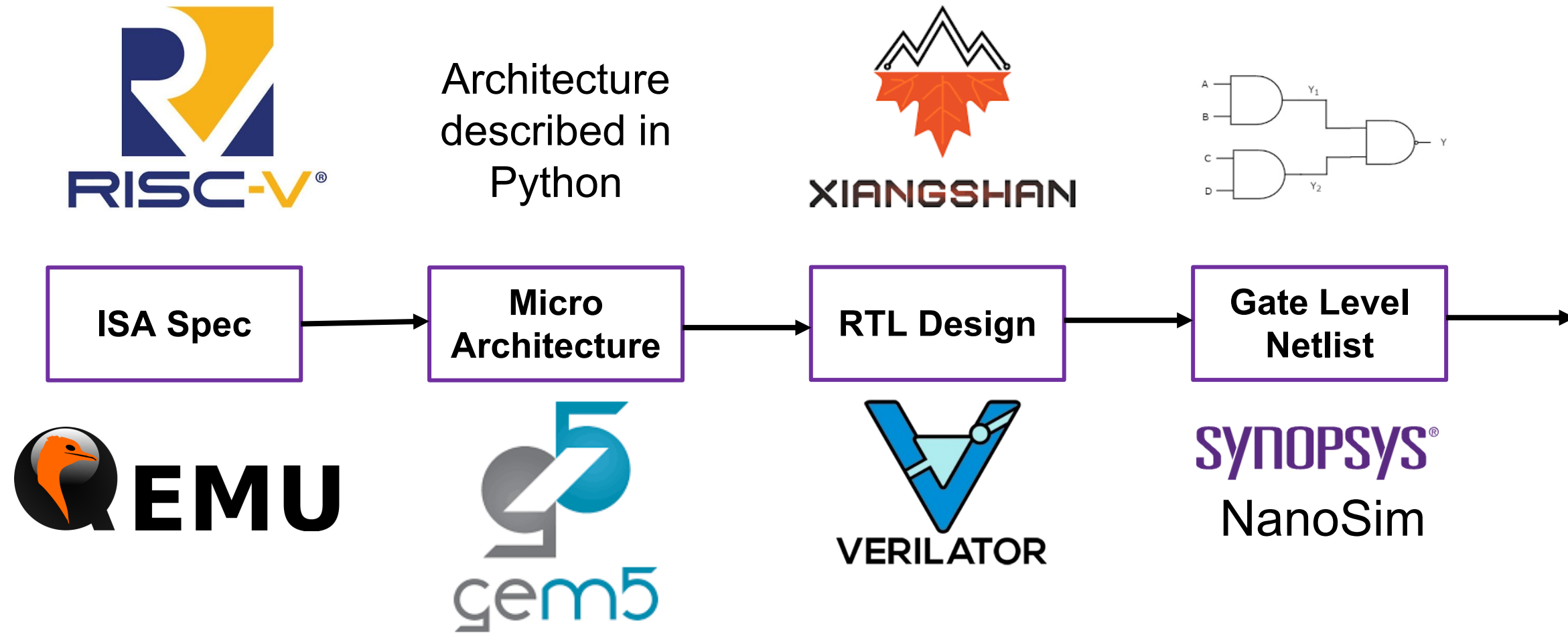


# 高性能RTL仿真器的设计和实现

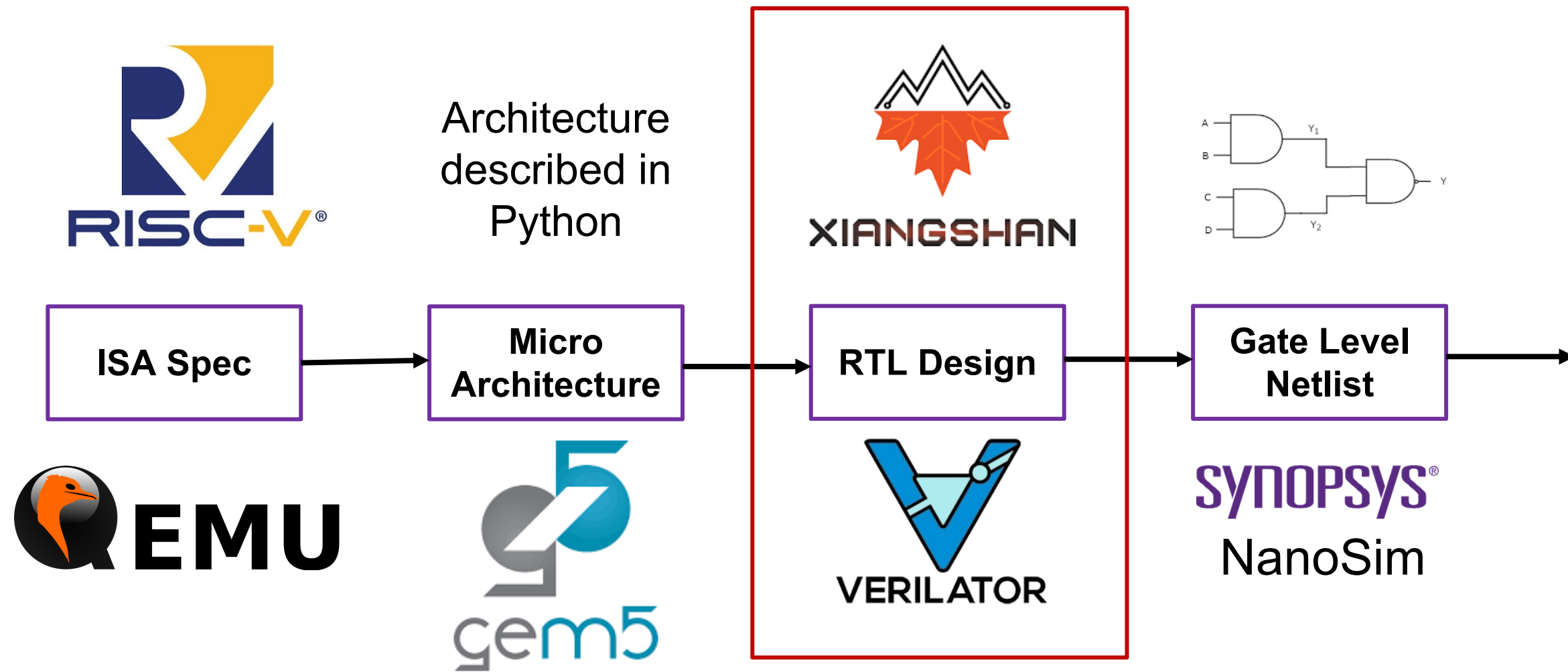
陈璐

2025.4.9

# ASIC设计流程中的仿真阶段





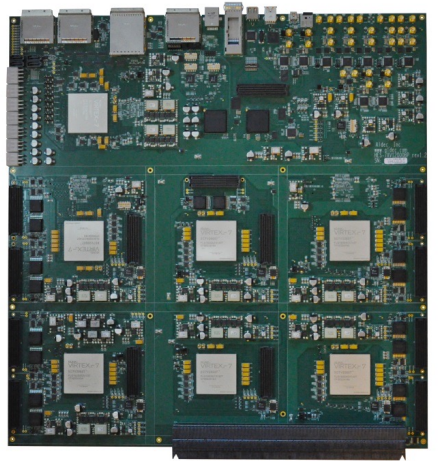

# ASIC设计流程中的仿真阶段



- RTL仿真被广泛应用于设计空间探索、验证、调试和初步性能探索

# RTL 仿真方式

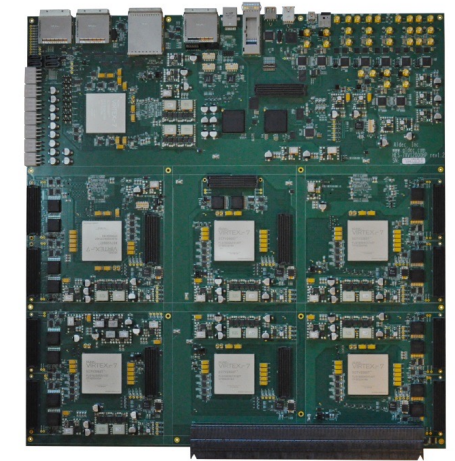
- 软件仿真：通过软件在通用计算机上模拟硬件电路的行为
  - 100% 信号可见，成本低，易于调试
  - 但对于复杂处理器的仿真过于耗时

	软件仿真	FPGA	硬件仿真加速器
速度	~1KHz	~100MHz	~1MHz
成本	<10万	<40万	>1000万
调试难度	低	高	低
支持的设计规模	大	中	大
	<div> VERILATOR</div> <div> SYNOPSYS<sup>®</sup> VCS</div>		<div>Cadence Palladium</div> 

# RTL 仿真方式

- FPGA：将RTL代码综合到FPGA中，以物理原型加速验证流程
  - RTL代码 → FPGA综合工具 → 烧录到FPGA芯片 → 实时运行
  - 运行速度接近真实芯片

	软件仿真	FPGA	硬件仿真加速器
速度	~1KHz	~100MHz	~1MHz
成本	<10万	<40万	>1000万
调试难度	低	高	低
支持的设计规模	大	中	大



Cadence  
Palladium



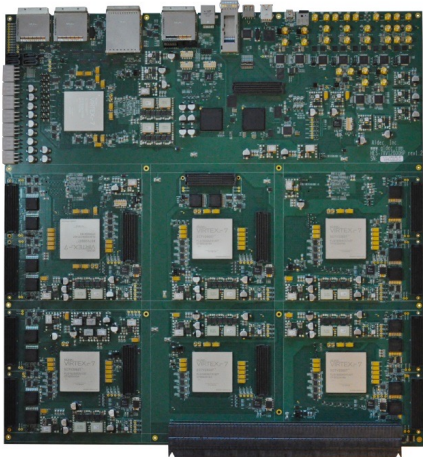
# RTL 仿真方式

- 仿真加速器：专用硬件平台，通过并行处理架构大幅提升仿真速度
  - 编译->加载到硬件仿真器->仿真

	软件仿真	FPGA	硬件仿真加速器
速度	~1KHz	~100MHz	~1MHz
成本	<10万	<40万	>1000万
调试难度	低	高	低
支持的设计规模	大	中	大



SYNOPSYS<sup>®</sup>  
VCS



Cadence  
Palladium





# 软件仿真在不同规模设计上的速度

- 软件仿真是最常用的仿真方式，随着电路规模的增大，软件仿真的速度快速下降，成为了验证过程的瓶颈<sup>[4]</sup>

Name	Configuration	IR node	IR edge	Speed
stuCore	In-Order Single-Issue	9,933	17,369	~ 900 KHz
Rocket	In-Order Single-Issue	234,807	293,164	~ 30 KHz
BOOM-Large	Out-of-Order Triple-Issue	571,038	827,619	~ 9 KHz
XiangShan	Out-of-Order Six-Issue	6,218,427	9,007,005	~ 0.9 KHz

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

# 软件仿真：事件驱动 vs. 全周期

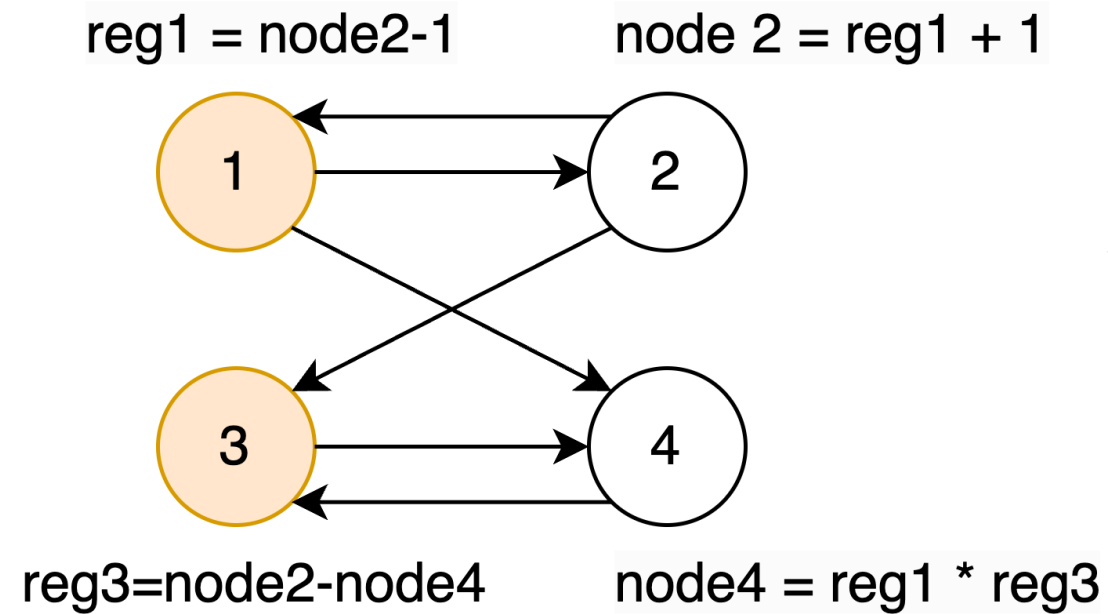
- 事件驱动：节点在运行时动态调度（VCS）
  - 将信号/电路状态的变化作为事件，在电路中传播
  - 可以建模任意延迟，被广泛地应用在综合后的门级仿真中
- 全周期：在编译时对节点排序，运行时静态调度（Verilator）
  - 最小仿真粒度为一个时钟周期，主要用于同步逻辑
  - 在RTL仿真中通常比事件驱动快

Field	Full-Cycle	Event-Driven
Wire Timing Model	zero delay	complex model
Schedule Overhead	small overhead	large overhead
Degree of Parallelism	large	small
Wave Accuracy	limited to cycle	very high

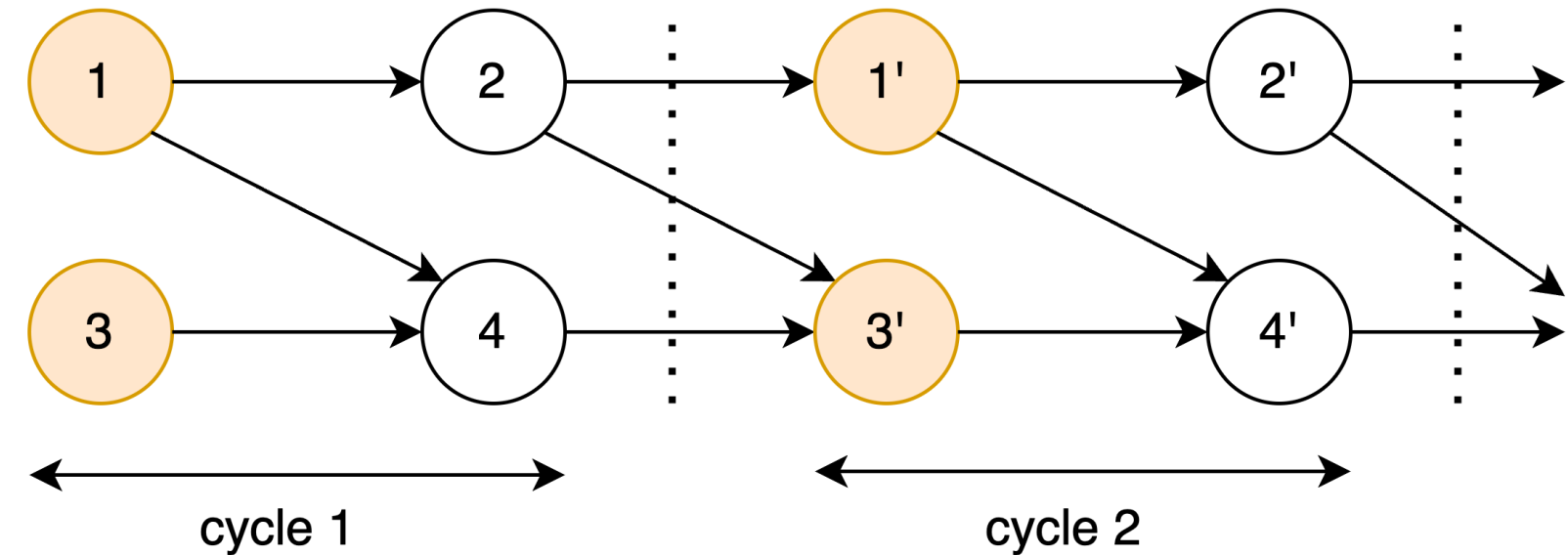
[2]



# 全周期仿真建模



simulation graph



RTL  
implementation

```
reg [31:0] reg1;  
reg [31:0] reg3;  
wire [31:0] node2 = reg1 + 32'h1;  
wire [31:0] node4 = reg1 * reg3;  
always @(posedge clock) begin  
    reg1 <= node2 - 31'h1;  
    reg3 <= node2 - node4;  
end
```

Test.v

verilator

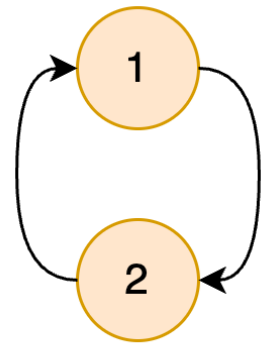
```
vlSelf->Test_D0T_reg3 = (((IData)(1U) + vlSelf->Test_D0T_reg1) - vlSelf->Test_D0T_node4);  
vlSelf->Test_D0T_reg1 = (vlSelf->Test_D0T_node2 - (IData)(1U));  
vlSelf->Test_D0T_node2 = (((IData)(1U) + vlSelf->Test_D0T_reg1);  
vlSelf->Test_D0T_node4 = (vlSelf->Test_D0T_reg1 * vlSelf->Test_D0T_reg3);
```

Test.cpp

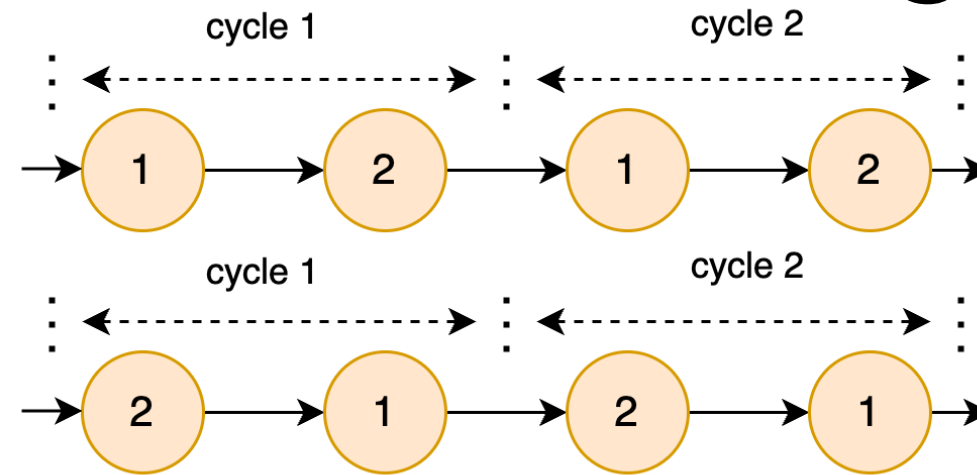
# 全周期仿真建模

- 组合逻辑不会引入环，但时序逻辑（e.g. 寄存器）可能引入环

$\text{reg1} = \text{reg2} + 1$



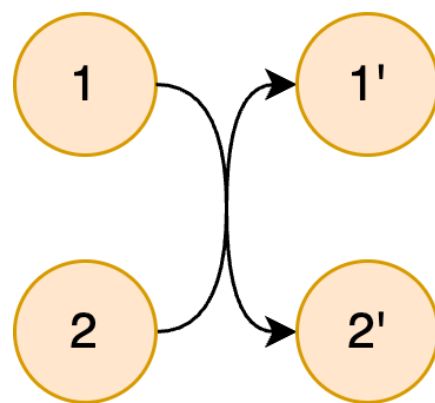
$\text{reg2} = \text{reg1} * 2$



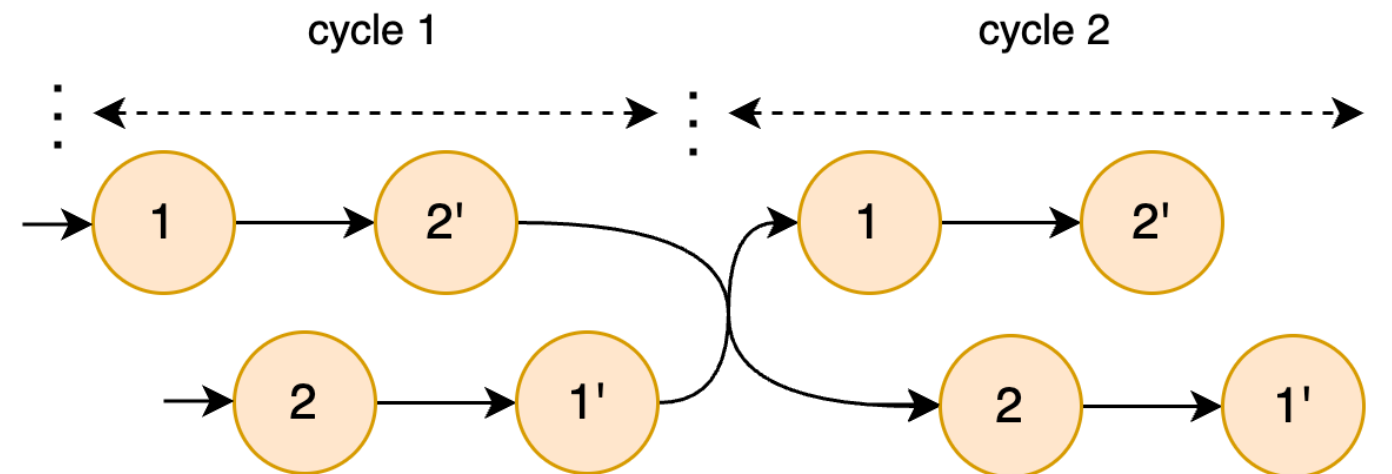
后更新的寄存器错误使用了下一周期的寄存器值

- 构造有向无环图：将寄存器拆分成两个节点，周期内锁存，并周期结束后进行更新

$\text{reg1}' = \text{reg2} + 1$



$\text{reg2}' = \text{reg1} * 2$



# 两种全周期仿真框架

- Verilator vs. ESSENT<sup>[3]</sup>

```
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**

[3] S.Beamer and D.Donofrio, "Efficiently exploiting low activity factors to accelerate rtl simulation," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20. IEEE Press, 2020.

# 建模与分析

- 每周期仿真开销

$$T = (E + A_{succ}) * af + A_{exam} * N$$

```
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()  
    ...
```

# 建模与分析

- 每周期仿真开销

$$T = (E + A_{succ}) * af + A_{exam} * N$$

- $E$ : 计算节点新值的开销

```
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()  
    ...
```

# 建模与分析

- 每周期仿真开销

$$T = (E + A_{succ}) * af + A_{exam} * N$$

- $E$ : 计算节点新值的开销
- $A_{succ}$ : 激活后继节点的开销

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()

...



# 建模与分析

- 每周期仿真开销

$$T = (E + A_{succ}) * af + A_{exam} * N$$

- $E$ : 计算节点新值的开销
- $A_{succ}$ : 激活后继节点的开销
- $af$ : 活动因子

```
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()  
    ...
```

# 建模与分析

- 每周期仿真开销

$$T = (E + A_{succ}) * af + A_{exam} * N$$

- $E$ : 计算节点新值的开销
- $A_{succ}$ : 激活后继节点的开销
- $af$ : 活动因子
- $A_{exam}$ : 检查激活位的开销

```
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()  
    ...
```

# 建模与分析

- 每周期仿真开销

$$T = (E + A_{succ}) * af + A_{exam} * N$$

- $E$ : 计算节点新值的开销
- $A_{succ}$ : 激活后继节点的开销
- $af$ : 活动因子
- $A_{exam}$ : 检查激活位的开销
- $N$ : 节点数目

```
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优化策略

- $T = ((E + A_{succ}) * af + A_{exam}) * N$
- 通过三种层次的优化来减少上述5个因素的开销

	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	+++	---			---

# 超节点层优化 1

- 将节点打包成超节点, 以超节点为粒度激活.  $A_{exam} \downarrow af \uparrow$ 
  - $A_{exam} \downarrow$ : 超节点中的所有节点共享同一个激活位
  - 保持较小  $af$ : 将可能同时激活的节点放入同一个超节点中

```
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()  
    ...
```

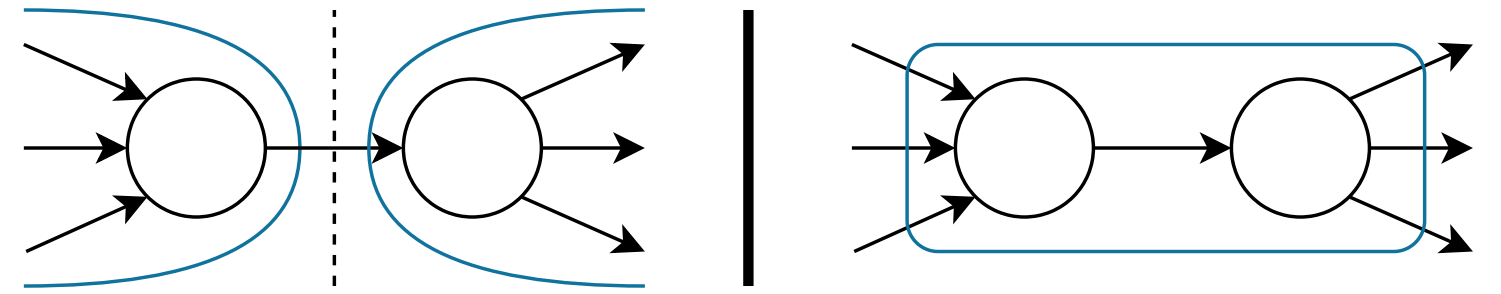


```
eval_super1():  
    eval_node1()  
    eval_node2()  
cycle():  
    if (active[1]) eval_super1()  
    if (active[2]) eval_super2()  
    ...
```

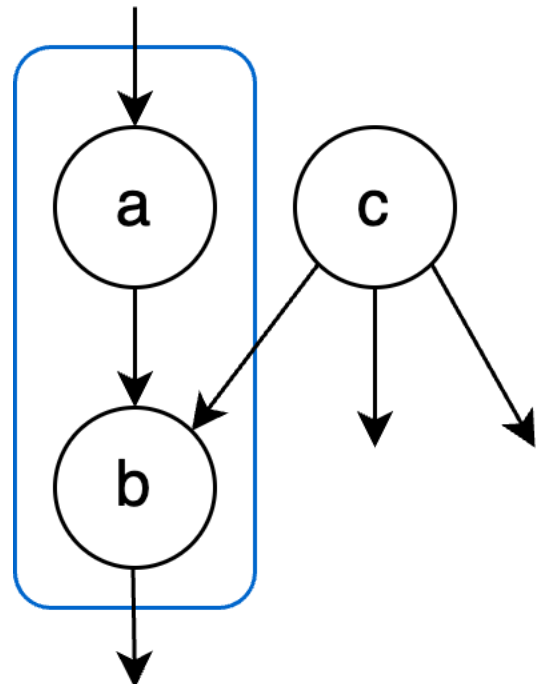
# 超节点层优化 1

- 针对仿真问题增强了有向图划分算法Kernighan's algorithm<sup>[4]</sup>

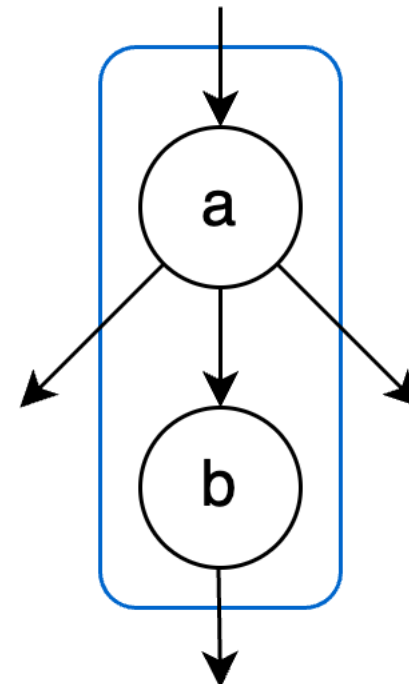
- Kernighan's algorithm 无法将连接较弱的节点划分在一起



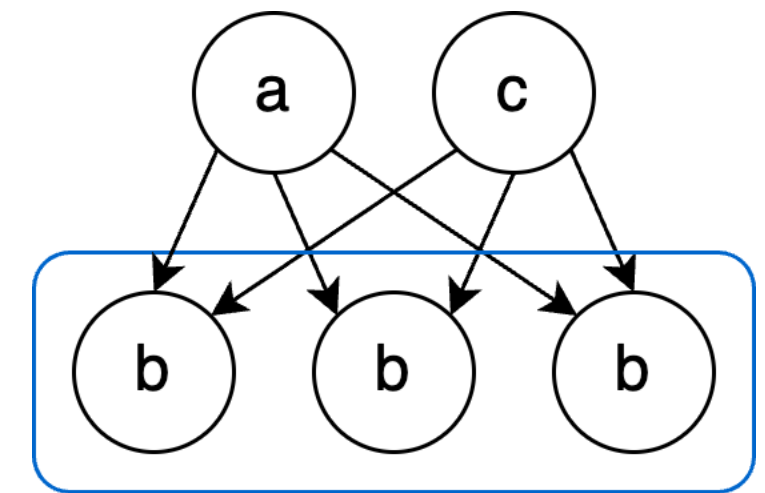
- 步骤1: 基于以下观察合并节点:



nodes with out-degree 1



nodes with in-degree 1



siblings with the same predecessors

- Step2: 应用Kernighan's algorithm



## 超节点层优化 2

- 由于电路中 $af$ 较小，多个连续的激活位通常为0
  - 利用SIMD思想，在快速路径上使用一条指令同时比较多个active位
  - $A_{exam} \downarrow$

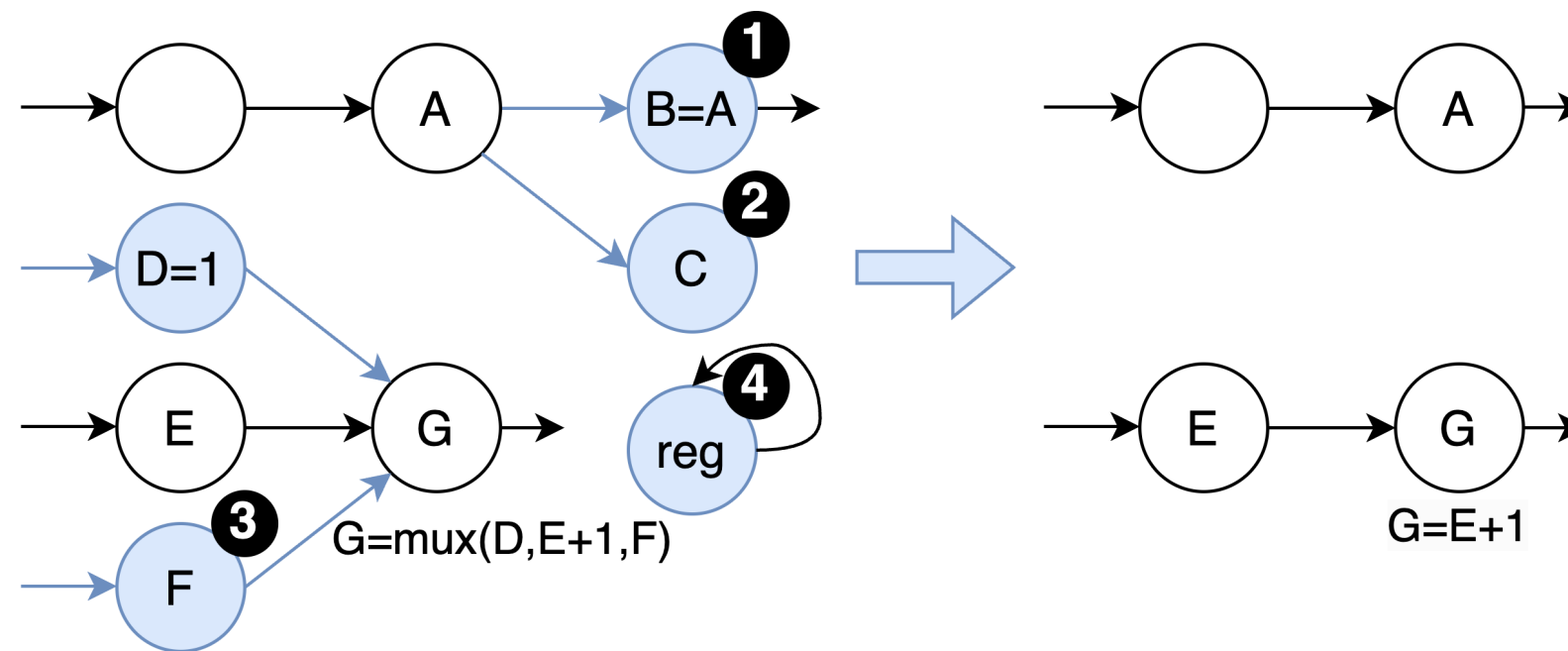
```
eval_super1():  
    eval_node1()  
    eval_node2()  
cycle():  
    if (active[1]) eval_super1()  
    if (active[2]) eval_super2()  
    ...
```



```
eval_super1():  
    eval_node1()  
    eval_node2()  
cycle():  
    if (active[8:1] != 0)  
        if (active[1]) eval_super1()  
        if (active[2]) eval_super2()  
    ...
```

# 节点层优化1

- 冗余节点消除  $N\downarrow$
- ①别名节点: 表示相同信号的重复节点
- ②死节点: 未被其他节点使用的节点
- ③被短路的节点: 由于其他控制信号导致未被选中的节点
- ④未使用寄存器: 未被其他节点使用的寄存器, 但可能自更新

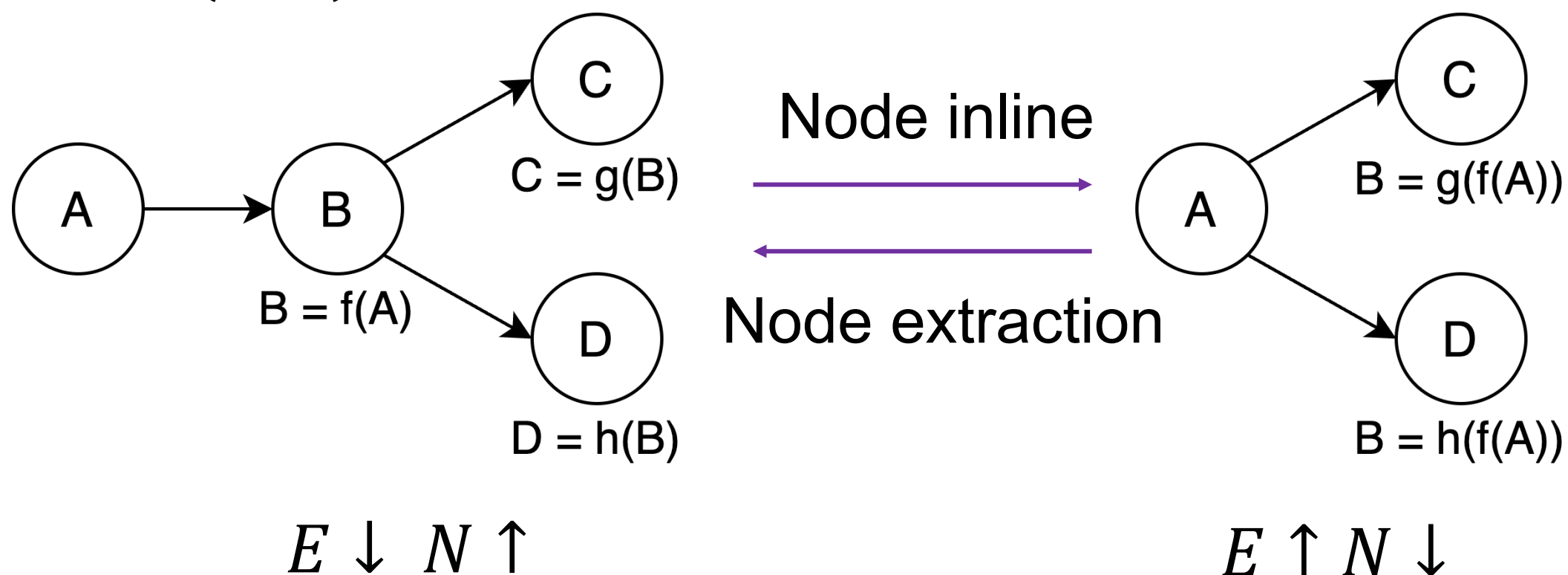


## 节点层优化 2

- 节点内联和节点提取：权衡  $N$  和  $E$  的开销
- GSIM建立开销模型来确定是否内联/提取节点
  - 如果 $cost(f(A)) \times \#reference > cost(f(A)) + node\ overhead$  则提取节点，反之则内联该节点

$$cost_{extraction} = cost(f(A)) + node\ overhead$$

$$cost_{inline} = cost(f(A)) \times \#reference$$



## 节点层优化 3

- 传统的寄存器更新策略会检查每一个寄存器是否复位，但是
  - 复位信号数量通常较少
  - 复位很少发生
- GSIM假设复位不发生，推测执行寄存器的更新逻辑，并在周期结束时进行统一检查 $E \downarrow$

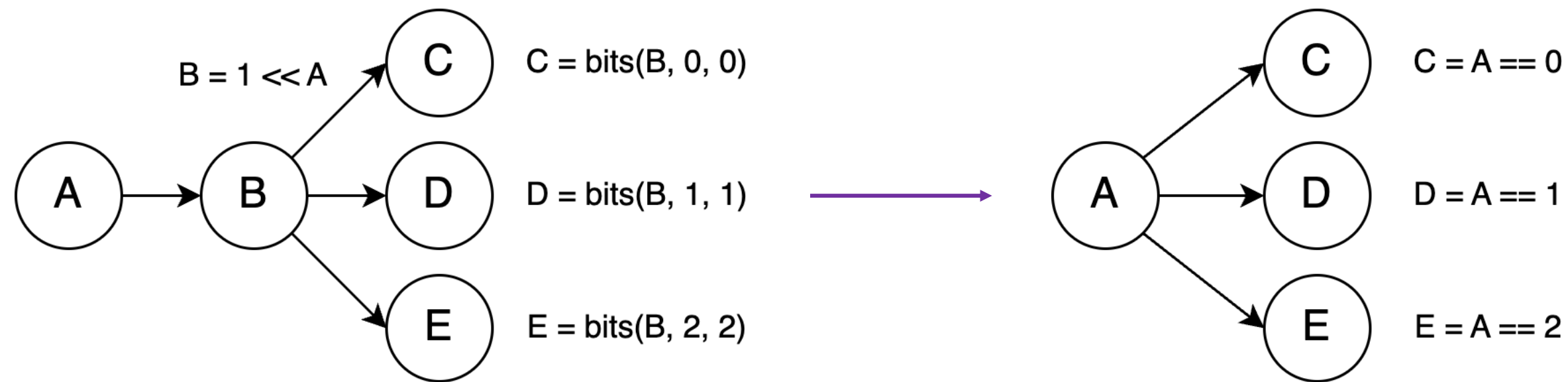
```
eval1():  
  reg1 = reset ? 0 : new_val  
  ...  
cycle():  
  if (active[1]) ...  
  ...
```



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

# 其他节点层优化

- 表达式优化  $E \downarrow$



- 优化激活后续节点的过程  $A_{succ} \downarrow$

- 评估条件操作和逻辑操作的开销，选择开销最低的方式

eval\_node1():

save old value

update new value

if (old != new) active[succ] = true

eval\_node1():

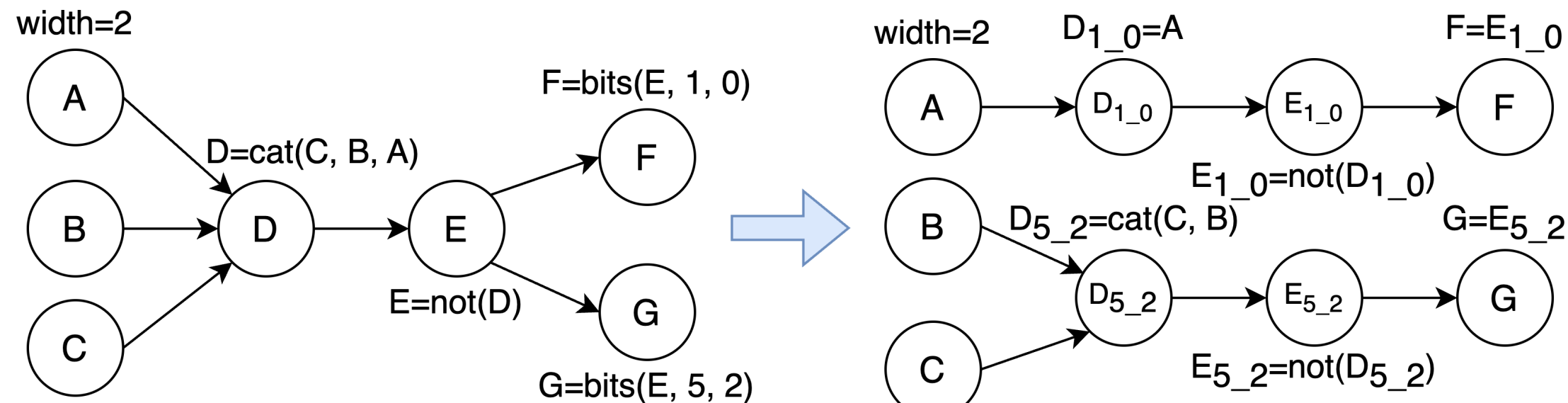
save old value

update new value

active[succ] |= mask

# 比特级优化

- 在很多长信号中，每周期只有部分位发生改变
  - 在香山中，23.2%对节点的访问只使用了其中某部分位
- GSIM使用了位级的节点拆分策略
  - 对节点中每一位进行数据流分析
  - 依据访问边界对节点进行拆分



$af \downarrow N \uparrow$

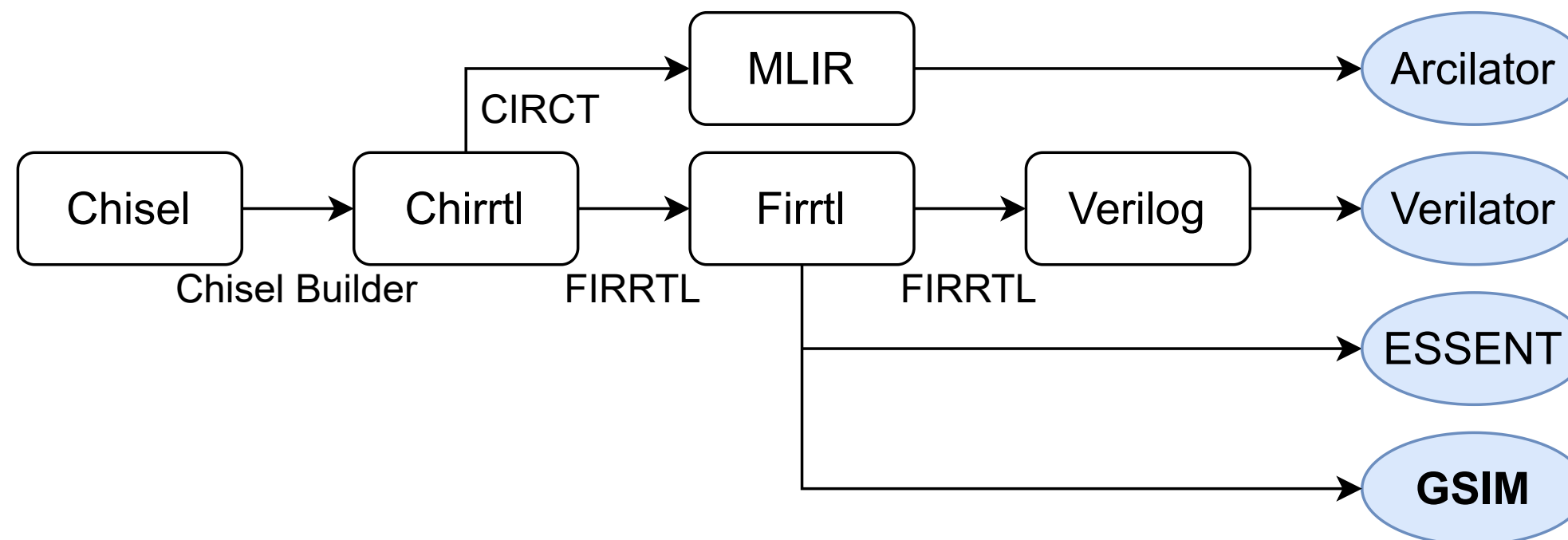


# 实验设置

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

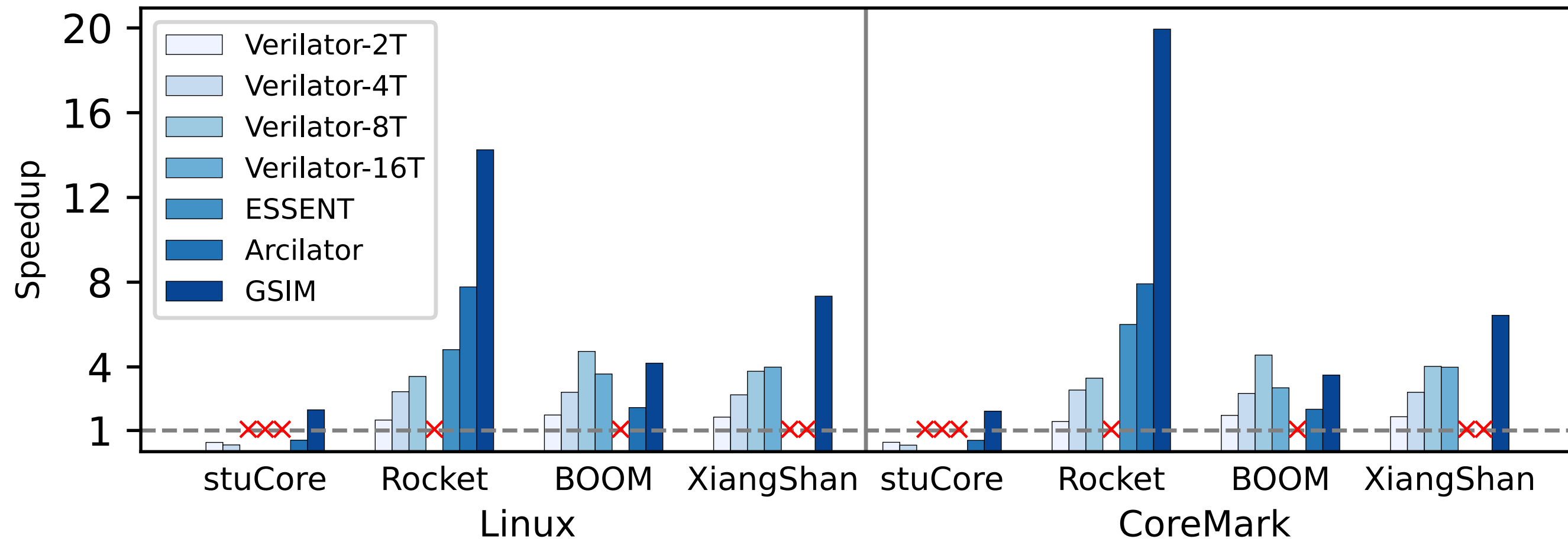
Name	Configuration	IR node	IR edge	commit hash	commit date
stuCore	In-Order Single-Issue	9933	17369	——	——
Rocket	In-Order Single-Issue	234807	293164	e3773366a5	Sep 26, 2023
BOOM-Large	Out-of-Order Triple-Issue	571038	827619	9459af0c1f	Oct 17, 2023
XiangShan	Out-of-Order Six-Issue	6218427	9007005	a42a7ffe5e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

- Simulators: Verilator, Arcilator, ESSENT, GSIM



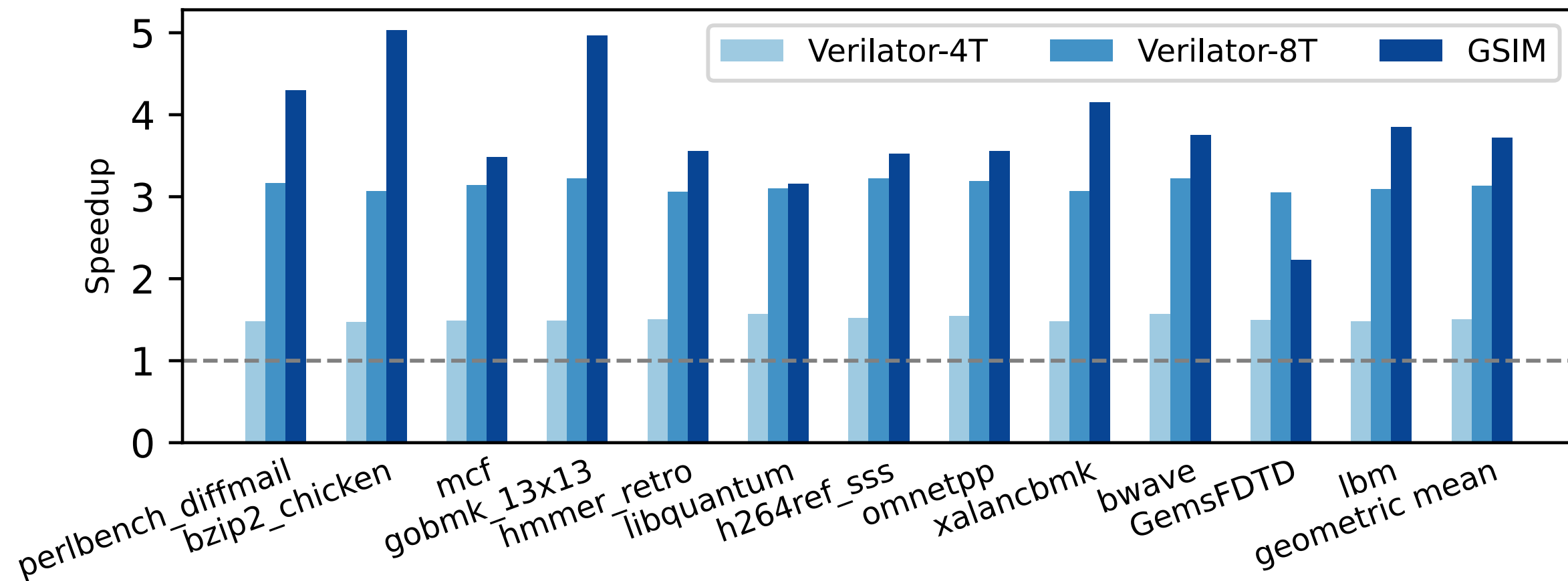
# 整体性能：运行CoreMark和Linux

- 性能远超其他单线程RTL仿真器
  - 仿真香山启动Linux性能为verilator的7.34倍
  - 仿真Rocket运行CoreMark性能为verilator的19.94倍
- 在大多数处理器核上性能超过多线程verilator



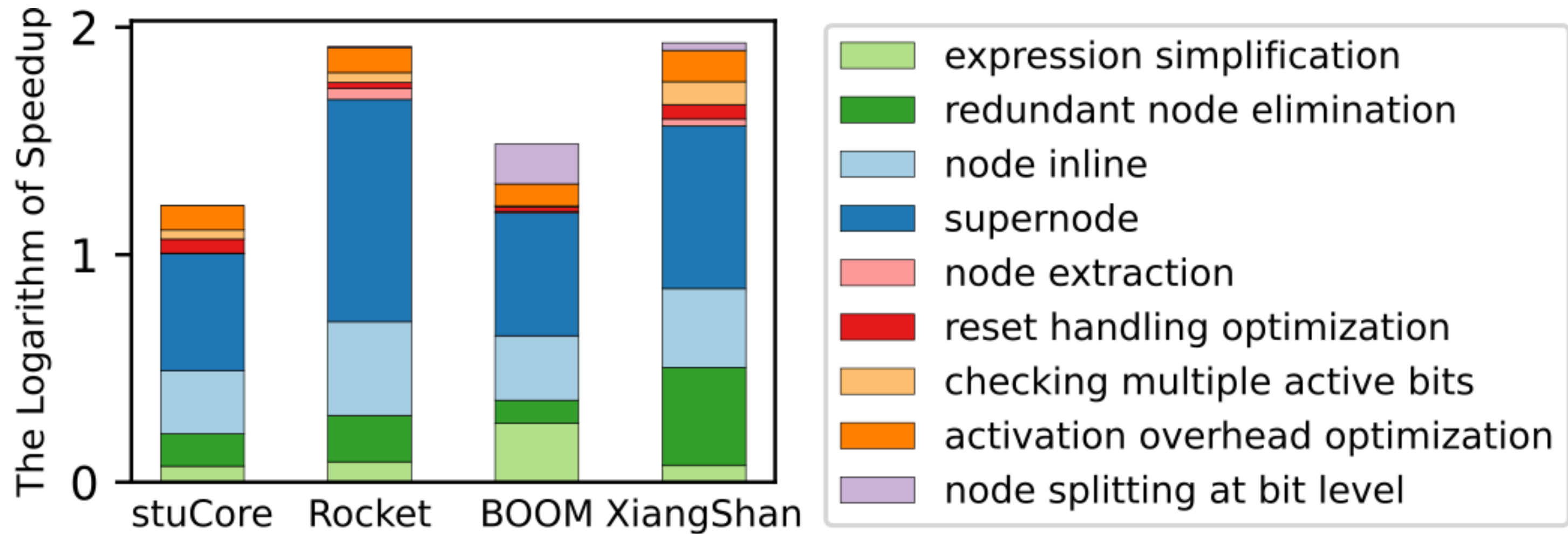
# 仿真香山运行SPEC CPU2006 的性能表现

- 通过SimPoint从SPEC CPU2006 采样得到包含40M条指令的片段，并构建检查点
- 仿真性能为单线程Verilator的3.72倍，8线程Verilator的1.18倍



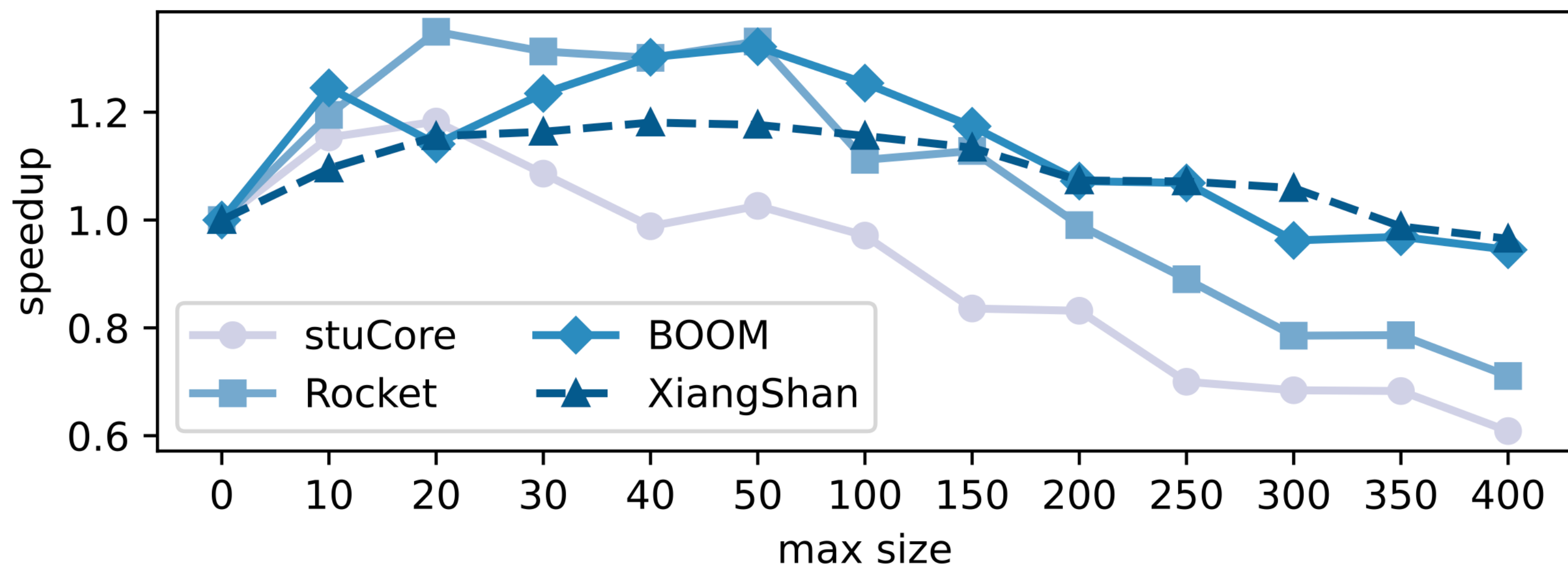
# 性能分解

- 在未加优化的基础上，增量式地添加所有优化



# 测试超节点最大节点数

- 超节点大小增大将减小 $A_{exam}$  但增加  $af$
- 最优大小范围为20-50



# 不同构建超节点的划分算法

- 仿真BOOM运行CoreMark比较不同划分算法的性能
- GSIM在 $A_{exam}$ ,  $A_{succ}$  和  $af$  中取得了更好的平衡

	partition time (s)	supernode $A_{exam}$	activation times $A_{succ}$	active node $af$	speed (Hz)
None	—	536721	64044	42398	913
Kernighan's	1.3	21769	13104	93327	4522
MFFC-based	88.9	36478	7310	124202	6802
GSIM	1.6	14261	8169	114255	8184



# 资源使用情况

- 编译时间和Verilator相当
- 生成的代码大小更小
- 数据大小比Verilator略大

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

# 未来工作

- 性能优化
  - 通过多线程进一步加速RTL仿真速度
  - 香山场景中加速 (e.g. 性能计数器, dpic)
- 功能测试
  - 引入更多软件负载/工具用于测试
- 更多特性
  - 生成波形
  - 支持更多HDL