On Decomposing Complex Test Cases for Efficient Post-silicon Validation

Harshitha C[1], Sundarapalli Harikrishna[1], Peddakotla Rohith[1], Sandeep Chandran[1], and Rajshekar Kalayappan[2]

[1]Indian Institute of Technology Palakkad, India
[2]Indian Institute of Technology Dharwad, India
{112003002,121901045,121901036}@smail.iitpkd.ac.in, sandeepchandran@iitpkd.ac.in, rajshekar.k@iitdh.ac.in

Abstract— In post-silicon validation, the first step when an erroneous behavior is uncovered by a long-running test case is to reproduce the observed behavior in a shorter execution. This makes it amenable to use a variety of tools and techniques to debug the error. In this work, we propose a tool called *Gru*, that takes a long execution trace as input and generates a set of executables, one for each section of the trace. Each generated executable is guaranteed to faithfully replicate the behavior observed in the corresponding section of the original, complex test case independently. This enables the generated executables to be run simultaneously across different silicon samples, thereby allowing further debugging activities to proceed in parallel. The generation of executables does not require the source code of the complex test case and hence supports privacy-aware debugging in scenarios involving sensitive Intellectual Properties (IPs). We demonstrate the effectiveness of this tool on a collection of 10 EEMBC benchmarks that are executed on a bare-metal LEON3 SoC.

I. Introduction

Recent industry-wide surveys have found very few designs to achieve first-silicon success [1]. This is because pre-silicon verification tools and strategies do not scale well with increasing design complexities. This makes post-silicon validation an indispensable step before a chip enters mass production.

During the post-silicon validation phase, long-running and complex test cases as well as representative target applications are executed on early silicon samples to uncover any design errors (or bugs) that may have been missed by pre-silicon verification techniques. However, the limited visibility into the internal functioning of the chip poses a significant challenge in debugging the erroneous behavior seen. Design-for-Debug (DFD) structures such as trace buffers are inserted into the chip during the design phase and exercised during the post-silicon validation phase to increase such visibility [2, 3]. The near-native execution speeds offered by the early silicon samples results in large volumes of execution trace being captured even for a relatively short executions (of a few ms) [4]. Therefore, debugging the erroneous behavior observed by analyzing only the collected traces becomes a tedious and time consuming task.

This work was partly sponsored by Semiconductor Research Corporation (SRC).

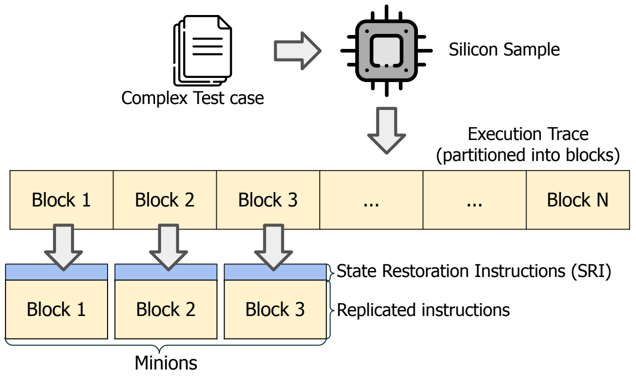


Fig. 1. Overview of the proposed approach

In this work, we propose a tool called *Gru*, that decomposes a large execution trace and synthesizes a set of smaller test cases called *Minions*, from it. A synthesized test case (or Minion) is guaranteed to replicate the behavior of a small section of the complex test case when it is re-executed on the silicon sample. Such decomposition of the original behavior offers the following advantages. Each Minion can be executed independently of the other and hence, further debugging activities can proceed in parallel. This could potentially reduce the time taken to localize the bug. The size of Minions created by the tool is configurable. This enables synthesis of Minions that are small enough for their behaviors to be analyzed by other bug localization and verification tools. Finally, our tool requires only the execution trace to create Minions. This helps in replicating errors that were observed at the customer site without the need for the customer to share any sensitive intellectual properties (IPs) such as the source code or executable with the validation team.

Figure 1 illustrates our proposed technique. The execution trace generated by running a complex test case on the silicon is partitioned into fixed sized *Trace Blocks*. The instructions captured in each trace block are replicated in the synthesized Minion. The replicated instructions will behave exactly as they did in the complex test case only if the starting state of each block matches with the corresponding state during the execution of the complex test case. We insert additional instructions, called *State Restoration Instructions (SRIs)*, into the Minion such that its execution will restore the state of the chip to the state that prevailed at the start of the corresponding trace block. The Minion is crafted in a way that SRIs execute first and then transfers control to the replicated instructions after the state is restored.

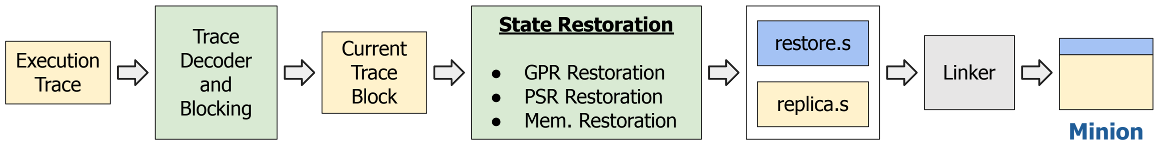


Fig. 2. Steps to generate a Minion

We discuss three techniques to infer the starting state of a trace block by analyzing the execution traces and elaborate the strategy to insert state restoration instructions automatically. Finally, through the execution traces captured by executing 10 EEMBC benchmarks on the bare-metal for $10ms$ (averaging $369K$ trace messages occupying $5.6MB$), we show that generating Minions, takes only $6s$ on average.

II. Related Work

Test case generation has been extensively studied in the past and several test case generation strategies have been developed. Several strategies adopt an exploratory approach where the test cases are generated in a random or pseudo-random manner [5]. These test generation strategies are fast and can generate a lot of test cases in a short duration. Although this helps in verifying large portions the input space, this strategy does not guarantee complete coverage of the input space. Formal-methods based test generation is another strategy to generate test cases that uses a model of the underlying system to direct test generation [6]. This guides test generation towards regions previously unexplored. These works are orthogonal to ours. We generate Minions to replicate an observed behavior and not to expose design errors.

Several works have proposed techniques to localize bugs during post-silicon validation using test cases [7–10]. These works embed self-checks into the test case intelligently. For example, a block of code is replicated inside the test case, and the results produced by the two copies of code are checked for equality. Any mismatch localizes the bugs to a small set of instructions [7]. Similar approach of replicating code blocks can be used to check for errors in memory and control flow [8]. Our work is orthogonal to such bug localization strategies and can complement them by generating Minions from an execution trace which can then be transformed suitably.

Our proposed tool positions itself to be the first step in debugging an error observed during post-silicon validation by decoupling the replication of erroneous behaviors observed from localizing and analyzing the root cause of the bug. This paves the way for using other tools and techniques on each Minion separately (in parallel). For example, the Minions generated can be executed on an Instruction Set Simulator (ISS) to obtain the "Golden" end state which can then be compared against the corresponding on-chip state to localize faulty state elements [11]. Other strategies such as Run-Pause-Resume (RPR) [12] can also be used on each Minion for further debugging of the observed error.

III. Gru Architecture

Figure 2 shows the steps involved in generating a Minion. The execution trace is passed through a trace decoder that converts the captured trace from binary to a human readable format. The trace is also partitioned into trace blocks as it is decoded. The instructions in the current trace block are copied into `replica.s` as is. The label `test` is given to the first instruction of the trace block (which is copied into `replica.s`). We keep track of the smallest PC value encountered in the current trace block and then subtract this from the PC of all the instructions to determine the offset of each instruction. The instructions are written to `replica.s` at appropriate offsets and `nop`s are written if a particular PC within the minimum and maximum PC values seen in the trace is not observed (see line numbers 12 – 14 of `replica.s` in Figure 3(c)).

The next step is to analyze the current trace block to identify state elements which needs to be restored. All state elements that are produced in some earlier trace block and are consumed in the current trace block have to be restored when generating the corresponding Minion.

Once the state elements are identified, the values that should be written to these state elements to restore the state are identified using *State Inference Techniques* (discussed in Section IV). After the values are determined, SRIs corresponding to each state element are written to `restore.s` (discussed in Section V).

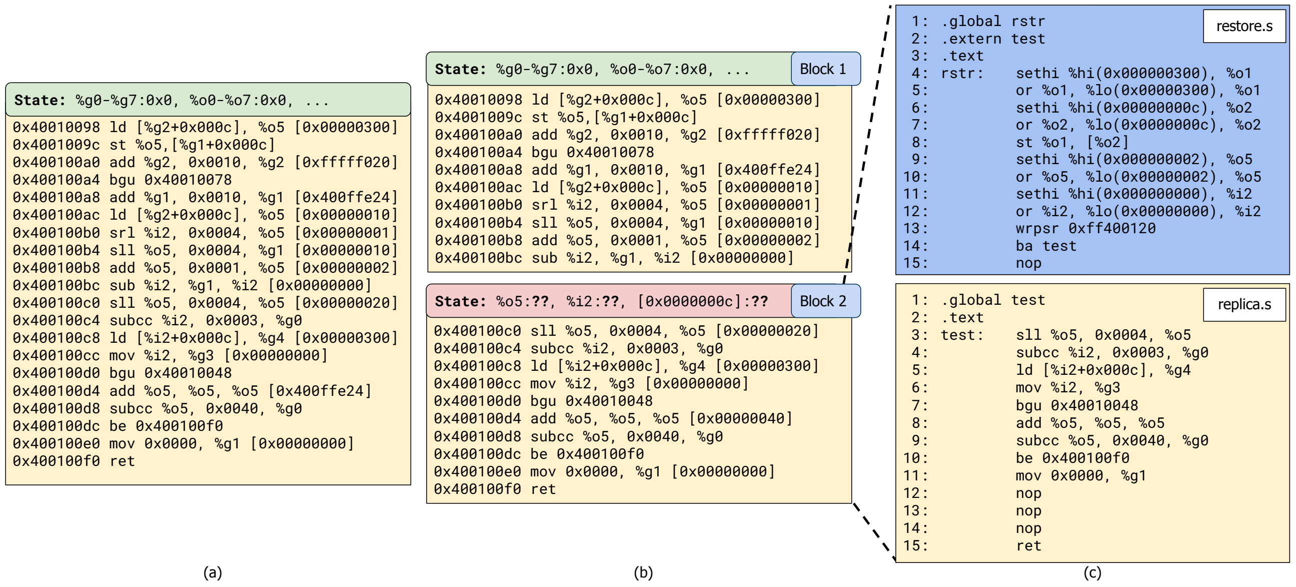
Once the `replica.s` and `restore.s` is constructed, it is compiled and linked using a standard compiler such as GCC to generate the Minion (discussed in Section V.D).

IV. State Inference Techniques

Figure 3(a) shows a sample execution trace collected from a LEON3 SoC and decoded into a human-readable form. Consider the first line of the sample trace. The first hexadecimal number (`0x40010098`) is the Program Counter (PC) of the instruction. This is followed by the instruction disassembly. The final hexadecimal number (`0x00000300`), if present, is the data value produced by the instruction (and written to the destination register). Similar information is captured in the instruction traces captured by other DFD hardware such as ARM CoreSight [13]. Figure 3(b) shows the captured trace partitioned into two trace blocks.

The state inference technique should find the most recent updates to (or values in) the state elements that are read before being written to in the current trace block. The registers `%o5` and `%i2` as well as the memory location `0x0000000c` shown in Figure 3(b) are examples of such state elements.

The registers to be restored are easily identified by maintaining two sets, *REG_RESTORE* and *REG_WRITE*. As an instruction in the current trace block is decoded, the set REG_WRITE is looked up for the source registers. If a source register is not found, it is added to REG_RESTORE. Then the destination register of the current instruction is added to REG_WRITE. The registers in REG_RESTORE are the registers whose values at the block boundary need to be deter-



```
State: %g0-%g7:0x0, %o0-%o7:0x0, ...
0x40010098 ld [%g2+0x000c], %o5 [0x00000300]
0x4001009c st %o5,[%g1+0x000c]
0x400100a0 add %g2, 0x0010, %g2 [0xfffff020]
0x400100a4 bgu 0x40010078
0x400100a8 add %g1, 0x0010, %g1 [0x400ffe24]
0x400100ac ld [%g2+0x000c], %o5 [0x00000010]
0x400100b0 srl %i2, 0x0004, %o5 [0x00000001]
0x400100b4 sll %o5, 0x0004, %g1 [0x00000010]
0x400100b8 add %o5, 0x0001, %o5 [0x00000002]
0x400100bc sub %i2, %g1, %i2 [0x00000000]
0x400100c0 sll %o5, 0x0004, %o5 [0x00000020]
0x400100c4 subcc %i2, 0x0003, %g0
0x400100c8 ld [%i2+0x000c], %g4 [0x00000300]
0x400100cc mov %i2, %g3 [0x00000000]
0x400100d0 bgu 0x40010048
0x400100d4 add %o5, %o5, %o5 [0x400ffe24]
0x400100d8 subcc %o5, 0x0040, %g0
0x400100dc be 0x400100f0
0x400100e0 mov 0x0000, %g1 [0x00000000]
0x400100f0 ret
```

```
State: %g0-%g7:0x0, %o0-%o7:0x0, ...
0x40010098 ld [%g2+0x000c], %o5 [0x00000300]
0x4001009c st %o5,[%g1+0x000c]
0x400100a0 add %g2, 0x0010, %g2 [0xfffff020]
0x400100a4 bgu 0x40010078
0x400100a8 add %g1, 0x0010, %g1 [0x400ffe24]
0x400100ac ld [%g2+0x000c], %o5 [0x00000010]
0x400100b0 srl %i2, 0x0004, %o5 [0x00000001]
0x400100b4 sll %o5, 0x0004, %g1 [0x00000010]
0x400100b8 add %o5, 0x0001, %o5 [0x00000002]
0x400100bc sub %i2, %g1, %i2 [0x00000000]
```

```
State: %o5:??, %i2:??, [0x0000000c]:??
0x400100c0 sll %o5, 0x0004, %o5 [0x00000020]
0x400100c4 subcc %i2, 0x0003, %g0
0x400100c8 ld [%i2+0x000c], %g4 [0x00000300]
0x400100cc mov %i2, %g3 [0x00000000]
0x400100d0 bgu 0x40010048
0x400100d4 add %o5, %o5, %o5 [0x00000040]
0x400100d8 subcc %o5, 0x0040, %g0
0x400100dc be 0x400100f0
0x400100e0 mov 0x0000, %g1 [0x00000000]
0x400100f0 ret
```

```
.global rstr
.extern test
.text
rstr:   sethi %hi(0x000000300), %o1
        or %o1, %lo(0x00000300), %o1
        sethi %hi(0x00000000c), %o2
        or %o2, %lo(0x0000000c), %o2
        st %o1, [%o2]
        sethi %hi(0x000000002), %o5
        or %o5, %lo(0x00000002), %o5
        sethi %hi(0x000000000), %i2
        or %i2, %lo(0x00000000), %i2
        wrpsr 0xff400120
        ba test
        nop
```

```
.global test
.text
test:   sll %o5, 0x0004, %o5
        subcc %i2, 0x0003, %g0
        ld [%i2+0x000c], %g4
        mov %i2, %g3
        bgu 0x40010048
        add %o5, %o5, %o5
        subcc %o5, 0x0040, %g0
        be 0x400100f0
        mov 0x0000, %g1
        nop
        nop
        nop
        ret
```

Fig. 3. (a) Sample execution trace captured by LEON3 SoC, (b) Trace blocks and state elements to restore for Block 2, (c) Minion (state restoration and replicated instructions) corresponding to Block 2

mined. We assign a Physical Register Identifier (PRI) to each register to uniquely identify the register to restore. This is needed in ISAs such as SPARCv8 where several architectural register names such as `%o2` of one register window and `%i2` of its adjacent window are aliases to the same physical register. Assigning PRI requires us to keep track of the Current Window Pointer (CWP) and the Window Invalid Mask (WIM) as the execution trace is decoded. This is easy because these fields are modified only by `save` and `restore` instructions. The identification of the memory locations to restore would require the knowledge of the values in the registers involved in the address computation. Therefore, this is done only after the values of the registers in REG_RESTORE is inferred using state inference techniques discussed next.

This work focuses on restoring only the architectural state (state of registers and memory) of the system. Therefore, only the functional behavior of the system such as the instructions and their sequence, and the results (values) produced by the instructions, is reproduced by the Minion. This makes our tool applicable to detect bugs that influence the final architectural state of the system and can complement other bug localization techniques [7, 11]. Faithful reproduction of timing behavior would also require the micro-architectural state (state of cache lines accessed and the memory-management unit mappings). We will cover timing behavior replication in a future work.

We propose three state inference techniques to determine the value in the state elements (registers and memory locations) thus identified.

A. Brute-force Method

The simplest technique to infer the values of state elements is to scan the execution trace backwards starting from the block boundary. This technique is called the *Brute-force method*. This technique is time consuming because of the backward scan that goes instruction by instruction.

After the values in the set REG_RESTORE has been identified using backward scans starting from the block boundary, we do another forward pass to identify the memory locations to restore. For each load and store instruction, a backward scan from it is initiated to identify the most recent update to the source registers involved. This is required to compute the address of the memory locations involved. Such backward scans will go at most till the start of the current block. A similar procedure is followed using the sets *MEM_RESTORE* and *MEM_WRITE* to identify memory location to restore. For the memory locations in MEM_RESTORE, a Brute-force strategy is adopted to determine the values to be restored. If a store instruction writes a double word into the memory (using `std` instead of `st`), then it would result in two bus transactions. In this case, the value associated with subsequent transactions have to be accounted against the adjacent memory location instead of the one available in the trace. This allows correct inference of the value in the adjacent memory location.

B. Snapshots-saving Method

The second technique, called *Snapshots-saving method*, aims to reduce the time taken for such backward scans by maintaining an incremental snapshot of the updates to the state elements in the current trace block. A snapshot includes the value at the end of the trace block for each state element that was updated. For example, a snapshot captured at the end of Block 1 (in Figure 3(b)) will have an entry <`%o5`,`0x00000002`> in its snapshot corresponding to the most recent value of `%o5`. Here, we only store the most recent value of `%o5`, although it was updated several times in the trace block. Also, the snapshot does not contain any entry for state elements that are not updated in the trace block. Therefore, in order to find the most recent value, this method goes from one snapshot to the previous snapshot, instead of going instruction by instruction. The rest of the procedure, where the state elements to restore are identified in the forward pass, remains the same as in the case of the Brute-force method.

C. Inverted Map Method

We further improve upon the Snapshots-saving method by entirely eliminating the need for a backward scan. The elimination of backward scans offers two benefits: (i) the time taken to search for the most recent value reduces, and (ii) there is no need to store multiple snapshots. This is achieved by maintaining a map of all state elements that were updated since the start of the execution trace and their recent values. This cumulative map of updated state elements and their most recent values is called the *Inverted Map*. The value stored against each state element is overwritten as and when the state element is updated by a more recent instruction. Therefore, the history of updates is lost, unlike in Snapshots-saving method where the history of updates is captured across several snapshots if the state element was updated in several trace blocks.

The Inverted Map is looked up to find the most recent value of any state element that needs to be restored. Such immediate look-ups take constant time if the Inverted Map is implemented using a Hash-table. The size of the Inverted Map can keep increasing as different state elements are updated over the course of the execution. However, the number of entries in the Inverted Map is bounded by the number of state elements (all registers and memory locations).

When a load or store instruction is encountered, the values in the source registers at that point is readily available in the Inverted Map. Therefore, the address computation can be performed immediately, thereby avoiding any backward scans. The rest of procedure to infer the memory locations to be restored and their values remains the same as above.

V. State Restoration

A. Register State Restoration

Once the registers to restore have been identified and their corresponding values are determined using one of the state inference techniques, we emit a sequence of `sethi` and `or` instructions per register to be restored to set the register content to the determined 32-bit value. For example, consider the instructions at line numbers 9 and 10 of the `restore.s` shown in Figure 3(c). These two instructions set the register `%o5` to `0x00000002`. Similarly, the instructions at line numbers 11 and 12 set the register `%i2` to `0x00000000`.

B. Memory State Restoration

The memory state restoration is a three step process. The address of the memory location to restore is first written to a register. The value to store into that memory location is written to another register next. Finally, a store (`st`) instruction is emitted to write the value into the corresponding memory location. Let us consider the example shown in Figure 3(c). The line numbers 4 and 5 in `restore.s` writes the value to store `0x00000300` into the register `%o1`. The line numbers 6 and 7 writes the address of the memory location `0x0000000c` to the register `%o2`. The store instruction (`st %o1, [%o2]`) on line number 8 actually writes the value determined by the state inference techniques to the memory location whose state needs to be restored.

C. Processor-specific Register Restoration

In addition to emitting instructions that restore the state of General Purpose Registers (GPRs) on execution, we also have to insert an instruction that restores the state of the Processor (or Machine) Specific Register (PSR). The PSR in SPARCv8 ISA contains condition codes, the supervisor bit, CWP, WIM and other machine-specific details. Therefore, the starting state of the PSR dictates the behavior of several instructions such as branch, call, and privileged instructions.

Many instructions such as compare instructions, `save` and `restore` update the PSR implicitly. These updates to the PSR are not captured in the execution trace explicitly. Therefore the techniques to determine the value of state elements discussed earlier are not suitable to infer the value of the PSR.

Several bits in the PSR contain static information such as the version. These are known apriori and can be readily used in the SRI. However, PSR also contains bits that are influenced by the dynamic behavior of the instructions such as condition codes and register window pointers (WIM and CWP). Since the CWP and the WIM bits are kept track by us to generate the PRI, their values are readily available. A similar approach to keep track of the condition codes is tedious and error prone because SPARCv8 has many instructions that modify these bits.

We adopt a different strategy of inspecting the branching behavior in the current trace block to determine the starting state of the condition codes. If a branch is taken, then from the opcode we can infer the flags that influence the branch outcomes. Let us consider the `bgu` instruction (fifth instruction) of Block 2 (shown in Figure 3(b)). We know this branch was not taken because the PC of the second instruction after `bgu` (or instruction after the delay slot) is part of the straight-line execution. This indicates that the carry flag or the zero flag is set to 1 [14]. We repeat such inference for all the branch instructions in the current trace block. For example, consider the behavior of the `be` instruction (eighth instruction of Block 2). Since this branch is taken, we infer the value of zero flag to be 1. The final value of the condition codes is the intersection of the values inferred. In a scenario where conflicting values inferred, the values of the condition codes that were inferred from the branch instruction closer to the start of trace block takes precedence. If there are no branch instructions in the current block that are influenced by some flags, then the starting state of those flags do not matter and so the condition code bits are set to 0. Similarly, the value of the supervisor bit is inferred to be 1 if there is a privileged instruction in the current trace block that has succeeded (without generating any traps).

Once the values of these bits are inferred, a `wrpsr` instruction is written to `restore.s` (as seen in line number 13 of `restore.s` in Figure 3(c)).

D. Linking and Compilation

Since the execution of the memory state restoration instructions itself uses a few registers (`%o1` and `%o2` in our example), the instructions to restore memory state (line numbers 4 – 8 in `restore.s` of Figure 3(c)) are placed before the instructions to restore the register state (line numbers 9 – 12). The instruction to restore the `%psr` comes next (line number 13). Finally, the control is transferred to the replicated instructions using a

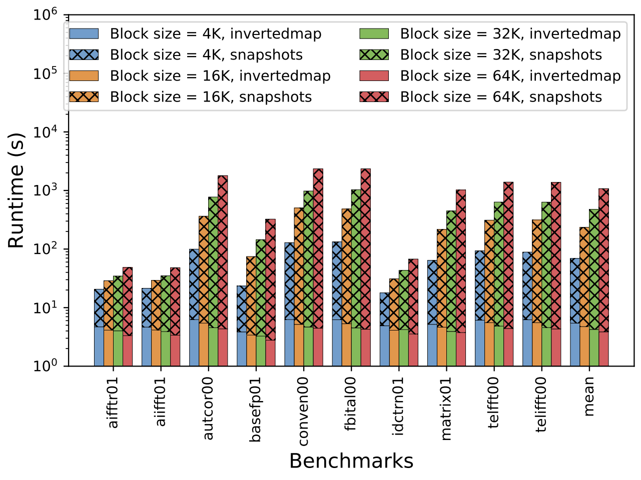


Fig. 4. Minion Generation Time

Branch Always (BA) instruction (`ba test` in line 14). A `nop` is added to the delay slot of the `ba` (line 15).

We replicate the PC values of the complex test case by using a linker script to guide the placement of the text section (of the replicated instructions) of the Minion. The object file generated from `restore.s` is placed at an address range that is not touched by the replicated instructions. The entry point of the Minion is set to the symbol `rstr`.

VI. Experiments

We used a LEON3 SoC with one SPARCv8 core (with 8 register windows), AMBA Advanced High-performance Bus (AHB), DDR controller and a Debug Support Unit (DSU). We enabled a $4KB$ Instruction Trace Buffer and a $4KB$ AHB Trace Buffer to collect LEON3 instruction trace as well as AHB bus traces.

We experimented with 10 benchmarks from the EEMBC AutoBench and TeleBench suites. These benchmarks were compiled using the Bare-C Cross Compiler for LEON3 (BCC v2.0.7), and could run without an Operating System (as bare-metal applications). We report the results of only the benchmarks whose execution time was more than $10ms$ because they are representative of long running test cases. The instruction and bus traces generated during the $10ms$ execution were captured. The resulting execution trace had $369K$ messages (occupying approx. $5.6MB$) on average (with a minimum and maximum of $316K$ and $398K$ messages respectively).

Gru was implemented in Python v3.10 and executed on an Intel Xeon Gold 5218R processor (2.1 GHz base frequency) with 32 GB RAM. The generated Minions were re-executed on the LEON3 SoC and its execution traces were collected. The traces generated from the Minion were then compared against the corresponding sections of the original trace. We did not observe any deviations in instruction behavior or their sequence, thereby validating our implementation.

A. Minion Generation Time

Figure 4 shows the time taken to generate Minions for different block sizes. We see that the time taken to generate Minions using the Inverted Map technique far outperforms the Snapshots-saving technique. The time taken to generate Minions when using Snapshots is up to three orders of magnitude greater (on average) than when using an Inverted Map. This shows that backward scans are quite time consuming and eliminating them is indeed beneficial. The generation time using the Brute-force method is another two orders of magnitude greater than Snapshots-saving method and hence is not shown here.

We also notice that, in the Inverted Map technique, the average time taken to generate Minions gradually reduces from $7.79s$ to $6.02s$ as the block size increases from $4KB$ to $64KB$, for an execution trace containing $369K$ trace messages (approx. $5.6MB$) on average, which is quite scalable. This is because the number of Minions to be generated reduces with increasing block size. The same trend is not seen in the case of Snapshot-saving technique because the amount of work done to generate a Minion increases with increasing block size.

B. Space Overhead

Figures 5 and 6 shows the total space consumed to store the Inverted Map and all the Snapshots respectively. We observe that the average space consumed by the Inverted Map is $5.65KB$ (and a maximum of $14.7KB$) which is nominal. The size of the Inverted Map is independent of the block size. The space consumed by the Snapshots reduces with increasing block size because the number of blocks (and therefore the number of Snapshots to store) decreases. We observe that the average space consumed reduced from $75.88KB$ to $22.39KB$ as the block size was increased from $4KB$ to $64KB$. The space consumed by the Snapshots is higher than that of the Inverted Map because it captures the history of how values in a state element have changed over time across Snapshots if the state element was modified across several trace blocks. Such history is not maintained in the Inverted Map.

C. State Restoration Overhead

Figure 7 shows the number of State restoration instructions as a percentage of the number of replicated instructions in the Minion. We observe that the average (geomean) state restoration instruction overhead decreases from 9.62% to 2.7% as the size of the block increases from $4KB$ to $64KB$. This overhead is not very significant and hence does not adversely affect the execution time of the Minion. However, these additional state restoration instructions added into each Minion enables it to be executed independently of the others. This potentially allows debugging to be performed on each Minion in parallel, thereby saving significant time.

VII. Conclusion and Future Work

We presented a tool called Gru that decomposes the execution of a complex test case into non-overlapping sections and generates Minion executables that replicate the observed behavior in each section independently. Further debugging activities such as bug localization can now be performed in parallel using these smaller executables. The proposed tool is also useful to debug errors in scenarios where the source code

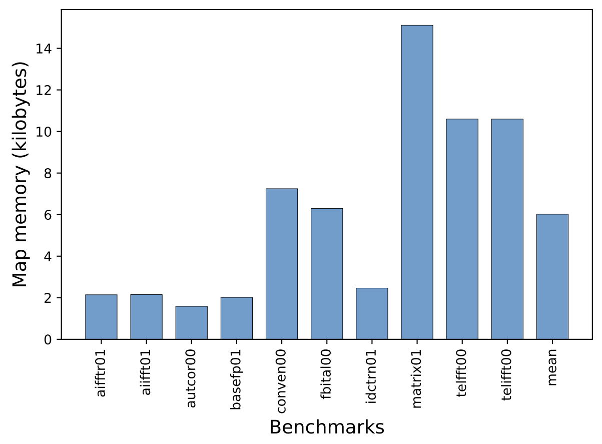


Fig. 5. Space consumed by Inverted Map

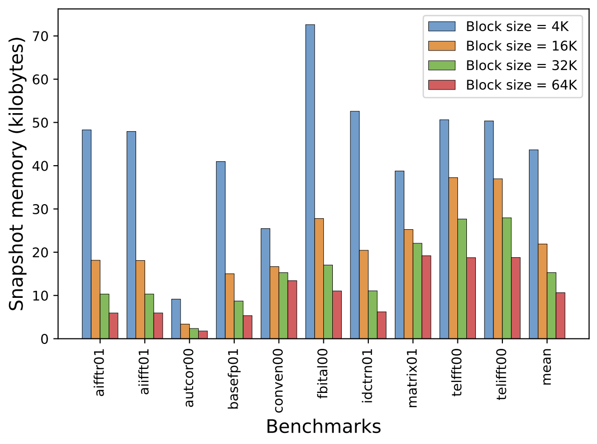


Fig. 6. Space consumed by Snapshots (cumulative)

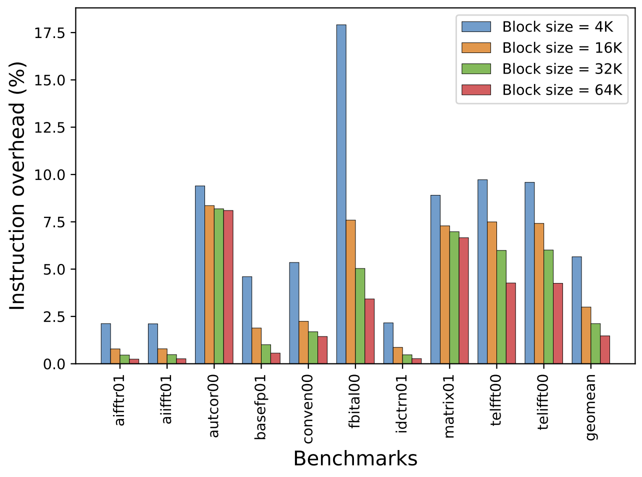


Fig. 7. Overhead of State Restoration Instructions

(or the executable) of the application that triggered the error is not available. We proposed three techniques to address the challenge of inferring the starting state for each Minion by analyzing the execution traces alone. We outlined a strategy to construct each Minion in such a way that the state restoration instructions inserted are executed first to restore the state of the system to the state that prevailed at the start of the corresponding section, and then transfers control to the replicated instructions. This ensured deterministic reproduction of the functional behavior observed in the corresponding section of the original execution of the complex test case. Through extensive validation on EEMBC benchmarks using a LEON3 SoC, we demonstrate the usefulness of the proposed tool. This demonstration establishes a firm base for replicating timing behaviors of complex test cases in future.

References

[1] W. R. Group and S. EDA, "Functional verification study - 2022." [Online]. Available: https://resources.sw.siemens.com/en-US/white-paper-2022-wilson-research-group-functional-verification-study-fpga-functional-verification-trend-report

[2] B. Kumar, J. Adhaduk, K. Basu, M. Fujita, and V. Singh, "A methodology to capture fine-grained internal visibility during multisession silicon debug," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 4, pp. 1002–1015, 2020.

[3] H. F. Ko, A. B. Kinsman, and N. Nicolici, "Design-for-debug architecture for distributed embedded logic analysis," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 19, no. 8, pp. 1380–1393, 2010.

[4] S. Chandran, P. R. Panda, S. R. Sarangi, A. Bhattacharyya, D. Chauhan, and S. Kumar, "Managing trace summaries to minimize stalls during postsilicon validation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 6, pp. 1881–1894, 2017.

[5] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, "A survey on adaptive random testing," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2052–2083, 2019.

[6] A. Banerjee, B. Pal, S. Das, A. Kumar, and P. Dasgupta, "Test generation games from formal specifications," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 827–832.

[7] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "Qed: Quick error detection tests for effective post-silicon validation," in *2010 IEEE International Test Conference*. IEEE, 2010, pp. 1–10.

[8] D. Lin, T. Hong, Y. Li, S. Eswaran, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "Effective post-silicon validation of system-on-chips using quick error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1573–1590, 2014.

[9] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz, "Symbolic quick error detection using symbolic initial state for pre-silicon verification," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 55–60.

[10] S.-B. Park, A. Bracy, H. Wang, and S. Mitra, "Blog: Post-silicon bug localization in processors using bug localization graphs," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 368–373.

[11] O. Friedler, W. Kadry, A. Morgenshtein, A. Nahir, and V. Sokhin, "Effective post-silicon failure localization using dynamic program slicing," in *2014 Design, Automation & Test*

in Europe Conference & Exhibition (DATE). IEEE, 2014, pp. 1–6.

[12] S.-L. Hong and K.-J. Lee, "A run-pause-resume silicon debug technique for multiple clock domain systems," in *2017 International Test Conference in Asia (ITC-Asia)*, 2017, pp. 46–51.

[13] ARM, "Coresight components technical reference manual." [Online]. Available: https://developer.arm.com/documentation/ddi0314/latest/

[14] S. I. Inc., "The sparc architecture manual version 8." [Online]. Available: https://gaisler.com/doc/sparcv8.pdf