SGEN2: Evolution of a Sequence-Based Stimulus Engine for Micro-Processor Verification

Stephan Bourduas (stephan.bourduas@cavium.com)
Christopher Mikulis (chris.mikulis@cavium.com)

1) Introduction

- SGEN is a sequence-based stimulus engine written in C++11 that generates assembly programs used to verify the Cavium ThunderX3® ARM microprocessor.
- SGEN1 was originally presented at DVCON 2017:
  - Limitations with the original methodology became apparent over time and a refactoring effort was undertaken to address them.
- SGEN2 is the new and improved version:
  - OOP techniques and C++11 features were used to add layers of abstraction to simplify creation of new sequences.
  - Automates resource management and initialization.
  - Reduces the amount of code required for new sequences.
  - Facilitates code sequence reuse.
  - 2x performance improvement.
  - Support for machine learning was added to automate exerciser tuning:
    - A genetic algorithm was used to improve the failure rate of a selected exerciser.

2) SGEN1

- SGEN evolved from a need to better control stimulus:
  - Bridge the gap between directed and knob-based generation.
  - C++11 was chosen for its rich set of data structures, 3rd party libraries, fast compile and run times and ease of debugging.
  - No constraints:
    - Randomization provided by C++11 lambda functions.
    - Successfully used to create directed random sequences in production.
  - First version required helper sequences and manual tracking of base registers to generate valid programs:
    - This made sequences difficult to reuse and necessitates a lot of boilerplate.
    - Makes maintenance and reuse difficult.

3) SGEN2

- SGEN2 provides layers of abstraction that automates repetitive tasks such as object configuration and resource reservation and release through the use of OOP techniques.
- The most important new classes are:
  - The register pool centralizes and simplifies register management by using RAII.
  - The instruction generator class provides a simple way for a user to create and randomize instruction objects.
  - Uses lazy initialization as well as callback hooks to enable customization.
  - The instruction generator class returns pre-configured generator objects using the most commonly used defaults.
  - The user can override the defaults by attaching <c++11> lambda functions.

4) Automation Through Machine Learning

- Hand-tuning exercisers is often necessary towards the end of a project when failure rates fall below 1%.
  - Hand-tuning is ad-hoc and time consuming.
  - We attempted to automate tuning by using a genetic algorithm.
    - Our goal was to generate initial exerciser states that had a higher chance of failure.
  - The final population of tuned exercisers were generated as follows:
    1. The initial population consisted of 1000 shortened exerciser runs.
    2. The pass/fail status of each test as well as configuration weights were saved.
    3. The next generation consisted of:
      - The weights of all failed tests from the prior generation.
      - Mutated weights from failing and passing tests.
      - Newly generated weights.
    4. Steps (2) and (3) were repeated a maximum of 10 times or until 100 failing tests were found.
  - The algorithm generated 100 failing states after 6 generations.
    - We ran 2000 exercisers initialized with the failing states and the failure rate increased from 1% to 2.5%.

5) Conclusions and Future Work

- The limitations of SGEN1 were addressed through a large refactoring effort to create SGEN2.
  - OOP techniques such as factories, generators, lazy initialization and RAII were used to provide layers of abstraction that greatly reduce the overhead associated with creating new stimulus.
  - Code reduction was significant — often by as much as 70-90%.
  - Resulting code was less buggy and easier to maintain and reuse.
  - Please see full paper for examples of more complex sequence code.
- A genetic algorithm was used to automate exerciser tuning.
  - Initial results have been promising — failure rates for a selected exerciser were increased from ~1% to ~2.5%.
- Future work includes:
  - Continuing to improve SGEN by adding features such as multi-threading and stateful sequences.
  - Explore other machine learning techniques such as clustering, partitioning and logarithmic regression to improve exerciser efficiency.