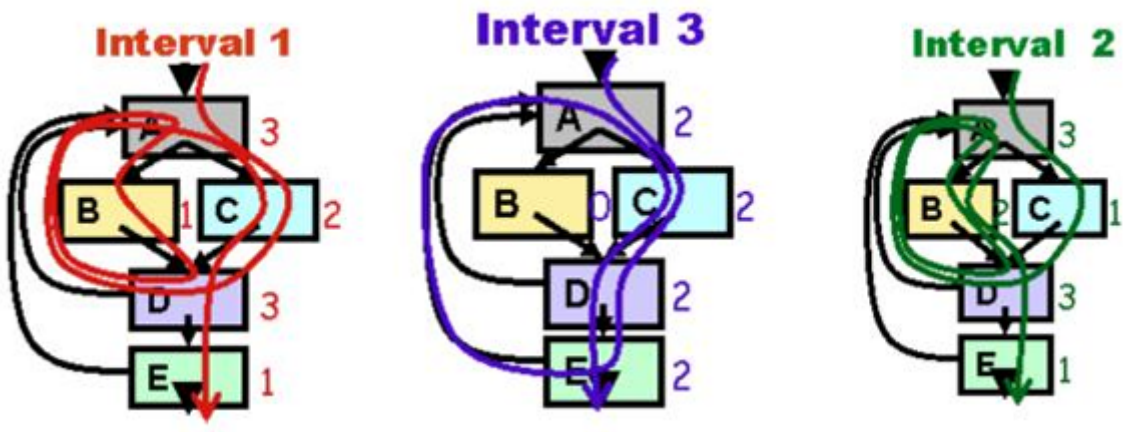
1 SimPoint基础知识

1.1 背景

SimPoint是基于离线聚类分析的一项技术，其动机是出于执行高效、准确的程序分析和架构仿真的需要

SimPoint基于这样一个观察：**程序在执行过程中往往会重复相似的行为模式。因此，可以通过识别这些重复的模式，选择代表性的程序片段进行详细模拟，而不是模拟整个程序执行过程。**

如果能统计出那几个主要的操作以及它们所占的比例，是不是只对那几个操作做仿真再加权就可以了呢？



1.2 基本块向量（BBV）

基本块

基本块：从头到尾执行的一段代码，只有一个出口和一个入口。SimPoint使用基本块的执行频率作为指标，比较应用程序执行的不同部分。因为程序在给定时间的行为与它在该间隔（interval）内执行的代码直接相关，而基本块的分布为我们提供了这些信息。

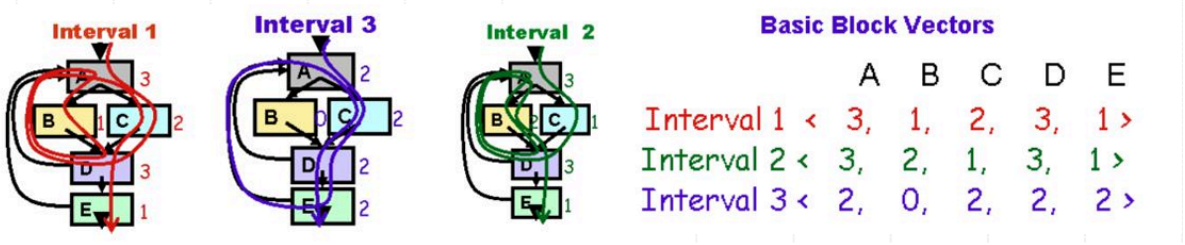
> 问题：基本块的类型是如何区分的？每个基本块都有一个唯一编号

程序在任意的时间间隔内将执行每个基本块一定次数，基于此可以得到每个执行间隔内的基本块向量，并告诉我们程序在哪个部位花费了时间。多个执行间隔会有多个基本块向量，通过对比基本块向量可以得到这些执行间隔之间的相似程度，如果两个基本块向量相似，那么这两个间隔在相同代码应该花费大致相同的时间，并且性能应该相似。

基本块向量

基本块向量（BBV）：一维数组（也可以视为多维向量），其中程序的每个静态基本块都对应数组中的一个元素。以间隔（interval）为单位收集基本块向量。在每个间隔结束时，记录在本间隔内进入（执行）每个基本块的次数。**数组中的每个元素是指在执行间隔内进入（执行）相应的基本块的次数乘以该基本块的指令数。**通过乘以每个 basic block 中的指令数量，可以确保我们对 instructions 的权重相同，无论它们位于大 basic block 还是 small basic block中。

> BBV：元素个数等于基本块个数，每个元素位置对应相同的基本块。



我们关注的不是某个时间间隔内基本块的实际执行次数，而是**各个基本块之间执行时间的比例关系**。因此，对BBV进行**归一化处理**，具体方法是将向量中的每个元素除以该向量所有元素的总和。

> 一个interval对应多个基本块和一个BBV。BBV归一化处理后记录该interval中各个基本块执行时间比例（次数 * 指令数 / 总指令数）。

基本块差异对比

得到归一化后的BBV如何对比计算，输出结果表示BBV之间的相似程度？

欧几里得距离：将每个向量视为D维空间中的单个点，计算两点之间的距离

$$EuclideanDist(a,b) = \sqrt{\sum_{i=1}^{D}(a_i - b_i)^2}$$

曼哈顿距离：计算两个D维向量各个维度之间的差异。

$$ManhattanDist(a,b) = \sum_{i=1}^{D}|a_i - b_i|$$

归一化处理后，两个间隔对应的BBV曼哈顿距离介于0—2之间。

使用曼哈顿表示差异，曼哈顿在表示高维数据中的差异更准确。

在聚类算法使用欧几里得，在低纬度数据中更准确。

1.3 聚类

许多执行间隔彼此相似，一种有效的表示方法是将具有相似行为的间隔分组在一起。这个问题类似于聚类问题。

聚类的目标是将一组点分成组，使得每个组内的点彼此相似（根据某种度量，通常是距离）。

使用随机线性投影降维，然后是k-means聚类算法，为了选择k的值，使用贝叶斯信息准则（BIC）分数。

以下步骤是聚类算法步骤示例：

1. 分析每个程序执行的基本块，为每1亿条指令（interval）的执行生成基本块向量。
2. 使用随机线性投影将BBV数据的维度降至15维。
3. 对降维后的数据尝试k-means聚类算法，k值从1到10。k-means的每次运行都会产生一个聚类，即**将intervals分成k个不同的聚类。**
4. 对每个聚类（k = 1...10），使用BIC计算其拟合度。选择得分至少达到最佳得分90%的聚类中k最小的那个。

降维

对于聚类问题，必须解决维度问题。所有聚类算法都受到所谓的"维度诅咒"的影响，这指的是当维度数量增加时，聚类数据变得极其困难：

- 对于基本块向量，维度的数量是程序中执行的基本块的数量，在我们的实验数据中范围从2,756到102,038，对于非常大的程序，这个数字可能增长到数百万。
- 另一个实际问题是聚类算法的运行时间取决于数据的维度，如果维度增长太大，就会变得很慢。

由于BBV可能有很高的维度，SimPoint使用随机投影将其降到较低的维度（通常是15维），以减少计算复杂度。

k-means

SimPoint使用k-means算法对降维后的BBV进行聚类。每个聚类代表程序执行的一个阶段或行为模式。**输出是一组最终的聚类中心和每个点所属的聚类的映射。**由于我们已将数据投影到15维，我们可以快速为k-means生成聚类。

k-means算法还有个缺点就是需要提前确定k的值，k表示最后需要聚合成几个类，simpoint中用的方法是按某种规则（非随机，实验多次选区的都是同一组k值，可能也是有某种算法）选择多个k值（介于1—MAXk之间，选取的个数等于进行k-Means算法的次数），分别计算得到聚类的结果后使用 BIC算法进行打分，打分最高的k值作为最终的选择并作为最优的聚类数量

贝叶斯信息准则

SimPoint使用贝叶斯信息准则（BIC）来自动选择最佳的聚类数量（k值）。BIC分数越高，聚类被评分为对数据的"拟合"越好。

1.4 采样点选择

对于最后“拟合”最好的每个聚类，SimPoint选择一个代表性的采样点（通常是离聚类中心最近的点）。

只有这些采样点才会用于详细的架构模拟。每个采样点被分配一个权重，基于其所代表的聚类在整个程序执行中所占的比例。

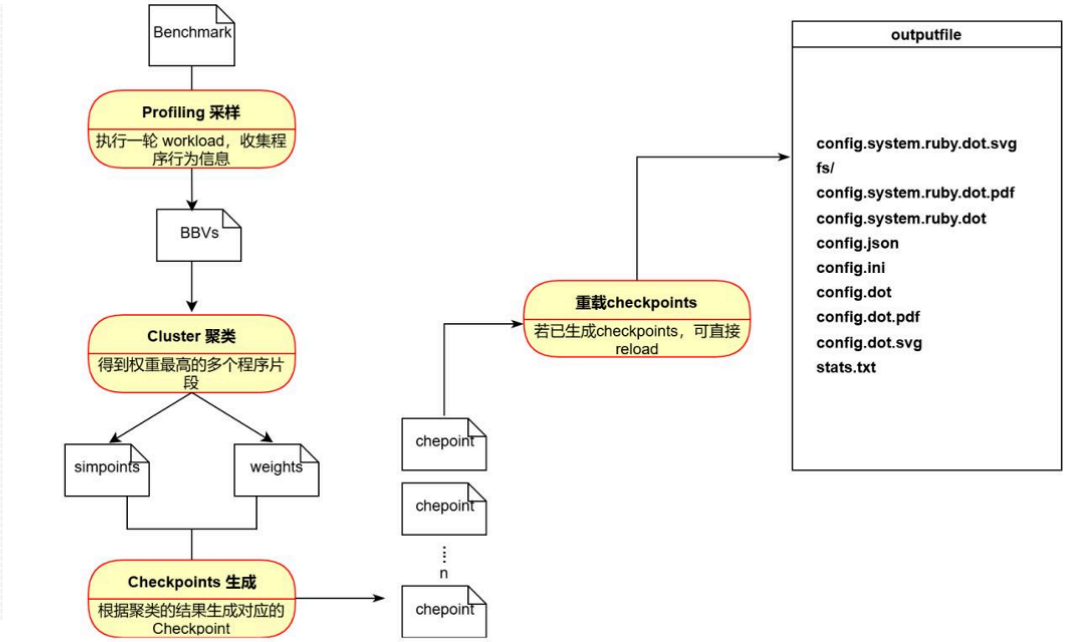
> 采样点：实际就是BBV，对应一个具体的interval

1.5 模拟

只对选定的采样点进行详细的架构模拟，而不是模拟整个程序。

使用采样点的模拟结果和它们的权重，重建整个程序执行的性能估计。

2 SimPoint在Gem5中的应用



2.1 采样

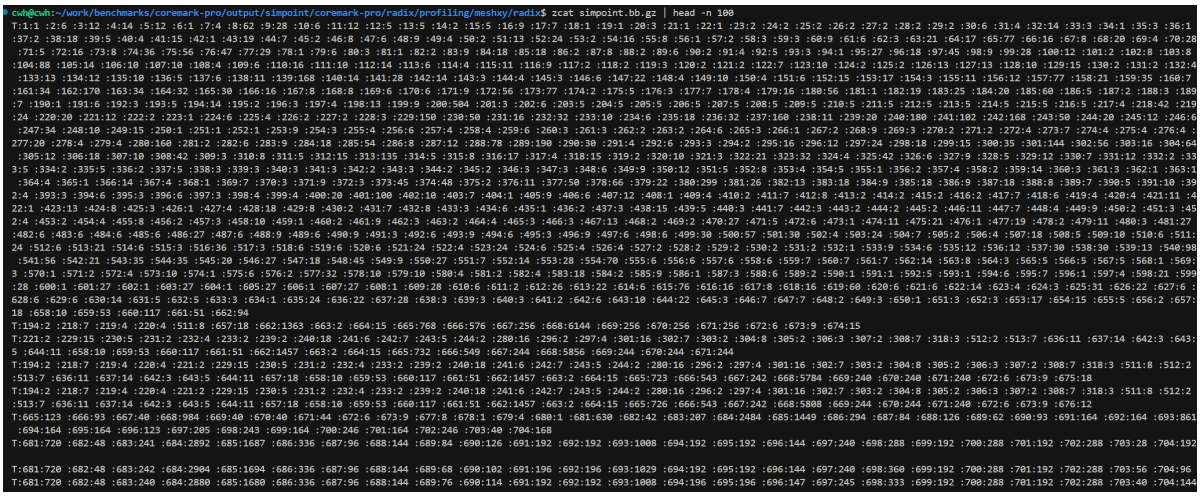
运行一遍应用程序，生成BBVs

- **interval大小经验法则：一个常见的做法是选择一个使得程序总共有 1000-3000 个采样点的 interval 大小。**
- **快速模拟，收集BBV信息，注意CPU type：--cpu-type=NonCachingSimpleCPU**
- **输出：simpoint.bb.gz文件**

```
# profiling
build/RISCV/gem5.opt \
    -d path/to/output/dir \
    configs/deprecated/example/se.py \
    # 为了快速生成BBVs，这里的CPU type使用NonCachingSimpleCPU
    --cpu-type=NonCachingSimpleCPU \
    -c path/to/benchmark.exe \
    --simpoint-profile --simpoint-interval 10000
```

simpoint.bb.gz文件格式：T[:BB_id:count]

- 每个T对应一个interval，BB_id为basic block的编号；
- count为对应的basic block的值，值的计算方式为在此interval中对应的block出现的次数和block中指令的个数的乘积。



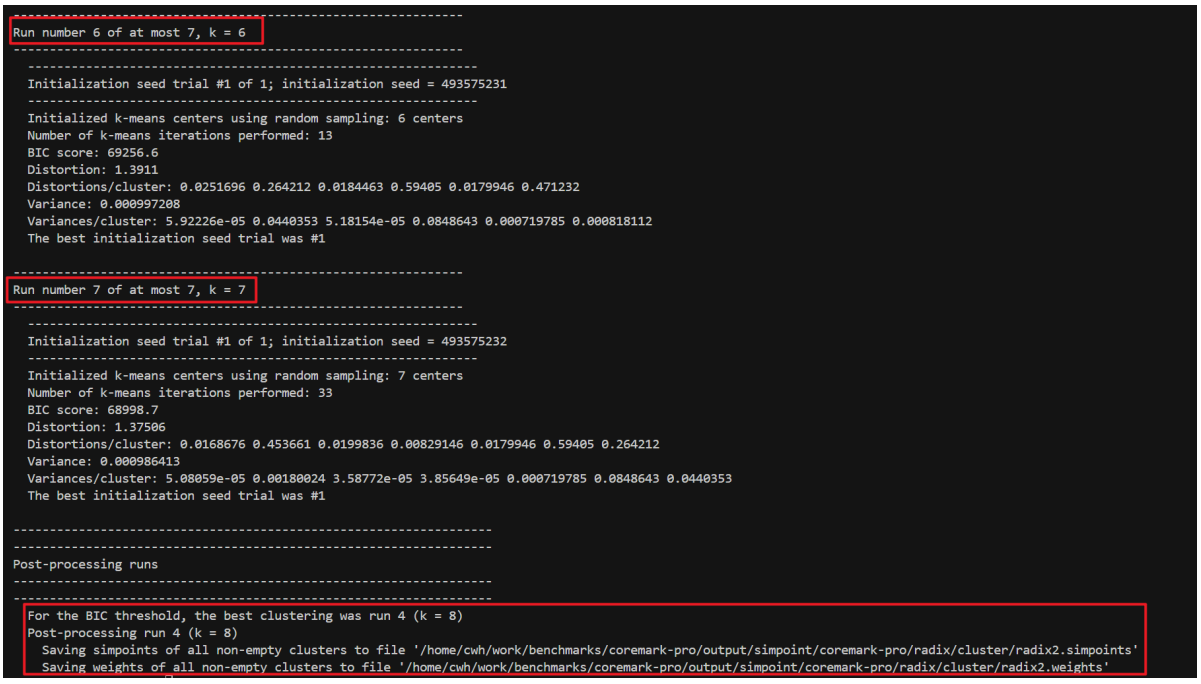
2.2 聚类

生成最佳k个聚类。

- **使用SimPoint工具分析Gem5生成的BBV文件。**
- **使用k-means算法何BIC算法确定最佳的聚类数量和采样点。**
- **maxK = 30是经验值**
- **输出：xx.simpoints 与 xx.weights 两个文件**

```
# cluster
simpoint.3.2/bin/simpoint \
    -maxK 30 \
    -numInitSeeds 1 \
    -loadFVFile path/to/simpoint.bb.gz \
    -inputVectorsGzipped \
    -saveSimpoints path/to/save/radix2.simpoints \
    -saveSimpointWeights path/to/save/radix2.weights
```

k-means算法还有个缺点就是需要提前确定k的值，k表示最后需要聚合成几个类，simpoint中用的方法是按某种规则（非随机，实验多次选区的都是同一组k值，可能也是有某种算法）选择多个k值（介于1—maxK之间，选取的个数等于进行k-Means算法的次数，默认是7次，即选7个k值），分别计算得到聚类的结果后使用 BIC算法进行打分，打分最高的k值作为最终的选择并作为最优的聚类数量



xx.simpoints文件内容：

每行包含两个数字，格式为 "采样点 聚类ID"，每个聚类都由一个特定的 interval 来代表，这个 interval 被认为最能代表该聚类的行为特征。性能模拟时，只需要模拟这些被选中的 intervals，每个被选中的 interval 代表了程序执行中的一个不同阶段或行为模式。

```
465 0
532 1
741 2
1398 3
1 4
475 5
1088 6
1372 7

```

- 例如，"465 0" 表示第465个采样点属于第0个聚类，也意味着第0个聚类的中心点是第465次采样

> 采样点 == BBV
>
> 第465次采样：第465个interval，地址是465 * interval大小

- 这个文件列出了8个代表性的采样点，每个对应一个聚类

xx.weights 文件内容：

```
0.249108 0
0.388294 1
0.0963597 2
0.00499643 3
0.00499643 4
0.234832 5
0.0028551 6
0.0185582 7
```

- 每行包含两个数字，格式为 "权重 聚类ID"
- 例如，"0.0378378 0" 表示第0个聚类的权重为0.0378378

- 较高的权重表示该聚类代表了程序执行中更大的部分

2.3 生成CheckoutPoints

拿到simpoints和weights文件后就可以知道各段interval的起始位置和权重了。可以使用GEM5快速forward到对应的interval，然后切换到GEM5的detailCPU模式开始运行。

但为了在后续仿真时跳过forward过程，加速仿真流程，GEM5提供了checkpoint功能：第一次运行时，快速forward到interval的起始点 后使用checkpoint功能把快照保存起来，以后运行时直接从快照恢复速度快很多。

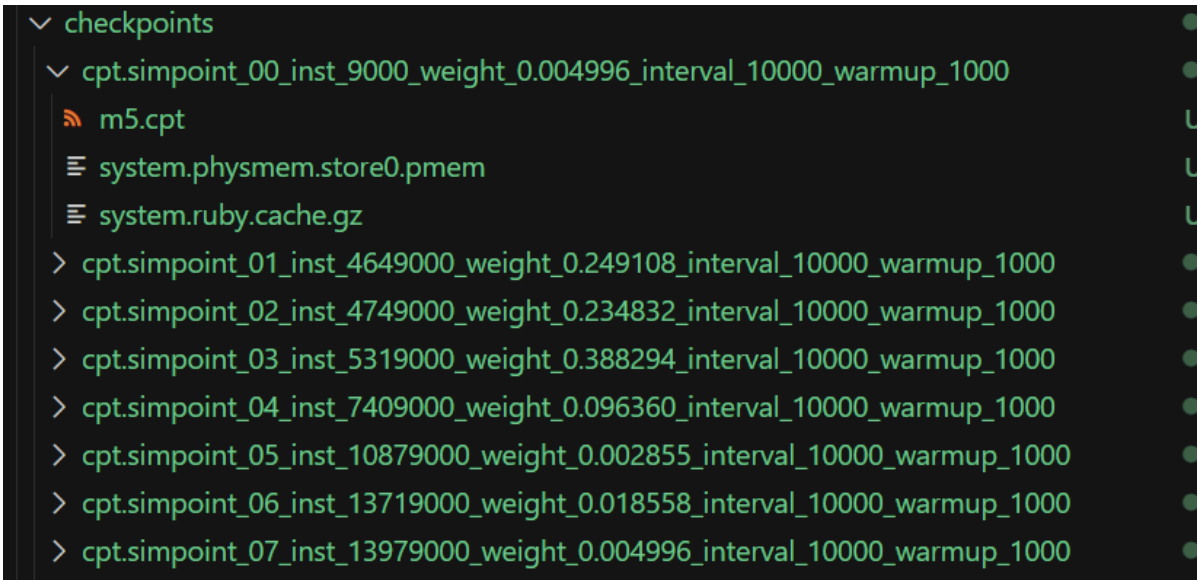
> 省去Gem5 Forward到对应interval的过程，所以这里使用的CPU type是：--cpu-type AtomicSimpleCPU

- 使用SimPoint生成的采样点信息。
- 设置Gem5以在指定的指令数后生成检查点。
- **CPU type是：--cpu-type AtomicSimpleCPU**

```
# checkpoints
build/RISCV/gem5.opt \
    -d path/to/output/dir \
    configs/deprecated/example/se.py \
    --take-simpoint-checkpoint=path/to/radix2.simpoints,
path/to/radix2.weights,10000,1000 \
    -n 16 \
    # 注意这里的CPU type
    --cpu-type AtomicSimpleCPU \
    --l1d_size=64kB \
    --l1i_size=64kB \
    --num-l2caches=16 \
    --l2_size=128kB \
    --num-dirs=16 \
    --ruby \
    --network=garnet \
    --topology=Mesh_XY \
    --mesh-rows=4 \
    -c path/to/benchmark.exe
```

checkpoint类似于上下文切换时保存的上下文，也类似于微架构中遇到分支指令时，保存的**checkpoint 可以理解成是在一个时刻CPU运行的快照，会保存架构寄存器和缓存memory的状态。**

输入为前面生成的 simpoints 和 weights 文件， 输出生成了与simpoints数目相同个checkpoint文件 文件夹里面存储了给定时刻（执行到各个聚类代表interval位置时）硬件的全部信息，比如cache配置，cpu类型等。



m5.cpt即为当前cpu运行状态的快照，比如寄存器值，页表映射关系之类的
system.physmem.store0.pmem则是当时的内存的状态。

因此有了cpt和pmem，gem5就可以从任意一个checkpoint进行restore重载

2.4 重载

由于CheckoutPoint保存的快照，Gem5可以从从任意一个 checkpoint进行restore重载，下图中的4表示从第4个CheckPoint恢复。

- 重载时是详细模拟采样点对应的intervals
- **重载时的架构参数除了CPU type，其它参数应该与生成checkpoint时相同，因为checkpoint保存的是上一步的状态**

  > 所以说，如果更改了底层架构，需要重新生成一遍checkpoint

- 注意--restore-simpoint-checkpoint -r 4 对应CheckPoint的编号是 3（checkpoint文件从0开始编号）

```
# reload
build/RISCV/gem5.opt \
    -d path/to/output/dir \
    ./configs/deprecated/example/se.py \
    # 这里以下配置应该与上一步生成checkpoints的配置相同
    -n 16 \
    # 这里的CPU type使用详细模拟的type
    --cpu-type TimingSimpleCPU  \
    --l1d_size=64kB \
    --l1i_size=64kB \
    --num-l2caches=16 \
    --l2_size=128kB \
    --num-dirs=16 \
    --ruby \
    --network=garnet \
    --topology=Mesh_XY \
    --mesh-rows=4 \
    --restore-simpoint-checkpoint -r 4 \
```

```
    --checkpoint-dir path/to/checkpoints \
    -c path/to/benchmark.exe
```

2.5 合并仿真结果

合并算法

使用 SimPoint 的最后一步是合并加权仿真点，以得出程序执行的总体性能估计。

- 收集每个SimPoint模拟的性能数据。
- 使用SimPoint权重文件（.weights）来加权平均这些结果，得到整个程序的性能估计。

每个权重代表属于其阶段的总执行的比例，总体性能估计值是一组模拟点估计值的加权平均值。

例如，如果有 3 个模拟点，它们的权重为 [0.22， 0.33， 0.45]，它们的 CPI 为 （CPI1， CPI2， CPI3） 则这些点的加权平均值为： CPI = 0.22 * CPI1 + 0.33 * CPI2 + 0.45 * CPI3

加权平均计算出的 CPI 是程序完整执行的估计值，此计算方法可用于任何指标

预热问题

处理器中的 Cache、MMU、分支预测器的冷启动会影响性能评估的准确性，因此需要进行Warm-Up，对Cache、MMU、分支预测器进行数据预热。具体实现方式为提前多执行W(Warmuplength)条指令，例如：一个预期的 Checkpoint，时间节点为N，采样区间长度(cpt-interval的参数)为1，预热长度为 W。真正生成的 Checkpoint 节点为 N-W，处理器执行时，需要执行(N-W，N+I)，即 W+1条指令。收集性能数据时需要舍去(N-M,N)部分，只收集(N，N+I)部分的性能数据。

在Gem5重载检查点的m5out/stats.txt文件中会有两部分内容：

```
---------- Begin Simulation Statistics ----------

...

---------- End Simulation Statistics   ----------
```

第一部分就是预热，预热数据不考虑，只分析第二部分数据。

3 问题

SimPoint在生成检查点（BBVs）时（对应Gem5的采样阶段）是完整的执行一遍程序吗?

SimPoint在生成采样点时并不需要完整执行一遍程序。

SimPoint的优势在于它能在不完整执行程序的情况下识别出代表性的采样点。这大大减少了分析时间，同时仍能捕获程序的关键行为特征。

1. BBV 分析阶段：
    - 这是第一次"执行"程序，但不是完整或详细的执行。
    - 使用功能模拟器或二进制插桩工具。
    - 目的是收集基本块执行频率信息。

- 这个过程比真正的执行快得多，因为它忽略了许多微架构细节（在Gem5中使用--cpu-type=NonCachingSimpleCPU）。

2. 聚类分析阶段：
    - 完全离线，不涉及程序执行。
    - 使用收集到的 BBV 数据进行数学分析。
    - 识别相似的程序阶段并选择代表性采样点。
3. 检查点生成阶段：
    - 这是第一次"真正"执行程序，但只执行到选定的采样点。
    - 使用快速的功能模拟器（Gem5中使用：--cpu-type AtomicSimpleCPU）。
    - 目的是在每个采样点创建系统状态的快照（检查点）。
4. 详细模拟阶段：
    - 这是最终的详细执行阶段。
    - 从每个检查点开始，使用详细的微架构模拟器。
    - 只模拟每个采样点后的一小段指令（通常是一个完整的间隔）。

可以在社交网站上请求别人发SimPoint选好的点给他，使用别人的simpoints吗?

这肯定是不行的，因为两边的Binary很可能不一样，受到程序版本、编译器版本、优化 选项、库的版本的影响。A这边的第N个片段和B的第N个片段很可能不是同一个片段。