LightSSS：基于内存的轻量级仿真快照

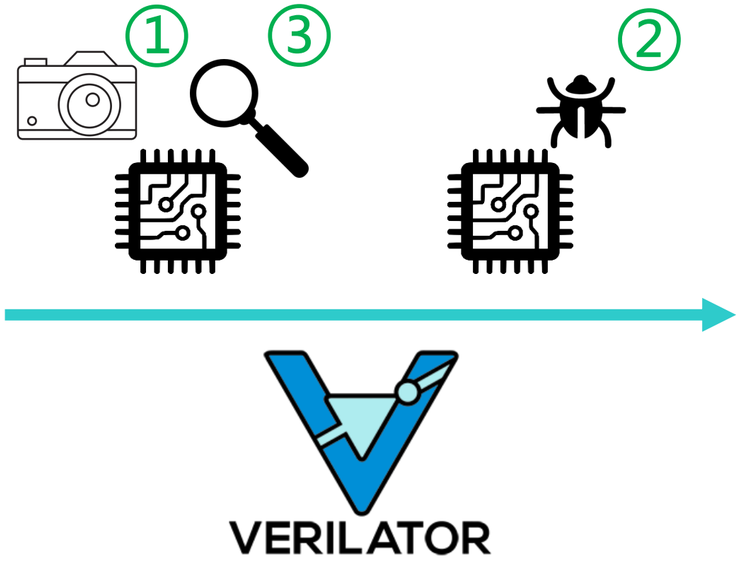
余子濠　金越
中科院计算所

2021.06@上海

中国科学院计算技术研究所
Institute Of Computing Technology Chinese Academy Of Sciences

复杂处理器需要长时间仿真

- 香山处理器[1]乱序6发射11级流水
  - https://github.com/OpenXiangShan/XiangShan
- Verilog代码122.7万行
- 使用Verilator仿真效率≈10K指令/s
  - 运行coremark 1轮循环 – 0.45分钟
  - 启动最小linux – 6.20分钟
  - 启动debian至登录提示符 – 9.67天



对电路状态进行快照

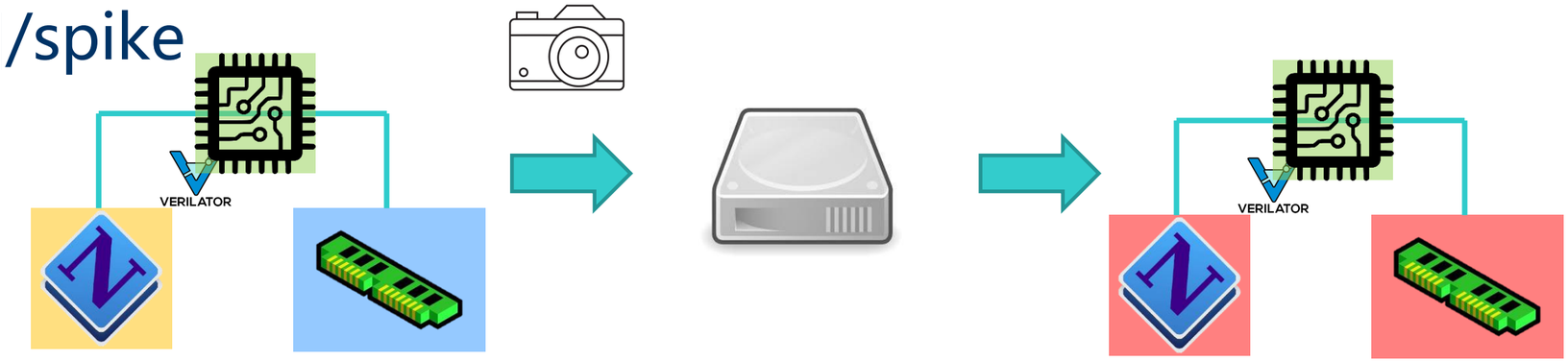
- 电路状态的快照功能对调试有很大帮助
  - 无需每次都从头开始运行仿真

相关系列报告
[1] 香山: 开源高性能RISC-V处理器, 周二14:30

现有快照无法适用于外部模型

- Verilator提供--savable选项, 可将电路模型的状态保存到文件
  - 仅支持通过Verilator生成的模型

- 但--savable选项无法保存外部模型的状态
  - 功能模型, 如NEMU[1][2]/spike
  - 性能模型, 如DRAMSim

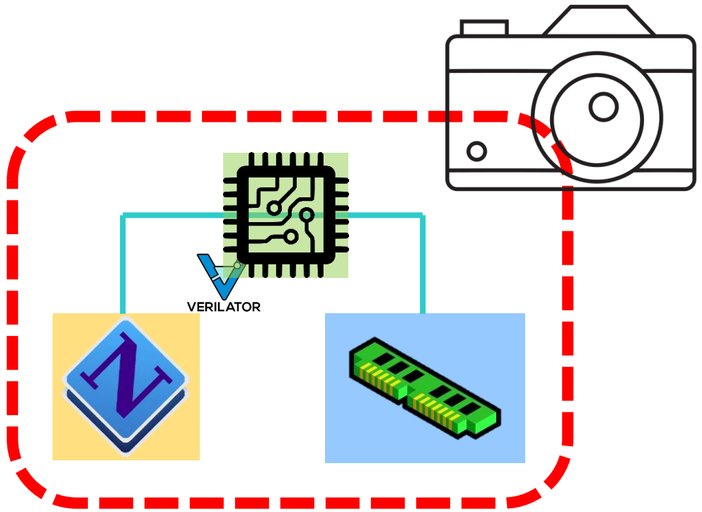


- 手动保存外部模型的状态需要理解内部细节, 既繁琐又易错
  - 状态遗漏 = 仿真结果错 = 重跑
  - 是否有更好的方法?

相关系列报告
[1] NEMU: 一个效率接近QEMU的高性能解释器, 周三13:20
[2] SMP-MArch-Diff: 支持多处理器和RV微结构状态的差分测试方法, 周四14:50

LightSSS - 一种轻量级仿真快照

- 需求
  - 如何在无需理解外部模型细节的情况下保存其状态?
  - 如何做到轻量级?
- 思考
  - 无需理解外部模型 -> 抽象
  - 状态 -> 数据结构 -> 内存的值
  - 轻量级 -> 内存

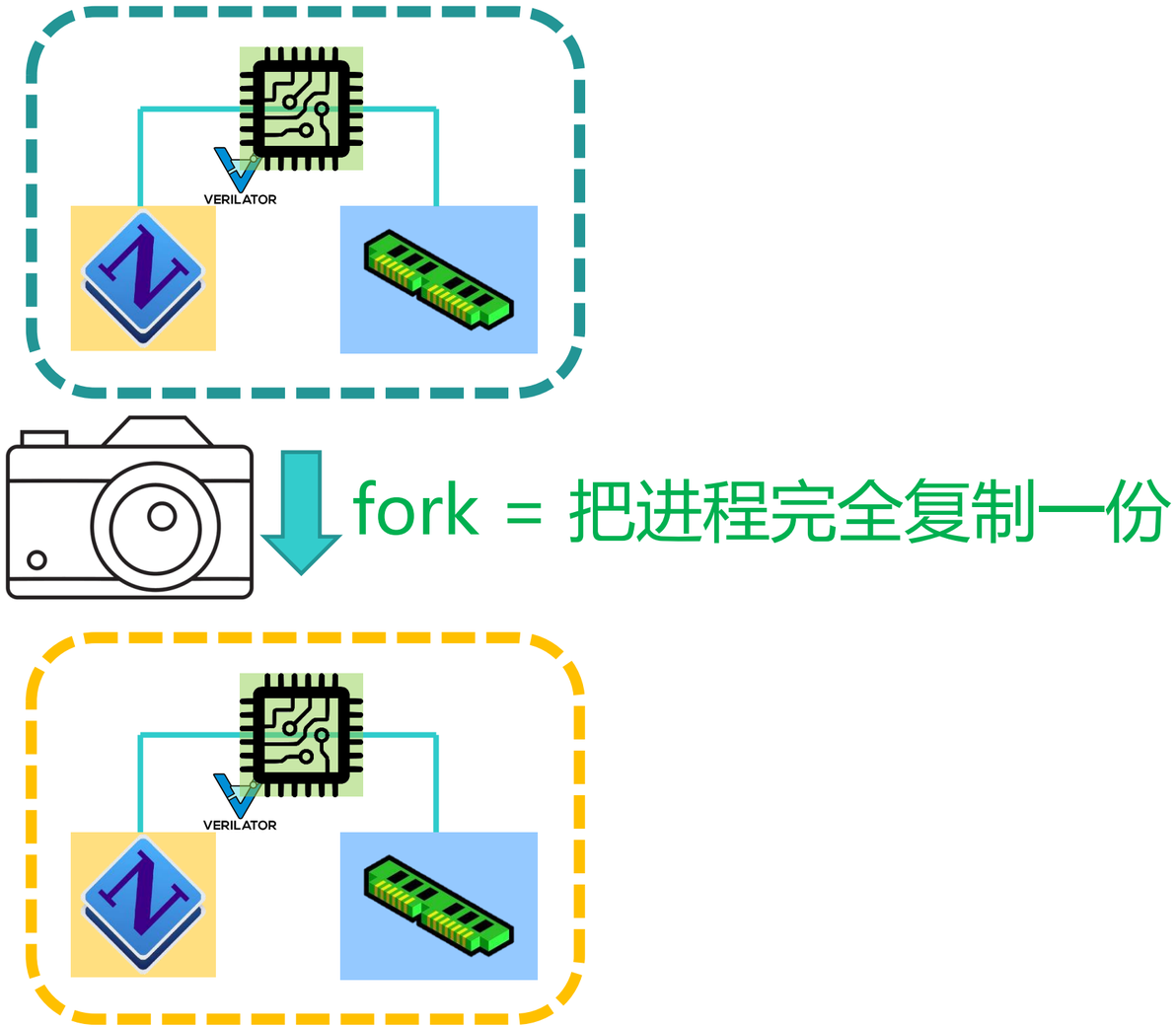


- 操作系统的礼物 - **进程抽象!**
  - 进程 = 正在执行的程序

- Light-weight Simulation SnapShot

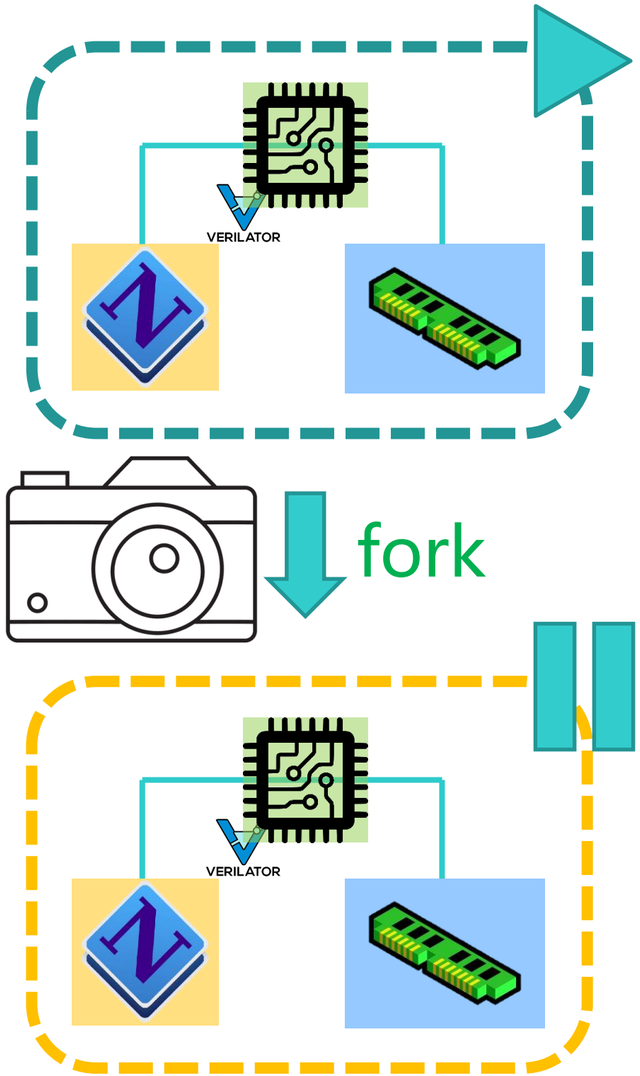
LightSSS的诀窍 - fork系统调用

- 核心思想: **从进程抽象视角看仿真状态，用fork对进程状态做快照**



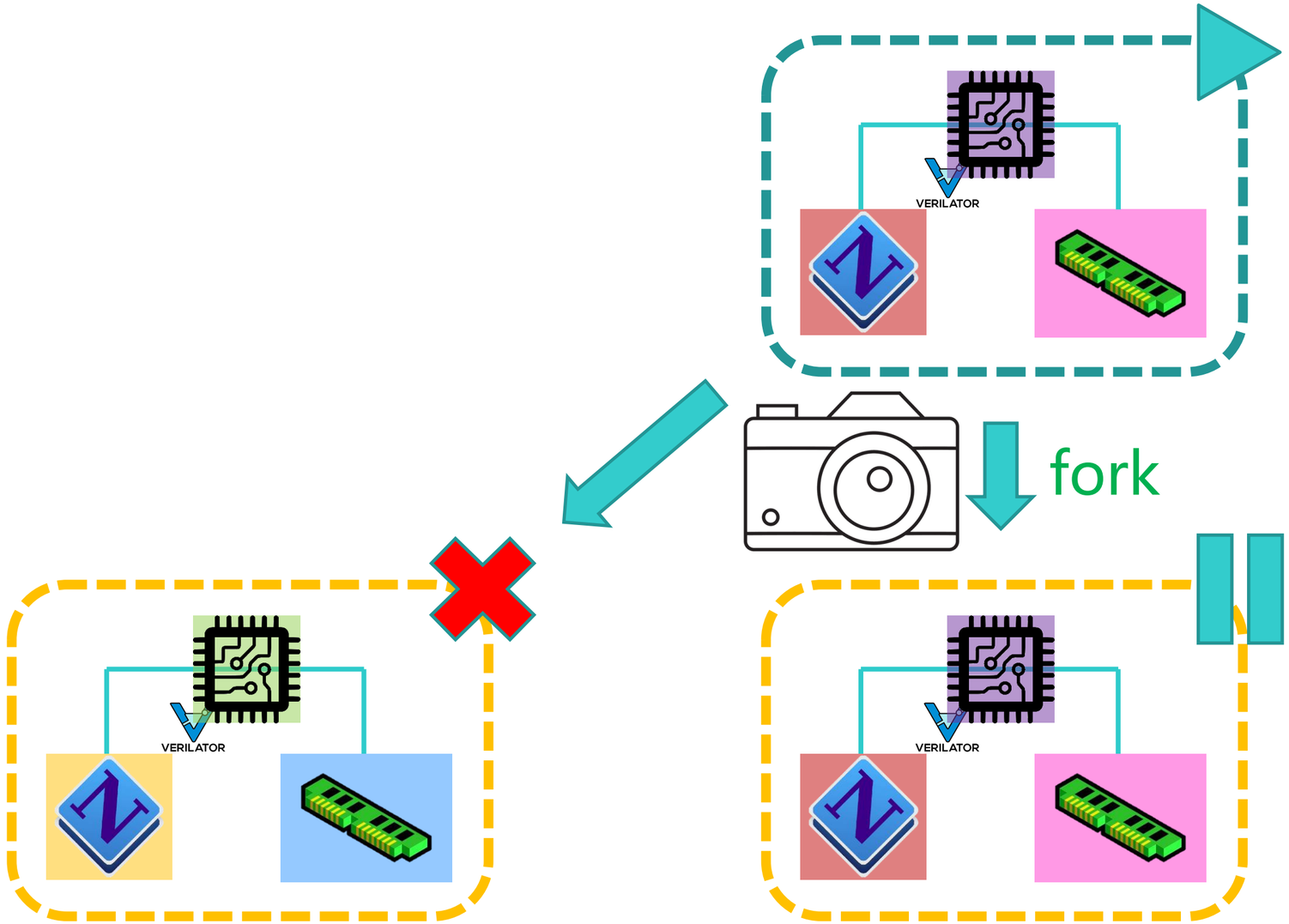
LightSSS工作过程

- (1) 通过fork保存当前状态, 子进程暂停, 父进程继续



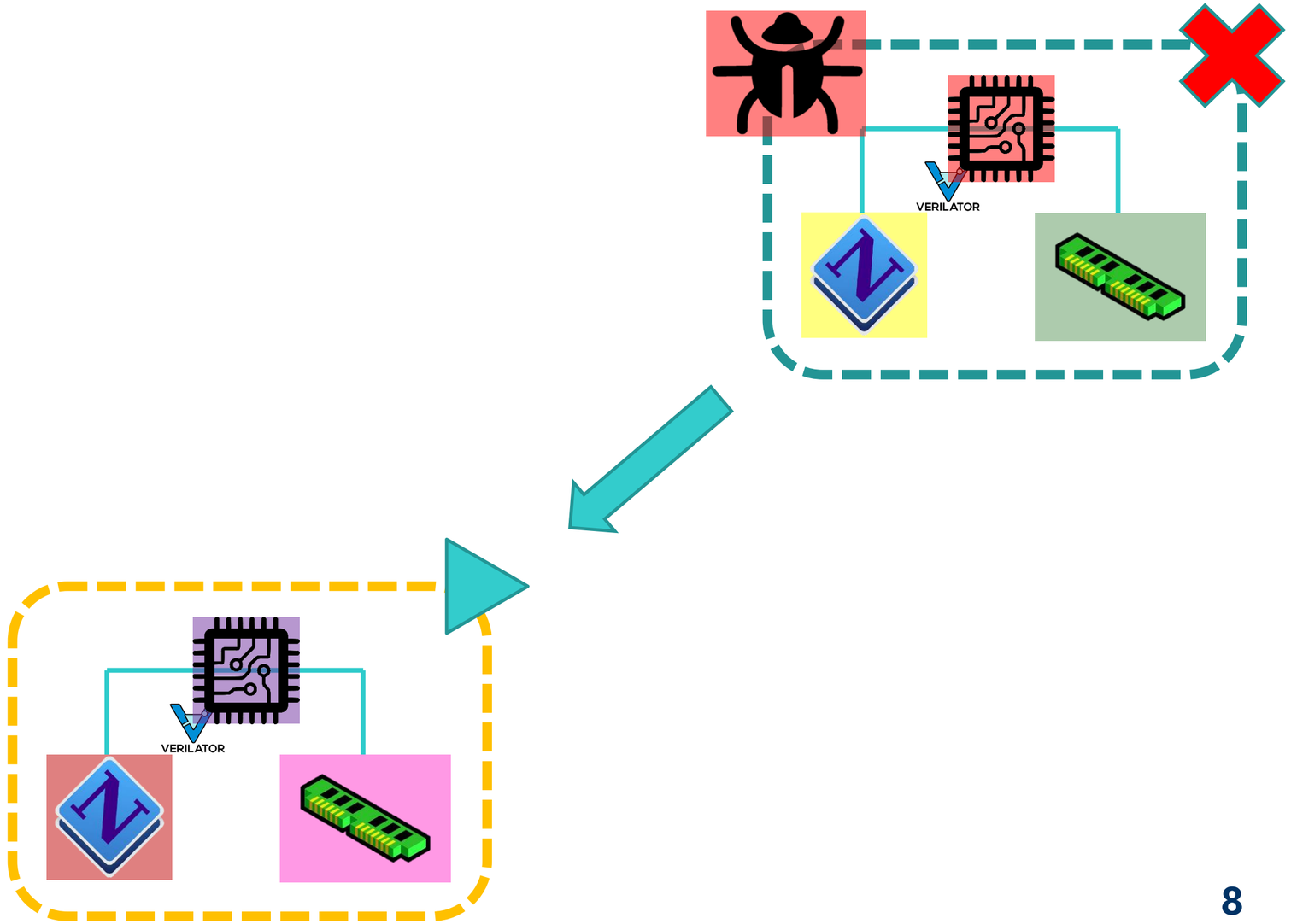
LightSSS工作过程

- (2) 父进程运行一段时间后, 若未出错, 则保存新状态, 并移除旧状态



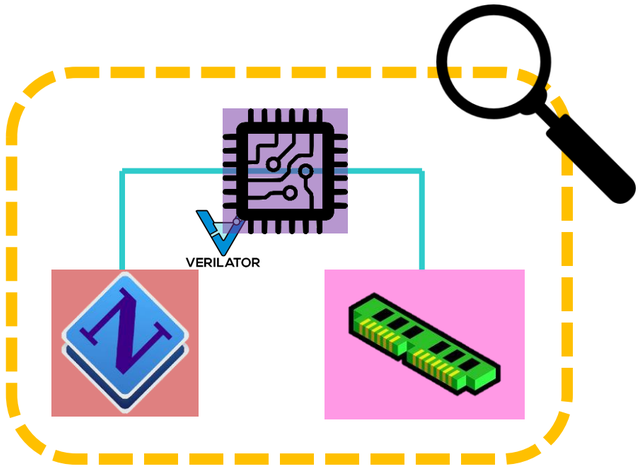
LightSSS工作过程

- (3) 若父进程仿真出错, 则唤醒最近保存的子进程, 自身退出



LightSSS工作过程

- (4) 子进程进入调试状态, 供开发者进行调试



评估分析

- **LightSSS的扩展性**
  - 实现LightSSS只需添加约100行代码
  - 此后无需额外添加任何代码, 即可支持各种外部模型的快照
    - 虚拟SD卡控制器, NEMU模拟器, DRAMSim

- **LightSSS的效率**
  - AMD EPYC 7742(256线程)/内存512GB/g++编译选项-j128/16线程模型

	Verilator 编译时间/s	g++ 编译时间/s	10轮coremark 运行时间/s	单次快照 时间开销/s
--savable	5765.49	19039.00	519.10	3.719
LightSSS	1515.47	6720.00	445.22	0.000535
效率/倍	**3.80**	**2.83**	**1.17**	**6951.40**

编译效率分析

- --savable选项生成了超大文件, 拖慢编译速度

```
-rw-r--r-- 1 jy xs   93M 4月  15 22:31 VXSSimSoC__stats.txt
-rw-r--r-- 1 jy xs   54M 4月  15 22:30 VXSSimSoC.h
-rw-r--r-- 1 jy xs  6.7M 4月  15 22:30 VXSSimSoC__33.cpp
-rw-r--r-- 1 jy xs  6.6M 4月  15 22:30 VXSSimSoC__Slow.cpp
-rw-r--r-- 1 jy xs  5.2M 4月  15 22:30 VXSSimSoC__1__Slow.cpp
-rw-r--r-- 1 jy xs  4.6M 4月  15 22:30 VXSSimSoC__210__Slow.cpp
```

添加--savable选项

```
-rw-r--r-- 1 jy xs   97M 4月  15 19:49 VXSSimSoC__Slow.cpp
-rw-r--r-- 1 jy xs   93M 4月  15 19:49 VXSSimSoC__stats.txt
-rw-r--r-- 1 jy xs   54M 4月  15 19:49 VXSSimSoC.h
-rw-r--r-- 1 jy xs  6.7M 4月  15 19:49 VXSSimSoC__33.cpp
-rw-r--r-- 1 jy xs  5.2M 4月  15 19:49 VXSSimSoC__1__Slow.cpp
-rw-r--r-- 1 jy xs  4.6M 4月  15 19:49 VXSSimSoC__210__Slow.cpp
```

```
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__tl2axi4_4__DOT__deq_1__DOT__ram_echo_tl_state_source__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__tl2axi4_4__DOT__deq_1__DOT__ram_id__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__tl2axi4_4__DOT__deq_1__DOT__ram_id__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__tl2axi4_4__DOT__deq_1__DOT__ram_size__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__tl2axi4_4__DOT__deq_1__DOT__ram_size__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__tl2axi4_4__DOT__deq_1__DOT__ram_burst__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__tl2axi4_4__DOT__deq_1__DOT__ram_wen__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__tl2axi4_4__DOT__deq_1__DOT__ram_wen__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility__DOT__ram_tl_state_source__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility__DOT__ram_tl_state_source__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility__DOT__ram_tl_state_size__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility__DOT__ram_tl_state_size__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility_1__DOT__ram_tl_state_source__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility_1__DOT__ram_tl_state_source__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility_1__DOT__ram_tl_state_size__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility_1__DOT__ram_tl_state_size__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility_2__DOT__ram_tl_state_source__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility_2__DOT__ram_tl_state_source__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility_2__DOT__ram_tl_state_size__v0;
<     os>>__Vdlyvset__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility_2__DOT__ram_tl_state_size__v0;
<     os>>__Vdlyvval__XSSimSoC__DOT__soc__DOT__axi4yank_6__DOT__QueueCompatibility_3__DOT__ram_tl_state_source__v0;
```

--savable选项额外产生的代码

- 用于保存模型中所有信号的值(状态)
- 这样的代码有44万行x2(保存和恢复)

- 多线程也无法加速编译过程
  - 其它文件都编译完了, 还要等它编译3小时

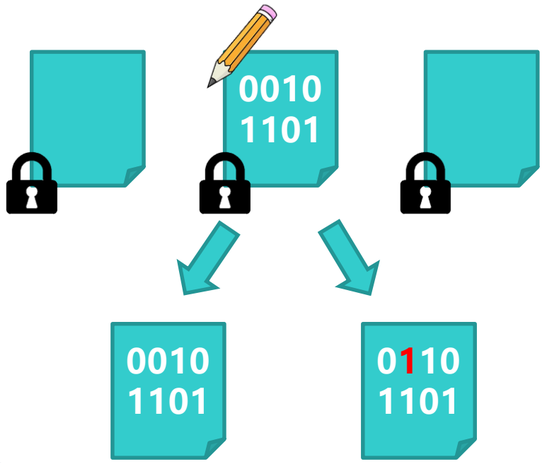
快照效率分析

- 一次fork系统调用的开销为535us

	--savable时间开销/s	LightSSS效率/倍
保存到内存		
verilator模型	0.048	**89.72**
仿真用物理内存(8GB数组)	1.223	
NEMU物理内存(8GB数组)	2.448	
DRAMSim	难以实现	
其它状态	0	
小计	3.719	**6951.40**
保存到文件		
将快照写入文件(压缩)	60.433	
总计		
总计	64.152	**119928.97**

神奇的fork

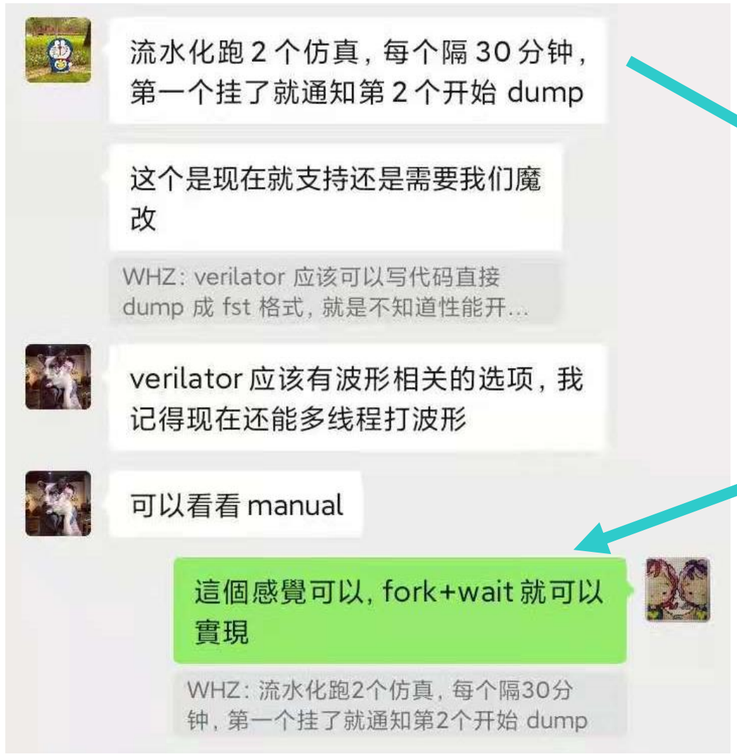
- 一次fork系统调用的开销为535us
  - 地址空间 > 8GB * 2 (仿真用物理内存+NEMU物理内存)
- 为什么fork的效率这么高?
  - 秘诀: 写时复制(Copy-On-Write)
    - 操作系统只修改进程页表属性(写保护, 引用计数器加1), 不拷贝数据
    - 后续写操作会触发缺页异常, 然后进行数据拷贝, 但写入的页面比例很小



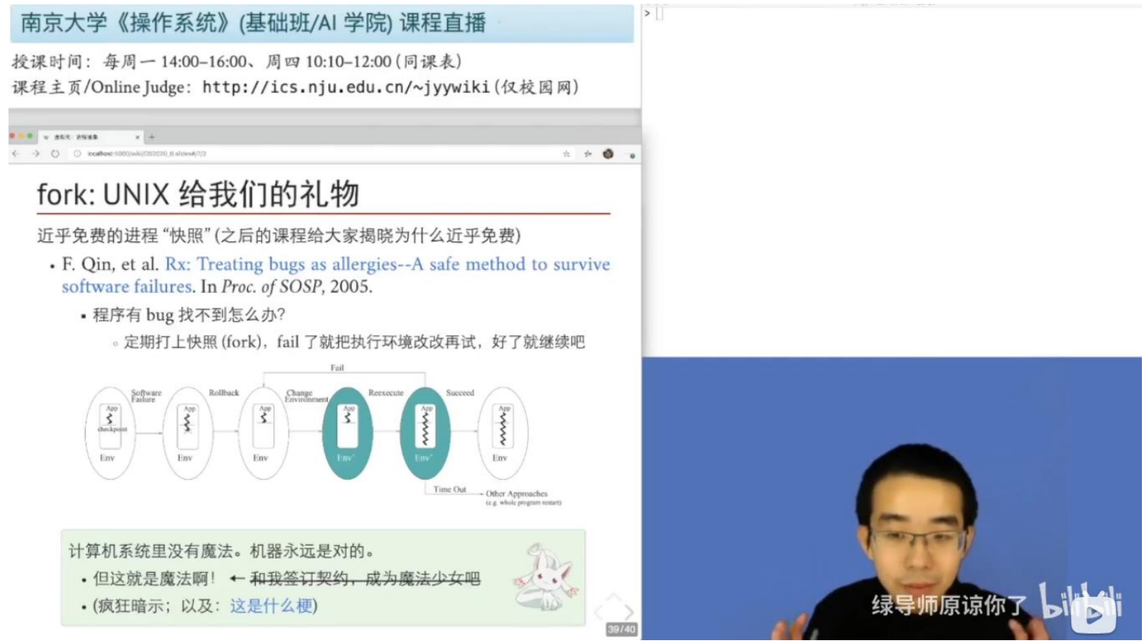
- 局限性
  - 快照只能在内存中, 不能持久存储
    - 进程状态还包含内核部分, 不能完全保存在文件
  - 外部模型不能拥有内存之外的仿真状态
    - 例如文件, 目前不支持文件状态的恢复
- 但对实际使用影响不大

花絮 - LightSSS的由来

- 一次讨论 + 一次课程



灵光一闪



疫情期间线上《操作系统》课程

- https://www.bilibili.com/video/BV1N741177F5?p=12
- 最新课程主页 http://jyywiki.cn/OS/2021

- LightSSS是操作系统原理在仿真领域的应用

小结

- LightSSS - 一种基于内存的轻量级仿真快照机制
- 核心思想 - **从进程抽象视角看仿真状态, 用fork对进程状态做快照**
- LightSSS优点
  - 可扩展性好 - 容易支持各种外部模型的快照
  - 效率高 - 和Verilator提供的--savable机制相比, 使用LightSSS
    - Verilator编译效率达到**3.80倍**
    - g++编译效率达到**2.83倍**
    - 10轮coremark运行效率达到**1.17倍**
    - 单次快照效率达到**6951.40倍**