**Gem5全系统跑simpoint**

目前网上找到的**simpoint案例都是使用se系统调用模式来跑，除了香山的 gem5模拟器(XS-GEM5)是用全系统模式做simpoint加速**

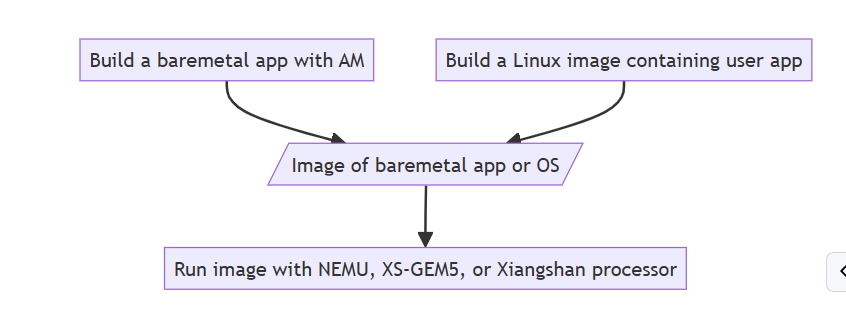
XS-GEM5：[OpenXiangShan/GEM5 at backport (github.com)](https://github.com/OpenXiangShan/GEM5/tree/backport)

[生成香山全系统负载和checkpoint的视频教程 | ShineZ's Homepage (shinezyy.github.io)](https://shinezyy.github.io/ArchShineZ/post/the-cost-of-fs/)

[SimPoint - XiangShan 官方文档](https://docs.xiangshan.cc/zh-cn/latest/tools/simpoint/)

### Run without checkpoint

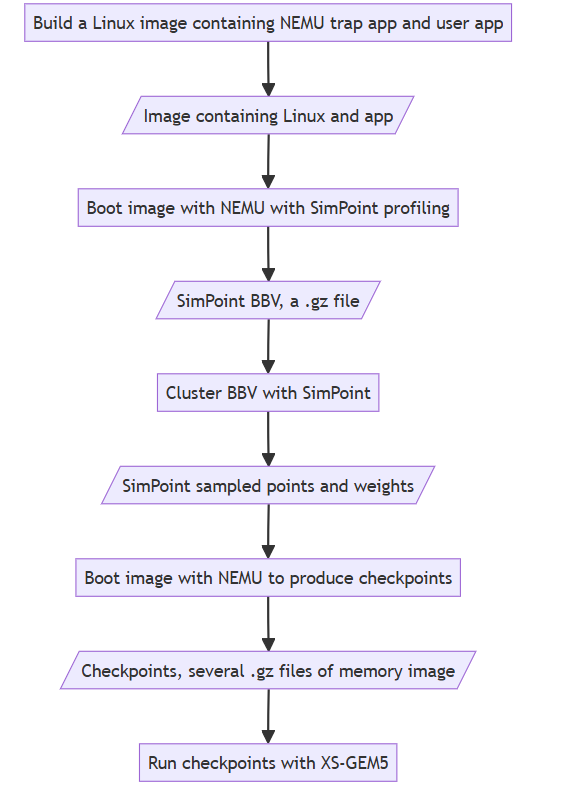
运行工作负载的典型流程与 NEMU、 XS-GEM5和香山处理器相似。它们都只支持全系统模拟。为了准备全系统模拟的工作负载，用户需要构建一个空白应用程序或在操作系统中运行用户程序。



### Run with checkpoints (使用simpoint)

香山使用**NEMU 作为checkpoints生成器，只有重载部分使用xs-gem5**，

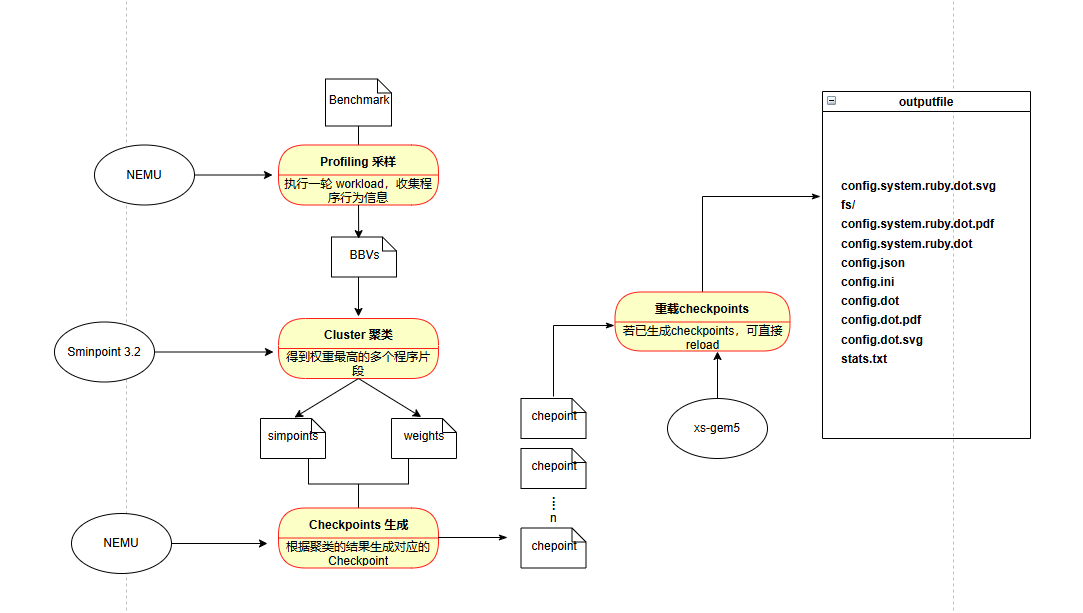
生成和运行checkpoint的流程如下所示



总结上图，使用xs-gem5，NEMU在**全系统仿真做simpoint**：

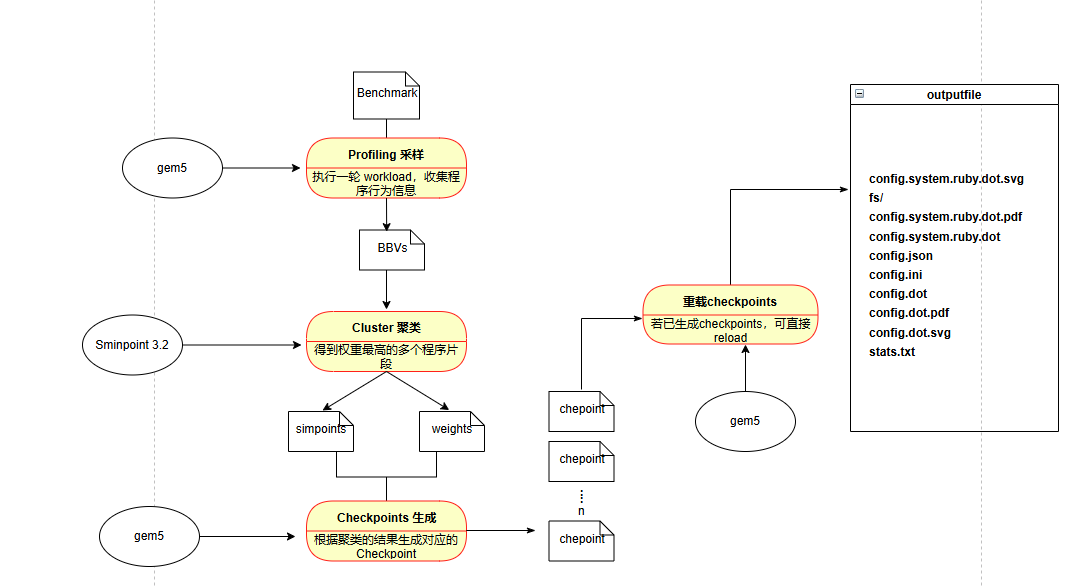
* 使用NEMU进行Profile采样和生成checkpoint
* 构建 RISC-V Linux 和 bootloader
* 编译 SPECCPU 2006
* 将 SPECCPU 2006 和 Linux 打包生成镜像
* 用 NEMU 模拟器运行镜像，进行 SimPoint profiling 和生成 checkpoint
* 使用Simpoint3.2进行聚类
* 使用xs-gem5进行重载

全系统simpoint流程图：



使用gem5的se模式做simpoint

se做simpoint流程图：



**香山全系统simpoint步骤：**

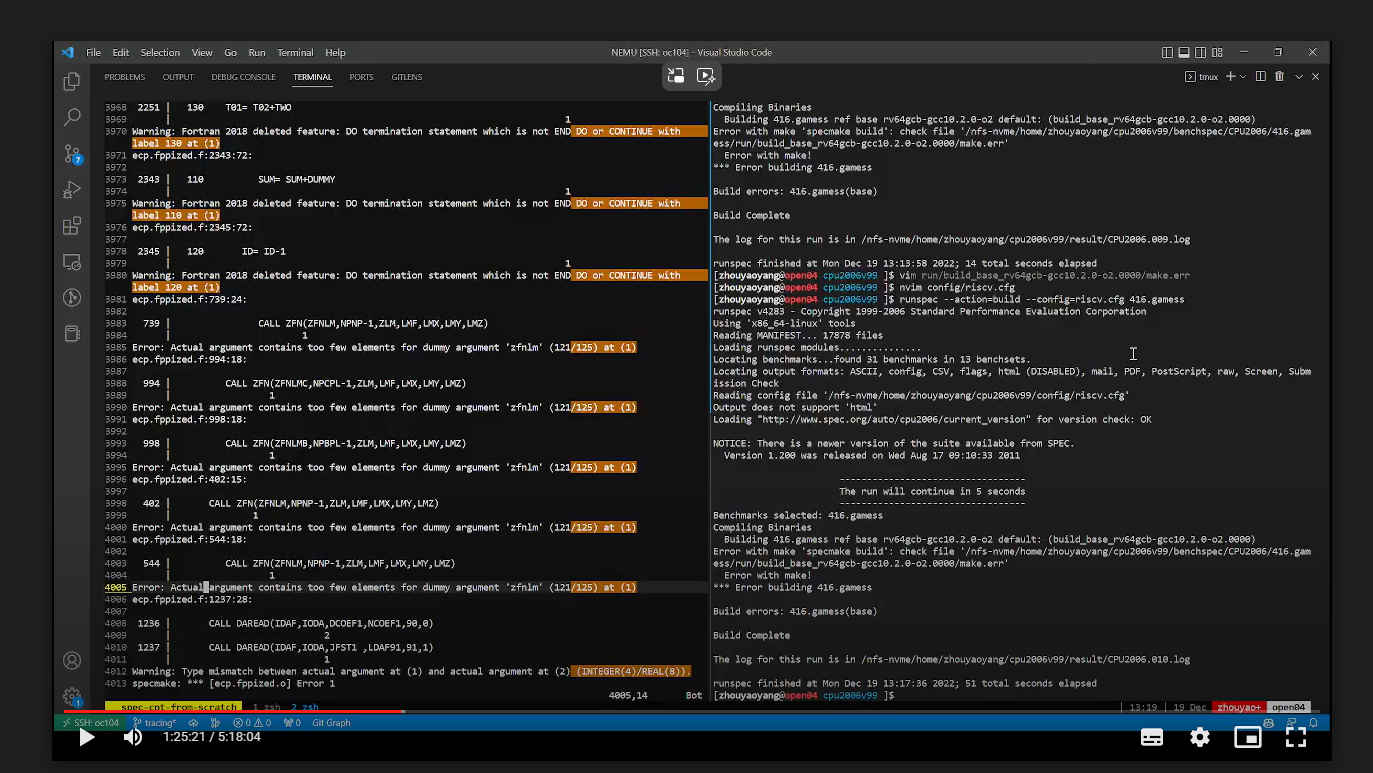
生成 SimPoint 检查点的过程包括3个单独的步骤：

1. 采样(使用NEMU)
2. 聚类(使用Simpoint3.2)
3. 生成checkpoints(使用NEMU)

步骤1：

下面这个5个多小时的视频介绍了如何[构建 SPECCPU，将其放入 Linux 中，然后在 NEMU 中运行以获取 BBV (即采样)](https://drive.google.com/file/d/1msr_YijlYN4rxpn71bod1LAoRWs5VtAL/view?usp=sharing)

视频地址：[2022-12-19 11-54-28.mkv - Google 云端硬盘](https://drive.google.com/file/d/1msr_YijlYN4rxpn71bod1LAoRWs5VtAL/view)



步骤2、3：

[基于步骤1的BBV进行聚类并生成检查点checkpoint](https://zhuanlan.zhihu.com/p/604396330)

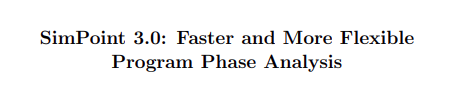
文档地址：[Make XiangShan Checkpoints (Cluster and Checkpointing part) - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/604396330)

**文献阅读：**

以下3篇为传统Simpoint原理相关论文(理解的不透彻，只能大概看懂)：

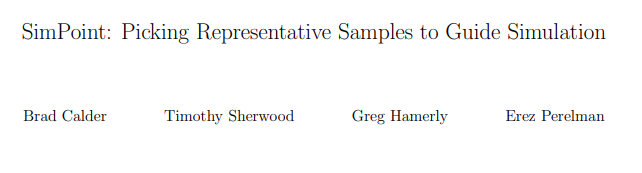
**论文1：**

[](simpoint/simpoint 参数介绍.pdf)



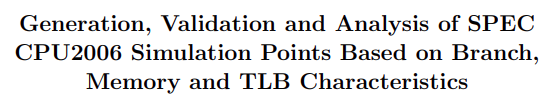
**论文2：**



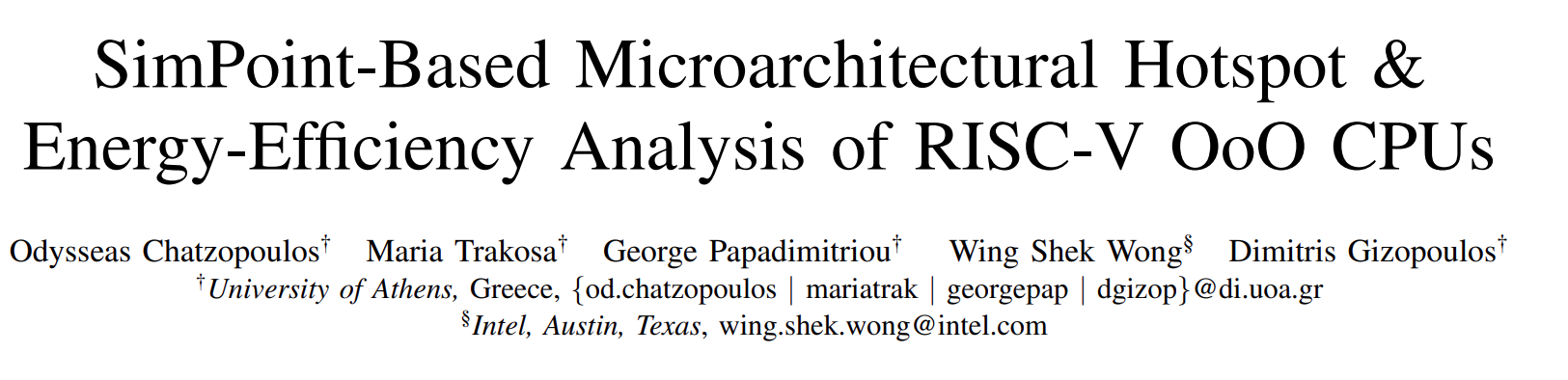


**论文3：**





**论文4：**



论文内容核心是：**以simpoint作为辅助**，**探究微架构对功耗的影响**，在不同的工作负载集上引入和应用 SimPoint 方法，使得能够评估比以前的研究平均**大2个数量级的工作量。**

论文提出使用几个软件模型和性能模拟器结合来估计这些设计的功耗，但它们**准确性有限，误差为21%**，或是**仿真慢，**故使用simpoints。

写论文的话可以**从simpoint作为辅助，提高准确性和仿真加速方面入手**？

使用simpoint能够快速 探索不同微处理器配置的设计空间，为我们提供性能、面积和能源效率等指标，在这方面写论文？

**论文内容：**

微架构在决定处理器或计算机系统的功耗方面 起着至关重要的作用。

在不同的工作负载集上引入和应用 SimPoint 方法， 我们揭示了 BOOM 的微架构和能量效率之间的关系。

提供一个全面的了解各种微架构配置如何影响性能和功耗。通过这项调查，我们试图揭示微架构，能源消耗和性能之间的关系。

Chipyard 是一个广泛使用的开源框架，该框架使得能够使用**敏捷开发方法设计和评估全系统硬件**。

我们扩展了 Chipyard 的检查点基础设施，以支持 SimPoint 方法。

这使得我们能够评估比以前的研究平均大2个数量级的工作量。

Chipyard 中引入的 SimPoint 方法的有效使用，所提出的实验流程可用于评估任意**大工作负载**的任何 CPU 设计——在我们的例子中，模拟时间减少了 45 倍。

通过使用 SimPoints，我们在略多于 2 天的时间内完成了我们的实验，与不考虑跟踪文件大小和处理时间限制的情况下所需的 3 个月以上的执行时间相比，实现了45倍加速。

Simpoint支持比以往任何时候都**大得多的工作负载的性能和功耗分析**，使整个社区受益。

SimPoint 显著地**减少了详细模拟所需的计算资源**，同时保持了高度的**准确性**。这使得它成为优化软件和硬件系统、便于识别关键瓶颈以及最终提高复杂计算环境的总体效率的宝贵工具。

**论文5：**



**论文内容：**

对**传统simpoint的优化：**

1、仿真加速(提高9.56倍)

2、提高精度(5.80%到3.89% )

与传统的**50M** simpoint对比，开发了一个开源框架，**结合LiveCache 和 detail-warm的热身**，使我们能够使用更小(**2M**) 的 Simpoint 尺寸而**不损失准确性，做到10倍加速**。

使用只有 200 万条指令(2M)的 Simpoints。与具有数亿条或更多指令的常规

simpoint 相比，模拟时间可以从**几小时大大减少到 几分钟**，而**不会丧失准确性**。

典型的 Simpoint 规模仍然是数千万到数亿条指令。够使用更小的 Simpoint 尺寸(200 万条指令)而不损失准确性。我们的评估结果表明，平均模拟 时间可以**加快 9.56 倍以上**的 50m规模和大部分的工作量 模拟可以在几十分钟内完成而不是小时。

使用更小的 Simpoint 大小Simpoint 执行之前是 **livecache 热身，然后是 detail 热身**，然后开始收集性能数据。

LiveCache 和 detail-warm 使我们能够**更新机器状态，保持模拟的准确性**

开发了一个开源框架，使我们能够使用更小(2M) 的 Simpoint 尺寸而不损失准确性。

使用 SPEC 2006 CPU 基准套件评估 2m 的 Simpoint 尺寸。与 50m 的 Simpoint 大小相比，平均模拟时间提高了 9 倍以上。

LiveCache 和 detail-warm 的结合使得 Simpoint 的

尺寸更小、更快、更准确。

200 万个 Simpoints 的平均误差(3.89%)低于流行的 5000 万个 Simpoints

(5.80%)。

**发论文点：**

个人感觉**以simpoint为主发论文的难度比较大**？ 目前想法是用**simpoint作为辅助**，simpoint仿真加速为辅，提高数量级，测试不同的异构带来的功耗、性能之间的关系

simpoint的功能**主要是仿真加速**，目前simpoint可以改进和优化的点：

1、减少误差率(但simpoint本身的准确率就挺高，再提高精度的话可能难度比较大且效果不明显，第五篇论文精度5%提高到3%)

1. 仿真加速再优化（第五篇论文中，改进的simpoint与传统的simpoint相比提高了10倍仿真速度）
   1. 正常仿真(不使用simpoint)数月量级，传统simpoint数小时量级，第五篇论文优化到数分钟量级。
   2. 加速已经**从数月提升到数分钟**了，感觉没有啥优化空间了，再优化感觉难度大

可以发论文、写论文的点：

1、结合simpoint作为辅助，对不同架构的仿真加速，测量不同架构对功耗、性能之间的关系。(感觉以simpoint可以改进优化的点发论文难度较大)

2、给定体系结构的有效性如何随其缓存大小等变化。研究人员通过simpoint技术可以减少评估架构修改的影响所需的机器月数

3、使用模拟点进行分支预测器、缓存和TLB研究。最终目标是找到最佳的分支预测器、缓存和 TLB 配置，它们在大多数基准测试中提供最佳性能。