M.S. THESIS

Coverage-driven Random Test Generation for Coarse-Grained Reconfigurable Architectures

재구성 가능 아키텍쳐의 기능 검증 커버리지 향상을 위한 무작위 테스트 생성

FEBRUARY 2016

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Eunjin Song

M.S. THESIS

Coverage-driven Random Test Generation for Coarse-Grained Reconfigurable Architectures

재구성 가능 아키텍쳐의 기능 검증 커버리지 향상을 위한 무작위 테스트 생성

FEBRUARY 2016

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Eunjin Song

Coverage-driven Random Test Generation for Coarse-Grained Reconfigurable Architectures

재구성 가능 아키텍쳐의 기능 검증 커버리지 향상을 위한 무작위 테스트 생성

지도교수 Bernhard Egger

이 논문을 공학석사학위논문으로 제출함

2015 년 10 월

서울대학교 대학원

컴퓨터 공학부

송 은 진

송 은 진의 석사학위논문을 인준함

2015 년 12 월

위 원 장	문 병 로	(인)
부위원장	Bernhard Egger	(인)
위 원	Srinivasa Rao Satti	(인)

Abstract

Coverage-driven Random Test Generation for Coarse-Grained Reconfigurable Architectures

Eunjin Song
Department of Computer Science and Engineering
College of Engineering
The Graduate School
Seoul National University

As the complexity of hardware designs keeps increasing, functional verification of microprocessor systems has become one of the main bottlenecks in the hardware development processes. Conventional verification methods are difficult to apply to coarse-grained reconfigurable architectures (CGRA) due to their high complexity and complicated requirements on the generated code. This thesis proposes a coverage-driven verification method for CGRAs to test functionalities through randomly generated test programs. The proposed verification is performed by a comparison with simulation results, and is thus suitable for pre- and post-silicon verification. Our random test program generator (RTPG) builds a graph model of the architecture directly from the CGRA's textual description and produces executable random test programs. The proposed RTPG adopts a guided place and routing algorithm to map operations and operands onto the heterogeneous functional units. To achieve maximum coverage, we employ a routing algorithm with various fitness functions and a heuristic approach for operation scheduling. The RTPG supports custom ISA extensions seamlessly without explicit knowledge about the semantics of operations. Experiments demonstrate that the proposed RTPG is versatile in generating test

programs by diverse test templates and quickly achieves a high coverage of the architecture's functionalities. We test the effectiveness of the method on a commercial CGRA, the Samsung Reconfigurable Processor. In a real world evaluation, the generated test programs were able to detect all randomly inserted faults as well as several yet unknown faults in the CGRA architecture.

Keywords: Coarse-Grained Reconfigurable Architecture, Functional Verification, Simulation-based Verification, Random Test Program Generator, Routing Algorithm, Coverage Analysis

Student Number: 2014-21761

Contents

List of Figures

Chapter 1

Introduction

In order to satisfy various requirements of microprocessor systems, several reconfigurable architectures with highly complex designs have been proposed in the recent years. The functional verification of such architecture designs is a crucial stage since it is one of the main obstacles to reduce time-to-market in the hardware design cycle [1, 2]. Formal methods are capable of proving the functional correctness of a higher-level design implementation [3, 4], however, the formal methods are not suitable for post-silicon verification. On the other hand, simulation-based verification is not only important to pre-silicon validation during the design phase of a microprocessor [5], but also to the post-silicon verification of a manufactured chip [6].

In simulation-based functional verification, a *test program* consisting of a sequence of instructions is executed on a microprocessor. The result of the execution is compared to reference values which are calculated by a functional simulator of the microprocessor. The quality of those test programs is estimated by their *coverage* of the microprocessor. There exist different kinds of coverage metrics: instruction coverage, operand coverage, or coverage of the interconnection network and so on. Depending on the purpose of the test, a

certain coverage is more important than others.

In many cases, the test programs for the simulation-based functional verification are generated by *random test program generators* (RTPG). RTPGs create a valid random sequence of instructions based on hardware and user-defined constraints [7]. Hardware constraints are defined as restrictions of the hardware specification, for example, the syntax and latency of instructions. User-defined constraints enable configuration of additional conditions for the test programs. These constraints allow test engineers to perform *directed random testing* such as testing with only specified instructions. This feature in RTPGs is essential to satisfy the various purposes of validation. RTPGs are preferable over high-level compiled programs since compilers usually generate similar code patterns with which it is hard to reach a sufficient coverage for verification.

In this thesis, we describe the design of an RTPG design for Coarse-Grained Reconfigurable Architectures (CGRA) that is suitable for both pre-silicon and post-silicon verification. CGRAs are composed of a variable structure with functional units (FU), register files (RF), and an interconnection network. In general, the components of CGRAs are statically reconfigurable in terms of the number or size, and functionality of those components. CGRAs provide sufficient parallelism and programmability that make them the ideal candidates for processing multimedia data streams in low-power systems.

RTPGs for CGRAs have to compute a valid *routing* of data values because the instruction's encoding of CGRAs does not directly include specific input operands. The input operands are routed through the complex and reconfigurable interconnection network. The proposed RTPG can generate valid test programs by only tracking the *type* of data values. Existing RTPGs [7, 8, 9] have trouble in changing instruction set architectures (ISA) by defining the semantics of each instruction in some architecture description language (ADL) [10]. The proposed RTPG is flexible to support variable instruction set architectures by requiring only the information about the *syntax* and the *class* of an instruction,

but not the exact semantics (with a few exceptions). In the absence of semantics, the proposed RTPG is not able to pre-compute the result of instructions which is needed comparison with the output values of the system-under-test (SUT). Therefore, in order to perform validation by comparison, the reference values are obtained by running the test program on an architecture simulator. These properties allow the RTPG to rapidly produce test programs for a variety of architectures with minimal user intervention.

For CGRAs the measure of coverage should be extended to all possible combinations of instructions. Since input operands are routed through the interconnection network comprising of multiplexers, latches and data connections, an RTPG for CGRAs must strive to exercise all possible routes. The routing process of our RTPG is based on the edge-centric scheduling algorithm [11]. The algorithm's routing decisions are guided by a parameterizable *fitness function* which can be tuned to achieve different results. During the routing process, the traversal concept of the shortest path algorithm is applied to traverse over the graph model which was built from a CGRA architecture description. The fitness function favors unexercised connections which allows the routing algorithm of the proposed RTPG to achieve full coverage in the interconnection network within a relatively low number of cycles. Moreover, we suggest several coverage-driven heuristic approaches to improve the coverage rate of the interconnection network.

We first evaluate the proposed RTPG in terms of coverage of the generated test programs and then run the test programs in a real-world test to detect faults in an actual CGRA chip, the Samsung Reconfigurable Processor (SRP) [12]. Experiments demonstrate that the RTPG quickly achieves a very high coverage not only in traditional measures (instruction coverage, register file coverage, etc) but especially also in exercising all possible routes in the interconnection network. The proposed routing algorithm favoring unexercised connections and the proposed heuristic approach are effective for the coverage

of the interconnection network. During the verification, all randomly inserted faults were detected and a number of yet unknown faults in the original VHDL design and the processor simulator were uncovered.

The contributions of this work are as follows:

- we propose an RTPG for coarse-grained statically reconfigurable architectures that can be used both for pre-silicon verification and post-silicon validation.
- to seamlessly support custom ISA extensions, the proposed RTPG tracks only the *data types* of values, not the values itself. Verification/validation is performed by comparing the computed values and reference values obtained by an architecture simulator.
- we describe a modified edge-centric scheduling algorithm with a parameterizable fitness function.
- we suggest a coverage analysis method for CGRAs and describe some heuristic approaches to achieve a higher coverage rate within a few hundred cycles.
- we demonstrate the effectiveness of the RTPG on the commercial Samsung Reconfigurable Processor. During pre-silicon verification, the generated test programs not only caught all randomly inserted faults but also lead to the discovery of yet-unknown faults both in the RTL and the architecture simulator.

The remainder of this thesis is organized as follows: Chapter 2 states the related work in random test program generation. Chapter 3 gives a brief introduction of CGRAs and introduces the reconfigurable architecture used for this work. It also contains the hardware modeling for the target architecture and fault models for each component of the target architecture. Chapter 4 describes the design and implementation of the RTPG. Coverage-driven heuristic approaches are suggested in Chapter 5. Chapter 6 contains an overview of pre-

silicon verification and post-silicon validation. Chapter 7 shows the result of experiments evaluating test programs generated by the RTPG, and lastly the conclusion of the thesis is provided in Chapter 8.

Chapter 2

Related Work

Due to the ever increasing complexity of chip designs, both academia and industry [7, 13, 14] 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 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, SAT, constraint satisfaction problems (CSP), or graph-based test program generation. Bin [15] and Adir [7] 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 [16] uses machine learning techniques to improve the initial stimuli for CSP-based RTPGs.

Qin [17] combines the CSP solver and simulation techniques to analyze the real hardware design, this enable to support dynamic array references. Corno [18] and Mishra [19, 9] 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 [20] proposes a pseudo-deterministic automatic test pattern generator based on extended FSMs. The test vectors are generated using a constraint or a SAT solver. The test programs exercise the processor at the gate level. Koo [21] also uses an FSM combined with reduction techniques to achieve high coverage with a small number of directed tests. In Sanches' work [22], 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 [23] 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 [24], Kamkin [8], and Rullmann [25] all propose augmented architecture description languages (ADL) to specify reconfigurable designs, however, none of the work can be adapted to CGRAs with reconfigurable interconnection networks. Velev's work [4] is the first to propose a formal verification framework for CGRAs. They apply Equality with Uninterpreted Functions and Memories (EUFM) to abstract functional units and memories while completely modeling the control of the CGRA. The framework is applied to the ADRES architecture [26]. 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 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 scope of the test. Ko [27] and Liu [28] tackle this problem by storing only a small set of trace signals that represent a larger number of states. Adir [29] proposes to execute the post-silicon test program on a pre-silicon validator to obtain exact coverage measurements. Ray [30] and Adir [31] integrate pre-silicon and post-silicon verification. The former [30] partitions pre-silicon checkers with full observability into limited-observability checkers with the same accuracy for post-silicon validation. The latter [31] extend coverage-driven verification methodology to the post-silicon verification domain by using similar 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. Being unaware of instructions' semantics, the proposed RTPG cannot pre-compute the correct values. Instead, testing is performed by comparing the results of the system under test to a reference implementation. In contrast to [29], 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 can be integrated as needed to improve certain aspects of testing.

Chapter 3

Coarse-Grained Reconfigurable Architectures

A Coarse-Grained Reconfigurable Architecture is usually devised for high performance and low power consumption. These properties are suitable for domain specific architecture such as mobile and multimedia embedded markets. In general, the processor architecture of CGRAs consists of the Coarse-Grained Array structure(CGA) and a general processor. The Very Long Instruction Word(VLIW) processor performs the role as the general processor which executes non-parallel friendly parts of a program to improve the performance. In contrast, the CGA processor exploits the higher loop-level parallelism of simple loop structures. These two types of processors may be tightly coupled or loosely coupled[32].

3.1 Target Architecture

The target architecture of the proposed random test program generat or is the Samsung Reconfigurable Processor[12]. The SRP architecture consists of functional units (FU), register files (RF), constant units (CU), and an interconnec-

tion network which connects those components using connections, multiplexers, and latches. The FUs are usually comprised of a 2-D array and often heterogeneous, which means, each FU supports different instruction set. According to the supported instruction set of each FU, the required number of input- and output ports are also different. The RFs can be configured as a different number of registers and read- and write ports. It may also differ in supported data widths. The CUs generate immediate values which could be an input operand of operations on the FUs. The interconnection network connects FUs, RFs, and CUs. Connections can be direct without delay or contain latches with a delay of few cycles. A predicate network comprising of predicate connections and predicate register files also exists to support predicated executions.

CGRAs are statically reconfigurable, in other words, they are reconfigurable during the design phase. The entire tool-chains including compilers, assemblers, simulators, debuggers and verification frameworks should be capable of adapting different architecture configurations within the valid range of the architecture's reconfigurability. The primary component of reconfigurability for the SRP architecture are: the number of FUs, the number and size of the RFs, the number and datawidth of CUs, the interconnection network (including the width of connections), and custom ISA extensions. A conceptual example of a CGRA with 12 heterogeneous FUs, three RFs, one CU and an interconnection network is shown in Figure 3.1. The green colored lines represent the predicate network.

The FUs of a CGRA do not support the control flow except loop control. A configuration memory stores the execution plan of the CGRA in so-called configuration lines. A configuration line indicates one cycle in the execution plan in decoded form. It includes the opcodes for each FU, the RF's write enable and read port signals, the immediate values for CUs, and the selection signal for each of the multiplexers. Although conventional processors directly include the input/output operands information in their instruction like register indices

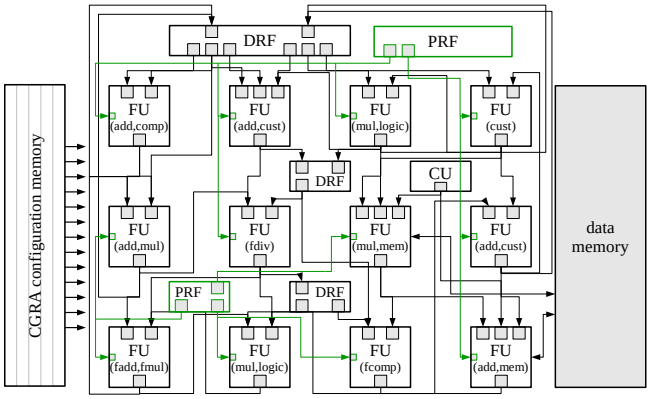


Figure 3.1 Example of a coarse-grained reconfigurable architecture.

or immediate values, a CGRA instruction does not contain the input/output operands. Instead, data of input ports at an execution time are used as input operands of scheduled instruction. For example, when an instruction is scheduled at t on a FU, the available data on input ports of the FU at time t used as the input operands and it generates the result on the output port of the FU at time $t + lat$ where lat stands for the latency of the instruction. If there is no available data at the input ports, the result of the instruction can not be defined.

The important part of the reconfigurability of a CGRA is custom ISA extension, that is able to define a new instruction flexibly. Other than related work[8], the proposed RTPG only requires the syntax of instructions to be defined. The syntax consists of the number and type of input and output operands, the instruction latency, and eventual side-effects of the instruction. The following is the definition of two operations, an integer addition `ADD`, and an integer-to-floating point conversion `I_F`:

```
<op name="ADD"  latency="1"
    class="ALU"
    syntax="(int:32)=(int:32,int:32)"
/>
```

```
<op name="I_F"  latency="3"
    class="ALU"
    syntax="(float:32)=(int:32)"
/>
```

The CGRA framework (compilers, simulators, and RTPGs) reads this description and interprets that the `ADD` instruction has two 32-bit integers as input operands and a 32-bit integer as its result with one cycle latency. The class `ALU` in the definition indicates that the instruction has no memory-related side effects. In the case of the `I_F` instruction, it takes one 32-bit integer input and generates a 32-bit float value after three cycles. The *semantics* of the instruction are not necessary to the CGRA compiler. Such new custom instructions are typically added by the programmer in the form of a high-level source program. In order to simulate custom instructions by the framework's simulators, therefore, an implementation in a high-level language should be provided.

In this work we assume that at least the following parameters of the architecture are reconfigurable:

- the number of FUs, RFs, and CUs in the CGRA
- for each PE: the number and data width for each input and output port and the supported instructions
- for each RF: the number and data width of registers, the number of read/write ports
- for each CU: the data width
- the interconnection network comprising multiplexers, latches, and connections, plus the data width of each component
- the ISA, including custom extensions
- the syntax (and semantics, if necessary) for each instruction

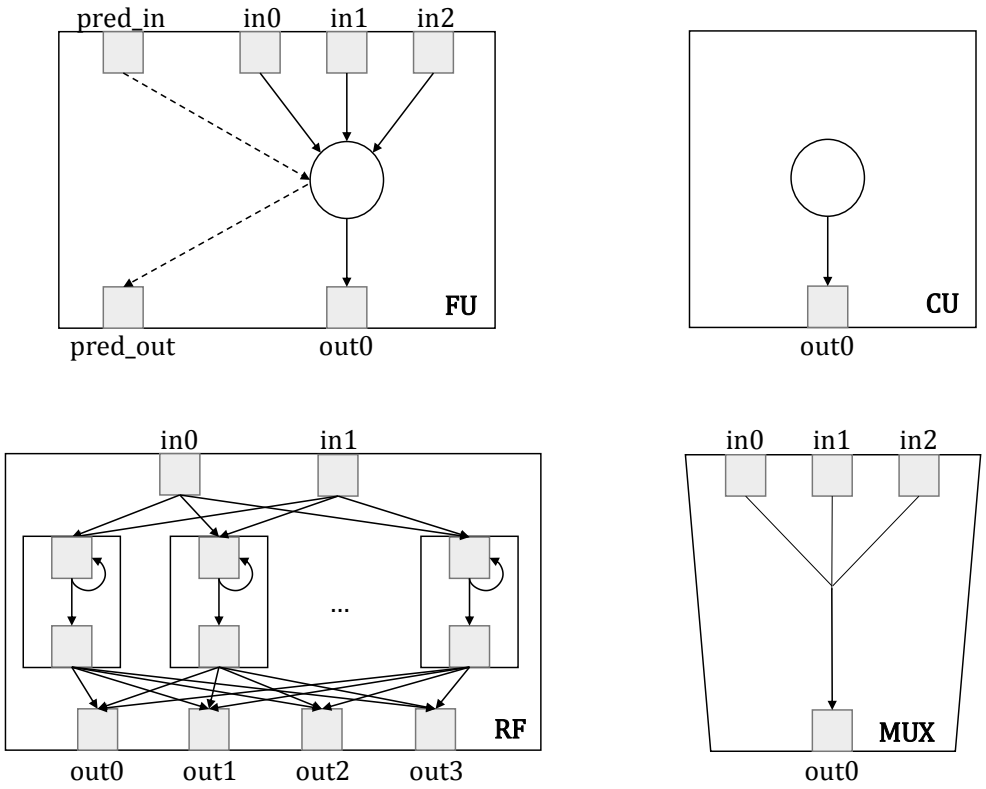


Figure 3.2 Graph models for different hardware entities. From left-to-right, top-to-bottom: FU, CU, RF, and multiplexer.

3.2 Hardware Model

In this section, we describe a graph modeling for components of our target architecture. Figure 3.2 shows the models for FUs, CUs, RFs, MUXs.

The model of a FU has input ports and output ports for ordinary instructions. An operation uses the data of the input ports as its input operands and generates the result into the output port. Because CGRAs are only able to handle loop controls, a branch instruction is performed by predicate operations. There are a predicate input and a predicate output ports for predicate operations. The predicate operations like `cga_pred_ne` generate the result into the predicate output port instead of the data output port. The result can be

provided to a predicate input port of other FUs. The model of a CU represents one output port and a generator. A CU generates a constant for FUs, the configured data width determines the range of the constant. The model of an RF is comprised of input ports, output ports, and registers. The input and output ports of RFs are connected to the read and write ports of its registers respectively. The connection from the write port of a register to itself has 1 cycle latency, this enables that the register maintains its value regardless of cycles. The model of a MUX consists of the input ports and one output port. MUXs are used to route the data among the components of this CGRA architecture. A connection between two components, a source and a sink, can be modeled as a directed edge with the data width and latency properties. For example, the connections in the predicate network have 1-bit data width. We also modeled the data memory virtually. The proposed RTPG uses a value with a type during the verification, so we separate the data memory into several sections by the type of a value.

3.3 Fault Model

In this section, we describe the functional fault model for the CGRA model in the previous section to verify the functionality of each component. First of all, we consider the register files and the memory units.

In the model of the register files, there are multiple input ports and output ports, and each register has the read/write port. The functionality of a register could be defined as reading the register, writing the register and maintaining a value of the register. As we described in the above section, a register file of the CGRA is not a special component like the registers of the general processor. This component is just a part of a path between a value generator (e.g. an output port of a CU) and a consumer (e.g. an input port of a FU). For the functional verification of this register file, we could check values of the registers after reading/writing the registers. In the case of a data memory unit, the

functionality is related to load and store operations. These memory related operations could possibly cause a fault by an incorrect operand, in other words, a memory access with an invalid address leads to an unexpected error. Therefore input operands of a memory related operation should be verified to have proper memory addresses. The functional verification for the data memory units is also performed by comparing their values before and after a store operation.

The next thing is about the value generators, CUs and FUs. A CU usually produces constants for several connected FUs. A possible fault of CUs is generating an incorrect value different from the configured value. This is also detectable by the comparison of the values. The functionality of heterogeneous FUs could be defined as operation executions of supported instructions. During the process of an execution on a FU, an error could possibly occur in the decoding process of a scheduled instruction, calculating the instruction and generating the output value into its output operand. These faults are detected by the comparison between the output value and its reference value of the simulator.

Also, an undefined output would be generated by a predicate operations or an absence of input values. In this case, the RTPG could not verify the fault by the comparison with an expected value since it is not possible to get the expected value. The verification of this undefined faulty operation requires the exact specification of the behavior of the target architecture. The proposed RTPG can generate a test program which makes the undefined values, but it is hard to examine faults of them.

Lastly, the faults in components of the interconnection network may be an incorrect routing of its data caused by a malfunction of MUXs, an unexpected data change during the transfer. The functional verification of these components also could be done by the value comparisons of each port.

3.4 Coverage Metrics

In this section, we introduce several coverage metrics which represent the range of a target architecture covered by a group of test programs. The metrics help to estimate the quality of the test programs, as well as the test program generator. Tracing the metrics during the test program generation and utilizing the information could be used to improve the quality of the test programs. We considered the instruction coverage per FU, read/write coverage of registers, the interconnection coverage. The instruction coverage is measured per FU. By comparing the types of supported instructions and the types of scheduled instruction in test programs, we could calculate the instruction coverage per FU. The register read/write coverage can be estimated using routing information of the test program. If a route contains reading or writing a register, we consider that the register has read or written. Therefore, the read/write coverage of each register could be estimated using the route information of test programs. In the case of the coverage of the interconnection network, a connection is classified as an exercised or unexercised connection with the route information of test programs. We could get the interconnection network coverage just by counting unexercised connections. In addition, the interconnection network is separated into the data network and the predicate network by the data width of its connections. A connection with greater than 1 bit is classified as the data network, otherwise the connection is considered as the predicate network.

For the functional verification of CGRAs, these coverage metrics demonstrate the range of a target architecture which is verified by a group of test programs. Except for the instruction coverage, the register read/write coverage could be a part of the coverage of the interconnection network. The full coverage of the interconnection network assures the verification of faults not only in components of the interconnection network like connections and MUXs, but also in reading and writing of register files, in the constant generation of

CUs, and in the functional operation of FUs. This verification is performed by comparing the reference value of the simulator. The details of the simulation based verification are outlined in Chapter 6.

Chapter 4

Random Test Program Generator

Conventional RTPGs [15, 20, 8, 19] are not well-suited to support CGRAs for a number of reasons: first, the vast majority of test generators aim at single-issue microprocessors. Scalar processors automatically resolve hardware hazards by delaying/reordering instructions in the instruction stream; consequently, an RTPG for such architectures does not need to consider hardware hazards in order to generate valid test programs (of course, triggering hardware hazards to test said functionality is desirable). On the other hand, CGRAs — and VLIW processors — do not provide this support at the hardware level. Instead, the compiler (or the RTPG) is responsible to generate instruction sequences that are hazard-free. Velev [4] formally proves the functional correctness of CGRA's interconnection network; this formal proof, however, cannot be applied to post-silicon validation.

Second, existing RTPGs require the semantics of every instruction of the ISA in order to pre-compute the outcome of the computation. Advanced features, such as Genesys-Pro's biased results [7], also necessitate the instructions' semantics to be known. Existing frameworks for CGRAs and RTPGs that support custom ISA extension require that the semantics of custom instructions

be provided in an ADL [24, 8]. In the proposed CGRA framework, only the instruction's syntax needs to be provided in order for the instruction to be usable (see Chapter 3). An implementation of the instruction is only necessary if the instruction is to be simulated by a functional or binary simulator. This has the advantage that architecture designers and application programmers can easily extend the ISA and test the effect of custom instructions as opposed to traditional CGRA frameworks where extending the ISA requires the knowledge and skills of a hardware engineer.

Third, generating valid instruction sequences for CGRA and, in a limited sense also for VLIW processors, requires a much more complex scheduler than for single-issue microprocessors. One reason is the aforementioned lack of hardware hazard resolution. In addition to that, a scheduler must not only consider whether an instruction can be scheduled on a FU at a given time, but also make sure the input operands are correctly routed through the interconnection network. 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 unacceptably long test program generation times.

In addition to similar requirements as for RTPGs for traditional architectures such as instruction/register file coverage, an RTPG for CGRAs should thus meet the following requirements:

1. the generated test program must be hazard-free
2. data operands need to be routed through the interconnection network
3. the generated test program should exercise all possible routes through the interconnection network
4. only the syntax is required, but not the knowledge of the semantics of

instructions (a notable exception is the side effect of instructions, see Chapter 3)

Since the semantics of instructions are not available, the RTPG's code generation framework must be able to schedule valid instruction sequences in the absence of any knowledge about the instruction.

4.1 Test Generation Engine

The main component of the proposed RTPG is an instruction scheduler operating on a graph representation of the architecture model described in Section 3.2 The random instruction scheduler employs an algorithm based on a compiler technique for CGRAs called place-and-route [33]. Our implementation conceptually follows an improved version of the place-and-route algorithm using edge-centric scheduling [11]. Scheduling an instruction is a two-step process: first, the instruction's operation graph is mapped onto one of the FUs. Input, output and internal ports are checked for hazards. If the placement succeeds, then the instructions input operands are routed to the proper value providers in a second step. Unlike a compiler which is given a data-flow graph (DFG) representing the program, we construct the DFG on-the-fly. For a given FU and a specific time, the instruction to be scheduled is selected randomly. In the absence of a DFG, there are no designated data providers for each of the instruction's inputs; in fact, *any* data provider which supplies valid data of the required type can serve as a potential provider for the input operand. The process of finding concrete data providers for each input operand is outlined in the next paragraph. For instructions with side-effects such as memory operations connecting the inputs to data providers of the correct type is not sufficient and may lead to incorrect results. For such instructions, the scheduler must verify that the memory address formed by the input operands conforms to the alignment requirements and denotes a valid memory range. If the scheduler fails to route one of the

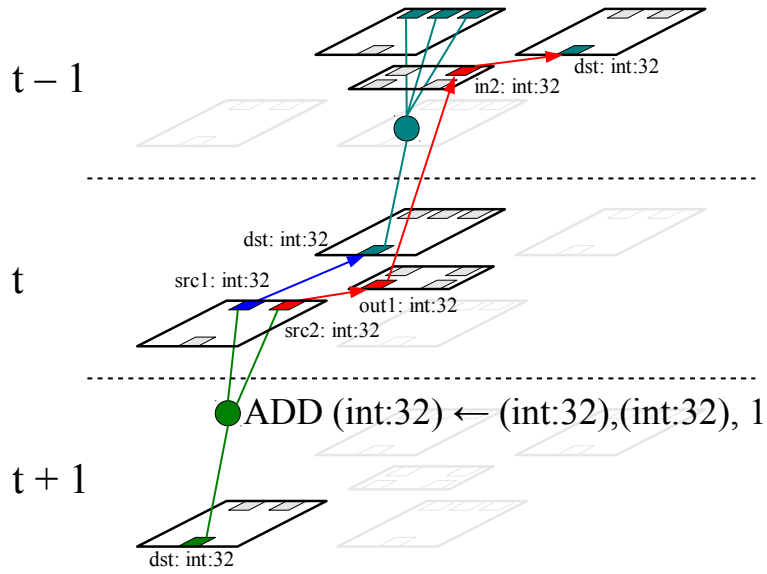


Figure 4.1 Placement of an instruction and routing of its input operands.

input operands, the placement of the instruction is undone and the scheduling process is restarted.

Figure 4.1 illustrates placement of a one-cycle latency `ADD` instruction onto a FU at time t. First, the output (the result) of the instruction is placed on the selected PE's output port at $t + 1$. Then, the first input operand `src1` is routed along all valid connections through the graph. In the example, the input operand is routed to the output port of another FU which produces the result of some instruction at time t. The second operand first enters an RF through an output port, routes the value through a register (not shown), and then continues routing at one of the input ports of the RF at time $t - 1$. From there, a valid datum is found at the output port of yet another FU. If the place-and-route algorithm fails at any stage, any previous placement and routing have to be undone.

The overall scheduling process (with time limit constraint) of the proposed RTPG is summarized in Algorithm 1.

Algorithm 1 Overview of test program generation

```
time ← 0
while time < time_limit do
    for all FUs do
        I ← GetCandidateInstruction(FU)
        if CanPlace(FU, I) and SearchRouteFromInputs(FU, I) then
            Confirm this schedule of < FU, I, time >
        else
            Skip this FU at this time
        end if
    end for
    time ← time + 1
end while
```

4.2 Routing Algorithm

The routing algorithm starts at the sink (an instruction's input operand), and then proceeds backwards in time through the architecture graph. This leads to an exponential fanout: upon entering a multiplexer through its output port, the route can potentially continue through any of its input ports. Similarly for register files: upon entering an RF through one of its output ports, the router can continue at any register. At a specific register, the possible choices are: stay at the register for one more cycle or exit the RF through one of the input ports. An unbounded search would very quickly lead to unacceptably long test generation times. The routing algorithm is therefore guided by a parameterizable *fitness function*. This fitness function can be designed in several ways, depending on the testing purposes. If the goal of the test is full coverage of the interconnection network, for example, the fitness function prefers routes through yet unexercised connections higher over routes that proceed along already employed connections.

As briefly described in the previous section, we construct the DFG during the scheduling process, more specifically during this routing process to find an available data provider for the input operand. From the sink node, the algorithm conducts a backward traversal on the interconnection network by utilizing connections as edges. At each traversal step, it validates that the next port is appropriate for the placement of the sink operand data. After finishing the traversal, the results are several candidate paths from the sink operand to a suitable data provider. Among the candidate routes, we generally prefer the candidate with the highest score of the fitness function. The proposed RTPG employs the traversal concepts of Dijkstra's shortest path algorithm to exploit the fitness function effectively. The advantages of this approach are that it produces only one candidate from the sink to a data provider, and it also prevents unbounded traversals which easily cause high memory utilization and unacceptably long generation times. Algorithm 2 gives an overview of the routing algorithm. The RTPG performs the routing algorithm per input operands during scheduling of an instruction.

In order to apply the routing algorithm, each port with a scheduling time is viewed as a node and each connection is regarded as a directed edge in the data flow graph. Each connection is assigned a weight according to the fitness function; therefore it is important to employ a fitness function to perform the routing algorithm properly. Our plan for the fitness function classifies two types of edges. The first one affects the testing purpose, in other words, improves the coverage rate (unexercised edge). The other type is the opposite; it has no effect on the coverage rate (exercised edge).

Giving a positive weight to exercised edges causes the difference of the evaluation between paths according to the length of each path. In this case, an unused edge among a long route which consists of many used edges is unable to achieve a high priority during the routing process. Therefore, to prefer unexercised connections, the weight of used edges should be *0* and the unused edges

Algorithm 2 Routing algorithm for one input operand

▷ initialization for a *input_operand* on *input_port*

$Q \leftarrow$ initialized priority queue

$T \leftarrow$ scheduling time

$source\_node \leftarrow CreateNode(input\_port, 0, T)$ ▷ port, weight, time

$Q.add(source\_node)$

$C \leftarrow$ empty list for candidate destinations

▷ start traversal from *source_node*

while Q is not empty **do**

  $cnode \leftarrow Q.extract\_min\_weight()$

  ▷ allow visiting a node more than once depending on its total weight

  if $GetMinTotalWeight(cnode) > cnode.weight$ **then**

    $UpdateMinTotalWeight(cnode)$

    if *cnode* can provide data for *input_operand* **then**

      $C.add(cnode)$

    else

      for all e in random-ordered edges connected to *cnode.port* **do**

        $neighbor \leftarrow e.source\_port$

        if $CanPlace(neighbor, input\_operand)$ **and** $GetMinTotalWeight(neighbor) > cnode.weight + e.weight$ **then**

          $tnode \leftarrow CreateNode(neighbor, cnode.weight + e.weight, cnode.time - e.latency)$ ▷ port, weight,time

          $Q.add(tnode)$

        end if

      end for

    end if

  end if

end while

have *a negative weight*. This configuration guarantees that if there is any chance of increasing the coverage, the routing algorithm can generate the candidates including one or more edges which have a negative weight.

Although Dijkstra's shortest path algorithm does not cope with negative weights, we fixed the traversal algorithm to allow that one node can be visited more than once depending on its weight (Algorithm 2 Line 8). The total weight refers to the summation of the weights of all edges on a route from the source node, and the minimum total weight of every node is traced during the routing process. The $GetMinTotalWeight(cnode)$ function returns the minimum total weight of a given *cnode* that was last updated. When the total weight of the current *cnode* is smaller than the return value, the node is allowed to be visited again, and minimum total weight of the node is updated.

Moreover, we assume that a given architecture description does not include connections representing a cycle with no latency that is nonexistent in real architecture. Therefore, the data graph model can be viewed as a directed acyclic graph.

As a result, the routing algorithm can produce candidate routes achieving high coverage despite the negative weight of edges. It allows us to get the maximum coverage within fewer test programs than a random routing algorithm which has a very low probability of generating those candidates. If such a candidate does not exist, the algorithm can consider all possible paths equivalently in a random manner. It produces the candidate paths regardless of the length of a possible path, because the weight of edges in the path is 0. Furthermore, it considers outgoing edges in random order.

4.3 Data Type Specification

During the routing process, a type tracker tracks the types of all data values that are flowing through the architecture model. In the current implementation a total of six different types are tracked: integer, known integer value, float,

known float value, memory base address and unknown type.

The *integer* and *float* types denote integral and real numbers of unknown values, respectively. Integer instructions typically take two integer input operands and compute an integer output, while floating point instructions take floating point values as inputs and compute a floating point value. For such instructions, the actual value of the input/output data is not important to generate a correct instruction sequence. *Known integer value* and *known float value* types are used whenever the result of a computation must be predictable. This is, for example, the case for values involved in loop boundary checks. For memory operations, the type *memory base address* provides a pointer to a valid memory region and its size. The RTPG inserts code that initializes the global register files with values of these five types at the beginning of each test program.

The *unknown* type is introduced to indicate a result of an operation with a predicate input. The result has an undefined value, therefore, its type should be changed to some other type (*unknown*), in order that the type tracker can distinguish the result as an uncertain value. The next section contains details of the predicate network testing.

4.4 Predicate Network

As we described in Section 3.1, our target architecture has the predicate network to support the predicate operations. Scheduling instructions and routing from their operands are not sufficient to cover the predicate network so that the RTPG produces additional routes from predicate input ports and write enable signals of register files.

After an instruction scheduling is completed on a FU, we try to route the predicate input with probability 1/4. In the routing process, the same routing algorithm of the data network is adopted. The predicate input data is usually from special MUXs (one, zero, ..) or a predicate output of FUs. If a route from a data source to the predicate input port is found, the result of the instruction

on FU at the time will become an undefined value. The output value should not be taken as a source of other instructions which will be further scheduled.

Our RTPG prevents from the situation by changing the type of the result into another type(`dtUnknown` meaning that this value can not be chosen as a source of an instruction. Since our routing algorithm is based on a data type, the simple change ensures that this value will not be selected. The proposed RTPG can configure the size of a test program by the number of instructions or the number of total cycles. When it is configured by only the number of total cycles, the RTPG performs predicate network testing. It is required to generate a loop skeleton for CGRAs at initialization time when the predicate network testing is set, because after generating a test program, it is not guaranteed that the loop skeleton can be created. Therefore, we only do the predicate network testing if the limit of total cycles is given.

A write enable signal for register files is also generated through the predicate network. This signal is necessary to run a test program on the real machine. The write enable signal is required when a route includes a write port of register file. After selecting a candidate route during the routing algorithm, the candidate is checked whether the route includes a write port of register files or not. If the route contains a write port of a register file, we try to route a write enable signal from the write enable signal port of the register file using a new backward router. If the process fails, the next candidate with the second priority is considered as a new candidate. Generating a write enable signal for all routes slows down to reach the maximum coverage of the interconnection network, however, this process also can exercise the write enable signal ports and related connections.

4.5 Test Template Language

There exists a template language for each RTPG to tune a generated test program. We first defined a test template language of our RTPG, however, we noticed problems with this approach:

1. using a test template language takes a lot of learning effort in a real environment.

2. minor changes of the architecture even require extensive modifications to the RTPG framework which could be parsing the template, supporting the modifications in the RTPG framework, and maintaining backwards compatibility to previous versions.

Because of the reasons, we took a somewhat different approach: test templates are written in C++ using a constraint specification language (CSL) library. A test engineer can define test templates by using the CSL library which represents the API of the RTPG.

The code in Figure 4.2 shows a test template. The test template generates a random CGRA schedule with 100 cycles. The `NofIterationContraint` inserts a loop skeleton and configures the number of loop iterations. The `RouterConstraint` determines the routing process and the fitness function we described in the previous section. `MemoryConstraints` is utilized to define memory ranges with a type of its values. It also supports that the initial values of the range could be specified using different value generators; `FloatGenerator` in this test template. The `RandomOperationGenerator` is a kind of the `OperationGenerator` which specifies the candidate operations and its weight during the scheduling phase. The `RandomOperationGenerator` selects an operation randomly based on the weight of operations. The `OperationGenerator` can be configured on all FUs or one FU. In this example, the RTPG only considers the operation group of `cga_load` on the `fu00`.

When the `ConnectionHistoryConstraint` is set, our RTPG maintains the history of exercised connections; it first reads the exercised connections from the log file before the generation process, and after the generation of a test program it saves the updated exercised connections into the log file. The history enables that the RTPG can generate a series of test programs with a higher coverage

```
void CCEGen::Configure(void)
{
  // define the type of the schedule
  Schedule *s = Add(new RandomCGASchedule());

  // maintain connection information for coverage
  s->Add(new ConnectionHistoryConstraint());

  // Code constraints
  s->Add(new NofCycleConstraint(100));
  s->Add(new LoopIterationConstraint(10));
  s->Add(new RouterConstraint(128, 128, 100, 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("add"),  100);
  // local operation generator constraint
  og = new RandomOperationGenerator();
  og->Add(new OperationGroup("cga_load"), 100);

  EntityConstraint *e = new FuConstraint("fu00");
  e->Add(og);
  s->Add(e);
}
```

Figure 4.2 Example of a test template written using the CSL library.

through the routing algorithm. This functionality is required to achieve a high coverage; especially so for CGRAs where the maximum length of a test program is typically limited to a few hundred cycles only. The CSL library provides many other types of constraints to allow more elaborate test generation, however, the details of them are not the subject of this thesis.

4.6 Stuck-at fault testing

The versatile test template language enables the proposed RTPG to generate a test program for a well-known fault model, stuck-at fault 0 or 1. For the stuck-at fault 0 testing, the interconnection network could be verified by flooding values of 0 into them. Since the RTPG is not aware of the semantics of instructions, it could not generate a test program for a stuck-at-fault testing by itself. However, a verification engineer who knows about the semantics could utilize the information with provided constraints. In order to verify the fault model of our target architecture, a test program only composed of instructions whose result as 0 if and only if all inputs are 0. These instructions could be configured by the `OperationGenerator` constraint. Furthermore, initial values of register files and memories should be initialized to 0 and CUs also must generate only 0. The `IntegerGenerator` constraint can indicate those initialization and configuration values as a given range of values. In terms of the stuck-at fault 1 testing, the approach is same as the stuck-at fault testing 0. However, a test program generated by this way does not cover memory-related components in CGRA. Because memory-related instructions usually take a base address or memory offset as its operands, these instructions are not applicable for a stuck-at fault 0 or 1 fault model.

Chapter 5

Coverage-driven Analysis

5.1 Available Coverage Analysis

An available coverage of a test program generator is an important metric for test program generators. It is required to evaluate the quality of a test program which would be generated by them. Because RTPGs generate a test program randomly, the evaluation is based on the metric.

In addition, as CGRAs could be reconfigurable, their available coverage could be changed as the modification of their architecture. Moreover, the proposed RTPG provides various types of constraints directly affecting the available coverage. For example, when a test template is configured some operations having less than two input operands, the connections connected to second and third input port of a FU have no chance of exercising. In this case, the available coverage of the test template becomes smaller. We need to figure out such available coverage to estimate the range which is covered by a given constraint set.

5.1.1 Getting available coverage

The available coverage usually depends on a supported instruction set of each FU, in detail, a type and a number of operands of each instruction. Even if the ISA indicates that an FU supports an instruction, there are many cases in which the instruction can not be scheduled. The input ports may structurally not be connected to any of data sources, or there does not exist available data sources of operands. Therefore, we needed to figure out that each instruction can be scheduled on FUs as a given ISA and constraints before calculating the maximum coverage. Algorithm 3 represents the measuring process of the coverage.

This algorithm consists of three parts. The first part is flooding the data from CUs, register files and special MUXs. From output ports of the components, the algorithm starts a graph traversal with the data type and width information and saves the information to every port on all paths of the traversal. That information of the ports represents that each type and width of data can be reachable to each port. As we described above, instructions should be verified that the instruction can produce an output on specified FUs before flooding the data information of the output. This is the second part of the algorithm. It iteratively checks the instructions whether the instruction can generate the output on each FU. If it found an instruction which can newly produce the output, it floods the output data from the output port of the FU into the interconnection network. These process continue iteratively until it can not find that kind of instruction. Then, the last part is the backward traversal from each input port of FUs with the data type and width of operands of schedulable instructions. For each traversal step, the next neighbor port should have the same history as the information of the source operand. During this backward traversal, all visited connections are marked as an exercisable connection. Using this history, we can get the maximum coverage of the interconnection network.

Algorithm 3 Available Coverage Analysis

```
▷ first part
for all e in CUs, Registerfiles do
    for all p in output ports of e do
        flood (d_type, d_width) from p on the network
    end for
end for
▷ second part
while any new schedulable instruction has updated do
    for all FU in FUs do
        for all i in instructions do
            if (i, f) is in schedulable_set or i is not schedulable on the FU then
                skip i
            end if
            insert (i, f) to schedulable_set
            for all p in output ports of i do
                flood (d_type, d_width) from p on the network
            end for
        end for
    end for
end while
▷ third part
for all f, i in schedulable_set do
    for all o in operands of i do
        p ← o-th input port of f
        backward traversal from p with (d_type, d_width)
            ▷ all visited connections of this traversal are reachable
    end for
end for
```

5.2 Improvement of Coverage Rate

We described the coverage-driven routing algorithm in Section 4.2, however, our RTPG still strives to reach full coverage of the interconnection network within proper cycles. One reason is that there is less chance of using some input ports of a FU. This is caused by the supported instruction set of a FU. Thus, we adopt a heuristic approach for the instruction selection process which is described in the following section. Furthermore, we present several templates for getting the specific range of the coverage using provided constraints.

5.2.1 Instruction Selection Strategy

In the proposed RTPG, the instruction selection is basically processed as given constraints. There are two ways of the instruction selection: random or weighted. If a test engineer wants to generate a test program with some specific instructions for some other purpose, like the functionality of FUs, it would not be a problem that only the constrained instructions are selected in the test generation process. However, for the full coverage of the interconnection network which is an important metric of CGRAs, the random selection method is not sufficient to reach the maximum coverage within a reasonable time. For the interconnection network coverage, usually a few connections are covered with a very low probability. These connections are often related to the instructions which have more than two operands. In general, there exist much more instructions having less than three operands in an instruction set. Apart from that, it is obvious that the utilization of first two of the input ports is higher than last two of the input ports. It is because the operands of an instruction are typically mapped into the input ports of an FU consecutively: the first operand is mapped into the first input port. In addition, not all selected instructions could be scheduled. A selected instruction is actually scheduled when a selected FU supports the instruction and the data sources of all operands are available. This

means that the more there are operands in an instruction, the less probability of scheduling it has. Therefore, if the instruction selection process proceeds randomly, it is not easy and takes a long time to attain the full coverage of the interconnection network. In terms of the improvement of the coverage, we decided to give a more weight to instructions which have an enough number of operands to exercise uncovered connections in the interconnection network. To determine where and how to give a more chance, we calculate how many uncovered connections each input port of FUs connected to. Getting the information is similar to calculate the maximum coverage. For each FU, it can be estimated by a backward traversal from each input port of FUs based on schedulable instructions. If there are any unexercised connections during the traversal, the starting input port is indicated the candidate of bonus port. The highest port number among the candidates will be the bonus port of each FU. Using this bonus port information, when our RTPG try to select instruction for an FU, we give a more weight to instructions which use the bonus port of the FU. We do not always choose those instructions; we select an instruction by default after several tries. By using this approach, we can choose more instructions which have a probability of increasing the coverage than the other instructions. Figure 7.2 in experiment section demonstrate that this strategy is very effective to reach the full coverage within a few test programs.

5.2.2 Directed Test Program Generation

When it takes a long time to get the full coverage, it can be divided into several sections, and reach the full coverage of the sections respectively. This strategy may be effective on some cases, but not all cases. In this section, we show that the interconnection coverage can be split into the data connection network and the predicate connection network. A test engineer is able to generate the test programs for various purposes by using the provided constraints. For the data connection network, the constraint set of the test program just excludes the

predicate network testing. Only testing the data network has a higher coverage rate of the data network than testing both data and predicate network since the output result of an operation becomes an invalid value if the predicate input of the FU which scheduled the operation is routed at the same time. The invalid output could not be a source of an operand of other instructions, therefore, scheduling an instruction has less probability than only testing the data network. This may lower the coverage rate of the data connection network as well as the interconnection network. For the predicate network coverage, the strategy testing on the divided network respectively is more effective. This is because that predicate operations generating its result on the predicate output port have not enough chance to scheduling if all supported operations are considered at the instruction selection phase. Figure 5.1 shows the constraint set for the predicate network testing. The `OperationGenerator` constraint could be set candidate operations and its weight. The operation group of `cga_pred` and `cga_fpred` indicates the group of predicate operations, they are normally composed of comparative operation generally requiring two input operands. Therefore we also additionally insert the `MemoryConstraint` and the operation constraint of `cga_mov`.

Those constrained operations and the routing for predicate input ports are enough to cover the predicate network. The predicate network of the target CGRA are composed of the predicate input ports, predicate output ports, and "one" and "zero" MUXs and write enable signal ports of the register files, and connections between those components. All FUs supporting predicate operations could generate the result into its predicate output port, and connections in the predicate network exercised during the routing process for predicate input ports. The connections which are related to write enable signals also could be exercised during the routing from input operands for predicate operations. Those constrained operations are only considered in the scheduling process, as a result, the coverage rate of the predicate network could rise rapidly than testing

```
void CCEGen::Configure(void)
{
  // define the type of the schedule
  Schedule *s = Add(new RandomCGASchedule());
  s->Add(new RouterConstraint(128, 128, 1024, 1024));

  // History constraints for the connection coverage
  s->Add(new ConnectionHistoryConstraint());

  // Code constraints
  s->Add(new NofCycleConstraint(50)); // for the predicate input routing

  // Memory Constraints (not necessary for the predicate network testing)
  s->Add(new MemoryConstraint(0, 400, CSL::dtInteger));
  s->Add(new MemoryConstraint(600, 400, CSL::dtFloat));

  // Operation generator constraints
  OperationGenerator *og = new RandomOperationGenerator();
  s->Add(og);
  og->Add(new OperationGroup("cga_pred cga_fpred"),100);
  og->Add(new OperationGroup("cga_mov"),  100);
}
```

Figure 5.1 A test template for predicate network testing.

both networks.

We evaluated these test templates in terms of the coverage rate. The result in Section 7.3 demonstrates that this technique is effective for the full connection coverage especially the coverage including the predicate network. In fact, a test engineer should understand the semantics of instructions and the structure of a target architecture to verify the architecture in this way. However, it is almost not necessary to modify the verification framework thanks to various kinds of provided constraints. The full connection coverage by a shorter length of total cycles could be achieved with a proper combination of test templates.

Chapter 6

Verification Framework

In this chapter, we describe the verification framework in terms of the pre-silicon and post-silicon verification.

6.1 Pre-silicon verification

For pre-silicon verification, states of internal registers and the architecture are accessible by the VHDL simulator. We adopted the RTL checker of the testing framework as shown in Cho [13]. The RTL checker is a kind of the RTL simulator, but it compares the values in each data port with expected values calculated by a cycle-accurate functional simulator. Each value on data ports of the RTL checker and the cycle-accurate functional simulator are compared every `n` cycles, so the greater value of `n` makes that the verification process is completed faster. If a fault is found during the comparison, the framework proceeds the cycle-by-cycle verification from the last valid checkpoint in order to check the exact position of the fault.

The RTL checker and the cycle-accurate functional simulator require semantics of instructions. The SRP framework allows that a developer defines a

new custom instruction in itself. In order to add a custom instruction, the SRP framework demands the syntax of the instruction and C code which indicates the same operation of the custom instruction. The compiler in the SRP framework can not schedule the custom instruction directly, instead, the compiler schedules the corresponding segment of the C code as a general function call. The SRP toolchain links the code segment to the simulators to execute the custom instruction on the simulators.

6.2 Post-silicon verification

For the post-silicon verification, it is unable to read registers while a test program is running on the CGRA. Alternatively, the framework appends the additional code to each generated test programs so that the values of all registers after executing a test program are saved in memories. Then post-silicon verification can be performed by comparing between the values and reference values which are obtained by the functional CGRA simulator. Although the output values of scheduled instruction are utilized as input operands of next instructions or stored in registers or memory, the proposed RTPG is not aware of the semantics of instructions; it can not guarantee that a wrong input value of an instruction will affect the computed value of the instruction.

6.3 Operation of Verification framework

Figure 6.1 displays the main operation of the verification framework. The verification framework receives the CGRA architecture description files and the test template which define a specific CGRA instance. First, the RTPG builds an architecture model, then generates a test program based on the architecture model and the test template. The SRP toolchain builds a cycle-accurate functional CGRA simulator using the same architecture description, and a hardware engineer also constructs the RTL model of the architecture.

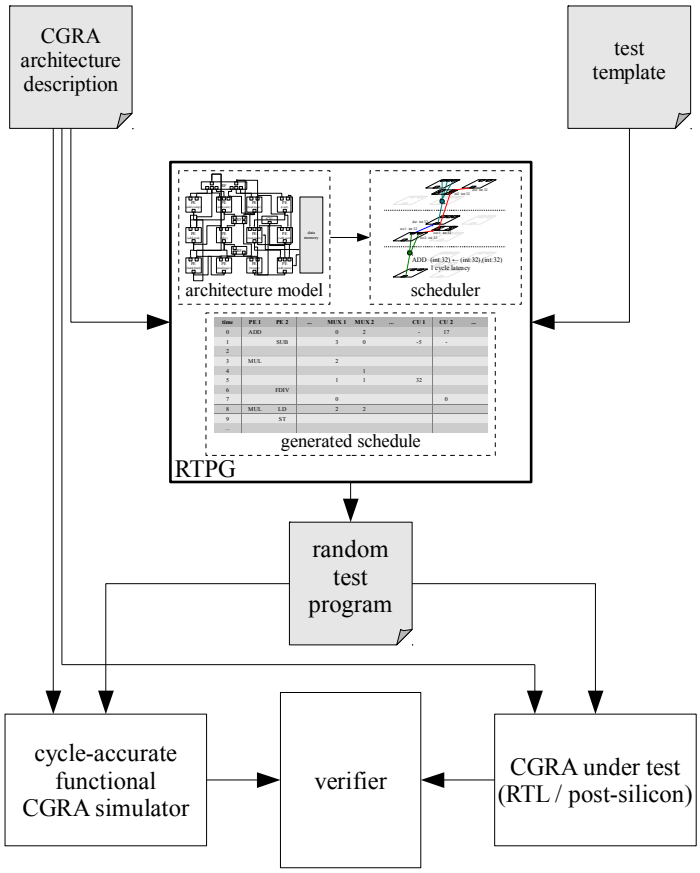


Figure 6.1 Operation of the verification framework.

In order to perform the pre-silicon functional verification, the RTL model runs on the RTL simulator. The cycle-accurate functional CGRA simulator and RTL simulator operate simultaneously, the verifier conducts the comparison of the values between two simulators. The values are on the input/output port of FUs, the output port of CUs, register files, and MUXs on the interconnection network and so on. In the post-silicon verification, the random test program generated by the RTPG are utilized to examine the integrity of the interconnection network. The cycle-accurate functional CGRA simulator stores the result of the test program to compare with the actual output of the test program on a manufactured chip.

Chapter 7

Experiments

We evaluate the proposed RTPG in terms of (1) the quality of the generated test programs and (2) the fault coverage achieved when running the test programs on a commercial CGRA implementation of the Samsung Reconfigurable Processor [12].

The quality of the generated tests is defined by the coverage of test programs. We measure the accumulated coverage over a series of 20 random test programs (50 cycles per test program) that are generated by one test template. We repeated the evaluation for each template 50 times and calculated the average of the measured values between evaluations. The evaluations with multiple short test programs are because of hardware constraints; the target architecture has a small size of configuration memories. The system under test is an instance of a SRP architecture with 4x4 FUs, 6 RFs, 8 CUs and an interconnection network comprising a total of about 5000 connections and 400 multiplexers. The FUs are heterogeneous and support a total of about 250 integer, floating point and custom instructions.

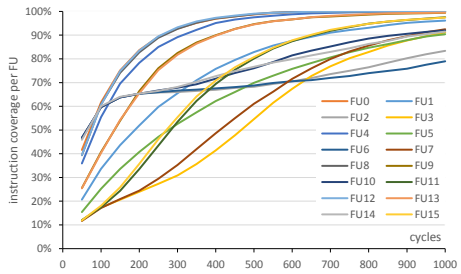
7.1 Coverage of Test Programs

The test template was configured to generate test programs of 50 cycles each, and a global operation generator randomly distributed all available instructions to the FUs. Also, for the improvement of coverage of the interconnection network, the tests were run with the instruction selection process considering the coverage of the interconnection network described in Section 5.2.1.

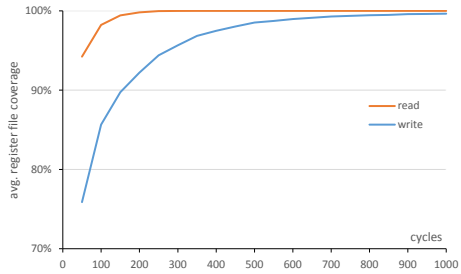
Figure 7.1 demonstrates the need for specialized routing algorithms for RTPG. For the left side figures (Figure 7.1 (a),(b) and (c)), we configured the fitness function to no favoring for unused connections. The right side figures (Figure 7.1 (d),(e) and (f) are obtained by the fitness function favoring unused connections.

The left side figures (Figure 7.1 (a), (b) and (c)) present the limit of the random routing process in terms of getting the full coverage. The coverage of the interconnection network affects the coverage of instructions per FU since we optimized the instruction selection process for the coverage of the interconnection network. The RTPG continuously try to schedule the weighted instructions to increase the coverage rate of the interconnection network until it reaches the maximum coverage. Therefore, the instruction coverage of Figure 7.1 (a) are generally lower than the instruction coverage of Figure 7.1 (d). A more elaborate instruction selection algorithm can achieve full instruction coverage within fewer cycles; however, the goal of this experiment is coverage of the interconnection network. The fitness function of this expriment is configured to value all connections equally, whether exercised or not, so the routing process of the proposed RTPG only proceeds in a random manner. Therefore the read/write coverage over all register files (Figure 7.1 (b)) and the coverage of the physical connections (Figure 7.1 (c)) have increased very slowly and it seems to be hard to achieve the full coverage.

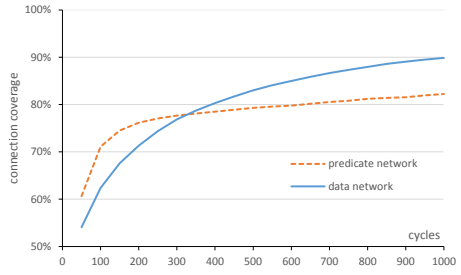
In Figure 7.1 (d), (e) and (f), the same test template is evaluated with



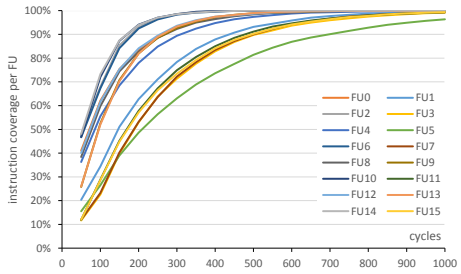
(a) FU instruction coverage



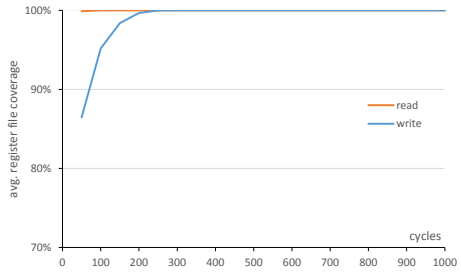
(b) RF read/write coverage



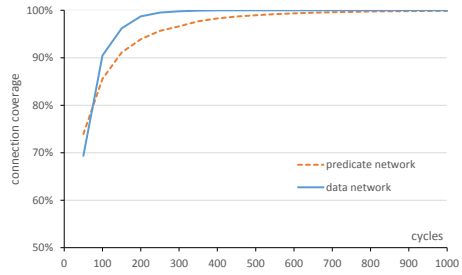
(c) Connection coverage



(d) FU instruction coverage



(e) RF read/write coverage



(f) Connection coverage

Figure 7.1 Individual instruction coverage per FU, average coverage of the RFs and the interconnection network of a test set with a fitness function that does not favor new connections ((a), (b) and (c)) and one that gives a higher weight to routes containing unexercised connections ((d), (e) and (f)).

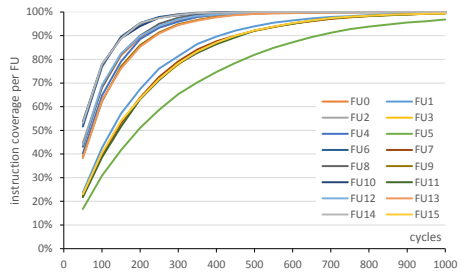
modified parameters for the fitness function. The modified parameters to the fitness function give higher priority to routes containing yet unexercised connections than routes whose connections have already been exercised. The effect of this is clearly visible in Figures 7.1 (d), (e) and (f). The instruction coverage in Figure 7.1 (d) shows a better instruction coverage for the individual FUs than in Figure 7.1 (a). The register file coverage achieves 100% and 84% for read and write coverage within 50 cycles and then reach to 100% for both reads and writes with 200 cycles. Similarly, the number of exercised data connections achieves already 90% after 100 cycles and almost all data connections was exercised within 300 cycles. For the coverage of predicate connections, the experiment in Section 7.3 shows more improved coverage with the directed test template described in the previous chapter.

The results clearly show that the proposed RTPG achieves a very high coverage for all measures within 300 to 400 cycles. The and CU coverage (not shown) achieves 100% after the 100 cycles. During this experiment, some unreachable connections was found; an analysis of the unexercised connections revealed that it is physically impossible to utilize these connections with the current ISA: the unexercised connections connect several FU's third input operand port to various multiplexers, however, there are no instructions to be scheduled that use more two input operands. These unreachable connections also detectable by the available coverage analysis in Section 5.1.

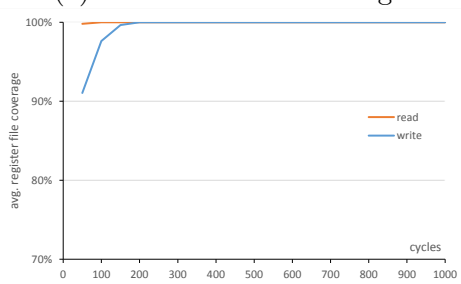
7.2 Improvement of Coverage Rate

7.2.1 Instruction Selection Strategy

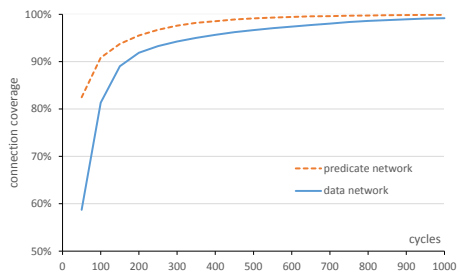
Figure 7.2 demonstrates the improvement of the instruction selection strategy we described in Section 5.2.1. The right side figures (d),(e) and (f) are same as in Figure 7.1 (d),(e) and (f). The test template for the left side figures did not adopt the optimization of the instruction selection process as compared with



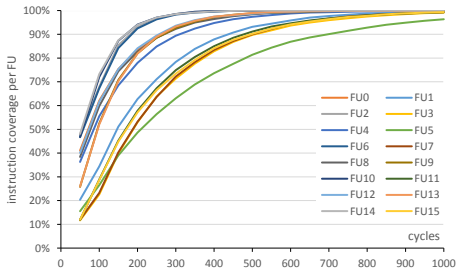
(a) FU instruction coverage



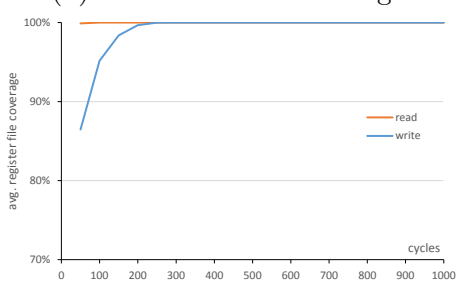
(b) RF read/write coverage



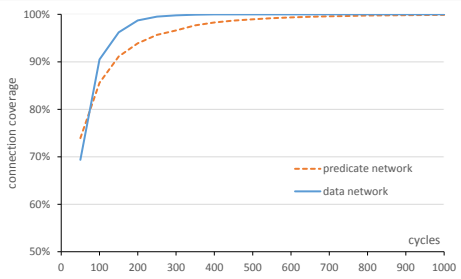
(c) Connection coverage



(d) FU instruction coverage



(e) RF read/write coverage



(f) Connection coverage

Figure 7.2 Individual instruction coverage per FU, average coverage of the RFs and the interconnection network of a test set without the optimization of instruction selection process ((a), (b) and (c)) and one that gives more weight to specific instructions ((d), (e) and (f)).

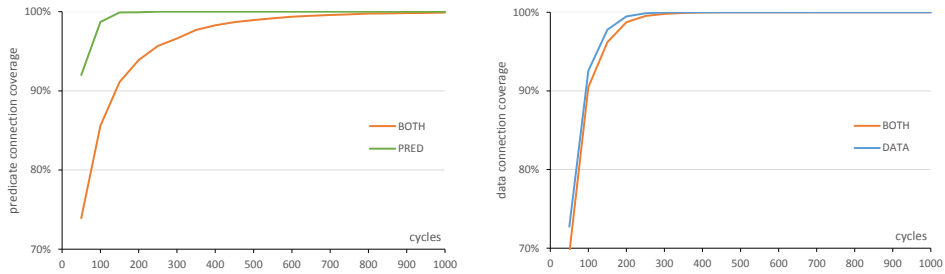
the test template of the right-side figures. Other constraints are completely equal to the right-side ones.

The instruction coverage and register read/write coverage in Figure 7.2 (a) and (b) shows the better coverage rate than Figure 7.2 (d) and (e). This is because that all candidate instructions are chosen with equal probability during the instruction selection process. In the case of Figure 7.2 (d) the RTPG try to select specific instructions until the full coverage of the interconnection network so that the instruction coverage rate could increase slowly. The register read/write coverage of Figure 7.2 (e) is also the consequence of the biased instruction selection. While trying to schedule those specific instructions continuously, only some of registers could be utilized due to the structure of our target architecture. However, as we mentioned in previous chapters, these coverage rates could be improved easily.

As shown in Figure 7.2 (c) and (f), there is significant effects on the coverage of the interconnection network. In terms of data connection network, the results show that much more cycles are required to reach the full coverage without the optimization in the instruction selection process. The coverage of the predicate network are naturally lower than Figure 7.2 (c), since predicate operations which make the predicate network coverage higher have less weight than the instructions helping the data network coverage. The maximum coverage of the predicate network could be achieved easily by other templates, this is proven in the next experiment. According to the result of the experiments, this enhancement for the coverage of the interconnection network is crucial to getting the full coverage within a small number of total cycles of the test sets.

7.3 Directed test program generation

We suggested the directed test generation for the full coverage of the interconnection network in Section 5.2.2, and Figure 7.3 represents the result of the strategy. We performed experiments with three test templates for only predicate



(a) Predicate network coverage (b) Data network coverage

Figure 7.3 Connection coverage for the predicate and data network is obtained by running the test programs for predicate network testing (`PRED`), data network testing (`DATA`) and both network (`BOTH`).

network testing (`PRED`), only data network testing (`DATA`) and all of interconnection network testing (`BOTH`).

Likewise the previous experiments, all the three test templates have the cycle constraint of 50 cycles and the history constraint for measuring the coverage rate. Since this experiment aimed the higher connection coverage rate, we also adopted the optimization for the instruction selection process. For the test programs of `PRED`, the test template in Figure 5.1 was used. In the case of the `DATA`, we configured the test template so that the RTPGs does not utilize any connection in the predicate network except for the loop structure. The data of `BOTH` were generated by the same test templates of the experiment in the Section 7.1.

Figure 7.3 (a) presents the connection coverage of the predicate network. The connection coverage metrics are calculated with the available coverage which was obtained without any constraints. The test template of the `PRED` got the maximum coverage of the predicate network within only 150 cycles, however, the `BOTH` shows the coverage rate increasing slowly and still could not guarantee the full coverage of the predicate network after running all test programs of 1000 cycles. The test programs of `DATA` only used the predicate

connections for the loop structure, so its coverage of the predicate network was very low (0.02%). The result of the data network coverage is shown in Figure 7.3 (b). The result of `DATA` and `BOTH` was not that different, but apparent. Due to the operation constraints of the `PRED`, the `PRED` could not exercise the data connection more than 67.4%.

Consequently, we could get the full coverage of the interconnection network with 150 cycles of test programs generated by the `PRED` template and about 300 cycles of test programs originated by the `DATA` template. The `BOTH` could not reach the maximum coverage of both networks even within 1000 cycles. The experiment result demonstrates that this approach is effective for the full coverage of the interconnection network.

7.4 Fault Coverage

We ran the test set on the RTL implementation of an instance of a commercial variant of the Samsung Reconfigurable Processor. As outlined in Chapter 6, the verification framework executes the binary test program simultaneously on an RTL simulator and a cycle-accurate functional CGRA simulator. After every cycle, all data values on every data port in the system under test are compared.

We have randomly inserted a total of 1029 distinct faults into the VHDL implementation of the SRP architecture. The faults cover all aspects of signal routing. This includes not only routing of data values in the interconnection network but also the input selection bits for the multiplexers or instruction selection on the FUs. For each of the 1029 faulty architectures, the fault generator inserted at least one and up to eight randomly generated faults from one of the following classes: selection errors in the multiplexers, missing write-enable signals, wrong register address decoding, and faulty registers in the RFs, or floating connection errors where the entire or a certain number of bits of a physical connection remain undefined. On average, each of the 1029 test architectures contains three faults.

Running the pre-silicon verification tests with the generated set of random test program resulted in a 100% success rate, i.e., the verifier successfully classified the architecture as faulty. Note that the current verification framework stops at the first inconsistency; there is no point in continuing the test once data values start to diverge from the reference set. This means that the verifier did not necessarily detect *all* inserted faults but rather that it detected one of the faults and then correctly concluded that the architecture contains a fault.

In addition to the randomly inserted faults, the proposed RTPG has also led to the discovery of two yet unknown errors in the original (and assumed-to-be error-free) VHDL specification of the SRP architecture. Both faults were caused by copy-pasting VHDL code and resulted in wrongly wired connections where one end of the connection was connected to the wrong input of a multiplexer. Existing test sets aiming at full coverage of PE functionality did not uncover the faults because the test generator does not consider connection coverage. The early tests during the development of the proposed RTPG framework have also revealed a number of bugs in the cycle-accurate CGRA simulator.

7.5 Discussion

Above experiments demonstrate that the proposed RTPG is versatile and is capable of generating a set of test programs that covers all the chip's functionality. Thanks to the fitness function favoring unexercised connections and other heuristics introduced in Chapter 5, the generated test programs achieve the full coverage of the interconnection network. Although the required length of random test programs for the full coverage is not determinate, we presented that the proposed RTPG noticeably improves the coverage rate rather than a solely random generator. The length is usually a few hundred cycles which is suitable for pre-silicon verification where the simulation speed is one of the limiting factors.

The fact that the proposed RTPG can generate valid test programs without

any knowledge about the instructions' semantics allows the RTPG to support seamlessly 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 instructions. In pre-silicon verification where an architecture is observable, 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 an input operand in a successive instruction. 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 is less of a problem in our post-silicon validation process because the functionality of FUs is tested separately in pre-silicon verification, but it may be a limiting factor if the test programs generated by the proposed RTPG are the only post-silicon validation step.

Chapter 8

Conclusion

In this thesis, we discuss the design and implementation of an RTPG for CGRAs in terms of coverage metrics. The place-and-route algorithm schedules random instructions on the FUs and routes the input operands through the interconnection network. In order to support the reconfigurable nature of CGRAs, a graph representation is constructed from the given architecture description files. Seamless support for custom ISA extensions is achieved by requiring only the syntax of an instruction and a C code segment for the instruction. The RTPG can schedule instructions without its semantics by tracking the types of the data values. In addition, we propose an analysis algorithm that computes the maximum available coverage for the given architecture description and constraints. Some heuristic approaches to improve the coverage rate of the generated test programs are also introduced. Experimental results show that the test programs achieve maximal coverage within a few hundred cycles and the proposed approaches are effective to test the interconnection network. Applied to a real instance of the Samsung Reconfigurable Processor, the generated test programs triggered 100% of 1029 randomly inserted faults and even exposed several to-date unknown bugs in the architecture.

Bibliography

[1] J. Bergeron, *Writing Testbenches: Functional Verification of Hdl Models*. Kluwer Academic Publishers, 2003.

[2] H. Foster, "Trends in functional verification: A 2014 industry study," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pp. 1–6, June 2015.

[3] E. M. C. Jr., O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.

[4] M. Velev and P. Gao, "Exploiting abstraction for efficient formal verification of DSPs with arrays of reconfigurable functional units," in *Formal Methods and Software Engineering* (S. Qin and Z. Qiu, eds.), vol. 6991 of *Lecture Notes in Computer Science*, pp. 307–322, Springer Berlin Heidelberg, 2011.

[5] B. Bentley, "High level validation of next-generation microprocessors," in *Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop*, HLDVT '02, (Washington, DC, USA), pp. 31–, IEEE Computer Society, 2002.

[6] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," in *Processor Design* (J. Nurmi, ed.), pp. 447–481, Springer Netherlands, 2007.

[7] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-Pro: Innovations in test program generation for functional processor verification," *IEEE Des. Test*, vol. 21, pp. 84–93, Mar. 2004.

[8] A. Kamkin, E. Kornykhin, and D. Vorobyev, "Reconfigurable model-based test program generator for microprocessors," in *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011*, pp. 47–54, 2011.

[9] P. Mishra and N. Dutt, "Specification-driven directed test generation for validation of pipelined processors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, pp. 42:1–42:36, July 2008.

[10] P. Mishra, A. Shrivastava, and N. Dutt, "Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable socs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, pp. 626–658, June 2004.

[11] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 166–176, ACM, 2008.

[12] D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim, "Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor," in *Field-Programmable Technology (FPT), 2012 International Conference on*, pp. 67–70, 2012.

[13] Y. Cho, S. Jeong, J. Jeong, H. Shim, Y. Han, S. Ryu, and J. Kim, "Case study: Verification framework of samsung reconfigurable processor," in *13th International Workshop on Microprocessor Test and Verification (MTV), 2012*, pp. 19–23, 2012.

[14] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey, "Functional verification of a multiple-issue, out-of-order, superscalar alpha processor – the DEC Alpha 21264 microprocessor," in *Proceedings of the 35th annual Design Automation Conference*, DAC '98, (New York, NY, USA), pp. 638–643, ACM, 1998.

[15] E. Bin, R. Emek, G. Shurek, and A. Ziv, "Using a constraint satisfaction formulation and solution techniques for random test program generation," *IBM Syst. J.*, vol. 41, pp. 386–402, July 2002.

[16] S. Fine, A. Freund, I. Jaeger, Y. Mansour, Y. Naveh, and A. Ziv, "Harnessing machine learning to improve the success rate of stimuli generation," *IEEE Transactions on Computers*, vol. 55, no. 11, pp. 1344–1355, 2006.

[17] X. Qin and P. Mishra, "Scalable test generation by interleaving concrete and symbolic execution," in *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*, pp. 104–109, Jan 2014.

[18] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero, "Fully automatic test program generation for microprocessor cores," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, (Washington, DC, USA), pp. 11006–, IEEE Computer Society, 2003.

[19] P. Mishra and N. Dutt, "Functional coverage driven test generation for validation of pipelined processors," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, (Washington, DC, USA), pp. 678–683, IEEE Computer Society, 2005.

[20] G. D. Guglielmo, L. D. Guglielmo, F. Fummi, and G. Pravadelli, "Efficient generation of stimuli for functional verification by backjumping across extended FSMs," *J. Electron. Test.*, vol. 27, pp. 137–162, Apr. 2011.

[21] H.-M. Koo and P. Mishra, "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, (New York, NY, USA), pp. 137–142, ACM, 2008.

[22] E. Sanchez, G. Squillero, and A. Tonda, "Automatic generation of software-based functional failing test for speed debug and on-silicon timing verification," in *Microprocessor Test and Verification (MTV), 2011 12th International Workshop on*, pp. 51 –55, dec. 2011.

[23] N. Foutris, D. Gizopoulos, M. Psarakis, X. Vera, and A. Gonzalez, "Accelerating microprocessor silicon validation by exposing ISA diversity," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, (New York, NY, USA), pp. 386–397, ACM, 2011.

[24] J. Filho, S. Masekowsky, T. Schweizer, and W. Rosenstiel, "CGADL: An architecture description language for coarse-grained reconfigurable arrays," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 9, pp. 1247–1259, 2009.

[25] M. Rullmann, S. Siegel, R. Merker, J. A. O. Filho, T. Schweizer, T. Oppold, and W. Rosenstiel, "Efficient mapping and functional verification of parallel algorithms on a multiücontext reconfigurable architecture," in *Architecture of Computing Systems (ARCS), 2007 20th International Conference on*, pp. 1–10, 2007.

[26] B. Mei, B. Sutter, T. Aa, M. Wouters, A. Kanstein, and S. Dupont, "Implementation of a coarse-grained reconfigurable media processor for AVC decoder," *Journal of Signal Processing Systems*, vol. 51, no. 3, pp. 225–243, 2008.

[27] H. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *Design, Automation and Test in Europe, 2008. DATE '08*, pp. 1298–1303, 2008.

[28] X. Liu and Q. Xu, "Trace signal selection for visibility enhancement in post-silicon validation," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, (3001 Leuven, Belgium, Belgium), pp. 1338–1343, European Design and Automation Association, 2009.

[29] A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann, "Reaching coverage closure in post-silicon validation," in *Hardware and Software: Verification and Testing* (S. Barner, I. Harris, D. Kroening, and O. Raz, eds.), vol. 6504 of *Lecture Notes in Computer Science*, pp. 60–75, Springer Berlin Heidelberg, 2011.

[30] S. Ray and W. Hunt, "Connecting pre-silicon and post-silicon verification," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pp. 160–163, 2009.

[31] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann, "A unified methodology for pre-silicon verification and post-silicon validation," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pp. 1–6, 2011.

[32] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *Field Programmable Logic and Application* (P. Y. K. Cheung and G. Constantinides, eds.), vol. 2778 of *Lecture Notes in Computer Science*, pp. 61–70, Springer Berlin Heidelberg, 2003.

[33] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "Piperench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

요약

점차 복잡해지는 하드웨어 디자인의 기능 검증을 수행하는 것은 하드웨어 개발 과정에서 발생하는 병목의 상당부분을 차지하고 있다. 디자인 과정에서 재구성이 가능한 아키텍쳐의 경우 아키텍쳐의 구성이나 명령어 구조가 변경될 수 있고, 이러한 부분을 고려한다면 일반적인 아키텍쳐를 검증하기 위해 설계된 기존의 기법들을 통해서는 재구성 가능 아키텍쳐의 기능 검증을 수행하는 것에 어려움이 있을 수 있다.

이 논문에서는 그러한 재구성 가능한 아키텍쳐를 위한 무작위 테스트 프로그램 생성 기법에 대해 소개한다. 소개된 방법은 시뮬레이션을 통한 검증을 수행하는 방식으로 하드웨어 공정 전 디자인 과정과 공정 후 기능 검증에 모두 사용될 수 있다. 테스트 프로그램을 만들기 위해서는 먼저 해당 아키텍쳐의 구성을 텍스트 기반의 설정 문서로부터 정보를 받아와 모델링을 수행한다. 하나의 재구성 가능한 아키텍처를 하나의 그래프 모델로 모델링하고, 이 모델을 기반으로 무작위 검증 프로그램을 생성하여 실행가능한 형태의 결과를 출력한다. 다형성을 가진 처리 장치에 명령어를 스케쥴링 하기 위해서, Place-and-Routing 알고리즘을 기반으로 하여 무작위 프로그램을 생성한다. 순수하게 무작위로만 생성된 테스트 프로그램이 검증하는 아키텍쳐의 커버리지는 커버리지가 높아질수록 천천히 증가할 수 밖에 없으므로, 적합성 함수를 정의하여 커버리지 향상에 목적을 둔 테스트 프로그램을 생성할 수 있게 하였다. 또한 연결 구성요소의 커버리지를 위한 휴리스틱 기법을 명령어를 선택하는 부분에 적용하였다. 이 무작위 테스트 프로그램 생성기는 새로 추가된 명령어나 재구성 된 아키텍쳐를 텍스트 기반의 설정 문서에서 읽어 아키텍쳐 모델을 생성하므로, 추가적인 작업이 필요없이 재구성 된 아키텍쳐를 지원할 수 있다.

실험에서는 이 테스트 프로그램 생성기가 다양한 테스팅 목적으로 사용될 수 있고, 작은 크기의 테스트 프로그램으로 높은 커버리지를 달성할 수 있음을 확인하였다. 이 테스트 생성 기법 및 검증 프레임워크는 Samsung Reconfigurable

Processor의 실제 기능 검증을 위하여 사용되었고, 검증을 위하여 무작위로 추가된 에러는 물론 아키텍쳐 내에 알려지지 않았던 에러를 모두 검증을 통해 발견할 수 있었다.

주요어: 재구성 가능 아키텍쳐, 하드웨어 기능 검증, 시뮬레이션 기반 검증, 무작위 테스트 생성
학번: 2014-21761