Micro-processor verification using a C++11 sequence-based stimulus engine.

Stephan Bourduas[1] stephan.bourduas@cavium.com
Chris Mikulis[1] chris.mikulis@cavium.com

[1]600 Nickerson Rd, Marlborough, MA 01752

Abstract— The *universal verification methodology* (UVM) [1] has proven insufficient for stimulus generation for micro-processor core verification. To address our stimulus needs for verifying the Cavium ThunderX and ThunderX2 ARM cores we have created a sequence-based assembly generator using C++11 [2], which introduced many new features that make developing in C++ easier and more efficient. This paper will present an overview of our sequence-based assembly generator *SGen* and will discuss how it is being effectively used to verify Cavium's ARM server cores.

I. Introduction

The *universal verification methodology* (UVM) [1] has become the de facto industry standard for design verification. Promoted as a way to accelerate verification through reuse and interoperability, it nevertheless falls short in the area of stimulus generation for micro-processor core verification. To address our stimulus needs for verifying the Cavium ThunderX and ThunderX2 ARM cores we have created a sequence-based assembly generator using C++11 [2]. We chose to use C++11 because it introduced many new features [3] that make developing in C++ easier and more efficient than previous iterations of the language.

This paper will present an overview of our sequence-based assembly generator *SGen.* First, we will present some background and related work. Second, the motivation for creating SGen will be discussed. Third, our tool chain will be presented. Fourth, some of the internal components of SGen will be described, focusing on how randomization is achieved despite the lack of a constraint solver. Finally, we will show how SGen is being effectively used in production to verify Cavium's ARM server core by presenting data collected from exerciser (exer) runs.

A. Background and Related Work

Figure 1 shows a typical UVM test bench architecture consisting of an active agent that injects stimulus into the device under test (DUT) and a passive agent that monitors the output. A checker that listens to each monitor can correlate the input and output values and perform any required checking and possibly collect coverage data. The architecture shown in fig. 1 assumes that stimulus is generated by a SystemVerilog (SV) sequencer/driver pair. This assumption does not hold true for micro-processor verification where the stimulus is an actual program loaded into memory that the core will execute.

Figure 2 shows an abstracted view of our core test bench. The figure shows that our environment includes several C++ components, making ours a mixed language environment. The RTL components that make up the DUT are the core and memory controller. At simulation startup, the program to be executed is loaded into a sparse memory (C++). The core/mem controller pair interact with a bus-functional model (BFM) of our last-level cache (LLC) written in SV. There are various checkers and monitors connected to each major interface. The most important checker in our environment is the ARM instruction-set simulator that is written in C++.

Clearly, the UVM sequence/driver paradigm is insufficient for our core verification needs. To address our stimulus requirements for verifying the Cavium ThunderX and ThunderX2 ARM cores we needed to explore alternative stimulus generation techniques.

Industry and academia have been exploring several techniques for generating test programs for processor verification. Examples include: linear programming [4], finite state machines (FSM) [5], constraint-satisfaction problem (CSP) solvers [6], graph-based generation [7], genetic programming [8], and finally random selection. Most of these solutions have steep learning curves and are impractical for verifying large and complex components; particularly a pipelined processor with a large verification space.

Some examples of industrial tools are: Raven (ARM), Cadence Perspec, Breker TrekSOC and IBM Genesys-Pro [9]. All of the aforementioned tools make use of graphical user interfaces (GUIs) for specifying

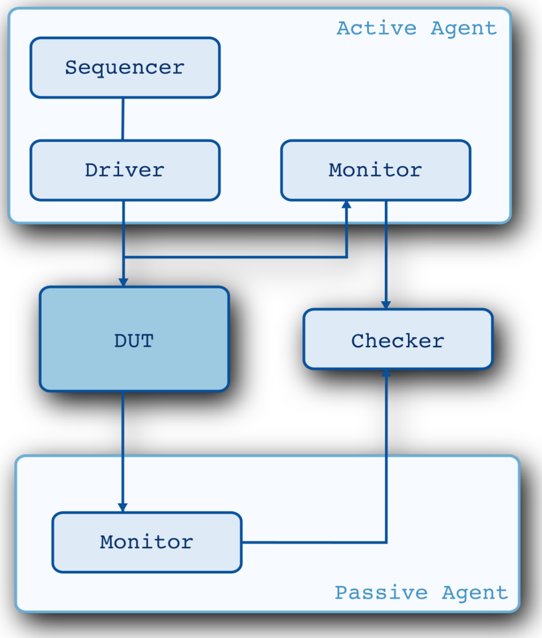

Figure 1: Generic UVM testbench architecture.

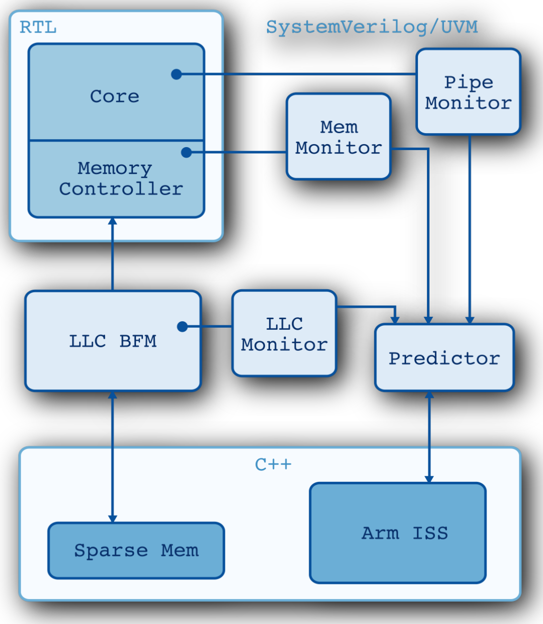

Figure 2: Core testbench architecture.

tool inputs. For example, the Raven tool allows users to create templates using a GUI. The tool then uses the template to generate instructions. The Raven tool currently ships with a library of templates that can be used for basic bring-up and random exerciser runs. The Cadence Perspec tool uses a UML-like graphical syntax to specify system-level tests for SoC verification. The Breker TrekSOC tool generates self-checking C tests for system-level verification. All of these tools require ramp-up time to learn their syntax. Since they are proprietary tools customization is difficult or impossible. Keeping our stimulus generators in-house enables us to avoid these restrictions.

Further, since Cavium already has a random generator called *PPIGen* that has been used successfully for several product generations (MIPS and ARM), we chose to leverage our current infrastructure when developing our new tool.

B. Motivation

Prior to the development of SGen, we had two ways to stimulate our design. The first was to hand-write a C or assembly test and the second was to use the knob-based instruction generator *PPIGen*. These two approaches cover opposite ends of the stimulus spectrum. The hand-written tests are only good for one-off verification tasks and cannot be leveraged for generating random stimulus. The PPIGen tool is *too* random and is not easily steered towards specific scenarios that we are interested in hitting. As with any knob-based tool, the controllability decreases over time as the number of knobs increases.

To meet our requirements we needed to bridge the gap between directed and fully random stimulus. Thus, we created a tool that adapts the sequence-based approach to stimulus promoted by UVM, thereby giving us the flexibility to create libraries of sequences that can be as random (or not) as necessary.

Even though the industry has been moving towards using SystemVerilog (SV) [10] and UVM, C++ has certain advantages that appealed to us. First, the standard libraries provide a richer set of data structures and algorithms when compared to SV; including string manipulation (important for generating assembly tests). Second, the compile times are much faster than SV, especially incremental compiles. Third, the lack of a licence requirement was an important factor that favoured C++ over SV. Fourth, debugging C++ code is easy with gdb and debug print statements. Fifth, C++ is very well documented and there are a lot of sources of community-driven support such as wikis and forums. Finally, C++ is *fast*!

Despite all of the aforementioned advantages of C++, we likely would have chosen a scripting language

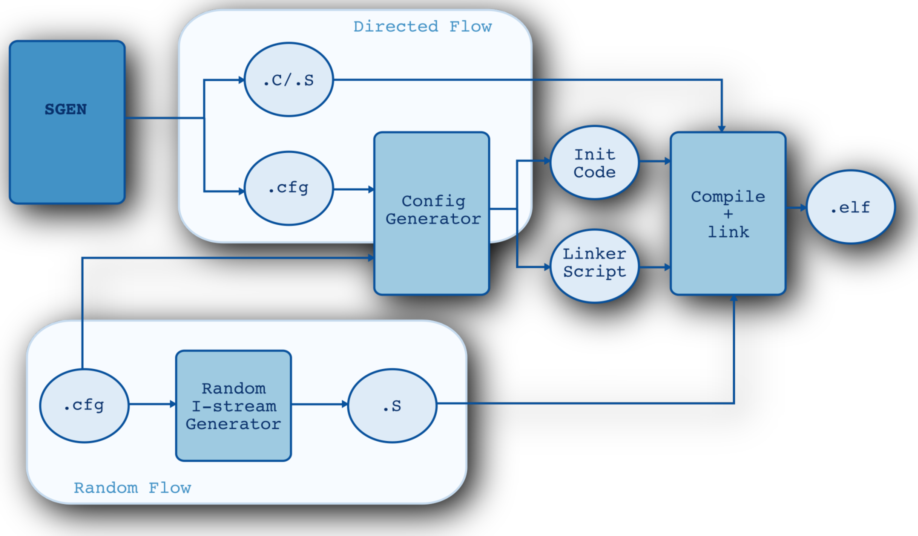

Figure 3: The PPIGen tool supports directed and random modes. The SGen tool piggybacks onto the directed mode.

such as Python instead had it not been for C++11. The new features introduced in C++11 [3] represent a significant improvement to the language over C++98. There are many new features but the key enablers for us were:

- *Lambda* functions [11] enable the creation of closures that capture variables in scope. These are used extensively in SGen to facilitate randomization and user-defined overrides (discussed in section II.B).
- The improved random number generation libraries [12] provide a wide array of number distributions (e.g. uniform, Bernoulli, Poisson) and number generators that are part of the C++11 standard. These libraries can be used to build sophisticated random number generation into an application. They were used in SGen to create the `weighted_set` class that enables generic random picking from a set of objects (section II.B).
- The polymorphic function wrapper [13] is a utility class that facilitates passing around and storing references to *callable* [14] objects. This utility is used to store references to lambda functions throughout the SGen codebase.
- Regular expressions [15] have been added to the standard library. It is now possible to do regular expression matching and advanced string manipulation without the need for external libraries.
- The `auto` specifier [16] is a useful feature that improves the readability and maintainability of code. For variables and return types, the `auto` keyword specifies that the type will be deduced by the compiler. The example in section II.B shows the specifier in working code.

C. The toolchain

The PPIGen tool has two modes of operation for generating test programs: random and directed (shown in fig. 3). In the random flow, the program input is a single config file that optionally constrains random variables that affect the program generation and system configuration. In the directed flow, the input is a C or assembly (.S) file and a config file containing configuration overrides. An important difference between both modes is that in the directed mode, the configuration of registers assumes default (reset) values unless otherwise specified in the config file, whereas in random mode configuration registers are randomized.

The PPIGen tool performs two basic functions. The first is to generate an instruction stream when in random (or exer) mode. The second is to read a config (.cfg) file and to generate appropriate init code and linker scripts. We wanted to leverage the configuration and linker script generation logic provided by PPIGen so we developed SGen to produce an assembly program and an appropriately formatted config file that is

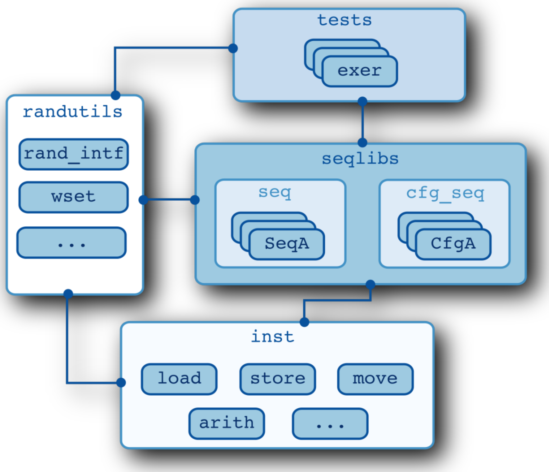

Figure 4: Basic architecture of the SGen program.

fed through PPIGen's directed flow. Figure 3 shows the PPIGen flow for directed and random test case generation as well as how the SGen fits into that flow.

II. SGen Overview

This section will discuss the internal structure of the SGen tool and provide code samples that show basic usage. Some of the more important classes used to create the sequence libraries will be discussed. Finally, a fully working test will be presented that shows what typical SGen syntax looks like.

A. Software Architecture

Figure 4 shows the software architecture of the SGen tool. The basic building block for generating an assembly test is an *instruction*; the `inst` library contains a class hierarchy of instructions organized by type (e.g. load, store, etc). The `seq` library is composed of sequences that are each written with a specific intent. Similarly, the `tests` library is comprised of tests that may use one or more sequences in order to achieve a specific verification goal. A random exerciser (or exer) test can be generated by randomly mixing sequences together. Finally, the `randutils` package provides utility classes that enable randomization.

B. Randomization — No constraints, no problem!

The lack of a constraint engine did not deter us from using C++. In our experience, constraints are often difficult to debug and can significantly impact simulation performance. We have found that randomizing variables using procedural code is sufficient to meet our needs. In addition, constraints in SV have their own syntax that one has to learn; since SGen is pure C++ there is no such requirement. In fact, it is often easier to express complex relationships between variables using procedural code. The random selection of variables is made possible by a parametrizable *weighted set* (`wset`) class and a *random interface* (`rand_intf`) base class.

The `wset` class is a container that can hold items and their associated weights. The class has a `pick` method that is called to randomly select an item from the set. Since the class is parametrizable, it can hold objects of any type, including other weighted sets for nested picking. An useful feature of the `pick` method is that it is *recursive*. That is, it does not matter whether the weighted set items are weighted sets or scalars, the pick method is "smart" enough to detect a nested knob and a user need only call the `pick` method once.

In lieu of a constraint solver, the SGen tool provides a random interface class that maintains a list of `lambda` [11] functions (or `closures` [17]) that are executed in fifo order when the `randomize` method is called. The callback functions are registered with the random interface class via the `push_callback` method. If multiple lambda functions set the same variable, the last one "wins". The random interface class also

Listing 1: The weighted set class.

```
// Create 2 weighted sets of chars with 2 items each.
// Note that the syntax uses C++11 list initialization.
wset<char> w1( {{'a', 100}, {'b', 200}} );
wset<char> w2( {{'c', 200}, {'d', 800}} );

// Create a nested weighted set
wset<wset<char>> w;

// Add items to the weighted set
// w2 has a higher weight.
w.add_item(w1, 100);
w.add_item(w2, 900);

// Pick a character from nested wset.
// Template magic will make the call recursive.
char c = w.pick();
```

Listing 2: The random interface class.

```
// foo.h
class foo : public rand_intf {
  int x;

  foo() {
    push_callback(
      // Set x to 1 by default
      [this]() { x = 1; }
    );

    push_check(
      // make sure x is never greater than 10;
      [this]() -> bool { return (x <= 10); }
    );
  };
};

// user_code.cc
// Create a weighted set with 2 items with equal weights.
// The set contains an illegal value of 20 that will
// cause the simulation to fail if picked.
wset<int> w( {{10, 100}, {20, 100}} );
auto f = new foo();
f->push_callback(
    // Add another callback to override the default
    [&](){ f->x = w.pick(); }
  );
f->randomize(); // execute all lambda functions in fifo order
    .
```

provides a mechanism to check for "constraint violations" by executing a list of "rule check" callbacks after the `randomize` method is called. A failed check will cause the tool to print an error message and exit.

Listing 2 shows an example of class `foo` that derives from `rand_intf` and sets a callback in its constructor that sets `x` to a default value of 1. Further, the constructor also adds a check that will enforce the rule that `x` is never greater than 10. The user code in listing 2 shows an additional callback being registered that will assign `x` to a random value by using a weighted set. Note that the set contains the illegal value of 20, which will violate the condition that `x` never be greater than 10. If picked, the value of 20 will cause the program to exit due to randomization failure. Since the program is C++, so-called "constraint violations" can be debugged using a debugger.

C. Anatomy of a sequence

When a sequence is run it generates a block of instructions that consist of a preamble and a main body. The preamble is used to set up base, scratch, index and offset registers that are used by the instructions in the main body of the sequence. Without proper setup code in the preamble, the program would access illegal addresses and start faulting continuously. Thus, proper register initialization is required for the test to perform useful work.

Since register initialization is required for most sequences, a helper class and associated sequence that randomizes and initializes registers was created. The `reg_helper` class randomizes which registers will be used for base, scratch, etc. The `reg_init_seq` will generate the preamble code that will initialize all the registers to sane and legal values. For example, base registers will be loaded with a valid address.

While developing the sequence library we found it useful to partition the sequences into *top-level* and *light-weight* sequences. A *top-level* sequence requires a proper preamble and is usually instantiated from the test. Figure 5 shows a top-level sequence `foo_seq` instantiating a register init sequence (`reg_init_seq`) that will generate the preamble. A *light-weight* sequence is typically a short sequence that requires some configuration from a top-level sequence. Figure 5 shows the main body of `foo_seq` executing the light-weight sequences `bar` and `baz`. A loop that executes a short block of code ten times is an example of a light-weight

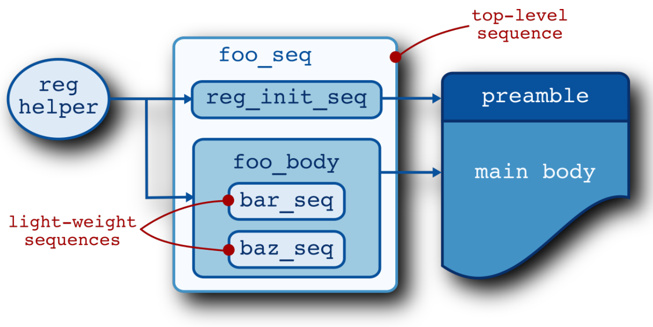

Figure 5: Top-level sequence `foo_seq` using the `reg_-helper` and `reg_init_seq` to generate a preamble while also executing nested light-weight sequences `bar` and `baz`.

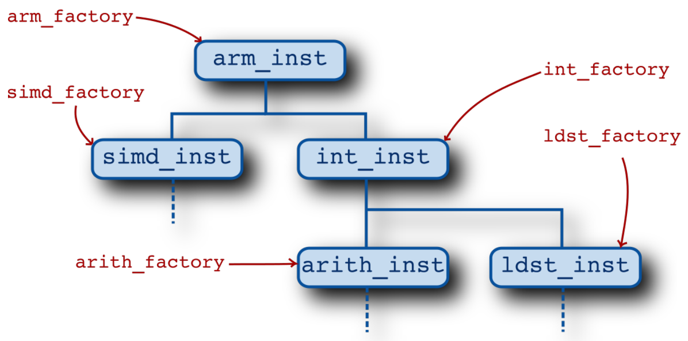

Figure 6: The SGen factories mirror the class inheritance tree.

sequence. It does not make sense to provide a full preamble for such a short sequence since the init code would likely be longer than the main body.

D. Factories

Similar to UVM, SGen uses factories to create instances of registered types; they are used heavily for creating instruction and sequence instances. However, unlike the typical usage of factories in UVM, SGen provides a hierarchy of factories that mirror the class hierarchy of the instruction and sequence libraries. Figure 6 shows how the instruction types in the `inst` package are organized. The `arm_inst` class is at the top of the hierarchy, therefore any class that derives from it can be instantiated using the `arm_factory` class. The same holds true for all of the other types in the hierarchy.

The factory class provides a utility method `get_types` that returns a vector of type names that can be used to instantiate types belonging to that factory. By selecting the appropriate factory, a test writer can restrict which instructions their sequence or test will generate. Further, the user does not even have to know the names of the individual types registered with the factory. This makes writing generic sequences and test cases easier.

E. Putting it all together

The previous subsections have discussed the basic building blocks needed to create random test programs using SGen. This subsection will present an example test that actually compiles. Listing 3 shows the `body` method of a test that will generate only simd instructions. Note that the listing makes use of the `bare_sequence` class, which is a special sequence type whose body method is a callback type that the user must override. This class is useful for creating a sequence “on-the-fly” that we don’t wish to add to the standard sequence library.

The program in listing 3 shows the following:

- line 3: A `bare_sequence` class is instantiated.
- line 6: The sequence body is assigned to a lambda function.
- lines 14 to 16: A register helper (`reg_helper`) is instantiated, configured, and randomized.
- line 18: A register init sequence (`reg_init_seq`) is instantiated using the sequence factory.
- line 19: The register helper instance that was created earlier associated with the `reg_init_seq` instance.
- line 21: The register init sequence will generate the init code in the preamble.
- line 27: Create a weighted set instance to hold instruction objects.
- line 28: The simd factory will only create simd instructions.
- line 29: The factory method `get_types` returns a vector of type names that can be used to create instruction instances later.
- line 35: Use the factory to create instruction instances.
- line 36: Constrain registers used by the instruction instance using the register helper object.
- line 38: Add instruction instance to the weighted set with a random weight.
- lines 42 to 48: Randomly pick instructions from the weighted set and send them to the driver.
- line 52: Execute the sequence until the end-of-test condition is reached (10k generated instructions in this case).

Note that by using the simd factory `get_types` method to get the list of registered instructions, the user need not know which instructions are supported by the tool. As more instructions are added the code in listing 3 will automatically include them in subsequent runs.

III. Results and Discussion

This section presents data on incremental compiles, regression and exerciser runs, and run times.

A. Incremental compile

As mentioned previously, one of the motivations for choosing C++ was fast compile times. Table 1 shows the incremental compile and link times for when a base class is modified, forcing multiple dependencies to re-compile and link. The instruction package has the most reverse dependencies, therefore modifying a file in `inst` will cause many files to recompile. Table 1 shows the compile and link times for when a base class is modified in each package. The worst case is when each package has modified files that trigger the most compiles. As table 1 shows, the worst case compile time is approximately 5 seconds.

B. Regressing Changes

An RTL designer has recently started running a small sample of random exers as part of their regression strategy for qualifying check ins. Data for recent runs by this RTL designer is presented in table 2. The regressions include mostly directed tests as well as a small selection of PPIGen and SGen exers. The exer runs contain only PPIGen and SGen random exercisers. Table 2 shows that for regressions, SGen appears to catch RTL bugs more consistently than the knob-based PPIGen. Note that directed failures do not necessarily prevent changes from being released because our regression suite is often not 100% clean. This was one reason why the RTL designer chose to start running random exercisers as part of their regressions. The reg3 series with 21 failures is the only regression run where the directed failures would have prevented release. Table 2 also shows that both tools caught RTL errors but SGen had a greater number of total failures. The RTL designer has observed that the SGen tool consistently catches bugs that the standard regression suite and PPIGen would have missed.

C. Exerciser Runs

25k exers per generator were run over a one month period. The data collected from those runs is presented in this subsection. Out of 50k runs, there were a total of 1731 failures. Interestingly, PPIGen only accounted for 171 of those failures and SGen accounted for 1560. On first glance it would appear that SGen is doing a

Listing 3: Fully working example of a test that executes only simd instructions.

```

// A bare sequence has no body and must be set by the user.
seq::bare_sequence seq("simd");
seq.set_driver(asm_driver_p());

seq.set_body([&]()
  {
      using namespace seq;
      using namespace randutils;
      using namespace inst::simd;

      // Instantiate a reg helper utility class to help with randomizing base,
      // offset, scratch and index registers.
      auto rh = std::make_shared<reg_helper>();
      rh->num_base_regs(12);                          // reserve 12 base regs
      rh->randomize();

      auto reg_seq = seq_factory::instance().create("reg_init_seq");
      reg_seq->reg_helper_ptr(rh);                    // set the reg helper instance
      reg_seq.set_driver(asm_driver_p());
      reg_seq->start();                               // This will generate the preamble code


      // Create and constrain instructions and add them to a wset with random
      // weights.

      wset<std::shared_ptr<simd_instruction>> inst_knob; // Create a wset to hold instruction objects
      auto& f = simd_inst_factory_t::instance();      // we only want simd instructions
      auto types = f.get_types();                     // Returns a vector of type names registered with the factory

      // Loop over type vector creating and configuring instances of each
      // registered type.
      for(auto& t : types)
      {
          auto inst = f.create(t);                    // create an instruction instance
          rh->constrain_regs(inst);                   // constrain the registers used by the instruction
          inst_knob.add_item(inst
                          , knobs::weight.pick()*5);  // add the instruction to the wset with a random weight
      };

      // Generate istream.
      int maxn = knobs::num.pick();   // Generate a random number of instructions
      for(int n = 0; n < maxn; ++n)
      {
          auto i = inst_knob.pick();     // pick and instruction
          i->randomize();                // randomize fields
          seq.driver_p->do_item(*i);  // send instruction to .S file
      };
  }); // end of sequence body

// execute sequence until we've generated 10k instructions.
while(asm_driver_p()->inst_count() < 10000)
{
   seq.start();                        // sequence body executed every time start is called
};
```

Table 1: Incremental compile and link times reported by unix `time` utility.

Modified base class	user time	cpu time
inst	4.5s	1.6s
seq	3.5s	1.3s
test	1.0s	0.5s
all	4.9s	1.8s

Table 2: Failures broken down by test type for recent regression and exerciser runs.

		Failures		
Series	Tests	Directed	PPIGen	SGen
reg0	421	2	0	6
reg1	446	1	0	15
reg2	417	2	0	29
reg3	410	21	16	34
reg4	388	1	1	8
exer0	47	–	0	5
exer1	46	–	0	6
exer2	41	–	9	26
exer3	35	–	1	0
exer4	55	–	1	0

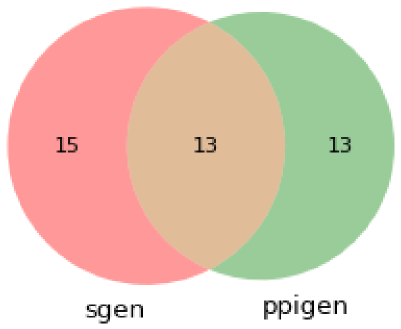

Figure 7: Venn diagram showing the overlap of failure buckets for the SGen and PPIGen generators.

better job of generating failures when compared to PPIGen. However, once we sort and classify the failures into *buckets* (or bins) based on failure signature, we see that both tools are doing a good job of generating failures.

The total number of buckets generated by both tools was 41. Figure 7 shows the overlap of buckets between the two tools. There were 15 buckets unique to SGen and 13 buckets unique to PPIGen. There were 13 buckets that overlapped. This result is encouraging because it shows that the new tool is adding value by uncovering unique buckets that PPIGen is missing.

Despite the similar bucket counts for SGen and PPIGen (28 and 26 respectively), the fact that SGen generated far more failures for the same number of total runs supports the premise that a sequence-based methodology is more efficient at finding bugs than a knob-based one. Finally, it should be noted that PPIGen is a more mature and feature-complete generator — SGen does not currently support all of the ARM v8.1 instructions. The fact they they perform similarly is highly encouraging.

D. Speed

A primary reason for selecting C++ was runtime speed. A typical exerciser run generates tests with 25k to 50k instructions. For 500 runs, the average SGen execution time was 709ms and the number of instruction generated per second averaged 31k. This run time is extremely fast and adds no overhead (in terms of computes and licenses) to our simulation times.

IV. Conclusion

In an effort to bridge the gap between directed and fully random stimulus methods, we have created *SGen*, a sequence-based assembly generator using C++11. Despite not having a constraint engine, we nevertheless are able to describe complex relationships between random variables using the random interface class in conjunction with the new *lambda* functions introduced in C++11 [11].

SGen is currently being used to verify the Cavium ThunderX2 core and results show that it is better at uncovering certain types of errors than our existing tools. In fact, SGen was able to generate far more failures than PPIGen for the same number of runs. Even though the bucket counts were similar there were still a good number of buckets unique to each tool. We can therefore conclude that the goal of augmenting our stimulus was achieved.

In addition to generating failures, we also showed that the compile and run times are extremely fast. This makes developing and debugging new sequences efficient and it also reduces the cost (in term of computes) associated with adding SGen to our verification flow.

Finally, we expect that results will improve as we continue to add new sequences and instructions to the tool.

References

[1] *Universal Verification Methodology (UVM)*. 2014. URL: http://accellera.org/downloads/standards/uvm.

[2] *C11 Standard*. ISO/IEC 9899:2011. 2011. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[3] *C++11 — Wikipedia, The Free Encyclopedia.* [Online; accessed 17-August-2016]. 2016. URL: https://en.wikipedia.org/w/index.php?title=C%2B%2B11&oldid=732204738.

[4] Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. "Functional Vector Generation for HDL Models Using Linear Programming and 3-satisfiability". In: *Proceedings of the 35th Annual Design Automation Conference*. DAC '98. San Francisco, California, USA: ACM, 1998, pp. 528–533. ISBN: 0-89791-964-5. DOI: 10.1145/277044.277187. URL: http://doi.acm.org/10.1145/277044.277187.

[5] Kwang-Ting Cheng and A. S. Krishnakumar. "Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model". In: *ACM Trans. Des. Autom. Electron. Syst.* 1.1 (Jan. 1996), pp. 57–79. ISSN: 1084-4309. DOI: 10.1145/225871.225880. URL: http://doi.acm.org/10.1145/225871.225880.

[6] Yehuda Naveh et al. *Constraint-Based Random Stimuli Generation for Hardware Verification," AI Magazine.*

[7] P. Mishra and N. Dutt. "Graph-based functional test program generation for pipelined processors". In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings.* Vol. 1. Feb. 2004, 182–187 Vol.1. DOI: 10.1109/DATE.2004.1268846.

[8] Fulvio Corno et al. "Automatic test program generation for pipelined processors". In: *Proceedings of the 2003 ACM symposium on Applied computing.* ACM. 2003, pp. 736–740.

[9] Allon Adir et al. "Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification". In: *IEEE Des. Test* 21.2 (Mar. 2004), pp. 84–93. ISSN: 0740-7475. DOI: 10.1109/MDT.2004.1277900. URL: http://dx.doi.org/10.1109/MDT.2004.1277900.

[10] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)* (Feb. 2013), pp. 1–1315. DOI: 10.1109/IEEESTD.2013.6469140.

[11] *C++ reference — lambda functions.* [Online; accessed 17-August-2016]. 2016. URL: http://en.cppreference.com/mwiki/index.php?title=cpp/language/lambda&oldid=87234.

[12] *C++ reference — Pseudo random number generation.* [Online; accessed 17-August-2016]. 2016. URL: http://en.cppreference.com/mwiki/index.php?title=cpp/numeric/random&oldid=86199.

[13] *C++ reference — std::function.* [Online; accessed 17-August-2016]. 2016. URL: http://en.cppreference.com/mwiki/index.php?title=cpp/utility/functional/function&oldid=85833.

[14] C++ reference. *C++ concepts: Callable.* 2015. URL: http://en.cppreference.com/w/cpp/concept/Callable.

[15] C++ reference. *Regular expressions library.* 2015. URL: http://en.cppreference.com/w/cpp/regex.

[16] *C++ reference — auto specifier.* [Online; accessed 17-August-2016]. 2016. URL: http://en.cppreference.com/mwiki/index.php?title=cpp/language/auto&oldid=85958.

[17] *Closure (computer programming) — Wikipedia, The Free Encyclopedia.* [Online; accessed 17-August-2016]. 2016. URL: https://en.wikipedia.org/w/index.php?title=Closure_(computer_programming)&oldid=731035439.