) [2]. Originally based on the ADRES design [3], the processor has evolved into its fourth generation running at frequencies up to 1.2GHz with a configurable number of processing elements. The SRP has been integrated into a number of products such as smartphones, cameras, printers, and SmartTVs and is used for tasks such as audio and video decoding and convolutional neural network (CNN) processing [4, 5, 6, 7, 8, 9]. The SRP is a dual-mode processor operating either in VLIW mode or in CGRA mode. This duality allows the processor to process control-flow intensive code in VLIW mode and execute loops in CGRA mode using the power of all PEs. A custom C compiler seamlessly generates code for both modes and inserts the necessary glue to switch from VLIW to CGRA mode and vice-versa. Data, i.e., live-in or live-out values between the two modes are passed using a central data register file or the processor’s on-chip memory, eliminating the need for costly off-chip memory data transfers as typically necessary for separate accelerators. Figure 1 shows a block diagram of the Samsung Reconfigurable Processor.

Coarse-grained reconfigurable array processors are attractive accelerators in the embedded systems domain due to their programmability, large raw computation capabilities at low energy consumption, and reduced development cost [3, 10]. The SRP CGRA operates in data-flow mode and consists of an array of processing elements (PEs) and register files (RFs) connected by an interconnection network. In addition to the dynamic reconfigurability when routing data through the on-chip network, the implementations also offers 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 [3, 5]. The entire toolchain from compilers over debuggers to simulators seamlessly supports the reconfigurable nature of the SRP.
The reconfigurable nature of the hardware poses special challenges for functional verification and validation. Existing formal methods proving functional correctness \cite{11, 12} are limited with respect to the wide range of reconfigurable options. For this reason a compiler-based approach is adopted: the system under test (SUT) executes the 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 \cite{13}. 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, RTPG for CGRAs not only have to generate valid instruction sequences 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. In this work, a modified edge-centric modulo scheduling algorithm for commercial CGRA compilers \cite{14} is employed to achieve these goals.
Another requirement for the presented RTPG is seamless support for custom ISA extensions. Existing RTPGs [13, 15, 16] require the semantics of every instruction to be defined in an architecture description language (ADL). This work raises this abstraction by only requiring the syntax of the intrinsics but not their semantics.

While the presented RTPG for the Samsung Reconfigurable Processor supports both the VLIW and the CGRA mode of the processor, this article only describes the design and implementation of the CGRA part; verification of the VLIW mode is out of the scope of this paper. The RTPG is evaluated on the SRP CGRA in terms of coverage of the generated test programs. The experiments show that the presented 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 Samsung Reconfigurable Processor.

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

• An RTPG for modulo-scheduled CGRAs for pre-silicon verification and post-silicon validation is presented.
• Seamless support for custom ISA extensions is achieved by requiring only the syntax but not the semantics of intrinsics.
• A detailed description of the Constraint Specification Language (CSL) used to generate versatile test programs.
• High coverage is achieved through adopting a modified edge-centric scheduling algorithm.

The paper is organized as follows: Section 2 introduces CGRAs. Section 3 discusses the design and implementation of the RTPG. The test description language and the verification framework are outlined in Sections 4 and 5. An evaluation using the generated test programs is given in Section 6. Section 7 discusses related work, and Section 8 concludes this paper.
2. The SRP Coarse-Grained Reconfigurable Array

CGRAs are low-power, high-performance accelerators typically used in embedded systems to provide image, video, and audio processing [3]. This work considers the CGRA mode of the SRP that, like most CGRAs, operates 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 [17] is used. This section first provides a general overview of 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 heterogeneous in their architecture and functionality. Register files (RF) provide temporary and fast storage of data values, constant values are generated by constant units (CU). Figure 2 shows an example of a CGRA with twelve PEs, five register files (labeled DRF/PRF), and
two CUs. Unlike traditional ISAs, operand values are not directly encoded in an instruction. Instead, 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 the predicate interconnection network (dotted lines in Figure 2).

The code to control the execution of the array is stored in the **configuration memory**. The configuration memory is a wide memory organized as a number of **configuration lines**. Each configuration line specifies the execution plan of the CGRA for one cycle. Each component of the CGRA is controlled by a specific part of the configuration line. That is, the configuration lines contain the opcodes for each of the PEs, the read/write register addresses for each RF, the constant values to be generated by the CUs, and the mux control signals for the interconnection network. It is not uncommon for configuration lines to be wider than 1000 bit [18, 19].

### 2.2. Execution Model

CGRAs execute software-pipelineable loop kernels [20] in data-flow mode. The kernel of a loop is encoded as a number of configuration lines and stored in the configuration memory. The **loop stop** hardware entity controls termination of a loop. When the loop stop input signal goes low, the CGRA ends the currently executing loop and returns to VLIW mode. Because the SRP CGRA supports predicated execution, no special loop prologue and epilogue code has to be generated. Instead, the loop kernel is directly executed on the CGRA. Predicate values, generated by the compiler and routed through the on-chip predicate network, control execution of operations during the loop prologue and epilogue phases. The interested reader is referred to DRESC [21] for details.
Figure 3 illustrates the process of encoding a data-flow graph into the configuration lines of a CGRA. Figure 3 (a) shows the available hardware units in this example, comprising two PEs, PE01 and PE05, a constant unit CU, and a register file RF. The code to be executed repeatedly adds the value 5 to register r1; Figure 3 (b) shows the data-flow graph of the loop body. Figure 3 (c) gives a possible execution plan for this data-flow graph. The constant unit CU, required to generate the value 5, is only connected to PE05, hence the addition operation is placed on PE05 in cycle 2 of the loop body (marked blue). Similarly, because the register file RF is not directly connected to PE05, the value of r1 needs to be routed through PE01 in cycle 1 (marked red). The result of the addition is written back to the register file in cycle 3 (shown in green). In addition to placing the operations onto the PEs at the given cycles, the compiler also needs to configure the data routing between the components. In cycle 1, the value of register r1 propagates through read port 3 of the register file and the connected multiplexer to PE01’s first input port i0. The mov operation on that PE effectively forwards the value of r1 to the output port. In cycle 2, the value at the output port of PE01 is routed through the multiplexer to the first input port of PE05. The value 5 is produced by CU and forwarded to the second input port. The add operation on PE05 generates the desired result by adding r1 to the value
5 at its output port after one cycle. In cycle 3, the result at PE05’s output port is sent to write port wp0 of the register file; this also requires setting the register address (r1) and the write enable signal (we) of the register file’s write port wp0.

For the sake of simplicity, the loop index variable computation and termination condition as well as the related routing of predicate values are not shown; a detailed description of this topic can be found in [21]. The 3-cycle execution plan is encoded into three configuration lines as illustrated by Figure 3(d). Since the CGRA operates in data-flow mode, the three lines are repeatedly executed until the loop stop condition becomes true. Actual compilers overlap the execution of individual loop bodies using the modulo scheduling technique [17, 22].

All components of the CGRA operate in lock-step, and there is no control flow except loop control. A stall caused by a memory access causes the entire array to stall. The hardware does not provide 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. In this context, reconfigurability refers to the at-design time reconfigurability of the array. Typically, CGRAs are defined by a high-level description of the architecture [3]. 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 snippet of C code [2, 3].

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 at
which cycle. Other than single-issue or VLIW processors and as demonstrated by Figure 3, the routing of data values through the interconnection network is also programmable. 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. The CGRAs targeted in this work are typically so complex that even simple loops are difficult to map by hand and a modulo scheduler is required.

2.4. The Samsung Reconfigurable Processor

The SRP is a dual-mode VLIW/CGRA processor based on the ADRES architecture [23]. 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. Typical instances range from 2x2 PEs with two register files and two constant units to 8x8 PEs with 36 register files and 32 constant units [9, 24]. The architecture description of the most commonly used 4x4 SRP design comprises 16 PEs, 11 register files, 8 constant units, 314 multiplexers, and around 1250 wires in the interconnection network. Note that these numbers do not include internal routing components (wires, multiplexers) inside PEs and RFs.

Figure 4 shows an excerpt of a simplified 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 port, \(wp0\), 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 whose definition is given in a
<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 4: High-level SRP architecture description.
separate file (not shown). Line 27 shows the declaration of a constant unit that is able to generate a 17-bit constant. The multiplexers and latches are defined in lines 31–33. The interconnection network, finally, is defined by a list of connections between ports of hardware units shown in lines 36–40.

Three separate files define the syntax, the semantics, and the properties of individual operations. An operation’s syntax defines the number and the types of the input and output operands plus the operation’s latency, whereas its semantics defines the effect of the 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 5 contains short snippets of the three files defining operations. Figure 5 (a) shows the syntax definitions of three general purpose and one user-defined operation. The latency attribute defines the latency of the operation. The syntax node specifies the type and number of input and output operands, and the class node contains a list of classes, further defined by the property file, that describe the properties of the operation. Figure 5 (b) lists the semantics of a user-defined operations CUSTOM_MIN in C. Figure 5 (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.

The definition of the binary encoding of VLIW instruction and CGRA configuration lines is given in separate high-level configuration files that are not shown here for the sake of brevity. These files define the opcodes of PE operations and the selection signals for the various multiplexers and read/write ports of the entire architecture along with their exact position in the binary encoding.

This high-level textual architecture definition is supported by the entire SRP toolchain (compiler, debuggers, simulators, and the presented RTPG). To compile an application for a different architecture, a programmer simply selects the
Figure 5: High-level operation specification.

(a) Syntax declaration

```plaintext
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

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

(c) Operation properties and constraints
architecture and recompiles the source code to build the application binary, debuggers, and simulators capable of debugging/simulating the selected architecture. Likewise, the presented RTPG respects the architecture definition and generates test programs for the selected architecture.

For the reminder of this work, the following parameters of the CGRA are assumed to be reconfigurable:

- the number of PEs, RFs, and CUs
- 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/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 sections describe the test program generation process in detail.

3. Test Program Generation

Traditional RTPGs [15, 25, 26, 27] are not well-suited to support modulo-scheduled CGRAs for a number of reasons. First, the vast majority of test generators aim at single-issue microprocessors. Test code can be structured as an arbitrarily long sequence of instructions during which the individual components of the processor can be tested one-by-one. Modulo-scheduled CGRAs, on the other hand, are loop accelerators and have limited capabilities to execute long sequences of sequential code because of the constrained size of the configuration memory. RTPGs modulo-scheduled CGRAs for must aim at exercising all hardware configurations in as few cycles as possible and make sure the program can be executed iteratively. Furthermore, scalar processors automatically resolve hardware hazards by delaying/reordering operations in the instruction
stream; consequently, RTPGs for such architectures do not need to consider hazards. Because CGRAs do not provide hazard-resolution at the hardware level, the RTPG is responsible to generate instruction sequences that are hazard-free.

Second, existing RTPGs require the semantics of every operation in the ISA to compare the test result with the pre-computed outcome. Advanced features, such as Genesys-Pro’s biased results [13], also necessitate the semantics to be known. The reconfigurable nature of CGRAs allows extensive customizations for specific application domains. Part of that customization are custom operations that range from simple operations such as saturation to operations of high complexity. Existing frameworks for CGRAs and RTPGs that support custom ISA extensions require that the semantics of such custom operations are provided in an ADL [15, 28]. To support custom instructions seamlessly, the presented framework does not require the semantics of an operation to be known; valid test programs can be generated based only on the syntax and properties of an operation (see Section 2.4). This design has the advantage that architecture designers and application programmers can easily extend the ISA and test the effects of custom operations without requiring the know-how of a hardware engineer. This flexibility, however, means that the presented RTPG is not able to pre-compute the results of the generated test program. This imposes a number of limitations on the use-cases of the RTPG such as (a) the inability to generate tests that explore the operation of the CGRA with specific input values and (b) in the context of predicated execution, if a PE’s predicated execution signal has been generated by another PE as a result of an operation, the RTPG cannot predict whether the operation on that PE will be executed or not. In the context of validating/verifying the SRP CGRA, these limitations are less of a concern because the main focus lies on generating schedules that fully exercise the entire architecture in terms of all possible data and predicate routes and less on testing the correct operation of PEs with specific inputs (this, in fact, is performed by a different tool that is not part of this work). To ensure that the execution predicate of PEs is known, the presented RTPG can generate schedules that do not use computed predicates.
A third limitation of traditional RTPGs is that generating valid operation sequences for a CGRA requires a more complex scheduler than 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 arrive at the PE’s input ports at the correct time. This is achieved by routing the data from the data sources through the interconnection network to the PE’s inputs. As a consequence, an RTPG for CGRA processors 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 schedulers for RTPGs cannot easily cope with the massive increase in complexity and suffer from long test program generation times.

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

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

1. All possible routes through the interconnection network are exercised
2. All possible operations are executed on each PE
3. Data operands are routed through the interconnection network
4. The generated test program is hazard-free
5. Support for user-defined instructions
6. No knowledge of the semantics of operations is required, only their syntax and properties.
3.1. Test Generation Engine

The RTPG for the SRP can generate exhaustive tests for different entities of the hardware such as exercising all connections in the data or predicate interconnection network, reading or writing all registers of a specific RF, selecting all available inputs of a multiplexer at least once, or exercising all supported operations on a specific PE. It can also be used to explore corner cases around special operation sequences or data routes in the network by specifying the operation sequence and/or data routes and letting the RTPG generate random operation sequences and data flows around this corner case. Tests are specified in the Constraint Specification Language (CSL) that is discussed in Section 4.

At the core of the presented RTPG lies an edge-centric place-and-route scheduler \cite{22, 29} that operates on a graph representation of the architecture \cite{21}. This architecture model is a directed acyclic graph (DAG) generated directly from the high-level architecture description. The nodes of the graph represent physical ports or virtual entities that can accept or deliver a data value. The edges represent physical wires or logical connections across the time domain; the weight of the edge represents its delay. Physical wires have a zero weight (no time delay) whereas edges representing operation latencies, the delay of latches, or a register holding its value across clock cycles have a weight bigger than 0. An edge connecting the ALU of a PE with its output port, for example, has a variable weight that is set to the latency of the scheduled operation.

Figure 6 shows the graph models used to represent PEs, RFs, CUs, and MUXes. Gray boxes represent input/output ports, edges represent connections. Ports for operation selection, register address selection and mux control ports are not shown. The number of input/output ports, the number of registers in the RFs, and the width of connections vary according to the high-level hardware description. Entities are connected through edges representing the data and predicate networks. The latency of unlabeled connections is 0 cycles. The edge connecting the (virtual) node representing the operation of a PE, shown as a circle in Figure 6, is set to the latency of the scheduled operation. Each register in a register file contains two edges with a delay of one cycle; one to model the
fact that a register can be read one cycle after being written to, the other one to model a register can keep its data across clock cycles.

A test schedule for the SRP CGRA is generated by the following three steps:

1. Map loop control operations and data routes.
2. Map user-specified routes and/or operations.
3. Exercise architecture with randomly generated operations and data routes.

In the following, the individual steps are described in more detail.

3.1.1. Mapping Loop Control Operations and Data Routes

This first step is necessary to generate the loop control signals for modulo-scheduled CGRAs. It maps a simple for loop with a configurable but constant number of iterations to the CGRA. Since the CGRA operates in data-flow mode, the RTPG generates an operation sequence that increases the loop counter by 1 in every loop iteration and compares its value to the loop iteration bound. The result of the comparison is a predicate (true/false value) that is routed to the loop stop hardware entity that, once the predicate goes from high (true) to
low (false), exits the loop and switches the processor back to VLIW mode. The mapping of the loop control signals is performed first to ensure the necessary computational and data routing elements are available.

3.1.2. Mapping User-Specified Routes and/or Operations

In a second step, user-specified routes and operations are mapped to the architecture (see Section 4.6). Such constraints allow the explicit specification of data routes and operation placement such as, for example, the specification of a data route from a PE’s output port at a given time $t_1$ to a data input port of some other PE at time $t_2$ or a certain operation $\mathbf{op}$ to be executed at time $t_{\mathbf{op}}$ on a specific PE (an illustrative example is given in Figure 13). User-specified routing and operations are specified through schedule constraints (Section 4.1); similar to Step 1, these constraints are mapped first to ensure maximum availability of the required hardware resources.

3.1.3. Mapping Randomly Generated Operations and Data Routes

In this last step, the scheduler generates a random sequence of operations and data routes. The random test scheduler places operations one-by-one until the desired length of the schedule has been reached. Operation scheduling on a CGRA is a two-step process consisting of placing the operation on a PE followed by routing the input operands. In random test programs, the data sources of the required input operands can determined randomly; this allows the RTPG to select data sources and routes that exercise specific parts of the architecture.

The main loop of schedule generation is shown in Algorithm 1. Inputs to the scheduler are a model of the architecture-under-test and scheduling constraints as defined by a verification engineer (Section 4). The algorithm outputs a data flow graph representing the random test program that is transformed into binary code by an assembler. Lines 1–2 place operations cycle-by-cycle, one PE at a time, until a schedule of the desired length $\text{timeLimit}$ has been generated. A configurable OperationGenerator (see Section 3.2) selects a candidate operation to be placed on a given $\text{PE}$ at a given time $t$ (line 6). The RTPG then checks
Algorithm 1 Random test program generation loop

Input: Architecture and test generation constraints
Output: Random test program

1: for $t = 1$ to time_limit do
2:     for all PEs do
3:         success = false
4:         tries = 0
5:         repeat
6:             $op = OperationGenerator.GetCandidateOperation(PE, t)$
7:             success = $PE.CanPlace(op, t)$ and $op.ConnectInputs()$
8:             tries = tries + 1
9:         until success or (tries == MaxTries)
10:        if success then
11:            $OperationGenerator.ConfirmOperation(op)$
12:            $op.ConfirmRoutes()$
13:            $PE.Place(op, t)$
14:        else
15:            $PE.Place(NOP, t)$
16:        end if
17:     end for
18: end for

that placing the operation $op$ on $PE$ at time $t$ does not cause any data hazards on the involved input and output ports and tries to find suitable data providers that can satisfy the inputs of the operation (line 6, see also Section 3.3). This process is repeated until an operation is found that can be placed and have its inputs routed or up to $MaxTries$ times (lines 8–9). If an operation is found, it is placed on the PE and the data routes confirmed (lines 11–13). If no suitable operation can be found, a nop operation is scheduled on the $PE$ at time $t$ (line 15).

Figure 7 illustrates one step of Algorithm 1. An ADD operation with a latency of one cycle and an operation syntax $(\text{int}:32) \leftarrow (\text{int}:32, \text{int}:32)$ has been selected by the OperationGenerator (line 6). The method $PE.CanPlace(op, t)$ checks that its input ports at time $t$ and the output port at time $t + 1$ are free. Next, two suitable data sources for the two input operands are searched by $op.ConnectInputs()$. The syntax of ADD dictates two inputs 32-bit ints.
For each input port, the routing algorithm finds a data source (see Section 3.3 and Algorithm 2). Starting at the input port of the PE, the algorithm follows connection edges backwards in time through the interconnection network until suitable data providers have been found. Data source 1 (src1) in Figure 7 is provided by an output of type int generated by a previously placed operation at time t − 1 (blue data path). For the second input src2, a value available in cycle t − 2 at the output of a PE is selected. The value arrives at input 2 and the desired time t after crossing a local register file that holds the value for one cycle (red data path). Since all input can be connected, success is set to true in line 6 in Algorithm 1 and operation and its routes are confirmed.
3.2. Operation Selection

Operations selection is performed 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 $t + \text{latency}(op)$ 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. Different operation selection methods that determine what operations are available and with what probability are provided (Section 4.4).

Operation selection has a large impact on the test coverage. To quickly achieve high coverage in the interconnection network, the operation selection gives more weight to operations that exercise yet unexercised ports of a PE.

3.3. Routing Algorithm

Once an operation has been selected for a given PE, a routing algorithm based on an edge-based modulo scheduler [22] connects the required inputs of the operation to existing data providers. For a given input operand at time $t$, the sink, the task of the routing algorithm is to find an available path through the 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 total latency of $\delta t$, i.e., the route goes backwards in time. In terms of the DAG model of the architecture, edges are traversed in reverse direction and the sum of all traversed edges’ weights represents the total latency $\delta t$. Blindly following all possible connections would lead to an exponential fanout and long test program generation times. To prune the exponential search space, the scheduling algorithm uses a priority queue to prioritize edges with a weight $> 0$. The search for potential data providers is stopped after a predefined number of candidates has been found. The candidates data providers are then evaluated in terms of the test policy and the best provider is selected.
The search algorithm is guided by parameterizable priority functions that determine the priority of each connection and hardware port. The presented RTPG currently allows to set priorities for yet unexercised connections and ports; the more unexercised components a route is composed of, the higher its priority is. For exhaustive testing of the interconnection network, for example, choosing the provider whose route includes the most unexercised edges allows the RTPG to quickly reach a full coverage in the interconnection network.

Algorithm 2 illustrates the candidate search function \textit{ConnectInputs()} called in line 7 of Algorithm 1. \textit{ConnectInputs()} iterates over all input operands of an operation (line 1). A priority queue contains the end points of the search; an end point is represented by a 4-tuple \textit{(port, time, weight, hops)} where \textit{port} and \textit{time} represent the end point of the route. \textit{Weight} is the priority of the end point, and \textit{hops} counts the number of segments of the route to the end point. For each input, the priority queue is initialized with the 4-tuple \textit{(in, t, 0, 0)} where \textit{in} represents the physical port of the input and \textit{t} is the scheduling time of the corresponding operation (line 3). The algorithm repeatedly selects the head of the queue, i.e., the route that so far has the highest priority (line 6) and follows all connections leading into that port backwards to their sources (line 7). For each connection, the \textit{head} of the connection and the current time \textit{htime} are obtained (lines 8–9). To prevent an unbounded search, a maximum number of hops can be specified for a route after which the search is stopped (line 10). If the port \textit{head} provides the required data type, it is added to the list of candidates and the route is not followed further (lines 11–13). If \textit{head} does not provide the requested data type and is unoccupied at time \textit{htime}, the algorithm computes its priority and weight, and adds it to the priority queue (lines 14–16). The search ends when the priority queue is empty or the configurable maximum number of data provider candidates has been found (line 5). Finally, in lines 20–26, the candidate with the highest priority is selected and the route recorded in the DFG. In case no candidate was found, all routing is undone and the function returns \textit{false}. If a suitable data provider was found for each input of the operation, the function returns \textit{true}. 
Algorithm 2 Operation::ConnectInputs()

Input: operation, architecture model, priority function, MaxCandidates, MaxHops

Output: return true and save route if found

1: for all inputs in of operation do
2:   datatype = in.GetDatatype()
3:   queue = {(in, t, 0, 0)}
4:   candidates = Ø
5:   while (queue not empty) and (|candidates| < MaxCandidates) do
6:     (port, time, weight, hops) = queue.PopHead()
7:     for all connections con out of port do
8:       head = con.GetHead()
9:       htime = time – con.Latency()
10:      if (hops == MaxHops) then continue end if
11:     if (head provides datatype at time htime) then
12:       weight+ = con.Priority()
13:      candidates.insert(head, htime, weight, hops + 1)
14:     else if (head free at htime) then
15:       weight+ = con.Priority() + head.Priority()
16:      queue.insert(head, htime – head.Latency(),
17:                    weight, hops + 1)
18:     end if
19:   end for
20:   if (candidates not empty) then
21:     (port, time, weight, hops) = candidates.PopHead()
22:     ApplyRoute(in, t, port, time)
23:   else
24:     UndoRouting()
25:   return false
26: end if
27: end for
28: return true
3.4. Data Type Tracking

A distinctive feature of the presented RTPG is that the semantics of operations need not be known. Every route of a value from a source to a sink is annotated by a data type. Integer and float types denote integer and floating point numbers of unknown values. These values 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 pointers to memory regions. Together with a value of known integer type, the RTPG can generate valid memory accesses that satisfy the constraints (typically alignment restrictions) given by the operation's syntax. A special data type unknown is used to represent outputs of predicated operations whose predicate is unknown at test generation time. In this case, the RTPG cannot determine whether the output will be computed or the operation is blocked by the predicate (refer to the following section). To avoid propagation of values of type unknown, the routing algorithm never selects such values as data sources.

3.5. Predicate Network Testing

The predicate network is independent from the data interconnection network and enables predicated execution of operations on a PE. If the predicate delivered to a PE through its pred_in port is true, the scheduled operation is executed normally. If, however, the predicate is false, execution of the operation is suppressed, effectively replacing it with a nop operation. The routing algorithm presented in Section 3.3 and Algorithms 1 and 2 can be employed in unmodified form to also perform routing for the predicate network. This is possible because the routing algorithm is aware of the bit-width of connections and can thus support 1-bit predicate values without any changes.
4. Test Template Language

There seem to exist almost as many test template languages as there are RTPGs. The initial implementation of the SRP RTPG used a dedicated test template language; however, two obstacles were observed:

1. Using the RTPG in an industrial environment involves a significant learning curve that may be prohibitive.
2. Even minor feature additions require extensive modifications to the RTPG framework, from parsing the template over modifying the RTPG core itself to maintaining backwards compatibility with previous versions.

To avoid these difficulties, the presented RTPG takes a different approach: test templates are written in plain C++ using a constraint specification language (CSL) library. The template is compiled and linked with the RTPG into an executable. This CSL library effectively represents the API of the RTPG.

The code in Figure 8 shows a complete example of a SRP RTPG test template. The template generates a random CGRA schedule of 50 cycles (lines 4 and 7). The LoopIterationConstraint defines the number of times the modulo schedule is executed (line 8). The MemoryConstraints in lines 11 and 12 define memory ranges containing a specific type of value; additionally, the range of the initial values can be specified using value generators such as the FloatGenerator shown in line 13. The RandomOperationGenerator (line 18) is an operation generator that randomly selects operations based on weights. Operation generators can be global or local (i.e., assigned to individual PEs) to allow for more specific tests. In the example, a global operation generator is defined that generates logical and ALU operations with the same probability (lines 18-21). For PE00, a separate operation generator is defined that generates operations from the mem_load group (lines 24-28). When selecting an operation (line 6 in Algorithm I) local operation generators defined for specific PEs take precedence over the global one.

The CSL library is built around the concept of constraints. The CGRA CSL library is consequently designed around that principle: any class in the
void CCEGen::Configure(void)
{
  // define the type of the schedule
  Schedule *s = Add(new RandomCGASchedule());

  // Code constraints: 50 cycles, loop 100 times
  s->Add(new NoOfCycleConstraint(50));
  s->Add(new LoopIterationConstraint(100));

  // Memory constraints
  s->Add(new MemoryConstraint(0, 400, CSL::dtInteger));
  MemoryConstraint *m = new MemoryConstraint(0, 200, CSL::dtFloat);
  m->Add(new FloatGenerator(0.0f, 100.0f));
  s->Add(m);

  // Operation generator constraints
  // global operation generator constraint
  OperationGenerator *og = new RandomOperationGenerator();
  s->Add(og);
  og->Add(new OperationGroup("logic"), 100);
  og->Add(new OperationGroup("alu"), 100);

  // local operation generator constraint for PE00
  og = new RandomOperationGenerator();
  og->Add(new OperationGroup("mem_load"), 100);
  EntityConstraint *e = new PeConstraint("pe00");
  e->Add(og);
  s->Add(e);
}

Figure 8: Test program generation template written using the CSL library.
CSL library is a descendant of the CSL_API::Constraint superclass. Figure 9 shows the class hierarchy diagram of classes that are descendants of the CSL_API::Constraint class.

Constraints often need to be aggregated. The class CSL_API::ConstraintContainer implements a constraint container class that accepts instances of the CSL_API::Constraint class as additional constraints. The CGRA CSL library models compounds that can contain subconstraints as a subclass of CSL_API::ConstraintContainer.

4.1. Schedule Constraints

The basic unit of test program generation is a schedule. A schedule represents a sequence of random operations generated by satisfying all constraints of the hardware and the verification engineer. Schedulers in the CGRA CSL are all descendants of the CSL_API::Schedule class and implement scheduling for the different modes of the SRP (VLIW, CGRA).
Through constraints a variety of the schedule’s properties can be modified such as, but not limited to:

- **General schedule properties.** General schedule properties control basic properties of the generated code such as the length of the generated schedule, the number of loop iterations, or the routing parameters. The CSL exposes these properties through *code constraints*.

- **Hardware units to exercise.** Especially in CGRA processors, it may be desirable to test only specific parts of the entire array. The CSL’s *architecture constraints* allow control over which hardware entities are available for scheduling.

- **Operation constraints.** These constraints define so-called *operation generators* which are invoked by the CSL scheduler whenever a new operation needs to be generated. By constraining the operation generator further, the set of operations to be generated can be defined specifically for individual PEs.

- **Operand constraints.** Operand constraints control what kind of constant values are generated. Constant values generators are required to initialize the global register files and all available memory areas with values. Additionally, during scheduling, value generators provide constant values generated on the CUs.

- **Memory constraints.** Memory operations access on-chip or off-chip memory. The *memory constraints* allow the definition of what memory areas are available for testing.

- **Route constraints.** Route constraints allow the creation of specific test cases by defining data dependencies between a data source and a data sink. A route constraint defines a source, zero or more intermediate entities, and a sink. The RTPG scheduler generates a route carrying data from the source via all specified intermediate entities to the sink. Route constraints are discussed in detail in Section 4.6.
4.2. Router Constraints

Router constraints control the fanout of the generated routes and the selection of candidate routes (Section 3.3 Algorithm 2). A `RouterConstraint` takes a total of five arguments as shown in Figure 10. The first two arguments limit the fanout of the routing algorithm. Lower values restrict the search space more and result in a higher test program generation speed. Higher values allow the scheduler to explore more routes and candidates and result in a lower generation speed, however, in general the quality (i.e., the coverage) of the generated code is higher. The first argument, `max_hops`, limits the number of hops of a route. One hop is defined as routing through one port. The second argument, `max_candidates`, defines how many potential data provider candidates are searched before a route is selected. The third and fourth argument control the priority of candidate routes. A high value for `reuse_bonus` favors routes that connect to already existing data routes instead of using a register or a constant value as its source. The `newness_bonus` defines how much the router favors routes that have not yet been exercised. The fifth and last argument, finally, defines whether the same data provider can be used for more than one input operand of the same operation. All arguments have pre-set default values as shown in the listing in Figure 10. Figure 11 shows two data-flow graphs generated with different `RouterConstraint` parameters. With no bonus for unexercised connections, the routes tend to be as short as possible. With an increased bonus for unexercised connections and disallowing the identical data source to provide data for both input operands, the generated routes are much longer and comprise many more connections and entities of the architecture.
4.3. Architecture Constraints

Architecture constraints testing of selected parts of the architecture. Especially for CGRA processors, it may be desirable not to test the entire array. The **ArchConstraint** controls which hardware entities are available for scheduling. Hardware components that become unreachable as a result of restricting the test to specific entities are removed by a garbage collection pass. Figure 12 shows the effect of applying an **ArchConstraint** on the exercised architecture.

4.4. Operation Constraints

Operation constraints define which operations a schedule contains. The task of randomly selecting an operation is performed by an **operation generator**. Currently, two variants of operation generators are implemented: fully random and weighted random operation generation. Both accept probabilities for operations to be generated. While the random operation generator selects operations using an independent random variable, the weighted random operation generator uses a dependent random variable ensuring that operations are selected exactly as defined by their probabilities.
(a) 4x4 CGRA architecture with 16 PEs, 11 register files, 8 constant units, 314 multiplexers, and interconnections.

(b) 4x4 CGRA architecture restricted to 4 PEs. Required and reachable components are automatically included (4 register files, 2 constant units, and 91 multiplexers).

Figure 12: Effect of architecture constraints on the exercised architecture model.
Figure 13: CGRA schedule with varying constraints for different hardware entities.

4.5. Operand Constraints

Even though CGRA processors do not encode operands directly in the instruction words, there are situations when immediate operand values need to be generated. In addition, register files and memories need to be initialized with meaningful values of a certain type. Operand constraints provide immediate operands to the schedule by defining what type of value and what value ranges are generated and available in register files and memory blocks.

4.6. Route Constraints

These constraints define data dependencies or place operations in a schedule. This may be desirable to test a specific corner case. Route constraints are scheduled first, before any other operations are inserted into the schedule (Step 2 in the test generation process; see Section 3.1). Figure 13 shows an example schedule that places three operations and defines a data dependency from the output port of the operation to be scheduled on PE01 at time 4 to an input port to the operation to be placed on PE03 at time 7.
5. 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.

During pre-silicon verification, the RTL checker of the testing framework is employed as outlined in Cho [30]. 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.
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 is cannot be guaranteed that a wrong value on one of the inputs of an instruction affects 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 14 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 (Section 3). 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.
6. Evaluation

The presented RTPG is evaluated in terms of (1) the speed and 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 [2].

6.1. SRP Architectures Instances

The SUT are instances of the Samsung Reconfigurable Processor with 4x4, 3x3, 2x4, and 2x2 PEs [9, 24]. Figure 15 shows the layout of the PEs and the central register file(s). Predicate and local register files, CUs, and the interconnection network are not shown. Details of the exercised architectures are given in Table 1. The columns PE, RF, CF, and Interconnection Network represent physical entities whereas the column Architecture Model gives details about the architectural model used by the RTPG. The PEs support a total of 250 integer, floating point, and custom operations.

![Figure 15: SRP architecture instances under test.](image-url)
6.2. Test Program Generation and Metrics

The different architectures were evaluated with different configurations. Because of the random nature of the RTPG, each configuration is evaluated 20 times; reported are the arithmetic average over all 20 runs. A distinctive feature of the presented RTPG is the ability to create guided tests that aim to exercise all hardware entities and all supported operations; this is controlled by the parameters of the RouterConstraint (Section 4.2). For random routing, routes are selected randomly by initializing the RouterConstraint with the parameters \texttt{reuse\_bonus}=0, \texttt{newness\_bonus}=0, and \texttt{allow\_same\_source}=true (Figure 10). In guided routing, the parameters are \texttt{reuse\_bonus}=1000, \texttt{newness\_bonus}=10000, and \texttt{allow\_same\_route}=false.

The depth of the candidate search is controlled with the \texttt{max\_hops} and \texttt{max\_candidates} parameters; these parameters are varied to explore the effect of the candidate search on test program generation time and coverage. To fit into the SRP’s configuration memory, test programs are limited to a length of 100 cycles. Since it is impossible to exercise all components within 100 cycles, the RTPG generates a sequence of 20 random test programs, each 100 cycles long. The RTPG maintains the statistics on which components have been exercised across individual test programs of a sequence and is thus able to generate tests that fully exercise the entire architecture.

Metrics used in the evaluation of the presented RTPG are the test generation time and the coverage of the interconnection network, the register files (register read/write), the constant units, and the executed operations.

6.3. Test Program Time and Coverage for Different Architecture Instances

Table 2 compares the generated test programs in terms of test generation time and architecture coverage for the different architectures and routing methods. The route search parameters are limited to 64 hops and 64 candidates (columns \texttt{Depth} and \texttt{Cand.}). The column \texttt{Runtime} lists the time required to generate one test program of 100 cycles. The last six columns list the coverage of the entire test program sequence for the interconnection network (ICN), the executed
operations \((Ops)\), and the register files \((RF)\). Coverage in the interconnection network is defined as the ratio of the exercised connections over all available connections. Operation coverage is defined as the average operation coverage of all individual PEs; in other words, the number of executed different operations over the total number of operations a PE can execute. The register file coverage, finally, is the average of the read and write coverage over all registers in all register files. All results report the arithmetic average of 20 individual runs.

The results in Table 2 show that the presented RTPG with guided scheduling is able to fully exercise the interconnection network of all tested architectures. Full coverage is achieved after 400-700 cycles. Register file coverage reaches 100% after 200-300 cycles. Full operation coverage is obtained after 1100 cycles for all configurations. Guided scheduling clearly outperforms random scheduling; random scheduling does not achieve 100% coverage in the interconnection network after 2000 cycles for any architecture. A moderate increase in the test generation time is observed for guided scheduling and with an increasing complexity of the architecture, however, even for the 4x4 architecture with almost 600 nodes and 4000 edges in the architecture model, the test generation time

<table>
<thead>
<tr>
<th>Arch. ×</th>
<th>Configuration</th>
<th>Run-time [sec]</th>
<th>After 1K cycles</th>
<th>Full cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[ICN %] [Ops %] [RF %]</td>
<td>[ICN cycles] [Ops cycles] [RF cycles]</td>
<td></td>
</tr>
<tr>
<td>4x4</td>
<td>guided 64</td>
<td>47.0</td>
<td>100 100 100</td>
<td>700 1000 200</td>
</tr>
<tr>
<td></td>
<td>random 64</td>
<td>37.0</td>
<td>95 99 98</td>
<td>- 1100 -</td>
</tr>
<tr>
<td>3x3</td>
<td>guided 64</td>
<td>32.4</td>
<td>100 99 100</td>
<td>600 1100 200</td>
</tr>
<tr>
<td></td>
<td>random 64</td>
<td>24.5</td>
<td>95 99 100</td>
<td>- 500 -</td>
</tr>
<tr>
<td>2x4</td>
<td>guided 64</td>
<td>26.3</td>
<td>100 99 100</td>
<td>500 1100 200</td>
</tr>
<tr>
<td></td>
<td>random 64</td>
<td>20.7</td>
<td>95 99 100</td>
<td>- 1100 700</td>
</tr>
<tr>
<td>2x2</td>
<td>guided 64</td>
<td>16.8</td>
<td>100 100 100</td>
<td>400 900 300</td>
</tr>
<tr>
<td></td>
<td>random 64</td>
<td>13.9</td>
<td>96 100 97</td>
<td>- 900 -</td>
</tr>
</tbody>
</table>

Table 2: Average test generation time and coverage over 20 runs of a sequence of 20 test programs with a length of 100 cycles for different architectures and configurations. After 1K cycles lists the coverage in the interconnection network (ICN), the operations (Ops), and the register files (RF) in percent. Full indicates after how many cycles the configuration achieves full coverage (’-’ denotes the case when full coverage is not achieved within 2000 cycles).
of a 100-cycle long guided schedule is a moderate 47 seconds on an Intel Xeon E5530 running at 2.4 GHz. Test generation can be improved by reducing the detailed logging and by using a more efficient heap manager.

6.4. Effect of the Routing Algorithm on Coverage and Test Generation Time

The router parameters \textit{Depth (max\_hops)}, \textit{Candidates (max\_candidates)}, and the schedule \textit{Length} have a significant effect on test generation time and coverage. Table 3 shows the results of varying these parameters for the 4x4 SRP architecture and guided scheduling. The first four rows vary Depth, the maximum length of the selected data routes, from 128, 64, 32, to 16 while keeping Candidates and schedule Length fixed at 64 and 100, respectively. Despite the limited depth, the RTPG is able to generate test programs that exercise the entire architecture. At Depth = 16, the coverage of the interconnection network and the operations after 1000 cycles does not reach 100\% anymore, but nevertheless full coverage is achieved within 2000 cycles after 1400 and 1100 cycles, respectively. The effect of the Depth on test generation time is also moderate, suggesting that routes do not typically reach maximal depth because the Candidate limit is satisfied first. The second four rows vary the number of Candidates, i.e., the number of potential data sources for a given sink, while keeping the Depth and the schedule Length fixed at 64 and 100, respectively. The RTPG is able to fully exercise the entire architecture for all configurations. Full coverage is achieved after 700-800 cycles in the interconnection network and 1000-1100 cycles for the operations despite the limited number of candidates, i.e., routes covering the entire architecture are typically found within the first 16 candidates. Unlike coverage, the test program generation time increases significantly from 16 to 128 candidates, further reducing the benefit of searching a large number of candidates. The third group of rows displays results while varying the schedule Length from 100, 50, 32, to 16. This parameter has a strong correlation with the test generation time: at 16 cycles, the test program generation takes 3.6 seconds compared to 47 seconds for 100 cycles. This superlinear effect is caused by the fact that the longer the test programs
Table 3: Effect of router parameters on test generation time and coverage over 20 runs of a sequence of 20 test programs with a length of 100 cycles for the 4x4 SRP architecture. After 1K cycles lists the coverage in the interconnection network (ICN), the operations (Ops), and the register files (RF) in percent. Full indicates after how many cycles full coverage is achieved; ‘-’ denotes the case when full coverage is not achieved within 2000 cycles. Note that the configuration 64/64/100 is repeated in each group for better readability.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Cand.</th>
<th>Length</th>
<th>Run-time</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>[sec]</td>
<td>After 1K cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ICN</td>
<td>Ops</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>100</td>
<td>48.2</td>
<td>100 100 100</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>100</td>
<td>47.0</td>
<td>100 100 100</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>100</td>
<td>41.7</td>
<td>100 100 100</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>100</td>
<td>30.8</td>
<td>99 99 100</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>100</td>
<td>59.1</td>
<td>100 100 100</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>100</td>
<td>47.0</td>
<td>100 100 100</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>100</td>
<td>37.9</td>
<td>100 99 100</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>100</td>
<td>30.8</td>
<td>100 99 100</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>100</td>
<td>47.0</td>
<td>100 100 100</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>50</td>
<td>16.0</td>
<td>99 95 100</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>32</td>
<td>9.0</td>
<td>98 88 100</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>16</td>
<td>3.6</td>
<td>93 66 96</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>100</td>
<td>59.8</td>
<td>100 100 100</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>100</td>
<td>47.0</td>
<td>100 100 100</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>100</td>
<td>32.0</td>
<td>100 99 100</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>100</td>
<td>23.0</td>
<td>99 99 100</td>
</tr>
</tbody>
</table>

get, the more components are occupied so that longer data routes are needed to find max_candidate data providers. A sufficient schedule length is, however, required to allow the RTPG to exercise the full architecture: at 16 and 32 cycles full coverage is not achieved within 2000 cycles. The reason is that it takes a few cycles for valid data to reach “remote” PEs that are not connected to the central register files. The last four rows in Table 3 test the effect of varying the Depth and Candidates together at a schedule length of 100 cycles. Full coverage is achieved earlier for higher values at the expense of a longer test program generation time, but all combinations from 128/128 to 16/16 achieve full coverage within 2000 cycles.
Figure 16: Coverage of guided routing.
6.5. Coverage Analysis

The quality of random test programs is determined by the coverage of a number of criteria. The main focus of the presented RTPG lies on coverage in the data and predicate interconnection network as this criteria is unique to CGRAs. As outlined in Section 6.3 coverage of the interconnection network is defined as the ratio of exercised connections over the total number of connections. It is reasonable 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. Separate coverage ratios for the data and the predicate network are presented. Coverage for register files is defined as the number of registers read from or written to over the registers of all register files. Register file coverage is also analyzed in terms of read 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.

Figures 16 and 17 show detailed results of test programs for guided and random routing on the 4x4 SRP architecture instance (the first two rows of Table 2). The reported results represent the coverage in (a) the interconnection network, (b) the register files, and (c) the operations for a single run, i.e., a sequence of 20 test programs of 100 cycles each with router parameters Depth = 64 and Candidates = 64. With guided routing, unexercised routes prioritize over already exercised ones. Figure 16 (a) shows that full coverage in the data and predicate interconnection networks is reached after 400 and 700 test cycles, respectively. Giving preference to unexercised connections also has a strong effect on register file coverage, shown in Figure 16 (b); this an expected result since register files are modeled using latches and connections. Full read coverage is achieved after the first 100 cycles, full write coverage after 200 cycles. CU coverage (not shown) also reaches 100% after the first 100 cycles. Operation coverage, shown in Figure 16 (c), reaches 100% after 1100 cycles. This is a side-effect of guided routing because the operation generator gives preference to operations that facilitate the testing of yet unexercised connections.
Figure 17: Coverage of random routing.
With random selection, shown in Figure 17, only RF read coverage and operation coverage achieve 100% within the first 2000 cycles. Without prioritizing yet-untested connections and hardware components, the routing algorithm tends to select the closest (both in space and time) data sources leading to short but frequently exercised routes. The results clearly demonstrate the need for a guided random test program generation.

6.6. 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 4x4 SRP 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 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 presented RTPG also discovered two yet unknown errors in the original (and assumed-to-be error-free) implementation of the processor. 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.
6.7. 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.

6.8. Discussion

The presented RTPG for modulo-scheduled CGRAs has been tested with various instances of the Samsung Reconfigurable Processor, but is also applicable to other modulo-schedule CGRAs such as the ADRES architecture \[23\] and its derivatives. The experiments confirm that the RTPG is able to produce test programs that achieve full coverage within a few hundred cycles, making it an excellent candidate for pre-silicon verification where simulation speed may be an issue. The fact that the presented RTPG can generate valid test programs without 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. This poses no difficulty in pre-silicon verification where observability is not a problem. 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 which may be a limiting factor if the test programs generated by the presented RTPG are the sole post-silicon validation step.
7. 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 presented.

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 [25] and Adir [13] 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 [31] uses machine learning techniques to improve the initial stimuli for CSP-based RTPGs. Qin [32] combines the CSP solver and simulation techniques to analyze the real hardware design, this enable to support dynamic array references. Corno [33] and Mishra [27, 16] use graph-based algorithms to generate test programs. While Corno uses a predefined library of instructions, Mishra’s work extracts the structure of the pipelined processor directly from the architecture description language. This model is then fed to a symbolic model verifier. Di Guglielmo [26] proposes a pseudo-deterministic automatic test pattern generator based on extended FSMs. The test vectors are generated using a constraint or SAT solver. Koo [34] also uses an FSM combined with reduction techniques to achieve high coverage with a small number of directed tests. In Sanches’ work [35], 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 [36] 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 [28], Kamkin [15], and Rullmann [37] all present augmented architecture description languages (ADL) to specify reconfigurable designs, however, neither work can be adapted to CGRAs with reconfigurable interconnection networks. Velev’s work [12] is the first to present 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 [23]. 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 [38] and Liu [39] tackle this problem by storing only a small set of trace signals that represent a larger number of states. Adir [40] propose to execute the post-silicon test program on a pre-silicon validator to obtain exact coverage measurements for later comparison. Ray [41] and Adir [42] 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 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 [10], the presented 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.

8. Conclusion

This paper discussed 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.

Acknowledgments

We thank the RP Core Group at Samsung Electronics for their support and the access to the SRP. 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 ), the Neural Processing Research Center of the Samsung Advanced Institute of Technology, 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

URL https://doi.org/10.1145/3211332.3211342


URL http://dx.doi.org/10.1007/978-3-540-45234-8_7


URL http://doi.acm.org/10.1145/1629395.1629433


[9] B. Harris, I. Bae, B. Egger, Architectures and algorithms for on-device user customization of CNNs, Integration doi:https://doi.org/10.1016/j.vlsi.2018.11.001

URL http://dl.acm.org/citation.cfm?id=968880.969243


URL http://dx.doi.org/10.1007/978-3-642-24559-6_22


URL http://doi.acm.org/10.1145/3162018

URL http://doi.acm.org/10.1145/53990.54022


URL http://doi.acm.org/10.1145/1542452.1542456

URL http://dx.doi.org/10.1007/s11265-007-0152-8


URL http://doi.acm.org/10.1145/2155620.2155666


[42] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner,