2024
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SAN JOSE, CA, USA
MARCH 4-7, 2024

Crafting a Million Instructions/Sec RISCV-DV

HPC Techniques to Boost UVM Testbench Performance by Over 100x

Puneet Goel, Ritu Goel, Jyoti Dahiya

The Curious Case of RISC-V Verification

- High-end Processor Architecture involves intricate maneuvers like Instruction Pipelines, Re-ordering, and Hyperthreading
- Verification of such cores requires a huge stimulus ranging over 10^{15} random instructions*
- RISCV-DV† (coded in SystemVerilog) generates only about **10,000 instr/sec**
  - At this rate, it takes over **Three Thousand Machine Years** just to generate the stimulus

*https://semiengineering.com/what-makes-risc-v-verification-unique/
†https://github.com/chipsalliance/riscv-dv

The Curious Case of RISC-V Verification

- High-end Processor Architecture involves intricate maneuvers like Instruction Pipelines, Re-ordering, and Hyperthreading
- Verification of such cores requires a huge stimulus ranging over $\mathbf{10^{15}}$ random instructions*
- RISCV-DV† (coded in SystemVerilog) generates only about **10,000 instr/sec**
  - At this rate, it takes over **Three Thousand Machine Years** just to generate the stimulus

*https://semiengineering.com/what-makes-risc-v-verification-unique/
†https://github.com/chipsalliance/riscv-dv

In This Talk …

Section 1	**Why is my Testbench so Slow?**	
Section 2	**HPC Testbenching with eUVM**	
Section 3	**RISCV-DV Testbench Optimizations**	
	Constraint Reduction and Optimization	2.5x
	Optimizing Memory Allocation and Reuse	1.5x
	Dierct Binary Generation (Skipping Assembler/Linker)	> 2x
	Native Data Types and Algorithmic Optimizations	2x
Section 4	**The Road to Epiphany – A Parallelized RISCV-DV**	
	Parallelizing RISCV-DV (32 threads)	14x

In this Section …

Why is my Testbench so Slow?

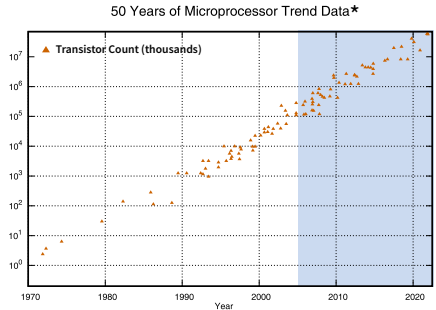
HPC Testbenching with eUVM

RISCV-DV Testbench Optimizations

The Road to Epiphany – A Parallelized RISCV-DV

The Free Lunch is Over

- Over the last 50 years, chip complexity has grown exponentially, owing to the Moore's Law
- Until 2005, thanks to Dennard's Scaling, processor performance also grew at the same rate
- In 2005, Herb Sutter wrote a seminal paper titled "The Free Lunch is Over"
- Modern processors focus on HPC techniques, including …
  - Concurrency – Multicore Parallelism
  - Programmable HW – Hybrid CPU/FPGAs

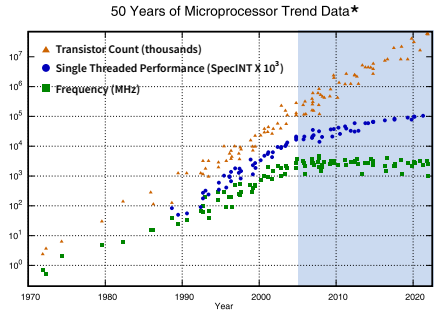


The Elephant in the Room

- Since its standardization in year 2005, SV has not added any HPC construct to HVL

The Free Lunch is Over

- Over the last 50 years, chip complexity has grown exponentially, owing to the Moore's Law
- Until 2005, thanks to Dennard's Scaling, processor performance also grew at the same rate
- In 2005, Herb Sutter wrote a seminal paper titled "The Free Lunch is Over"
- Modern processors focus on HPC techniques, including …
  - Concurrency – Multicore Parallelism
  - Programmable HW – Hybrid CPU/FPGAs

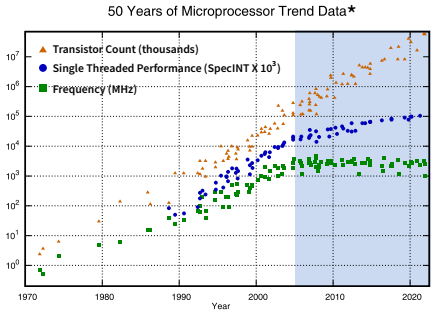


The Elephant in the Room

- Since its standardization in year 2005, SV has not added any HPC construct to HVL

The Free Lunch is Over

- Over the last 50 years, chip complexity has grown exponentially, owing to the Moore's Law
- Until 2005, thanks to Dennard's Scaling, processor performance also grew at the same rate
- In 2005, Herb Sutter wrote a seminal paper titled "The Free Lunch is Over"
- Modern processors focus on HPC techniques, including …
  - Concurrency – Multicore Parallelism
  - Programmable HW – Hybrid CPU/FPGAs



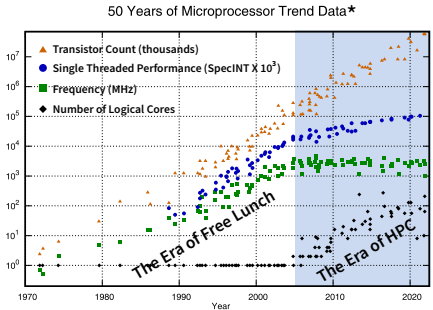
★ Data Sourced From: https://github.com/karlrupp/microprocessor-trend-data

The Elephant in the Room

- Since its standardization in year 2005, SV has not added any HPC construct to HVL

The Free Lunch is Over

- Over the last 50 years, chip complexity has grown exponentially, owing to the Moore's Law
- Until 2005, thanks to Dennard's Scaling, processor performance also grew at the same rate
- In 2005, Herb Sutter wrote a seminal paper titled "The Free Lunch is Over"
- Modern processors focus on HPC techniques, including …
  - Concurrency – Multicore Parallelism
  - Programmable HW – Hybrid CPU/FPGAs



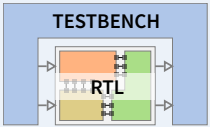
The Elephant in the Room

- Since its standardization in year 2005, SV has not added any HPC construct to HVL

Testbench is the New Bottleneck

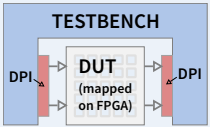
Multicore Simulation Perspective

- Modern simulators enable multicore parallelism for RTL simulation
- Behavioral character makes tool-level parallelism impossible for TB
  - SV lacks multicore semantics for the parallelization of TB
- SV testbench actually executes sequentially with respect to the RTL
  - As per Amdahl's law, the testbench becomes a bottleneck



Hybrid FPGA/CPUs: Co-Emulation Perspective

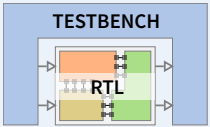
- RTL is synthesizable and can be mapped on FPGAs
- Behavioral nature of TB makes it impossible to map the TB on FPGA
  - DPI layer adds an additional drag on the SV co-simulation interface



Testbench is the New Bottleneck

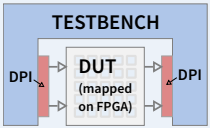
Multicore Simulation Perspective

- Modern simulators enable multicore parallelism for RTL simulation
- Behavioral character makes tool-level parallelism impossible for TB
  - SV lacks multicore semantics for the parallelization of TB
- SV testbench actually executes sequentially with respect to the RTL
  - As per Amdahl's law, the testbench becomes a bottleneck



Hybrid FPGA/CPUs: Co-Emulation Perspective

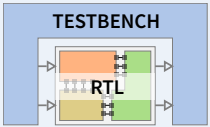
- RTL is synthesizable and can be mapped on FPGAs
- Behavioral nature of TB makes it impossible to map the TB on FPGA
  - DPI layer adds an additional drag on the SV co-simulation interface



Testbench is the New Bottleneck

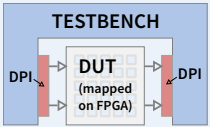
Multicore Simulation Perspective

- Modern simulators enable multicore parallelism for RTL simulation
- Behavioral character makes tool-level parallelism impossible for TB
  - SV lacks multicore semantics for the parallelization of TB
- SV testbench actually executes sequentially with respect to the RTL
  - As per Amdahl's law, the testbench becomes a bottleneck



Hybrid FPGA/CPUs: Co-Emulation Perspective

- RTL is synthesizable and can be mapped on FPGAs
- Behavioral nature of TB makes it impossible to map the TB on FPGA
  - DPI layer adds an additional drag on the SV co-simulation interface



Another Testbench Performance Gotcha

SV Lacks Native Data Types

- HVL data types (`byte`, `int` etc) have an implicit value change event with every arithmetic variable and expression

Native Data Processing

- Arithmetic algorithms coded in systems programming languages run an order of magnitude faster compared to SystemVerilog

fib.sv

```
module none;
  function automatic
    longint fib(longint n);
    if (n <= 1) return n;
    else
      return fib(n-1) + fib(n-2);
  endfunction
  initial
    $display(fib(42));
endmodule
```

fib.d

```
long fib(long n) {
  if (n <= 1) return n;
  else
    return fib(n-1) + fib(n-2);
}
void main() {
  import std.stdio;
  writeln(fib(42));
}
```

In this Section …

Why is my Testbench so Slow?

HPC Testbenching with eUVM

RISCV-DV Testbench Optimizations

The Road to Epiphany – A Parallelized RISCV-DV

An Introduction to Embedded UVM (eUVM)

eUVM is an HVL build on top of Dlang (an evolution of C++)

- Native Efficiency
- Multicore Powered
- 360° Portable Stimulus
- Modern Productivity
- Clean Pointer-Less Syntax
- HW/SW Coverification

eUVM Features

Dlang	ABI Compatibility with C/C++
	Object-Oriented Programming Paradigm
	Associative/Dynamic Arrays
	Automatic Garbage Collector
	Multicore Parallel Programming
	Executes on Embedded Android/Linux/Windows
eUVM	Multicore-Enabled Discrete Event Simulator
	Parallelized Constraint Solvers
	Multicore-Enabled Functional Coverage
	Multicore-Enabled UVM Implementation
	VPI/DPI/FLI/VHPI/Verilator Interface
	Co-Emulation with Altera/Xilinx FPGAs

An Introduction to Embedded UVM (eUVM)

eUVM is an HVL build on top of Dlang (an evolution of C++)

- Native Efficiency
- Multicore Powered
- 360° Portable Stimulus
- Modern Productivity
- Clean Pointer-Less Syntax
- HW/SW Coverification

eUVM Features

Dlang	ABI Compatibility with C/C++
	Object-Oriented Programming Paradigm
	Associative/Dynamic Arrays
	Automatic Garbage Collector
	Multicore Parallel Programming
	Executes on Embedded Android/Linux/Windows
eUVM	Multicore-Enabled Discrete Event Simulator
	Parallelized Constraint Solvers
	Multicore-Enabled Functional Coverage
	Multicore-Enabled UVM Implementation
	VPI/DPI/FLI/VHPI/Verilator Interface
	Co-Emulation with Altera/Xilinx FPGAs

An Introduction to Embedded UVM (eUVM)

eUVM is an HVL build on top of Dlang (an evolution of C++)

- Native Efficiency
- Multicore Powered
- 360° Portable Stimulus
- Modern Productivity
- Clean Pointer-Less Syntax
- HW/SW Coverification

eUVM Features

Dlang	ABI Compatibility with C/C++
	Object-Oriented Programming Paradigm
	Associative/Dynamic Arrays
	Automatic Garbage Collector
	Multicore Parallel Programming
	Executes on Embedded Android/Linux/Windows
eUVM	Multicore-Enabled Discrete Event Simulator
	Parallelized Constraint Solvers
	Multicore-Enabled Functional Coverage
	Multicore-Enabled UVM Implementation
	VPI/DPI/FLI/VHPI/Verilator Interface
	Co-Emulation with Altera/Xilinx FPGAs

An Introduction to Embedded UVM (eUVM)

eUVM is an HVL build on top of Dlang (an evolution of C++)

- Native Efficiency
- Multicore Powered
- 360° Portable Stimulus
- Modern Productivity
- Clean Pointer-Less Syntax
- HW/SW Coverification

eUVM Features

Dlang	ABI Compatibility with C/C++
	Object-Oriented Programming Paradigm
	Associative/Dynamic Arrays
	Automatic Garbage Collector
	Multicore Parallel Programming
	Executes on Embedded Android/Linux/Windows
eUVM	Multicore-Enabled Discrete Event Simulator
	Parallelized Constraint Solvers
	Multicore-Enabled Functional Coverage
	Multicore-Enabled UVM Implementation
	VPI/DPI/FLI/VHPI/Verilator Interface
	Co-Emulation with Altera/Xilinx FPGAs

An Introduction to Embedded UVM (eUVM)

eUVM is an HVL build on top of Dlang (an evolution of C++)

- Native Efficiency
- Multicore Powered
- 360° Portable Stimulus
- Modern Productivity
- Clean Pointer-Less Syntax
- HW/SW Coverification

uvm.d

```
import esdl;
import uvm;
class bus_trans: uvm_sequence_item {
  mixin uvm_object_utils;
  @rand ubvec!8 data;
  @rand uint[] payload;

  this(string name="") {
    super(name);
  }

  constraint!q{
    payload.length >= 8;
    payload.length < 256;
    foreach (i, elem; payload) {
      if (i > 0) payload[i] > payload[i-1];
    }
    unique [payload];
  } payload_cst;
}
```

An Introduction to Embedded UVM (eUVM)

eUVM is an HVL build on top of Dlang (an evolution of C++)

- Native Efficiency
- Multicore Powered
- 360° Portable Stimulus
- Modern Productivity
- Clean Pointer-Less Syntax
- HW/SW Coverification

coverification

```
override void run_phase(uvm_phase phase) {
  super.run_phase(phase);
  load_device_drivers();
  get_and_drive(phase);
}
override void connect_phase(uvm_phase phase) {
  fd = open("/dev/mem", O_RDWR | O_SYNC);
  if (fd < 0) assert(false, "Failed to open /dev/mem");
  mem = mmap(null, HPS_TO_FPGA_LW_SPAN, PROT_READ |
      PROT_WRITE, MAP_SHARED, fd, HPS_TO_FPGA_LW_BASE);
  if (mem == MAP_FAILED) {
    close(fd);
    assert(false, "Can't map memory");
  }
}
override void final_phase(uvm_phase phase) {
  super.final_phase(phase);
  munmap(mem, HPS_TO_FPGA_LW_SPAN);
  close(fd);
}
```

Perfomance Comparison of UVM Implementations

	Platform	UVM	PyUVM	SC-UVM	eUVM
	Language	SV	Python	C++	Dlang
HPC	Multicore-Enabled UVM	✗	✗	✗*	✓†
HPC	Native Data Types	✗	✗	✓	✓
HPC	ABI Compatibility with C/C++	✗	✗	✓	✓
Meta‡	User Defined Attributes	✗	◐	✗	✓
Meta‡	Code Introspection	✓	◐	◐	✓
Meta‡	Compile-Time Function Eval	✗	✗	◐	✓
Meta‡	Generative Programming	✗	◐	◐	✓

*While C++ supports parallelism, both SC-UVM and SystemC are single-threaded

†eUVM is yet the only Multicore-enabled Implementation of UVM

‡Advanced Metaprogramming features in Dlang enable compile-time constraint parsing, resulting in Ultra-Fast Constraint Solvers

Python Efficacy

- Being an interpreted language, Python is inherently slow and has been benchmarked to be 57x slower than C

Legend

✓ Full Support

◐ Partial Support

✗ Not Supported

Comparison Between Constraint Solvers

		SV	PyVSC	CRAVE	eUVM
	Language	SV	Python	C++	Dlang
Agility	BDD Solvers	✔	✘	✔	✔
	SAT Solvers	✔	✔	✔	✔
	Conditional Constraints	✔	✔	◕	✔
	Array/Loop Constraints	✔	✔	◕*	✔
	SV-Like Constraint Syntax	✔	✘	✘	✔
Speed	Native Rand Variables	✘	✘	✘	✔
	Compile-time Processing	◕	✘	✘	✔
	Multicore Solvers	✘	✘	✘	✔
	RISCV-DV Port	✔	◕	✘	✔

*CRAVE conditional and array/loop constraints are macro based

Solver Efficacy

- PyGen, the Python port of RISCV-DV, currently generates less than 100 instr/sec
- CRAVE (C++ Library) lags SV by over 10x

Legend

✔ Full Support
◕ Partial Support
✘ Not Supported

Testbench Parallelism in eUVM

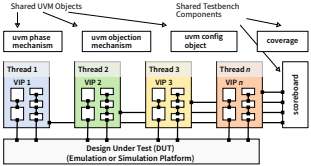


Figure: VIP-Level Parallelism in eUVM

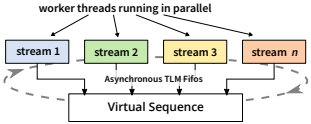


Figure: Sequence Parallelism in eUVM

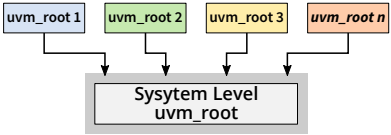


Figure: Multi-root Configuration in eUVM

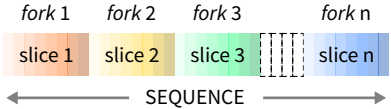


Figure: Parallelized Fork-Join

Tracing Testbench Performance in eUVM

- eUVM adds `uvm_trace` method to UVM
- It works just like `uvm_info` method, but it additionally prints the Wall-clock Time with the log message

uvm_trace usage

```
uvm_trace("GEN INSTR", "START", UVM_NONE);
// Code block to track Performance of
foreach (ref instr; instr_list) {
  randomize_instr(instr, is_debug_program);
}
uvm_trace("GEN INSTR", "END", UVM_NONE);
```

uvm_trace log

```
UVM_TRACE [6.946821] riscv_instr_stream.d(42) @0: uvm_dock.root.uvm_test_top [GEN INSTR] START
UVM_TRACE [13.526620] riscv_instr_stream.d(47) @0: uvm_dock.root.uvm_test_top [GEN INSTR] END
```

In this Section …

Why is my Testbench so Slow?

HPC Testbenching with eUVM

RISCV-DV Testbench Optimizations

The Road to Epiphany – A Parallelized RISCV-DV

Prefer Procedural Randomization Over Constraints

- Simple constraints can be replaced with a procedural randomization
- Dlang's algorithms library comes in handy with more complex constraints

instr-pick.sv

```
function riscv_instr_name_t pick_instr();
   riscv_instr_name_t instr;
   std::randomize(instr) with {
      instr inside {allowed_instrs};
      ! instr inside {disallowed_instrs};
   };
   return instr;
endfunction
```

instr-pick.d

```
riscv_instr_name_t pick_instr() {
  static riscv_instr_name_t[] instrs;
  instrs.length = 0;
  instrs ~=
    setDifference(allowed_instrs.sort,
                  disallowed_instrs.sort);
  size_t idx = urandom(0, instrs);
  return instrs[idx];
}
```

Compile-Time Constraint Filtering

- RISCV-DV implements randomization of about 600 instructions
  - Constraints are defined in common templatized base class
  - Constraint that applies to a specific instruction is implemented using a constraint guard

compr_cst.sv

```
constraint no_hint_illegal_instr_c {
  if (INSTR_NAME == C_JR) {
    rs1 != ZERO;
  }
}
```

Using Compile-Time Static If

- eUVM enables compile-time filtering of constraints
- Constraint gets defined only for the specific RISC-V instruction it applies to

compr_cst.d

```
static if (INSTR_NAME == C_JR) {
  constraint! q{
    rs1 != ZERO;
  } no_hint_illegal_instr_c;
}
```

Avoid Memory Allocation

Why is this Important?

- Memory allocation is a significant run-time cost
- Since memory is shared by all threads, memory allocation is not multicore friendly

Reusing Dynamic Arrays and Queues

- Declaring a dynamic array in a loop (or function) leads to repeated memory allocation/GC cycles
- This can be avoided by declaring the array statically scoped
  - Remember to reset the dynamic array/queue before putting it to reuse

instr-pick.d

```
riscv_instr_name_t pick_rand_instr() {
  // snip ....
  riscv_instr_name_t[] inter_set;

  inter_set ~= setDifference(setIntersection
      (instr_set, include_set, allowed_set),
       disallowed_instr[].sort());
  idx = urandom(0, inter_set.length);
  return(inter_set[idx]);
}
```

Avoid Memory Allocation

Why is this Important?

- Memory allocation is a significant run-time cost
- Since memory is shared by all threads, memory allocation is not multicore friendly

Reusing Dynamic Arrays and Queues

- Declaring a dynamic array in a loop (or function) leads to repeated memory allocation/GC cycles
- This can be avoided by declaring the array statically scoped
  - Remember to reset the dynamic array/queue before putting it to reuse

instr-pick.d

```
riscv_instr_name_t pick_rand_instr() {
  // snip ....
  static riscv_instr_name_t[] inter_set;

  inter_set ~= setDifference(setIntersection
      (instr_set, include_set, allowed_set),
       disallowed_instr[].sort());
  idx = urandom(0, inter_set.length);
  return(inter_set[idx]);
}
```

Avoid Memory Allocation

Why is this Important?

- Memory allocation is a significant run-time cost
- Since memory is shared by all threads, memory allocation is not multicore friendly

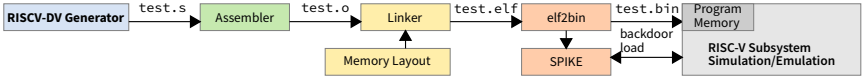
Reusing Dynamic Arrays and Queues

- Declaring a dynamic array in a loop (or function) leads to repeated memory allocation/GC cycles
- This can be avoided by declaring the array statically scoped
  - Remember to reset the dynamic array/queue before putting it to reuse

instr-pick.d

```
riscv_instr_name_t pick_rand_instr() {
  // snip ....
  static riscv_instr_name_t[] inter_set;
  inter_set.length = 0;
  inter_set ~= setDifference(setIntersection
      (instr_set, include_set, allowed_set),
       disallowed_instr[].sort());
  idx = urandom(0, inter_set.length);
  return(inter_set[idx]);
}
```

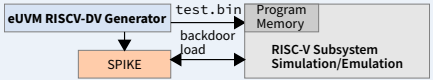
Optimizing RISC-V Functional Verification Flow



- In UVM terminology RISCV-DV plays the role of Sequence Generator (Sequencer)
- RISCV-DV writes out an ASM file that needs to be compiled and linked
- SPIKE (a high level C model) plays the role of reference model

RISCV-DV eUVM Port

- Generates a binary dump directly, and thus a monolith high-performance executable



In this Section …

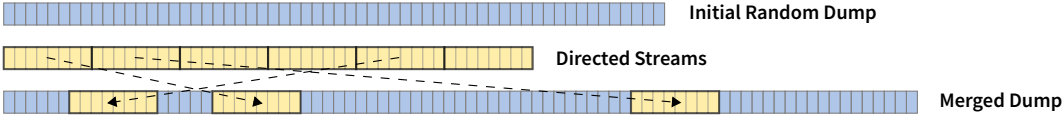
Why is my Testbench so Slow?

HPC Testbenching with eUVM

RISCV-DV Testbench Optimizations

The Road to Epiphany – A Parallelized RISCV-DV

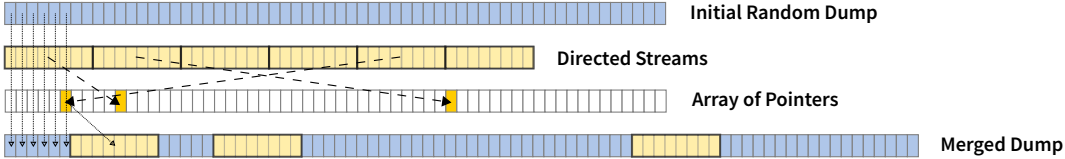
Analyzing RISCV-DV Performance



	Task	Complexity
1	Generate and randomize a huge dump of random instructions	$\mathcal{O}(n)$
2	Generate and randomize a large number of directed streams*	$\mathcal{O}(n)$
3	Insert multiple Directed Streams into the previously generated dump	$\mathcal{O}(n^2)$
4	Fix jump labels/addresses	$\mathcal{O}(n)$
5	Construct ASM string for every instruction	$\mathcal{O}(n)$

*A directed stream is a set of instructions defining a specific program construct (like a for loop)

Algorithmic Optimizations – Taming Non-Linear Complexity



Lazy Merging

- First create an array of null pointers of the size of the Random Dump
- Pick random locations where the Directed Streams need to be inserted
  - Replace the null pointer at that location with a pointer to the Directed Stream
- A Merged Dump is then created in a single iteration over the Pointer Array

Parallelizing the Random Instruction Dump

- Create multiple slices of the Random Instruction Dump – Lines 3-4
- Spawn a fork for each slice – Lines 7-8
- Make every fork stick to a separate thread – Line 11

par_random.d

```
1  Fork[] forks;
2  for (size_t i=0; i!=cfg.par_num_threads; ++i) {
3    size_t start_idx = i * instr_count/cfg.par_num_threads;     // start of the slice
4    size_t end_idx = (i + 1) * instr_count/cfg.par_num_threads; // end of the slice
5    Fork slice_fork = (size_t start, size_t end) {
6      return fork({
7          for (size_t i=start; i!=end; ++i)
8            randomize_instr(instr_list[i], is_debug_program);
9      });
10   } (start_idx, end_idx);
11   slice_fork.set_thread_affinity(i);
12   forks ~= slice_fork;
13 }
```

Parallelizing the Random Instruction Dump

- Create multiple slices of the Random Instruction Dump – Lines 3-4
- Spawn a fork for each slice – Lines 7-8
- Make every fork stick to a separate thread – Line 11

par_random.d

```
1  Fork[] forks;
2  for (size_t i=0; i!=cfg.par_num_threads; ++i) {
3    size_t start_idx = i * instr_count/cfg.par_num_threads;     // start of the slice
4    size_t end_idx = (i + 1) * instr_count/cfg.par_num_threads; // end of the slice
5    Fork slice_fork = (size_t start, size_t end) {
6      return fork({
7          for (size_t i=start; i!=end; ++i)
8            randomize_instr(instr_list[i], is_debug_program);
9      });
10   } (start_idx, end_idx);
11   slice_fork.set_thread_affinity(i);
12   forks ~= slice_fork;
13 }
```

Parallelizing the Random Instruction Dump

- Create multiple slices of the Random Instruction Dump – Lines 3-4
- Spawn a fork for each slice – Lines 7-8
- Make every fork stick to a separate thread – Line 11

par_random.d

```
1  Fork[] forks;
2  for (size_t i=0; i!=cfg.par_num_threads; ++i) {
3    size_t start_idx = i * instr_count/cfg.par_num_threads;     // start of the slice
4    size_t end_idx = (i + 1) * instr_count/cfg.par_num_threads; // end of the slice
5    Fork slice_fork = (size_t start, size_t end) {
6      return fork({
7          for (size_t i=start; i!=end; ++i)
8            randomize_instr(instr_list[i], is_debug_program);
9      });
10   } (start_idx, end_idx);
11   slice_fork.set_thread_affinity(i);
12   forks ~= slice_fork;
13 }
```

Parallelizing the Random Instruction Dump

- Create multiple slices of the Random Instruction Dump – Lines 3-4
- Spawn a fork for each slice – Lines 7-8
- Make every fork stick to a separate thread – Line 11

par_random.d

```
1  Fork[] forks;
2  for (size_t i=0; i!=cfg.par_num_threads; ++i) {
3    size_t start_idx = i * instr_count/cfg.par_num_threads;     // start of the slice
4    size_t end_idx = (i + 1) * instr_count/cfg.par_num_threads; // end of the slice
5    Fork slice_fork = (size_t start, size_t end) {
6      return fork({
7          for (size_t i=start; i!=end; ++i)
8            randomize_instr(instr_list[i], is_debug_program);
9      });
10   } (start_idx, end_idx);
11   slice_fork.set_thread_affinity(i);
12   forks ~= slice_fork;
13 }
```

Parallelizing Directed Streams Generation

- Determine the number of Directed Streams in a given category – Line 4
- Spawn a fork to generate the Directed Streams of the given category – Lines 7-8
- Stick every fork to a separate thread – Line 11

par_directed.d

```
1  Fork[] forks;
2  foreach (stream_name, ratio; directed_instr_stream_ratio) { // directed stream categories
3    uint  stream_idx = 0;
4    uint  insert_cnt = original_instr_cnt * ratio/1000; // number of directed streams
5    Fork dir_fork = (string name, uint ratio, uint idx, uint cnt) {
6      return fork({
7          generate_directed_instr_stream_idx(hart, label, orig_instr_cnt, kernel_mode,
8                                             name, ratio, instr_stream, idx, cnt);
9      });
10   } (stream_name, ratio, stream_idx, insert_cnt);
11   dir_fork.set_thread_affinity(forks.length);
12   stream_idx += insert_cnt;
13   forks ~= dir_fork;
14  }
```

Parallelizing Directed Streams Generation

- Determine the number of Directed Streams in a given category – Line 4
- Spawn a fork to generate the Directed Streams of the given category – Lines 7-8
- Stick every fork to a separate thread – Line 11

par_directed.d

```
Fork[] forks;
foreach (stream_name, ratio; directed_instr_stream_ratio) { // directed stream categories
  uint  stream_idx = 0;
  uint  insert_cnt = original_instr_cnt * ratio/1000; // number of directed streams
  Fork dir_fork = (string name, uint ratio, uint idx, uint cnt) {
    return fork({
        generate_directed_instr_stream_idx(hart, label, orig_instr_cnt, kernel_mode,
                                           name, ratio, instr_stream, idx, cnt);
    });
  } (stream_name, ratio, stream_idx, insert_cnt);
  dir_fork.set_thread_affinity(forks.length);
  stream_idx += insert_cnt;
  forks ~= dir_fork;
 }
```

Parallelizing Directed Streams Generation

- Determine the number of Directed Streams in a given category – Line 4
- Spawn a fork to generate the Directed Streams of the given category – Lines 7-8
- Stick every fork to a separate thread – Line 11

par_directed.d

```
1  Fork[] forks;
2  foreach (stream_name, ratio; directed_instr_stream_ratio) { // directed stream categories
3    uint  stream_idx = 0;
4    uint  insert_cnt = original_instr_cnt * ratio/1000; // number of directed streams
5    Fork dir_fork = (string name, uint ratio, uint idx, uint cnt) {
6      return fork({
7          generate_directed_instr_stream_idx(hart, label, orig_instr_cnt, kernel_mode,
8                                             name, ratio, instr_stream, idx, cnt);
9      });
10   } (stream_name, ratio, stream_idx, insert_cnt);
11   dir_fork.set_thread_affinity(forks.length);
12   stream_idx += insert_cnt;
13   forks ~= dir_fork;
14  }
```

Parallelizing Directed Streams Generation

- Determine the number of Directed Streams in a given category – Line 4
- Spawn a fork to generate the Directed Streams of the given category – Lines 7-8
- Stick every fork to a separate thread – Line 11

par_directed.d

```
1  Fork[] forks;
2  foreach (stream_name, ratio; directed_instr_stream_ratio) { // directed stream categories
3    uint  stream_idx = 0;
4    uint  insert_cnt = original_instr_cnt * ratio/1000; // number of directed streams
5    Fork dir_fork = (string name, uint ratio, uint idx, uint cnt) {
6      return fork({
7          generate_directed_instr_stream_idx(hart, label, orig_instr_cnt, kernel_mode,
8                                             name, ratio, instr_stream, idx, cnt);
9      });
10   } (stream_name, ratio, stream_idx, insert_cnt);
11   dir_fork.set_thread_affinity(forks.length);
12   stream_idx += insert_cnt;
13   forks ~= dir_fork;
14  }
```

Results and Conclusions

Performance Improvements for a 10 million instruction RISCV-DV test
(All timing values in seconds)

Instr Count	Thread Count	Execution Time	Performance	RAM Usage
10,000,000	1	57.86	1.00x	4.9 GB
10,000,000	2	31.22	1.85x	4.9 GB
10,000,000	4	18.03	3.21x	5.0 GB
10,000,000	8	10.35	5.59x	5.0 GB
10,000,000	16	5.53	10.46x	5.0 GB
10,000,000	32	4.23	13.68x	5.0 GB

The Importance of Shared-Memory Parallelization

- Running multiple simulations is not the most optimized way to utilize a multicore server
- If you are running a multicore RTL simulation, a single-threaded testbench becomes a bottleneck

The Memory Wall Perspective

- Modern CPUs (eg Apple M1) integrate a limited on-chip RAM
  - External RAM access is slow and power hungry

Hybrid CPU/FPGAs – Co-Emulation Perspective

- In a co-emulation setup, multiple CPU cores share a single FPGA core
  - The DuT gets mapped to the FPGA
  - A multicore-parallelized testbench is the best suited speedup scenario

The Importance of Shared-Memory Parallelization

- Running multiple simulations is not the most optimized way to utilize a multicore server
- If you are running a multicore RTL simulation, a single-threaded testbench becomes a bottleneck

The Memory Wall Perspective

- Modern CPUs (eg Apple M1) integrate a limited on-chip RAM
  - External RAM access is slow and power hungry

Hybrid CPU/FPGAs – Co-Emulation Perspective

- In a co-emulation setup, multiple CPU cores share a single FPGA core
  - The DuT gets mapped to the FPGA
  - A multicore-parallelized testbench is the best suited speedup scenario

The Importance of Shared-Memory Parallelization

- Running multiple simulations is not the most optimized way to utilize a multicore server
- If you are running a multicore RTL simulation, a single-threaded testbench becomes a bottleneck

The Memory Wall Perspective

- Modern CPUs (eg Apple M1) integrate a limited on-chip RAM
  - External RAM access is slow and power hungry

Hybrid CPU/FPGAs – Co-Emulation Perspective

- In a co-emulation setup, multiple CPU cores share a single FPGA core
  - The DuT gets mapped to the FPGA
  - A multicore-parallelized testbench is the best suited speedup scenario

The Importance of Shared-Memory Parallelization

- Running multiple simulations is not the most optimized way to utilize a multicore server
- If you are running a multicore RTL simulation, a single-threaded testbench becomes a bottleneck

The Memory Wall Perspective

- Modern CPUs (eg Apple M1) integrate a limited on-chip RAM
  - External RAM access is slow and power hungry

Hybrid CPU/FPGAs – Co-Emulation Perspective

- In a co-emulation setup, multiple CPU cores share a single FPGA core
  - The DuT gets mapped to the FPGA
  - A multicore-parallelized testbench is the best suited speedup scenario

Fork Me on Github

EUVM https://github.com/coverify/euvm

RISCV DV https://github.com/coverify/riscv_dv

Questions?