SGEN2: Evolution of a sequence-based stimulus engine for micro-processor verification.

Stephan Bourduas[1]
stephan.bourduas@cavium.com

Chris Mikulis[1]
chris.mikulis@cavium.com

[1]600 Nickerson Rd, Marlborough, MA 01752

Abstract— SGEN is an assembly generator written in C++11 that is currently being used to verify the ThunderX® ARM server core at Cavium. Over time, limitations of the original tool became apparent and a major refactoring effort was undertaken to address them, resulting in SGEN2. This paper will describe how OOP techniques have been used to greatly simplify the task of writing new sequences by removing the need for much of the boiler-plate code associated with resource reservation and initialization. Detailed code examples will be presented that highlight the improvements, which result in more concise and maintainable code. In addition to the code refactoring, automated exerciser tuning through the use of a genetic algorithm has been added to SGEN2; preliminary results show promise as we were able to increase the failure rate of a selected exerciser by 2×.

I. Introduction

In [1], we presented a sequence-based stimulus engine written in C++11 called SGEN. It is currently being used to verify the ThunderX® family of ARM processors at Cavium. Over the past 2 years, SGEN has gone from an experimental tool to an integral part of our verification flow. As such, it has undergone constant development and refinement. Limitations with the original tool became apparent over time and a major refactoring effort was undertaken to address them. The efforts culminated in SGEN2, which is superior to the original SGEN in terms of ease-of-use and performance. Through the use of object-oriented programming (OOP) techniques and C++11 features, layers of abstraction were added that hide and automate resource management and initialization. The improvements simplified the creation of new sequences and made it possible to easily implement complex scenarios that even commercial tools did not support. Finally, an unexpected benefit of the refactoring was a 2× performance improvement.

Another aspect of stimulus generation that we encountered was the need to manually tune our exercisers in order to find new failures, hit certain coverage goals or to recreate failures at block level. At the later stages of a project, the exerciser failure rates typically fall below 1%, which results in many wasted compute cycles. Instead of relying on an open-loop system where exercisers are run independently of prior runs, we experimented with machine learning techniques that try to use the results of prior runs to steer our stimulus towards interesting results. We implemented a genetic algorithm to try and accomplish this goal. Initial experiments show promising results as we were able to increase the failure rate of a selected exerciser.

This paper is divided into 4 sections. Section II will describe the original version of SGEN and its limitations. Section III discusses the major improvements that culminated in SGEN2. Section IV will present some more advanced code examples that illustrate how the new approach reduces and simplifies the code required to write a sequence. Finally, section V discusses how we used a genetic algorithm to improve our exerciser efficiency (i.e. failure rate).

II. SGEN 1.0

The SGEN tool evolved from a desire to better control our random stimulus. Targeting specific scenarios using our knob-based legacy tool was extremely difficult. Thus, we needed something to bridge the gap between fully random and hand-written tests. C++11 was chosen for it's rich set of data structures and algorithms, 3rd party libraries, short compile times, faster run speeds and easy debugging. The new features introduced by the C++11 standard represented a significant improvement over C++98 and many of the new features were key enablers for creating SGEN (lambda functions [2] in particular). Despite the lack of a constraint solver, SGEN is able to provide the ability to express complex dependencies between random variables [1].

To generate a valid assembly program, the registers used for storing addresses for memory accesses must be carefully managed so as to prevent them from being corrupted with garbage data during program execution.

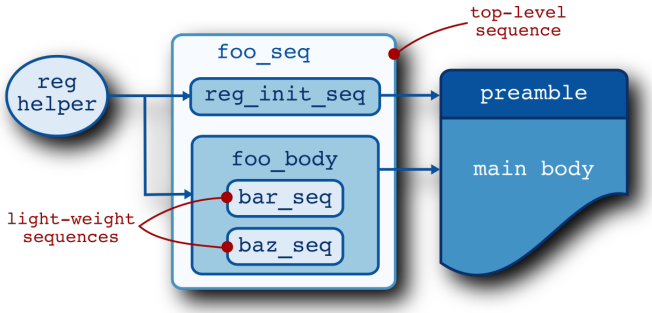

Figure 1: Top-level sequence `foo_seq` using the `reg_helper` and `reg_init_seq` to generate a preamble while also executing nested light-weight sequences `bar` and `baz` (Reproduced from [1]).

Such a corruption will render the test useless as the machine will send requests to illegal addresses. To avoid this, a sequence author had to carefully partition the set of registers into scratch and base registers; this partitioning had to be communicated to any children sequences as well. In order to facilitate the register management, several helper classes and initialization sequences were created [1]. Figure 1 shows this graphically.

Despite the helper classes, a developer still had to write a significant amount of setup code to create and configure instruction instances. Listing 1 shows a fully working example test written in the SGEN1 style. Note that the majority of the code is dedicated to reserving base registers, running a register initialization sequence, creating instruction objects and then configuring them by calling the "constrain_regs" function. Note that if the user forgets any of these steps the sequence will not work correctly when run on the core. This setup code is present in almost every sequence, which is a maintenance headache because a minor correction needs to be applied to multiple source files. Consequently, the sequences became monolithic and unwieldy; the complications of register management making reuse difficult. Further, it became impossible to interleave the outputs from multiple sequences because they would likely corrupt each other's base registers. This proved to be a major obstacle towards creating a library of reusable stimulus.

Listing 1: Fully working example of a test that executes only simd instructions (Reproduced from [1]).

```
1
2  // A bare sequence has no body and must be set by the user.
3  seq::bare_sequence seq("simd");
4  seq.set_driver(asm_driver_p());
5
6  seq.set_body([&]()
7    {
8      using namespace seq;
9      using namespace randutils;
10     using namespace inst::simd;
11
12     // Instantiate a reg helper utility class to help with randomizing base,
13     // offset, scratch and index registers.
14     auto rh = std::make_shared<reg_helper>();
15     rh->num_base_regs(12);                    // reserve 12 base regs
16     rh->randomize();
17
18     auto reg_seq = seq_factory::instance().create("reg_init_seq");
19     reg_seq->reg_helper_ptr(rh);              // set the reg helper instance
20     reg_seq.set_driver(asm_driver_p());
21     reg_seq->start();                         // This will generate the preamble code
22
23
24     // Create and constrain instructions and add them to a wset with random
25     // weights.
```

```
26
27      wset<std::shared_ptr<simd_instruction>> inst_knob; // Create a wset to hold instruction objects
28      auto& f = simd_inst_factory_t::instance();         // we only want simd instructions
29      auto types = f.get_types();                        // Returns a vector of type names registered with the factory
30
31      // Loop over type vector creating and configuring instances of each
32      // registered type.
33      for(auto& t : types)
34      {
35         auto inst = f.create(t);                        // create an instruction instance
36         rh->constrain_regs(inst);                       // constrain the registers used by the instruction
37         inst_knob.add_item(inst
38                            , knobs::weight.pick()*5);   // add the instruction to the wset with a random weight
39      };
40
41      // Generate istream.
42      int maxn = knobs::num.pick();   // Generate a random number of instructions
43      for(int n = 0; n < maxn; ++n)
44      {
45         auto i = inst_knob.pick();   // pick and instruction
46         i->randomize();              // randomize fields
47         seq.driver_p->do_item(*i);   // send instruction to .S file
48      };
49   }); // end of sequence body
50
51 // execute sequence until we've generated 10k instructions.
52 while(asm_driver_p()->inst_count() < 10000)
53 {
54    seq.start();                      // sequence body executed every time start is called
55 };
```

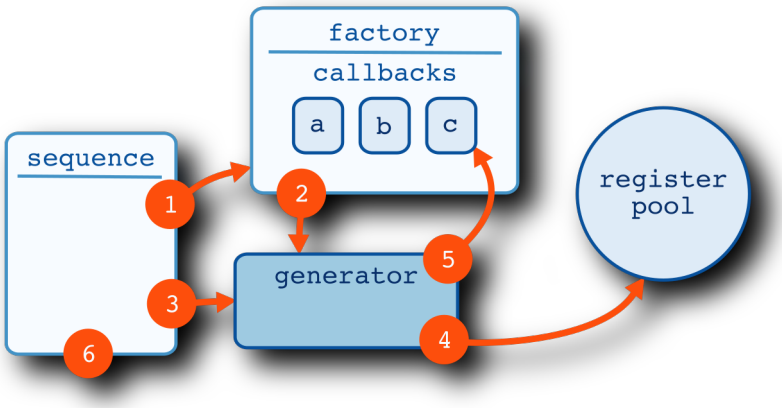

Figure 2: New flow that hides register management and initialization from the user.

III. SGEN 2.0

The ultimate goal of the sequence author is to generate some mix of instructions to achieve a certain result. On one occasion, we wished to mix barriers with a stream of memory access to find memory ordering violations. As discussed in section II, this would necessitate a lot of setup and configuration code. However, managing registers and configuring instructions are orthogonal to what the engineer is trying to accomplish. We realized that the setup code needed to be eliminated from the sequence, thereby freeing the engineer to focus on creating the desired behavior. To accomplish this, several new classes were created that hide the details of object creation and configuration from the user. Figure 2 shows the new structure of a sequence. The numbering in the figure shows the interactions between the various components in the order that they occur. Before explaining the interactions we will first describe each individual component.

A. The Register Pool

The register pool class centralizes the management of registers. A client can reserve a register by calling the "get" method, which returns a wrapper class that implements RAII (resource acquisition is initialization) [3]. RAII is a simple yet elegant concept where a wrapper object has ownership of a resource until it is destroyed, at which point the resource is automatically released. In the case of a register wrapper object, the register is returned to the free pool upon destruction. Listing 2 shows a block of code where a guard object is created upon entry. When execution reaches the end of the block, the guard object goes out of scope and is destroyed. The wrapper class will automatically return the register to the pool when it's destructor is called, thus removing the need for the coder to explicitly free the resource. The use of RAII to manage registers makes it possible to interleave sequences without needing to keep track of register allocations.

Listing 2: Reserving a register using the RegisterPool class.

```
{
    auto guard = RegisterPool::instance().get()     // reserved register managed by a guard object
    auto reg = guard->val();                        // access the managed object
     ...
};
// guard object destroyed; register automatically returned to register pool
```

B. The Instruction Generator

The instruction generator class provides a easy way for a user to obtain and randomize instruction objects. It is a variation of the *generator* [4] design pattern. The class provides a single "next" function that returns a randomly selected instruction that has already been configured using sane defaults. The generator class can be parametrized to return instructions of any supported type (e.g. loads, stores, atomics, etc). For convenience, the class also implements the necessary interface required by the *range-based for loop* [5] that was introduced in C++11. Listing 3 shows a generator object, parametrized to generate only load instructions, being instantiated and used in a range-based for loop. Note that the "next" function is called implicitly by the for loop as it iterates over the generator object.

Listing 3: Generator object being instantiated and used in a range-based for loop.

```
1 auto gen = InstGen<ld_inst>(); // Instantiate generator
2 for(auto i : gen)              // range-based for loop
3 {
4     i->randomize();
5     driver->do_item(*i);
6 };
```

Internally, the generator class uses *lazy initialization* [6] to defer register reservation and instruction instance creation until the first use of the "next" function. Lazy initialization simply means to defer some calculation or object creation as long as possible to improve performance or conserve memory. It is used here to delay or avoid needlessly locking a shared resource. For example, a test may instantiate several sequences that each instantiate generator objects. The test may randomly select to run only a subset of the sequences, which would make reserving registers for the unselected ones unnecessary. Lazy initialization prevents registers from being reserved and instruction objects being needlessly created by these sequences.

To support customization, *lamda* functions can be optionally attached to the generator. An example of customization via lambda functions in shown later in section IV.

C. The Instruction Generator Factory

The instruction generator factory class takes care of instantiating a generator class and then customizing it by attaching callbacks. The factory provides the most commonly used default configurations options such as: base register initialization, alignment of immediate values, exclusion of instructions that are illegal for the current configuration or that need to be used with care (e.g. branch instructions, and more). In the typical case, the defaults are usually sufficient and don't need to be overridden. For advanced usages, the default can be customized, but the gory details are hidden from the novice user.

D. The Register Callback Class

In SGEN1, setting base and scratch registers was accomplished using the "constrain_regs" virtual function as shown in listing 1. When called, the function would create a lambda function that would copy the weighted set object passed to it. The lambda function would then be pushed into the fifo of callbacks that are executed when the "randomize" method is eventually called on the instruction object. This process was repeated for every instruction instance created by the sequence, resulting in many unnecessary copies being made. Since the registers needed to be properly configured in each sequence, the code was duplicated in many source files.

The register callback class (RCB) was created to remove the need for the "constrain_*" functions to be called in each sequence. A client class can derive from the RCB and use the "register_*" functions to request that a field be set when the appropriate "set_*" method is called. This is shown visually in Figure 3, which shows objects "load" and "store", each having a field "Xn" that needs to be set. The figure shows the following sequence:

1. The load and store objects are created.
2. The newly created objects register their Xn objects with the RCB by calling "register_base".
3. The "set_base" method is called by an owning object/scope.
4. The RCB loops over the list of registered objects and sets them to appropriate values.

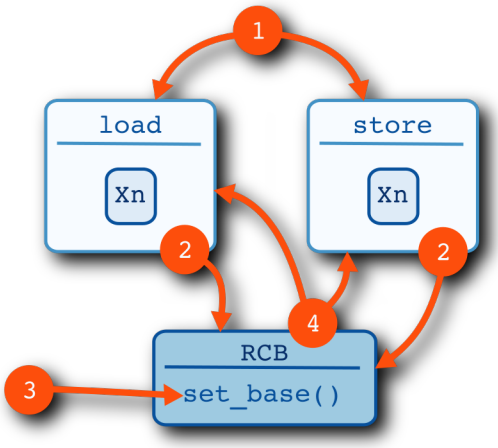

Figure 3: The register callback class provides a more efficient means of setting register values

The calling class shown in step 3 is typically a generator object described in section B. The generator passes a weighted set object that is used by the RCG to randomize the set values. The RCG makes no copies of the weighted set object (unlike the "constrain_*" methods), which result in a significant performance improvement when compared to SGEN1. We have recorded between 50–200% improvement.

E. Putting it all together

Now that the individual components have been described, we will outline here how they work together to provide the abstractions necessary to eliminate setup code from the sequence body. The events shown in fig. 2 are:

1. The sequence uses the generator factory to create a generator object.
2. The factory attaches to the generator object appropriate callbacks (i.e. lambda functions) that will perform the necessary initializations.
3. The sequence requests an instruction from the generator for the first time.
4. The generator reserves the necessary registers from the register pool.
5. The generator executes all attached callbacks before returning an instruction instance to the sequence.
6. The sequence terminates and goes out of scope, which causes the generator object to destruct, automatically returning all owned registers back to the pool.

Listing 4 shows what the refactored SIMD sequence from listing 1 looks like. Line 2 creates a generator instance using the factory; the resulting object is fully configured and ready for use. Instructions are randomly selected inside the loop on line 3. Recall that `inst_gen->next()` is called implicitly in the range-based for loop. It is immediately obvious that sequence body has been greatly simplified: *only 2 lines of code are required to create a generator and to get ready-to-use instruction objects!* Since the generator object has sole ownership of its registers, the output of this sequence can be safely interleaved with other sequences. Stimulus libraries can now be composed of smaller building blocks that can be easily combined to create new sequences.

Listing 4: SIMD sequence using the new Instruction-Generator class.

```
1 auto num randutils::random_number<int>::select(500, 1500);
2 auto inst_gen = InstructionGeneratorFactory<inst::simd>::create(this, num);
3 for(auto i : inst_gen);
4 {
5     i->randomize();
6     driver_p->do_item(*i);
7 };
```

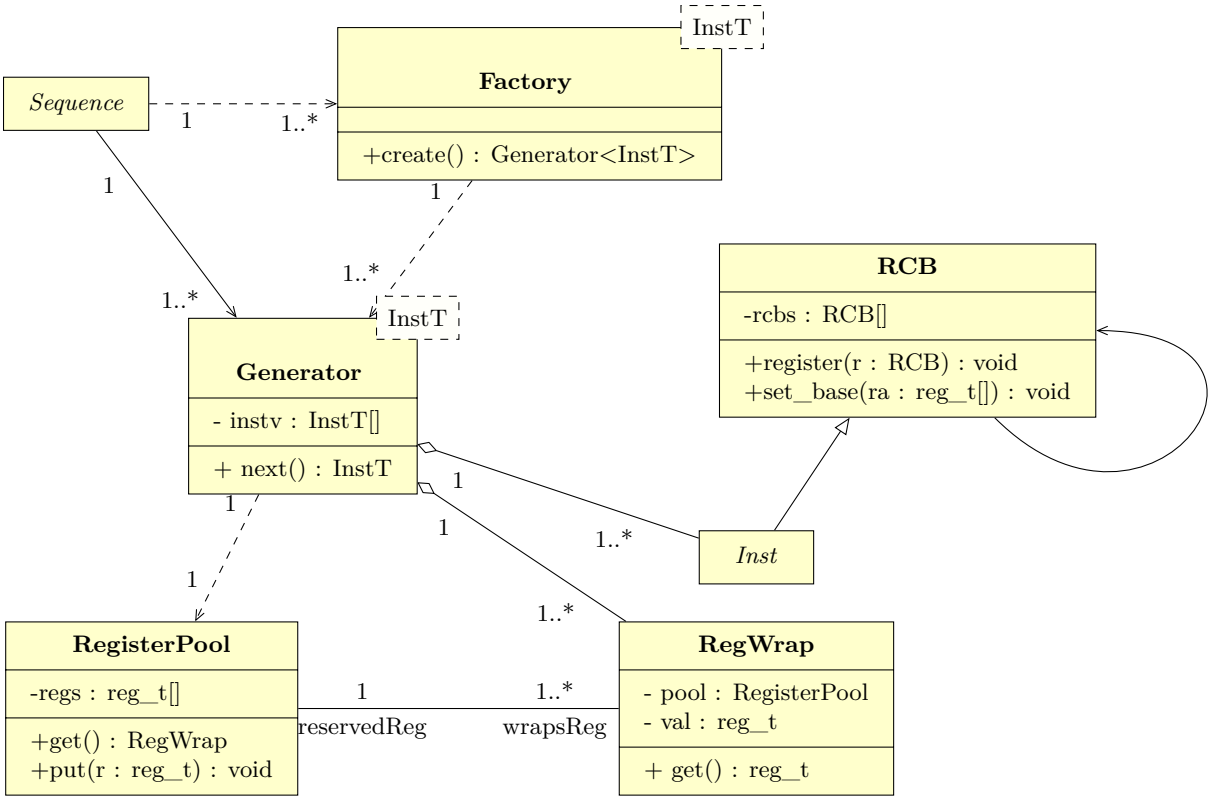

Figure 4: UML Class Diagram showing the relationships between the new classes described in this section.

Figure 4 shows the UML [7] class diagram of the new classes discussed in this section. The *Sequence* class uses the *Factory* class to create and configure a *Generator* object. The generator in turn uses the *RegisterPool* to create *RegWrap* objects that manage ownership of reserved registers. Note that the multiplicity of the *RegisterPool* object is 1 since there is only a single pool of registers in the system. The generator also creates (and owns) instructions, or instances of *Inst*. It also calls the "set_base" method that the *Inst* class inherits from *RCB* (the register callback class). The RCB class has a self-referencing association because it holds references to other RCB sub-classes.

In addition to improving flexibility and eliminating the need to configuration code, we also saw a significant performance increase due to improved encapsulation and program structure. A comparison of 10k runs of each tool showed that SGEN2 ran approximately *twice* as fast! SGEN1 generated instructions at a rate of 55k per second while SGEN2 generated them at a rate of 118k per second.

Finally, the new methodology facilitated the creation of a complex sequence that even several tool vendors and our legacy tool did not support. The new sequence "hops" back and forth between exception (i.e. privilege) levels. This behavior is essential for verifying the virtual memory subsystem. The *hopper* sequence will be discussed in more detail in section IV.

F. ASM Driver

Before moving on to the code examples, we discuss here a specialized driver component that has been added to SGEN2. A elf file contains many sections and it is often necessary to generate assembly code that will populate those sections. To enable the creation of complex sequences such as the "hopper" sequence mentioned in the previous section, it was necessary to create a specialized driver that allows a test writer to add code to any legal section. To accomplish this, sub-drivers were created for each section. The top-level driver simply owns all of the sub-drivers and provides "get" methods that return references to sub-drivers. Listing 5 shows example code where drivers for the EL3 and EL2 text sections are retrieved

from the top-level driver. Each sub-driver can be used to populate their sections with assembly code.

Listing 5: Using the ASM driver to populate the text segments for ELs 2 and 3.

```
// get reference to el3 driver
auto el3_driver = asm_driver_p->get_section_driver(asm_section_e::TEXT_EL3);
el3_driver->do_string("add x1, x2, x3");
 ...

// get reference to el2 driver
auto el2_driver = asm_driver_p->get_section_driver(asm_section_e::TEXT_EL2);
el2_driver->do_string("ldr x1, [x10]");
 ...
```

IV. Advanced Code Examples

Thus far, we have described the new features of SGEN2 and provided a simple example in listing 4. This section will present working examples of sequences that are actually being used to verify the ThunderX ARM server core. The code examples may be difficult to follow for those not fluent in C++11, however the important parts have been well commented. The goal of presenting these examples is to show how relatively easy it is to create new stimulus using SGEN2 without requiring a lot of boiler-plate code.

A. Overriding default callbacks

By default, the instruction generator factory will instantiate a generator object that will randomly return instructions from the set of all available instructions. Section III described how the factory attaches callbacks to the generator object that provide sane defaults that can safely be used most of the time. Two of those callbacks are:

1. A filter function (`filter_fn`) that is used to exclude certain instructions from the default generator (branches are excluded by default for example).
2. A weight function (`weight_fn`) that returns a weight that will determine the probability of an instruction being picked from all the available instructions.

By overriding these two functions, we can easily control which instructions our sequence will generate without needing to worry about resource allocation and management.

In order to exercise a specific part of the design, we had a need to create a sequence that generates only load/store, prefetch, atomic, SEV and IC instructions. Further, we wanted CAS and CASP instructions to be generated more frequently than the others. This was relatively easily accomplished using the instruction generator factory and overriding the default filter and weight functions. Listing 6 shows a sequence that overrides the default filter function with one that will accept the instructions we want. The default weight function is replaced with a custom one that increases the relative weight (and thus the frequency) of CAS and CASP instructions.

Listing 6: Example of a sequence that overrides the default callbacks that were installed by the factory.

```
// Instantiate our instruction generators
int maxn = randutils::random_number<>::select(100, 1000);
using InstFactT = InstructionGeneratorFactory<inst::arm_instruction>;
auto inst_gen = InstFactT::create(seq, *driver_p, maxn);

// Override the original filter function with a custom one that rejects all
// instructions except for loads and stores, atomic, prefetch and SEV and IC
// instructions.
inst_gen->filter_fn = [orig_filter_fn](const InstFactT::InstT& i)
{
    if((i->is_ldst()
        || i->is_atomic()
        || i->is_prefetch()
        || std::dynamic_pointer_cast<const inst::hint::SEV>(i)
        || std::dynamic_pointer_cast<const inst::IC>(i))
```

```
16      {
17          return false; // keep this instruction
18      }
19      else
20      {
21          return true; // discard this instruction
22      };
23  };
24
25  // Override the original weight function
26  auto def_weight_fn = inst_gen->weight_fn; // default weight function
27  inst_gen->weight_fn = [orig_weight_fn](const InstFactT::InstT& i)
28  {
29      auto w = def_weight_fn(i); // call default weight function
30      if(std::dynamic_pointer_cast<const inst::CAS>(i)
31              || std::dynamic_pointer_cast<const inst::CASP>(i)
32        )
33      {
34          // increase weight of CAS/CASP instructions
35          w*=randutils::random_number<int>::select(1,15);
36      };
37      return w;
38  };
39
40  // Now generate instructions using the inst_gen object
41  // from above.
42  for(auto i : inst_gen)
43  {
44      i->randomize();
45      driver_p->do_item(*i);
46  };
```

B. EL "HOPPER" SEQUENCE

The EL "hopper" sequence jumps back and forth between 2 execution levels. Figure 5 shows the program execution graphically. The program starts in el3 and executes some random number of instructions before jumping to the el2 start label. Once in el2, the program executes until the end of the first section and then jumps back to the corresponding el3 return label. The process of jumping to el2 and returning to el3 is repeated until execution reaches the end label.

The text sections corresponding to each execution level are generated using sub-drivers (previously discussed in section III.F). The sequence creates a random number of code *blocks*, where each block consists of the following:

1. A start and return label for each EL.
2. A block of random instructions for each EL.
3. A macro call that changes EL levels (e.g. `EL2_TO_EL3` and `EL3_to_EL2`).

Listing 7 shows the code for the sequence. The loop that creates the blocks is shown on line 49. Execution of the generated code will:

1. Branch to the EL3 start label.
2. Execute random instructions at EL3
3. Jump to the EL2 start label using the `EL3_TO_EL2` macro.
4. Execute random instructions at EL2.
5. Jump to the EL3 return label using the `EL2_to_EL3` macro.

Steps 1 to 5 are repeated `num_blocks` times to create a test that will jump back and forth several hundred times. It is obvious from listing 7 that this is a non-trivial example. Until now, this type of behavior could only be achieved using hand-crafted tests. Using SGEN2, the entire sequence is expressed in *less than 100 lines of code!*

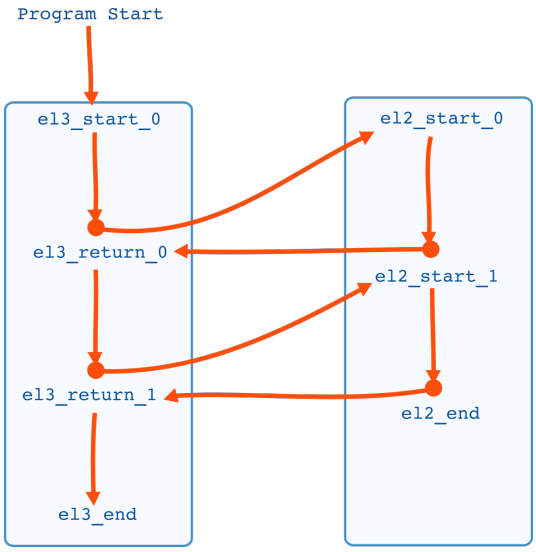

Figure 5: Program flow of the EL Hopper sequence shown graphically.

Listing 7: Example of a sequence that "hops" between EL levels 3 and 2 using the improved SGEN2 style.

```
// Get drivers for el2 and el3 text segments.
auto& el3_driver = asm_driver_p->get_section_driver(asm_section_e::TEXT_EL3);
auto& el2_driver = asm_driver_p->get_section_driver(asm_section_e::TEXT_EL2);

// Create instance specific start and return labels that will go in
// the main .text section
auto el3_start_label = tfm::format("%s_start", get_instance_name());
auto el3_return_label = tfm::format("%s_return", get_instance_name());

{
    // Load address of start label into scratch register and branch to
    // it.
    auto wrapper = RegisterPool::instance().get();
    auto xd = wrapper->val();
    asm_driver_p->do_item( inst::DLA(xd, el3_start_label) );
    asm_driver_p->do_item( inst::BR(xd) );
    asm_driver_p->do_string( el3_return_label +":" );
}; // wrapper deallocates and register returned to pool

// text el3 start label
el3_driver.do_string(tfm::format("%s:", el3_start_label));

// Declare a utility function that will return an instruction
// generator that will populate the appropriate .text section
// corresponding to the requested el value.
auto create_generator = [this, asm_driver_p](int el)
{
    asm_section_e section = (el == 3) ? asm_section_e::TEXT_EL3 : (el == 2) ? asm_section_e::TEXT_EL2 :
        asm_section_e::TEXT_EL1;
    auto& text_driver = asm_driver_p->get_section_driver(section);
    auto inst_gen = InstructionGeneratorFactory<inst::arm_instruction>::create(*this, text_driver);
    return inst_gen;
};

// Create a map of weighted sets where the key type is an int and the
// value type is an instruction generator.
std::map<int, decltype(create_generator(3))> gen_wset;
gen_wset[3] = create_generator(3); // create instruction generator for el3
gen_wset[2] = create_generator(2); // create instruction generator for el2
```

```
39
40
41  // Declare lambda function that will create unique labels
42  int label_count = 0;
43  auto gen_label = [this, &label_count](const int& el, const int& cnt) {
44     return tfm::format("%s_el%d_%d", get_instance_name(), el, cnt);
45  };
46
47  // Randomly select how many blocks to generate for each el level.
48  int num_blocks = randutils::random_number<int>::select(5, 20);
49  for(int num = 0; num < num_blocks; ++num)
50  {
51     std::string el3_label = gen_label(3, num);
52     std::string el2_label = gen_label(2, num);
53
54     // generate block of instructions for el3
55     int maxn = randutils::random_number<>::select(10, 50);
56     for(int n = 0; n < maxn; ++n)
57     {
58        auto i = gen_wset[3]->next_ptr();
59        i->randomize();
60        el3_driver.do_item(*i);
61     };
62     el3_driver.do_string("EL3_TO_EL2(%s)", el2_label); // jump to el2
63     el3_driver.do_string("%s:", el3_label);                  // return from el2 label
64
65     // generate block of instructions for el2
66     el2_driver.do_string("%s:", el2_label); // target label for EL3 to El2 jump
67     maxn = randutils::random_number<>::select(10, 50);
68     for(int n = 0; n < maxn; ++n)
69     {
70        auto i = gen_wset[2]->next_ptr();
71        i->randomize();
72        el2_driver.do_item(*i);
73     };
74     el2_driver.do_string("EL2_TO_EL3(%s)", el3_label); // jump back to el3.
75  };
76
77  {
78     // Return to el3_return_label that was declared above.
79     auto wrapper = RegisterPool::instance().get();
80     auto xd = wrapper->val();
81     el3_driver.do_item( inst::DLA(xd, el3_return_label) );
82     el3_driver.do_item( inst::BR(xd) );
83  }; // wrapper deallocates and register returned to pool
```

V. Automation through machine learning

Towards the end of a project, exerciser pass rates typically fall below 1% (often converging towards 0.1%). When the failure rates are so low, it is difficult to uncover new bugs for 2 reasons. First, finding new bugs may take days or weeks of runs due to the extremely low failure rates. Second, the exerciser recipes have reached a point of diminishing returns where newly generated tests are no longer stimulating the design in new ways. At this point, it is necessary to hand-tune exercisers in an attempt to steer them towards new behavior to hopefully uncover latent bugs. This process is ad-hoc and time consuming. In an effort to automate this process, we have added to SGEN2 the ability to tune our exercisers using a *genetic algorithm* [8].

A genetic algorithm is a heuristic algorithm inspired by the process of natural selection [8]. The algorithm starts with an initial population of candidate solutions that are *mutated* to create a new *generation*. The algorithm stops iterating when a generation satisfies the termination condition (e.g. maximum number of iterations reached, target criteria reached, etc).

For our initial experiments, we selected the sequence mentioned previously where we wished to mix barrier instructions with memory accesses. To achieve a useful mix, we needed to tune the relative weights of each instruction class. Too many barriers and the machine will sit idle for long periods; too few and the probabilities of anything interesting happening decrease. In either case, our chances of finding bugs decreases.

Our intent was to use a genetic algorithm to find the optimal ratio of barrier to memory access instructions. The sequence was instrumented so that all randomly generated weights and configuration parameters could be saved and loaded from a file.

In our environment, a typical random test consists of between 25–50k instructions and run times are typically between 1–4 hours. In order to speed up the algorithm, we decided to apply the genetic algorithm to shortened exerciser runs of approximately 10k instructions. This resulted in relatively short iteration times when compared to full exerciser runs. Since we are are ultimately after simulation failures, we defined our objective function to be simply whether a test passed or failed. We decided to avoid using coverage data for the reasons discussed in [9].

The experiment was conducted as follows:

1. The initial population consisted of 1000 shortened exercisers runs.
2. For each run, we recorded whether the test passed of failed and saved the weights to a file.
3. The next generation consisted of:
   - The weights of all failed tests from the previous generation.
   - Mutated weights from failing and passing tests.
   - Newly generated weights.

The mutator function must be carefully chosen so that a mutated test is similar to the original without being *too* similar or different. Too similar and the mutated test will be redundant; too different and the new test approaches random generation. In our experiment, the mutator function modifies each weight in a test by $\pm 1\%$. We felt that this allowed the ratio between instruction classes to change slowly enough to yield useful results.

Steps (2) and (3) were repeated a maximum of 10 times or until 100 failing states were found. The failing states were then used for normal length exerciser runs. For our experiment, the algorithm generated 100 failing states after 6 generations. When we ran the instrumented sequence initialized with those states the exerciser failure rate increased from 1% to 2.5% for 2000 tests. We believe that this proves that our approach has potential. We are currently in the process of instrumenting other sequences in order to collect more meaningful data.

VI. Conclusion

This paper discussed how the limitations of the original version of SGEN [1] were addressed to create SGEN2. The refactoring used object-oriented techniques such as *factories*, *generators*, and *lazy initialization* to provide layers of abstraction that greatly reduce the overhead associated with authoring new sequences. Sequences can now be written more concisely because setup code is no longer necessary. The new style enabled a SGEN1 sequence that was approximately 50 lines long to be rewritten using only 5 lines of code. In addition, register management has been hidden from the user such that they are dynamically acquired and released by a sequence using the principle of RAII. Since register reservations are owned by specific sequence instances, instruction streams from different sequences can now be safely interleaved. The refactoring also improved performance; simulation results showed that the new sequence style resulted in a $2\times$ speedup. Finally, we have used a genetic algorithm to automate exerciser tuning. Initial results have been promising and preliminary experiments showed that the application of the genetic algorithm to a selected exerciser could increase the failure rate from ~1% to ~2%.

In the future, we plan to continue creating new sequences and exploring new ways to generate stimulus such as using multi-threading to interleave sequences. We also plan to continue exploring how machine learning techniques such as clustering and regression analysis can help improve the effectiveness of generated stimulus.

References

[1] Stephan Bourduas and Christopher Mikulis. "Micro-processor Verification Using a C++11 Sequence-based Stimulus Engine". In: *Proceedings of DVCON*. San Jose, California, USA, 2017.

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

[3] *C++ reference — RAII*. [Online; accessed August-2017]. 2017.

[4] Wikipedia. *Generator (computer programming) — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-August-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Generator_(computer_programming)&oldid=785714360.

[5] C++ reference. *Range-based for loop*. 2017. URL: http://en.cppreference.com/w/cpp/language/range-for.

[6] Wikipedia. *Lazy initialization — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-August-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Lazy_initialization&oldid=780969164.

[7] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[8] Wikipedia. *Genetic algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-December-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=813376367.

[9] Stan Sokorac. "Optimizing Random Test Constraints Using Machine Learning Algorithms". In: *Proceedings of DVCON*. San Jose, California, USA, 2017.