GSIM: Accelerating RTL Simulation for Large-Scale Design

Lu Chen, Dingyi Zhao, Zihao Yu, Ninghui Sun, Yungang Bao

chenlu22z@ict.ac.cn

2025.6.25

Open-sourced at https://github.com/OpenXiangShan/gsim

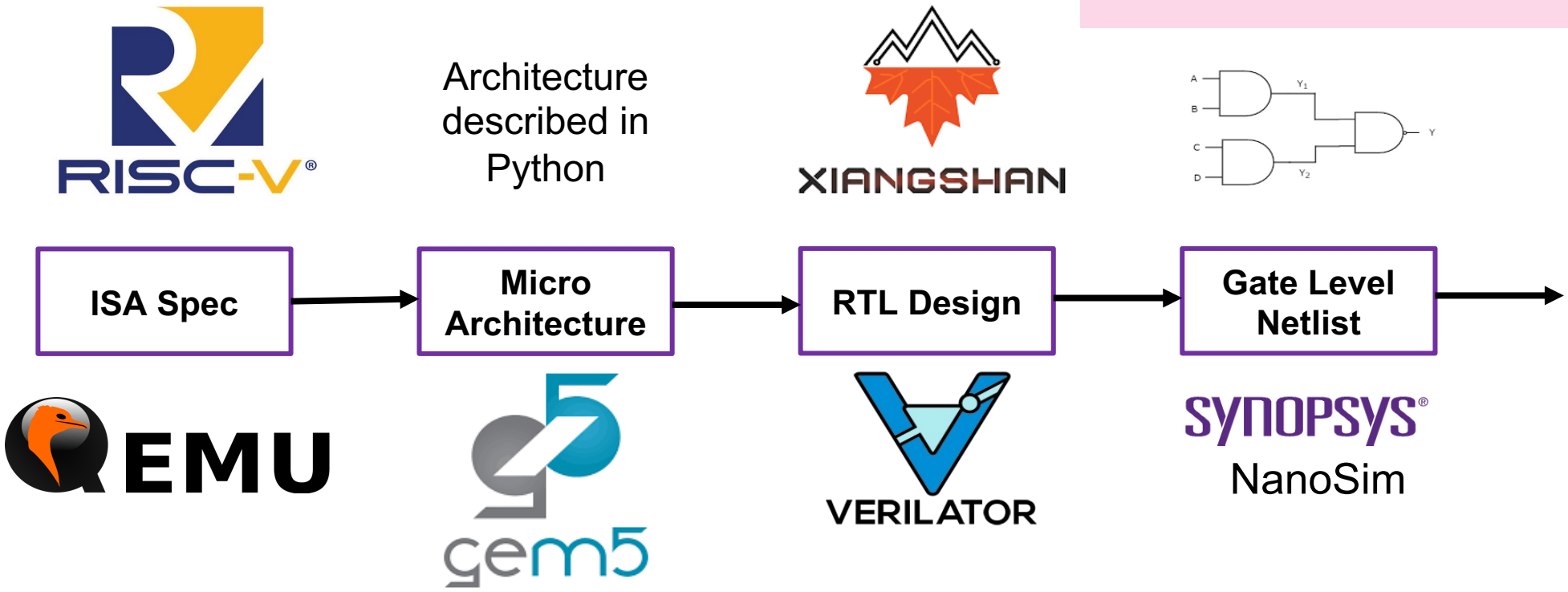
62 THE CHIPS TO SYSTEMS CONFERENCE

University of Chinese Academy of Sciences

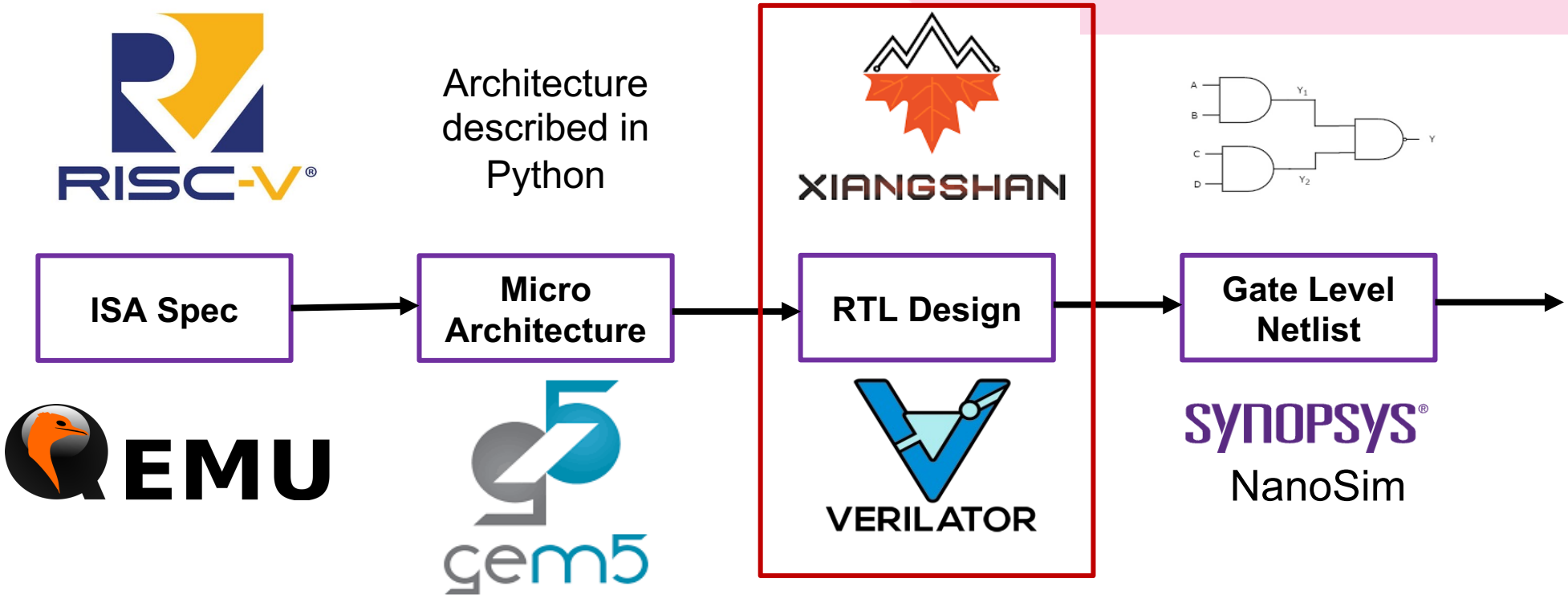
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

SPONSORED BY CEDA IEEE Council on Electronic Design Automation SIGDA

Simulation in ASIC Design Flow



Simulation in ASIC Design Flow



- RTL simulation is crucial in **Design space exploration, Verification, Debugging and Preliminary performance evaluation.**

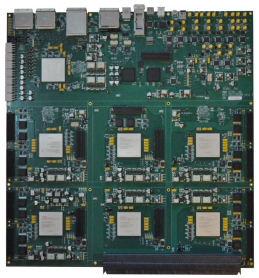
RTL Simulation Techniques

- Software simulation is the most commonly used method
  - 100% signal visibility, low cost and ease of debugging
  - least efficient

	Software Simulation	FPGA	Hardware Emulation
Speed	~1KHz	~100MHz	~1MHz
Cost	<10K$	<50K$	>1M$
Debug Difficulty	Low	High	Low
Design Scale	Large	Mid	Large

VERILATOR

SYNOPSYS®
VCS



Cadence
Palladium

The Slow Speed of Software Simulation

- The software simulation speed decreases dramatically with the increasing scale of modern digital circuits
- The slow speed becomes the bottleneck in verification[1]

Name	Configuration	IR node	IR edge	Speed
stuCore	In-Order Single-Issue	9933	17369	~ 900 KHz
Rocket	In-Order Single-Issue	234807	293164	~ 30 KHz
BOOM-Large	Out-of-Order Triple-Issue	571038	827619	~ 9 KHz
XiangShan[2]	Out-of-Order Six-Issue	6218427	9007005	~ 0.9 KHz

Verilator(single thread) simulation speed of booting Linux on different processors

[1] L. Lavagno, I. Markov, G. Martin, and L. Scheffer, *Electronic Design Automation for IC System Design, Verification, and Testing*. CRC Press, 2017. [Online]. Available: https://books.google.com/books?id= UeobDAAAQBAJ
[2]Y. Xu *et al*., "Towards Developing High Performance RISC-V Processors Using Agile Methodology," *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Chicago, IL, USA, 2022, pp. 1178-1199

Different Simulation Frameworks

- Verilator vs. ESSENT [2]

```
eval_node1():
        update new value

cycle():
        eval_node1()
        eval_node2()
        …
```

Verilator simulation framework

```
eval_node1():
        save old value
        update new value
        if (old != new) active[succ] = true

cycle():
        if (active[1]) eval_node1()
        if (active[2]) eval_node2()
        …
```

ESSENT simulation framework

[2] S.BeamerandD.Donofrio,"Efficientlyexploitinglowactivityfactorsto accelerate rtl simulation," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20. IEEE Press, 2020

Modeling and Analysis

- Simulation overhead $T = \left((E + A_{succ}) * af + A_{exam}\right) * N$

```
eval_node1():
        save old value
        update new value
        if (old != new) active[succ] = true

cycle():
        if (active[1]) eval_node1()
        if (active[2]) eval_node2()
        …
```

Modeling and Analysis

- Simulation overhead $T = \left((E + A_{succ}) * af + A_{exam}\right) * N$
  - E: evaluation overhead

```
eval_node1():
        save old value
        update new value
        if (old != new) active[succ] = true

cycle():
        if (active[1]) eval_node1()
        if (active[2]) eval_node2()
        …
```

Modeling and Analysis

- Simulation overhead $T = \left((E + A_{succ}) * af + A_{exam}\right) * N$
  - E: evaluation overhead
  - A_{succ}: overhead of activating successors

```
eval_node1():
        save old value
        update new value
        if (old != new) active[succ] = true

cycle():
        if (active[1]) eval_node1()
        if (active[2]) eval_node2()
        …
```

Modeling and Analysis

- Simulation overhead $T = \left((E + A_{succ}) * af + A_{exam}\right) * N$
  - E: evaluation overhead
  - A_{succ}: overhead of activating successors
  - af: activity factor

```
eval_node1():
        save old value
        update new value
        if (old != new) active[succ] = true

cycle():
        if (active[1]) eval_node1()
        if (active[2]) eval_node2()
        …
```

Modeling and Analysis

- Simulation overhead $T = \left((E + A_{succ}) * af + A_{exam}\right) * N$
  - E: evaluation overhead
  - A_{succ}: overhead of activating successors
  - af: activity factor
  - A_{exam}: overhead of examining **active** bit

```
eval_node1():
        save old value
        update new value
        if (old != new) active[succ] = true

cycle():
        if (active[1]) eval_node1()
        if (active[2]) eval_node2()
        …
```

Modeling and Analysis

- Simulation overhead $T = \left((E + A_{succ}) * af + A_{exam}\right) * N$
  - E: evaluation overhead
  - A_{succ}: overhead of activating successors
  - af: activity factor
  - A_{exam}: overhead of examining **active** bit
  - N: the number of nodes

```
eval_node1():
        save old value
        update new value
        if (old != new) active[succ] = true

cycle():
        if (active[1]) eval_node1()
        if (active[2]) eval_node2()
        …
```

GSIM: a high performance RTL simulator

- $T = \left((E + A_{succ}) * af + A_{exam}\right) * N$
- Aims to reduce the overhead of the five factors above through three levels of granularity

	optimization	N	E	A_{succ}	A_{exam}	af
supernode level	graph partition			---	---	+++
	examine active bit with SIMD				---	
node level	redundant nodes	---				
	node inline / extraction	tradeoff				
	reset optimization		---			
	expression simplification		---			
	activation optimization			---		
bit level	node splitting	+++	---			---

Supernode-Level Optimization

- Group nodes into supernodes. $A_{exam} \downarrow af \uparrow$
  - Reduce A_{exam}: nodes in a supernode share the same **active** bit
  - Keep af low: group nodes that are likely to be activated simultaneously

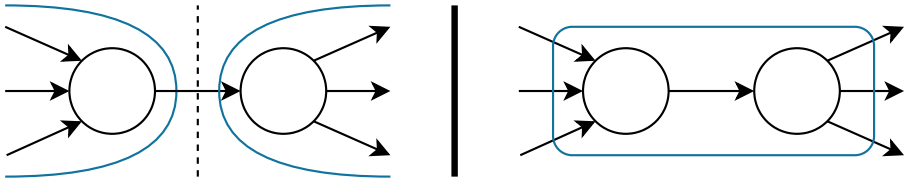
```
eval_node1():
        save old value
        update new value
        if (old != new) active[succ] = true

cycle():
        if (active[1]) eval_node1()
        if (active[2]) eval_node2()
        …
```

```
eval_super1():
        eval_node1()
        eval_node2()
cycle():
        if (active[1]) eval_super1()
        if (active[2]) eval_super2()
        …
```

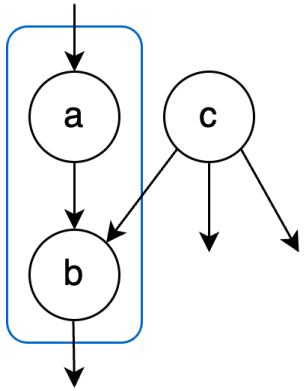
Supernode-Level Optimization

- GSIM employs enhanced Kernighan's algorithm[3]
  - Kernighan's algorithm fails to group nodes connected with few edges

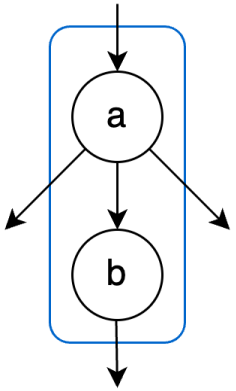


- Step1: group nodes according the following observations:

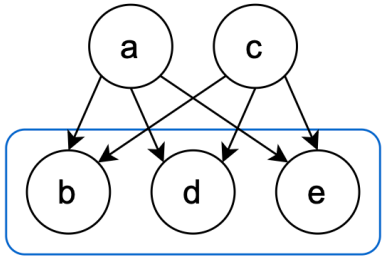
nodes with out-degree 1



nodes with in-degree 1



siblings with the same predecessors

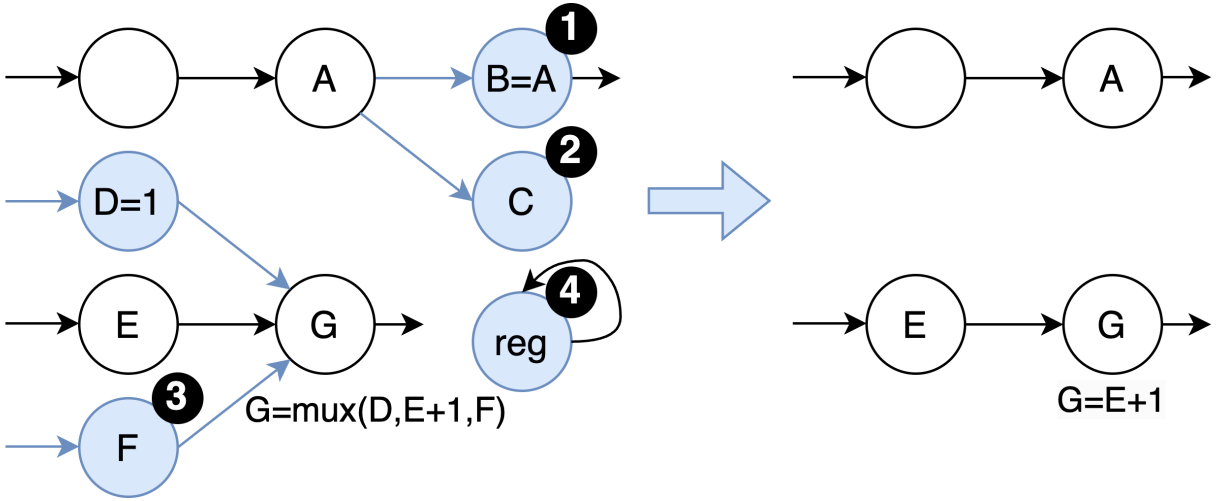


- Step2: adopts the original Kernighan's algorithm

[3] B. W. Kernighan, “Optimal sequential partitions of graphs,” Journal of the ACM (JACM), vol. 18, no. 1, pp. 34–40, 1971.

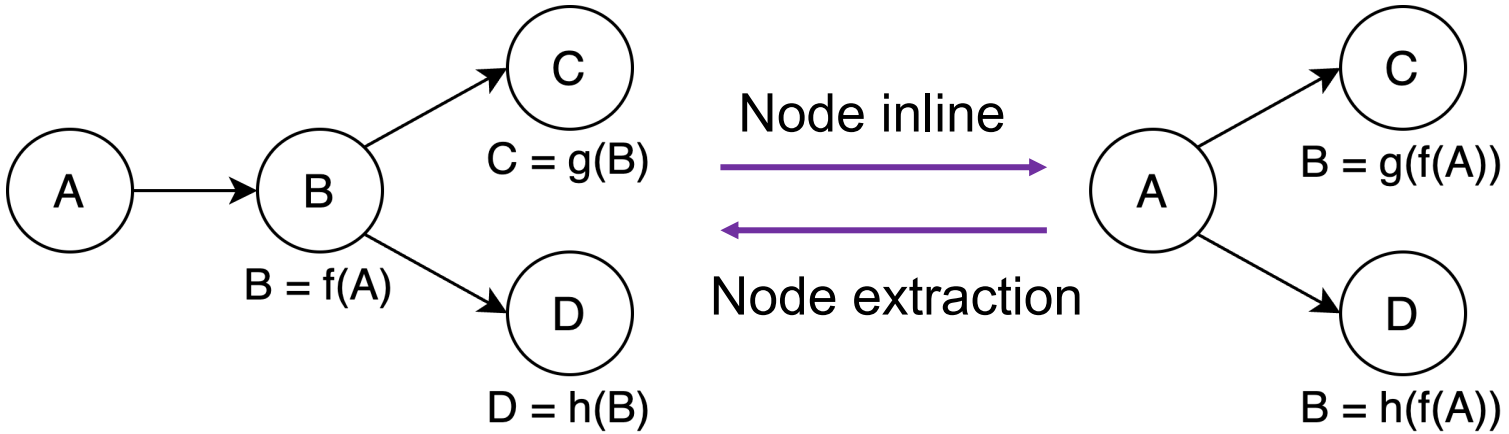
Node-Level Optimization 1

- Redundant node elimination $N\downarrow$
  - ❶**Alias Nodes**: duplicate nodes representing the same signal.
  - ❷**Dead Nodes**: nodes not used by other nodes.
  - ❸**Shorted Nodes**: nodes not selected due to another signal.
  - ❹**Unused Registers**: registers not used by other nodes, but they may be self-updated.



Node-Level Optimization 2

- Node inline and extraction: Trade-off between N and E
- GSIM models the cost of decision and select the best one
  - Extract cost: $cost\big(f(A)\big) + node\ overhead$
  - Inline cost: $cost\big(f(A)\big) \times \#reference$



$E \downarrow N \uparrow$

$E \uparrow N \downarrow$

Node-Level Optimization 3

- Traditional approach checks reset for each register, But:
  - The number of reset signals is small
  - Reset is hardly valid
- GSIM speculatively assumes reset does not happen, and checks each reset signal at the end of each cycle $E \downarrow$

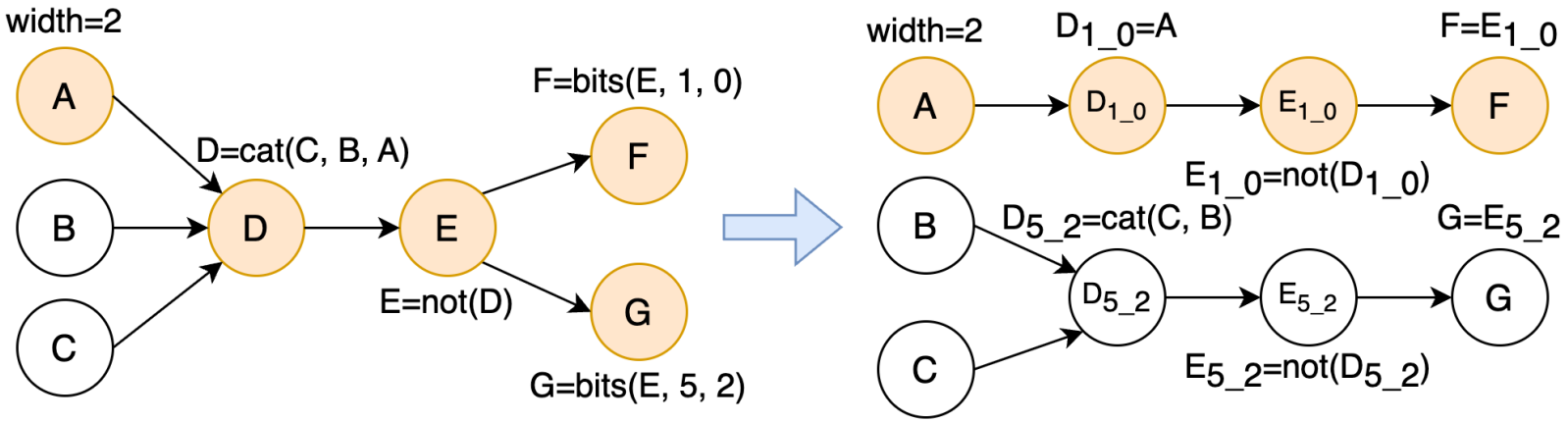
```
eval1():
  reg1 = reset ? 0 : new_val
  ...
cycle():
  if (active[1]) ...
  ...
```



```
eval1():
  reg1 = new_val
  ...
check_reset():
  if (reset)
    reg1 = 0
    reg2 = init_val2
    ...
    activate_all_succ()
```

Bit-Level Optimization

- In many long signals, only certain bits change each cycle
  - In XiangShan, 23.2% of the references to nodes only access a subset of their bits.
- GSIM adopts a bit-level splitting policy
  - Apply data flow analysis to each bit
  - Split nodes according to the reference boundaries



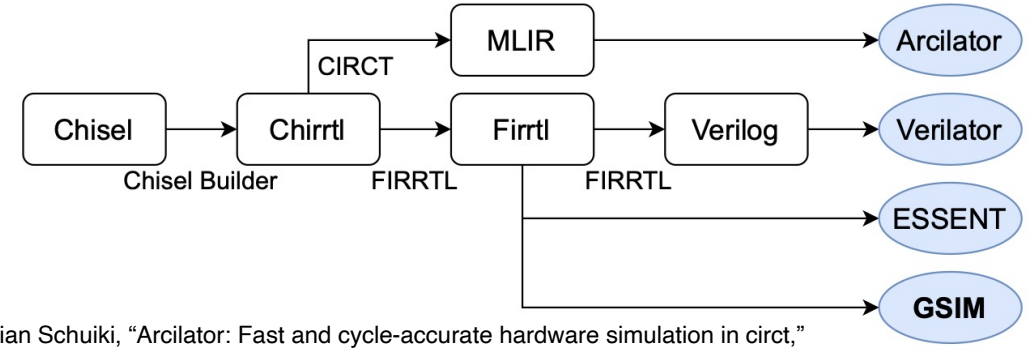
$af \downarrow N \uparrow$

Evaluation Setup

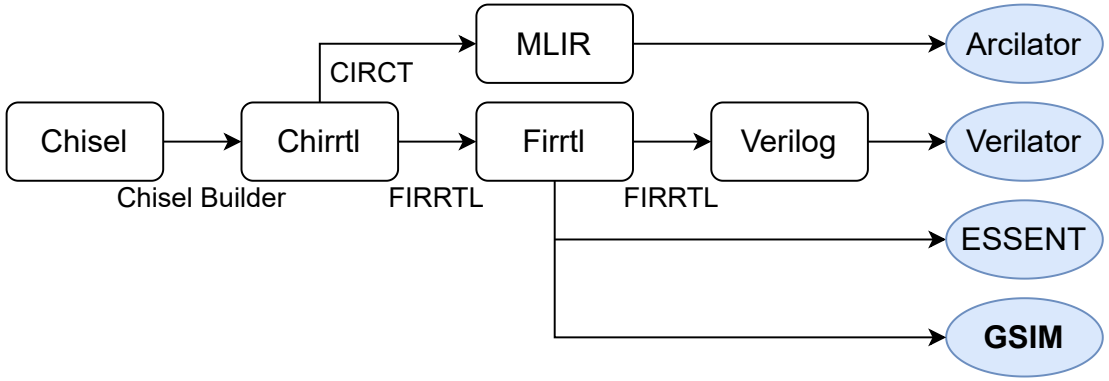
- Host: Intel 8-core i9-9900K at 3.6GHz with 64G of RAM
- Workload: Four processors developed in Chisel

Name	Configuration	IR node	IR edge	commit hash	commit date
stuCore	In-Order Single-Issue	9933	17369	——	——
Rocket	In-Order Single-Issue	234807	293164	e3773366a5	Sep 26, 2023
BOOM-Large	Out-of-Order Triple-Issue	571038	827619	9459af0c1f	Oct 17, 2023
XiangShan	Out-of-Order Six-Issue	6218427	9007005	a42a7ffe5e	Mar 1, 2024

- Simulators: Verilator, Arcilator[1], ESSENT, GSIM

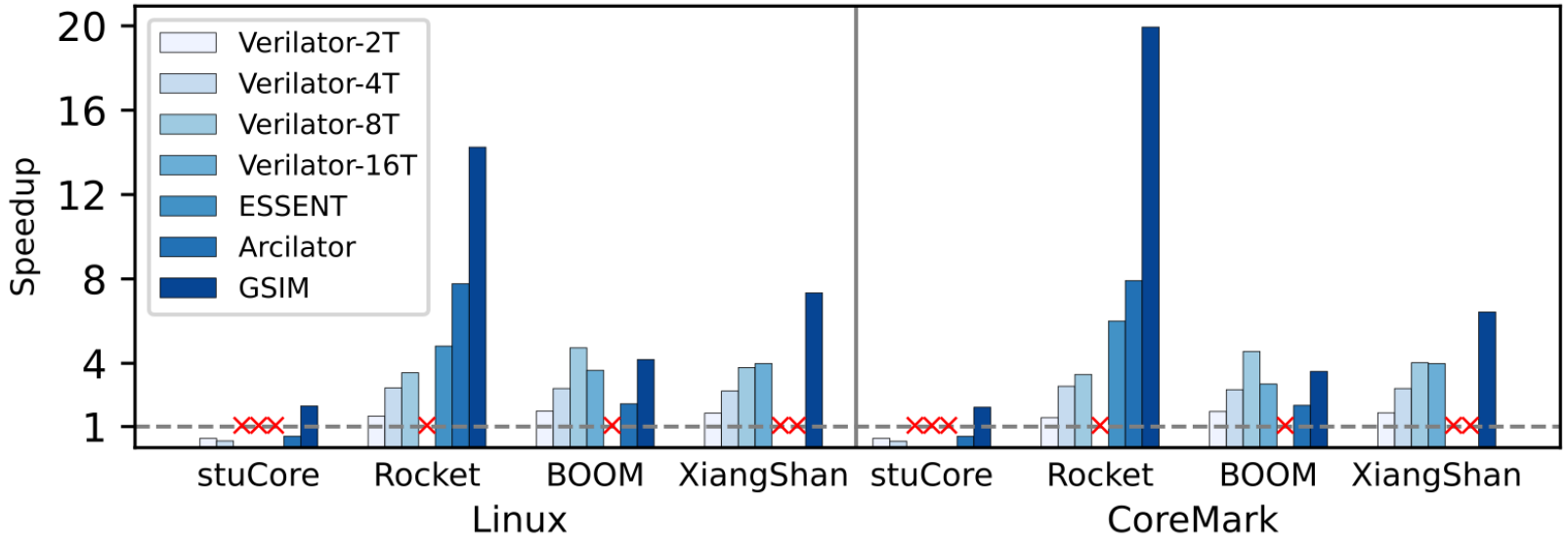


[1] Z. Y. B. H. T. G. Martin Erhart, Fabian Schuiki, “Arcilator: Fast and cycle-accurate hardware simulation in circt,”


MLIR
CIRCT
Arcilator
Chisel
Chirrtl
Firrtl
Verilog
Verilator
Chisel Builder
FIRRTL
FIRRTL
ESSENT
GSIM

Overall Performance on CoreMark and Linux

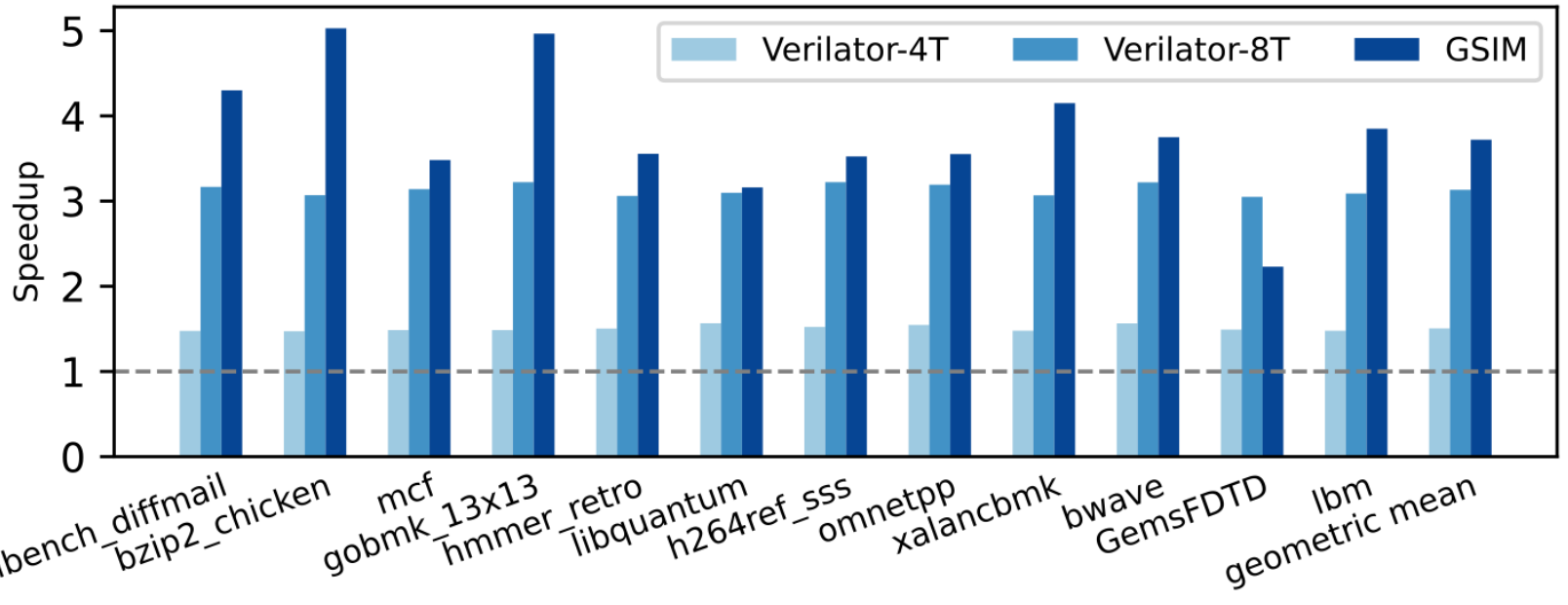
- Outperforms other single-threaded RTL simulators
  - 7.34x speedup over Verilator for booting Linux on XiangShan.
  - 19.94x speedup over Verilator for running CoreMark on Rocket.
- Outperforms multi-threading Verilator on most of the designs.



The speedup is normalized to the single thread version of Verilator.

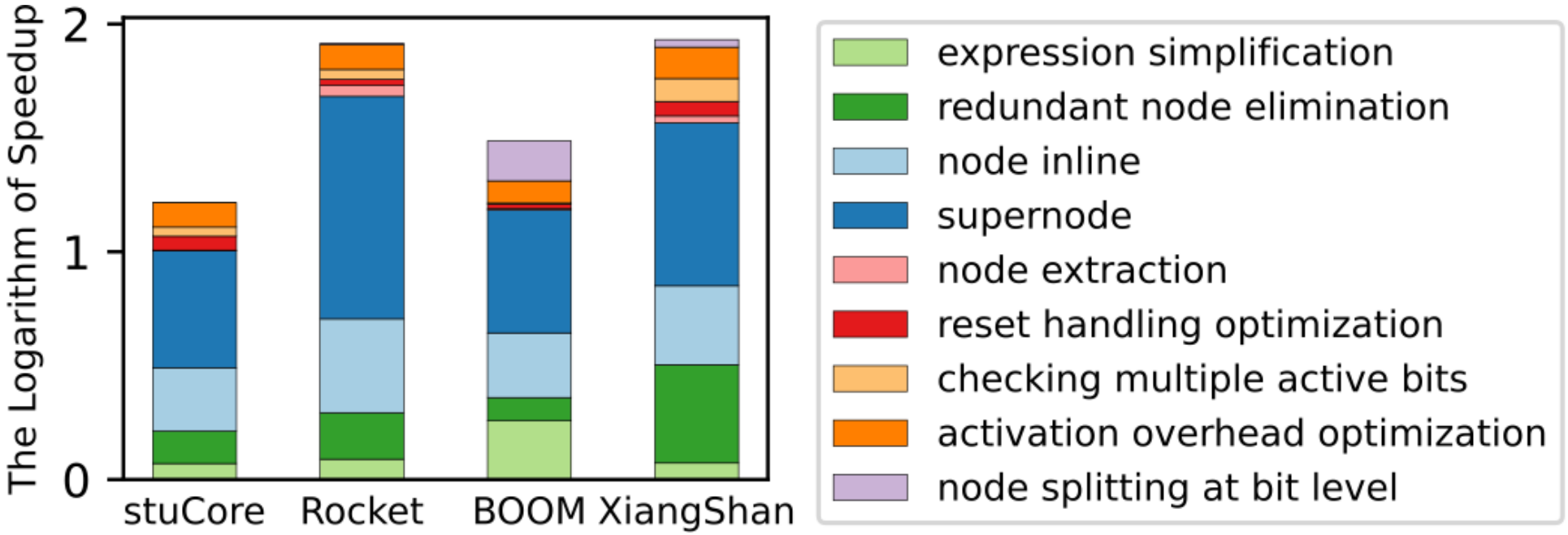
SPEC CPU2006 Performance on XiangShan

- Extract 40M-instruction segments from SPEC CPU2006 to create checkpoints using SimPoint.
- 3.72x faster than single-threaded Verilator, 1.18x of 8-threaded Verilator.



Performance Breakdown

- Apply all techniques incrementally with a baseline of no optimizations



Resource Usage

- Comparable emission time to Verilator
- The least amount of code
- Slightly larger data than Verilator

Design	Simulator	Emission Time(s)	Code Size(B)	Data Size(B)
stuCore	Verilator	1.1	394K	15K
	*ESSENT	7.7	659K	21K
	Arcilator	0.4	131K	13K
	GSIM	0.4	133K	16K
Rocket	Verilator	5.8	2.9M	62K
	ESSENT	37.8	1.6M	107K
	Arcilator	2.9	305K	49K
	GSIM	9.8	1.4M	72K
BOOM	Verilator	22.9	7.7M	954K
	*ESSENT	-	-	-
	Arcilator	4194.6	2.8M	799K
	GSIM	31.3	4.4M	976K
XiangShan	Verilator	374.6	40.7M	22.2M
	*ESSENT	-	-	-
	*Arcilator	-	-	-
	GSIM	389.1	25.4M	22.3M

Conclusion

- We explore the sources of computation overhead of RTL simulation and conclude them into five factors.
- We propose several techniques at the supernode level, node level and bit level.
- We implement these techniques in a fast RTL simulator GSIM
- **GSIM is open-sourced at https://github.com/OpenXiangShan/gsim**

Thank you!

Open-sourced at https://github.com/OpenXiangShan/gsim

chenlu22z@ict.ac.cn

SPONSORED BY CEDA® IEEE Council on Electronic Design Automation sigda acm