Random Test Program Generation for Reconfigurable Architectures

Seonghun Jeong*, Youngchul Cho*, Daeyong Shin†, Changyeon Jo†, Yenjo Han*,
Soojung Ryu*, Jeongwook Kim*, and Bernhard Egger†
*Samsung Advanced Institute of Technology, Samsung Electronics, Giheung, Korea
Telephone: +82 31 280 9518, Fax: +82 31 280 9587
{sss.jeong,rams.cho,yenjo.han,soojung.ryu,jw85.kim}@samsung.com
†School of Computer Science and Engineering, Seoul National University, Korea
Telephone: +82 2 880 1819, Fax: +82 2 880 1805
{daeyong,changyeon,bernhard}@csap.snu.ac.kr

***Abstract*—Automatic generation of test programs plays a major role in the verification of microprocessors. In this work, we propose a random test program generator (RTPG) framework for reconfigurable architectures. Reconfigurable architectures pose a number of problems to existing RTPGs. The proposed framework overcomes these problems by building a hardware model directly from the architecture description. Our RTPG only tracks the *type* of the data values. This enables the framework to seamlessly support custom ISA extensions whose semantics are not available to the RTPG. We implement the proposed RTPG framework for the Samsung Reconfigurable Processor (SRP), a low-power high-performance reconfigurable architecture consisting of a VLIW and a coarse-grained reconfigurable array processor. The experiments show that the framework is flexible, efficient and quickly achieves a high coverage in the generated test programs.**

I. Introduction

With the ever-growing complexity of hardware designs, coupled with the demand for greater performance and faster time-to-market, functional verification is widely acknowledged as the bottleneck of the hardware design cycle [3]. Although formal methods such as model checking [10] have advanced significantly the supported complexity of the models only allows for validation of relatively small hardware blocks.

Simulation-based verification thus plays an important role in the functional verification of microprocessor designs [2]. In simulation-based functional verification, the microprocessor executes a sequence of instructions, the test program. The outcome of the computation is compared to a reference value, either at the end or every n cycles during execution. An important measure for the quality of a (set of) test programs is the coverage. Depending on the goal of the test, different kinds of coverage are meaningful such as instruction coverage, operand coverage, etc.

The test programs are generated by *random test program generators* (RTPG). An RTPG generates a test program by randomly selecting instructions while adhering to a number of constraints. Hardware constraints describe constraints imposed by the hardware. Hardware constraints range from very simple (i.e., the syntax and latency of instructions) to complex (i.e., no add instruction must be scheduled directly after a multiply instruction). The simpler hardware constraints in existing RTPGs are typically extracted automatically from the architecture description. Complex constraints such as which instruction sequences are not supported by the hardware are typically added manually by a modeling engineer [1]. User-defined constraints, on the other hand, allow the verification engineer to specify certain conditions that the test program must adhere to. User-defined constraints are useful for *directed* random testing, for example, exercising certain instruction sequences mixed with randomly generated code.

Most RTPGs not only require knowledge of the syntax but also the semantics of an instruction. Knowledge of an instruction's syntax is necessary to generate code with the correct number of input/output arguments or correctly compute the latency of an instruction. The semantics of an operation are required to pre-compute the output of the test program or allow biased results [1].

In recent years, coarse-grained reconfigurable array (CGRA) processors have gained a lot of attention, both in academia [7], [15] and industry [13], [17]. CGRAs consist of a number of processing elements (PE), register files and an interconnection network. Typically, the functionality of the PEs as well as the interconnect are reconfigurable. The abundant parallelism and the programmability at a low power consumption of CGRAs make them ideal candidates for decoding (and encoding) multimedia data streams in embedded systems.

The reconfigurable nature makes it difficult to apply existing RTPGs to CGRAs. In an CGRA, the ISA, the number and functionality of PEs, as well as the interconnection network may be significantly different between two instances of the same architecture. Existing RTPGs are tailored to support a certain ISA and even though they support, for example, different latencies for instructions, the syntax and the semantics of instructions are not easily modifiable. Furthermore, due to the reconfigurable interconnection network, CGRAs cannot be described using a simple register-transfer language since not all PEs are directly connected to the register file(s). Instead, scheduling techniques such as simulated annealing [14] or

Presented by Seonghun Jeong
Corresponding author: Bernhard Egger (bernhard@csap.snu.ac.kr)

edge-centric modulo scheduling [17] are used to compile a data flow graph into a series of CGRA instructions.

In this paper, we present a directed random test program generation framework RDG (Random Diagnostics Generator) for Samsung Electronics' reconfigurable architecture, the Samsung Reconfigurable Processor (SRP) [17]. The SRP is a dual VLIW/CGRA processor with configurable VLIW issue width, number and functionality of PEs, register files, and interconnection network. The configuration of the SRP is stored in textual form and interpreted by the whole toolchain, including compilers, simulators, debuggers and profilers. The presented RTPG also reads the configuration of the SRP directly from the SRP configuration files. Similar to existing RTPGs, our RTPG needs to be aware of the instructions' syntax and latency; however, it can generate valid test programs without knowing the semantics of the instructions. This enables our RTPG to adapt without any user invention to different ISAs, customized instructions, VLIW issue widths, and heterogeneous PEs.

Since the RDG framework has no knowledge of an instruction's semantics it is impossible to pre-compute the outcome of executing a test program on the chip. Verification is thus performed by checking the values in the register files and at the PEs' output ports either periodically or after the test program has ended.

Certain classes of instructions require certain input operands. For example, the sum of the two input operands to a memory load instruction typically represent accessed memory address. Another example are branch instructions where one of the the input operands denotes the target instruction of the branch. For this reason, the RDG framework cannot be completely oblivious to such instructions' semantics. The SRP configuration files groups such instructions into instruction classes that allow our RDG framework to infer the constraints of an instruction's operands. While generating code, the RDG framework thus tracks the *type* of each datum. When selecting operands, the type of a datum in a register file or at the output port of a PE is checked against the instruction's operand constraints.

Our RDG framework is implemented as a C++ library, and consequently, test templates are written in plain C++ code. This has the advantage that the verification engineers do not have to learn a new language specifically for coding test templates. Additionally, customized extensions of the RDG framework are possible through sub-classing specific classes.

The experiments show that the presented RDG framework is able to adapt to reconfigurable architectures automatically while generating valid test programs that achieve a high coverage.

The contributions of this paper are as follows:

- We design and implement an RTPG for complex architectures that do not offer hardware-level hazard resolution. The implemented RTPG supports the Samsung Reconfigurable Processor, a chip consisting of a VLIW and a CGRA unit.
- We show that RTPGs do not need to know the semantics of instructions. By tracking the *type* of data values, the proposed RTPG can support custom ISA extensions.
- We propose a test template language entirely based on a constraint specification language library implemented in C++. This allows the verification engineer to familiarize himself quickly and enables him to even make small modifications to the test template language directly.

The remainder of this paper is organized as follows: Section II gives an overview of related work in random testing. In Section III, the reconfigurable architecture used for this work is described in more detail. Section IV contains the design and implementation of our random test generation framework. Section V evaluates the test programs generated by the random test generator, and Section VI concludes the paper.

II. Related Work

Functional verification has been and still is an active field of research. Many different methods ranging from low-level formal verification up to instruction-level functional verification have been developed in the past. 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 [4] and Adir [1] model the test program generation problem as a CSP. Their framework, Genesys-Pro, combines architecture-specific knowledge and testing knowledge and uses a CSP solver to generate efficient test programs. The test template language of Genesys-Pro is quite complex and allows, for example, biased result constraints. Corno [5] and Mishra [16] use graph-based algorithms to generate test programs. While Corno uses a predefined library of instructions, Mishra's work extracts the structure of the the pipelined processor directly from the architecture description language and then fed to a symbolic model verifier. Di Guglielmo [6], [9] proposes a pseudo-deterministic automatic test pattern generator (ATPG) 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 [11] also uses an FSM combined with reduction techniques to achieve high coverage with a small number of directed tests. In Sanches' work [18], an automatic, feedback-based approach that generates assembly instruction sequences for speed debug, 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 [8] 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.

In conclusion, several different and fully-automatic approaches have been proposed. None of the presented methods are easily applicable to reconfigurable architectures with a

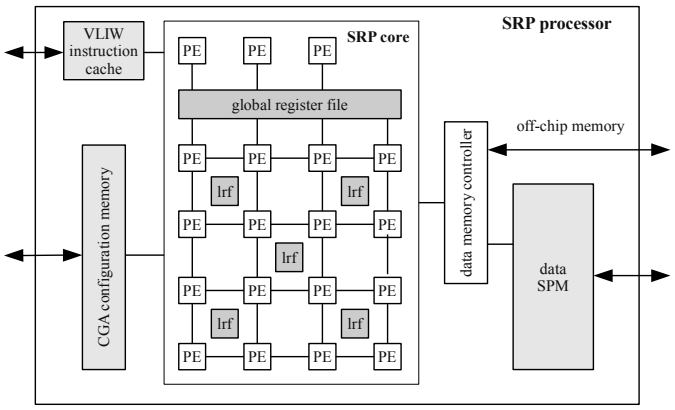

Fig. 1. The Samsung Reconfigurable Processor

large number of processing elements and an irregular interconnection network.

III. The Samsung Reconfigurable Processor

The Samsung Reconfigurable Processor (SRP) is a dual-mode VLIW/CGRA low-power processor targeting embedded systems (Figure 1). It comprises an SRP core, on-chip memory (instruction cache, configuration memory and data memory), a DMA controller and a bus interface. The SRP core itself is made up of a number of processing elements (PEs), global and local register files, and an interconnection network. The PEs are conceptually arranged on a 2D grid. The PEs are not homogeneous in their functionality; while typically all PEs support basic ALU instructions, only few support memory operations, yet other PEs support floating point arithmetic. The SRP operates in either VLIW or CGRA mode. CGRAs provide excellent performance at a low power consumption in code sequences that contain lots of parallelism. Code sections that are dominated by control flow, however, perform poorly in CGRA mode. Such code sections are thus compiled in VLIW mode. Data is conveyed between the two modes through a central data register file (cdrf), and the latency of the mode switch operation is in the order of the latency of a branch instruction. This allows for very efficient and frequent mode switches. The compiler makes extensive use of this property by scheduling control-intensive code in VLIW mode and compute software-pipelined code for loops.

The aim of the SRP is to provide excellent performance in code sections with lots of loop-level parallelism, yet different application domains exhibit different properties. SRP instances intended for, say, video decoding typically require a different configuration of the interconnection network and the functionality of the PEs than instances aiming at audio decoding. The SRP and its development environment enable very flexible and efficient architecture exploration by storing the entire high-level configuration of the SRP instance as human-readable text files. The entire development environment from compiler, to simulators, profilers, and debuggers take the architecture description as an input and thus adapt automatically to the SRP architecture. An engineer can thus

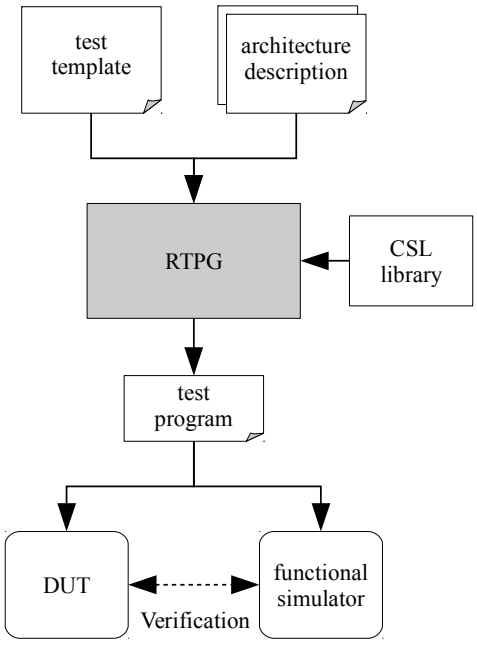

Fig. 2. RTPG framework

test the effect of adding/removing certain functionality from a PE or changing the interconnection network merely by editing the configuration file, recompiling and executing the code on a cycle-accurate simulator. In addition, the framework allows for rapid exploration of custom ISA extensions. Custom ISA extensions are instructions that are not part of the default instruction set architecture. Such extensions are added to an SRP instance by providing a C-implementation that simulated the requested functionality. Since these extensions are not part of the default ISA, the compiler has no knowledge of the semantics of such instructions. Consequently, the engineer has to invoke the custom ISA extensions explicitly. The framework models custom ISA extensions as simple C function calls. Of course, custom ISA extensions are supported throughout the entire SRP development environment.

IV. Directed Random Test Generation

This section describes the design and implementation of the proposed RTPG in detail.

The test template is written in C++. The so-called *constraint specification language* (CSL) is provided as a library. The RTPG is guided by the test template which makes calls to the CSL to specify the parameters of the test. In addition to the test template and the CSL library, the reconfigurable architecture's configuration files are another input to the RTPG. The RTPG then generates a test program which is run on the device-under-test (DUT) and a functional simulator. Verification is done online or offline either periodically or at the end of the run by comparing register contents and data in PE output ports. Figure 2 gives an overview of the design and the verification setup.

Traditional RTPGs are not well-suited to support reconfigurable architectures such as the SRP for a number of reasons:

first, the vast majority of test generators aim at single-issue microprocessors. Scalar processors resolve hardware hazards automatically by delaying/reordering instructions in the instruction stream. Both VLIW and CGRA processors, however, do not provide any such support at the hardware level. Instead, the compiler (or in our case the RTPG) is responsible to generate instruction sequences that are hazard-free.

Second, existing test generators require the semantics of every instruction of the ISA to be known in order to precompute the outcome of the computation. Additional features, such as Genesys-Pro's biased results [1] also require the instructions' semantics. An RTPG for a reconfigurable architecture also needs to support custom ISA extensions. Since the semantics of such 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 (there are important exceptions to this generalization which are discussed below).

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 PE at a given time, but also make sure the input data can be routed through the interconnection network. Only a limited number of PEs are directly connected to the central data register file, and for each local register file the directly connected PEs vary. An RTPG for reconfigurable architectures 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 constraints which leads to unacceptably long test program generation times.

A. Test Generation Engine

Our RTPG framework satisfies the above requirements through a number of design choices. First, instead of modeling the hardware and user-defined constraints as a CSP or SAT, we have implemented a random instruction scheduler. The fine-grained hardware model leads to massively more variables or constraints that need to be checked by a RTPG. Even though T.Larrabee has shown that for single-issue microprocessors most clauses belong to 2-SAT [12] (which is solvable in polynomial time), this does not hold for complex models. The scheduler of our RTPG framework uses weighted-random instruction selection with backtracking. It randomly selects an instruction based on the instruction's weight and then tries to satisfy the instruction's output and input operand requirements. The output operand requirement is simply that the output port of the given PE must be free when the to be scheduled instruction outputs its result. For the input operands, in VLIW mode, this step is completed by checking whether the connected register file(s) can provide the necessary types of input operands at the given time. In CGRA mode, the scheduler needs to find a route from PEs' input ports to output ports of other PEs, register files, or constant value generators that can provide a datum of the required value at the requested time. Note that routes may take longer than one cycle to transfer a datum from the source to the sink of the route. If the any of the input or output requirements cannot be satisfied, then the instruction selector randomly selects a new instruction until the instruction can be placed. If no instruction can be placed until a certain threshold is reached, the RTPG inserts a no-operation. In rare cases, the instruction selector can get stuck, for example, if the register file does not contain any data of a certain type. If so, the last selected instructions are unscheduled and the instruction selector starts over.

Close at hand with the instruction selector is a type tracker. The type tracker tracks the types of all data values that are available in registers, in memory, at PEs' output ports or provided by a constant generator at all times during the test program. The type tracker tracks four types: integer, known integer, float, and known float. 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. Certain classes of instructions require the actual value of certain input operands to be known. This is true for instructions accessing memory as well as branch/jump instructions. For the former the sum of two input operands denotes the memory address to be accessed, and for the latter one input operand determines the target of the control transfer. Known integer/float values are also necessary to support directed tests where the verification engineer wants to generate instruction sequences that take a specific value as an input operand. Data values of know type and value are represented by the *known integer* and *known float* type, respectively.

Known values are also used for loops. The RTPG inserts a loop skeleton if requested by the test template. To prevent infinite loops (or loops with a too large iteration count), a value of type *known integer* is used as the loop bound. If required the API of the CSL additionally allows the verification engineer to specify the exact number of loop iterations.

The architecture configuration of the SRP allow the RTPG to infer the type of every instruction's input and output operands, and this also holds for custom ISA extensions. The architecture configuration also explicitly denotes more specific constraints on input operands such as for memory access operations and control transfer instructions.

If requested, the RTPG inserts initialization code that initializes a minimal number of registers and memory cells to one of the tracked types at the beginning of each test program.

B. Test Template Language

There exist probably as many test template languages as there are RTPG. The first implementation of our RTPG also

defined its own test template language; however, we quickly noted two main problems with this approach:

1) there is a significant learning curve involved in using the RTPG in an industrial environment
2) even minor feature additions require extensive modifications to the RTPG framework, from parsing the template over modifying the RTPG core itself to implement the new features up to maintaining backwards compatibility to previous versions.

These difficulties have persuaded us to try a somewhat different approach: test templates are written in pure C++.

The RTPG's core components are already implemented in C++, and we have implemented a library, the so-called *Constraint Specification Language* (CSL) library, around the RTPG that allows the verification engineer to control various aspects of the random test program generation. A test template is written by overloading a specific method of the RTPG. The test template is then compiled and linked together with the RTPG core components and the CSL library into an executable program. This executable program generates a random test program every time it is invoked.

Writing test templates in C++ using the CSL library still incurs a small learning curve, namely, to familiarize oneself with the CSL API. However, since most engineers are already proficient in C++ the effort is much smaller. Additions and modification to the CSL are also much easier, and certain modification can be implemented by the verification engineers themselves. Per default, the CSL library only contains an immediate value generator that produces random values within a certain range. A verification engineer can easily create his own subclass and overload the method that selects the immediate value to, say, generate only values of special interest. Last, compiling the test template into an executable that is executed natively on the host allows our RTPG to achieve a very high throughput of generated instructions even for complex CGRA processors. The only disadvantage we have found is that the test templates quickly look bloated compared to domain-specific test template languages.

The code in Fig. 3 shows a working test template written in C++ using the CSL library. The test template generates a random VLIW schedule that is to contain 1000 instructions. The `NofIterationContraint` triggers insertion of a loop and simultaneously defines the number of iterations. The `RandomOperationGenerator` instance has a default weight of 50 for all instructions with four instructions explicitly set to 100. `OperationGenerators` can be applied globally (that is, for all PEs) or to single PEs. The call to `Emit()`, finally, triggers code generation. The CSL library provides much more functionality, however, it is outside the scope of this paper.

```
void CSL::RTPGenerator::CSL_main(unsigned int seed,
                                 string outputPath)
{
  /// 1. define (at least one) schedule
  RandomVLIWSchedule s;
  AppendSchedule(s);

  /// 2. Set seed value
  s.SetSeed(seed);

  /// 3. define constraints on the schedule
  /// - restrict the number of operations to generate
  s.Add(NofOperationConstraint(1000));
  /// - insert a loop and restrict the iterations
  s.Add(NofIterationConstraint(1000));

  // 4. Add Operation & Operand Constraints
  RandomOperationGenerator*
    rog = new RandomOperationGenerator();
  rog->SetDefaultWeight(50);
  rog->Add(Operation("add32_c2c"), Weight(100));
  rog->Add(Operation("fadd32"), Weight(100));
  rog->Add(Operation("asr32_c2c"), Weight(100));
  rog->Add(Operation("fmul32"), Weight(100));
  s.SetGlobalGenerator(rog);

  /// 5. emit the code
  Emit();

  /// 6. Print statistics
  s.PrintStatistics();
}
```

Fig. 3. Test template example

V. Evaluation

We have evaluated the proposed framework with twelve test templates on a SRP tailored for multimedia decoding. The architecture comprises a total of 20 PEs (VLIW+CGRA), two central data register files and several local register files. The PEs are heterogeneous; the number of distinct instructions that can be executed on a certain PE ranges from 78 to 180.

The twelve test templates cover all currently supported constraints in the CSL such as different instruction and operand weights, with/without control flow, with/without loops, on-chip memory accesses, off-chip memory accesses, floating point instructions and so on.

For the evaluation of the RDG framework we use three measures: instruction coverage, operand coverage and the time required to generate the test program. All twelve benchmarks have been run with a constraint limiting the total number of instructions to 500, 1000, and 4000.

Table I lists the results for 500 instructions. Columns 2 to 5 list the instruction coverage for the four PEs. Columns 6 to 9 indicate how many NOP instructions have been inserted into the schedule. Columns 10 to 13 list the coverage of the operands, and columns 14 and 15, finally, show the length of the generated schedule (in cycles) and the execution time (in seconds) of the RDG framework. Table I shows that even for a very small number of total instructions, the RDG framework achieves good coverage. Test template 1 and 12 are full-random (in addition to 1, 12 contains a loop), i.e., no constraints are given except for the limitation on the number of instructions. Coverage on PE1 and PE3 is below 50% because both PEs can execute over 170 different instructions. 500 instructions distributed evenly to four PEs is around 125 instructions per PE; in other words, there are not enough random instructions to achieve full coverage on PE1 and PE3. The operand coverage shows a similar result. Some test templates do not exercise all four PEs; in such cases the respective entry is marked with n/a.

Table II shows the average coverage of all twelve test templates for 500, 1000, and 4000 generated instructions. The execution time of the test generator even for 4000 instructions is still very moderate at 0.64 seconds. The results clearly show that the presented RDG framework achieves very good coverage at low instruction numbers.

TABLE I
RESULTS FOR 500 INSTRUCTIONS.

Test Template	Operation Coverage				NOP instructions				Operand Coverage				Schedule Length	Generation Time
1	72%	48%	70%	45%	30	12	41	7	60%	59%	66%	55%	259	0.30
2	100%	100%	100%	100%	56	0	71	3	100%	100%	66%	100%	256	0.29
3	100%	100%	100%	100%	23	3	24	2	100%	100%	66%	100%	253	0.26
4	100%	100%	100%	100%	4	16	3	16	100%	100%	66%	100%	252	0.24
5	100%	100%	100%	100%	56	119	71	3	100%	100%	66%	100%	253	0.29
6	100%	n/a	100%	n/a	3	116	131	119	100%	n/a	66%	n/a	251	0.25
7	100%	n/a	100%	n/a	0	113	0	119	100%	n/a	66%	n/a	251	0.22
8	94%	n/a	100%	n/a	1	138	2	128	100%	n/a	66%	n/a	252	0.22
9	88%	n/a	100%	n/a	11	115	10	122	96%	n/a	66%	n/a	263	0.23
10	89%	86%	100%	97%	25	10	22	11	72%	50%	66%	100%	265	0.27
11	n/a	95%	n/a	97%	131	3	135	2	n/a	68%	66%	80%	252	0.25
12	72%	48%	70%	45%	30	12	41	7	60%	59%	66%	55%	266	0.29
Avg.	92%	85%	94%	85%	31	54	45	45	90%	80%	91%	86%	256	0.26

TABLE II
RESULTS FOR 500, 1000, AND 4000 INSTRUCTIONS.

# Instructions	Operation Coverage				NOP instructions				Operand Coverage				Schedule Length	Generation Time
500	92%	85%	94%	85%	31	54	45	45	90%	80%	91%	86%	256	0.26
1000	98%	93%	99%	91%	60	92	85	90	95%	85%	95%	91%	510	0.32
4000	100%	100%	100%	85%	245	372	253	367	99%	98%	100%	97%	2032	0.64

VI. CONCLUSION

In this paper, we describe the design and implementation of an RTPG for reconfigurable architectures. The proposed RTPG supports reconfigurable VLIW and CGRA processors. It constructs a model of the architecture by reading the architecture configuration. The main idea of our RTPG is to track only data types instead of data values. By tracking only data types, the RTPG does not need to know the semantics of every instruction and can thus easily support custom ISA extensions. Instead of defining a test template language, we provide a constraint specification language (CSL) library written in C++. The CSL API contains all the necessary methods to specify directed random test templates. Experiments for the Samsung Reconfigurable Processor show that the proposed RTPG very quickly achieves a high coverage while adhering to the user-imposed constraints.

REFERENCES

[1] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Des. Test*, 21(2):84–93, March 2004.

[2] 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, pages 31–, Washington, DC, USA, 2002. IEEE Computer Society.

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

[4] 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.*, 41(3):386–402, July 2002.

[5] 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, pages 11006–, Washington, DC, USA, 2003. IEEE Computer Society.

[6] G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. A pseudo-deterministic functional atpg based on efsm traversing. In *Proceedings of the Sixth International Workshop on Microprocessor Test and Verification*, MTV '05, pages 70–75, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Jong eun Lee, Kiyoung Choi, and N.D. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *Design Test of Computers, IEEE*, 20(1):26 – 33, jan-feb 2003.

[8] Nikos Foutris, Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera, and Antonio Gonzalez. Accelerating microprocessor silicon validation by exposing isa diversity. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 386–397, New York, NY, USA, 2011. ACM.

[9] Giuseppe Di Guglielmo, Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli. Efficient generation of stimuli for functional verification by backjumping across extended fsms. *J. Electron. Test.*, 27(2):137–162, April 2011.

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

[11] Heon-Mo Koo and Prabhat 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, pages 137–142, New York, NY, USA, 2008. ACM.

[12] T. Larrabee. Efficient generation of test patterns using boolean difference. In *Test Conference, 1989. Proceedings. Meeting the Tests of Time., International*, pages 795 –801, aug 1989.

[13] Won-Jong Lee, Shi-Hwa Lee, Jae-Ho Nah, Jin-Woo Kim, Youngsam Shin, Jaedon Lee, and Seok-Yoon Jung. Sgrt: a scalable mobile gpu architecture based on ray tracing. In *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12, pages 2:1–2:1, New York, NY, USA, 2012. ACM.

[14] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. In *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pages 166 – 173, dec. 2002.

[15] Bingfeng Mei, Serge Vernalde, Diederik Verkest, and Rudy Lauwereins. Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study. In *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, DATE '04, pages 1224–1229, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Prabhat Mishra and Nikil Dutt. Graph-based functional test program generation for pipelined processors. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, DATE '04, pages 10182–, Washington, DC, USA, 2004. IEEE Computer Society.

[17] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. *SIGPLAN Not.*, 44(7):21–30, June 2009.

[18] 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*, pages 51 –55, dec. 2011.