北京大学
PEKING UNIVERSITY

2021级科研实践

基于超前执行的硬件数据预取的实现与分析

梁书豪

目录

北京大学
PEKING UNIVERSITY

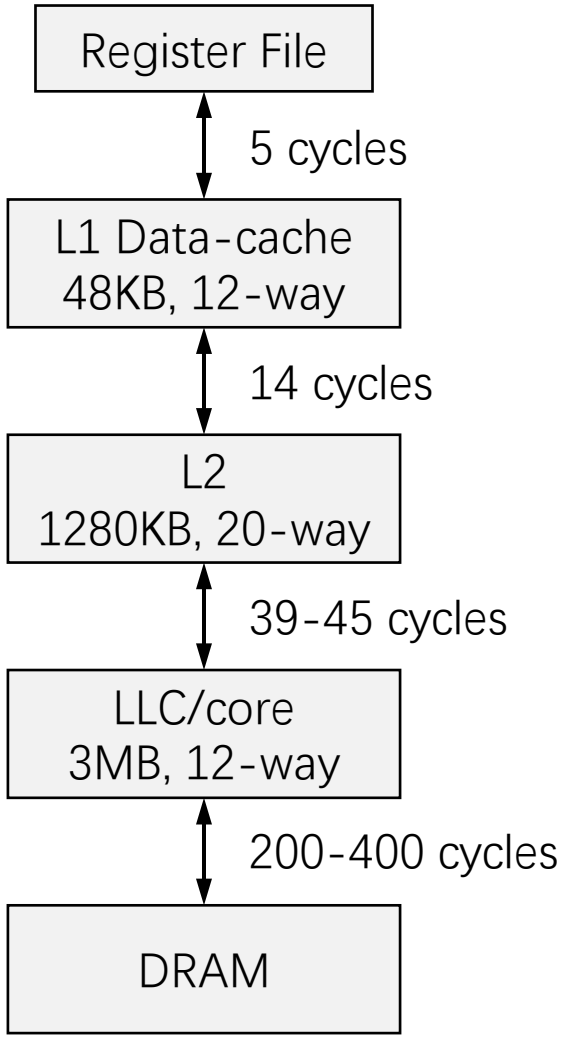
1.
研究背景

存储墙 / 全窗口停顿 / 超前执行

1. 研究背景

存储墙

- 随着内存访问延迟和处理器周期的差距进一步扩大，处理器正在遭受越来越严重的**全窗口停顿**
- 乱序处理器虽然可以乱序执行指令，但未完成指令会阻塞提交，导致资源无法释放，后续指令无法执行
- 与乱序执行的深度相关的**硬件资源**
  - ROB
  - 物理寄存器堆
  - 发射队列
  - 访存队列



Intel Tiger Lake (2020) 的存储层次结构*

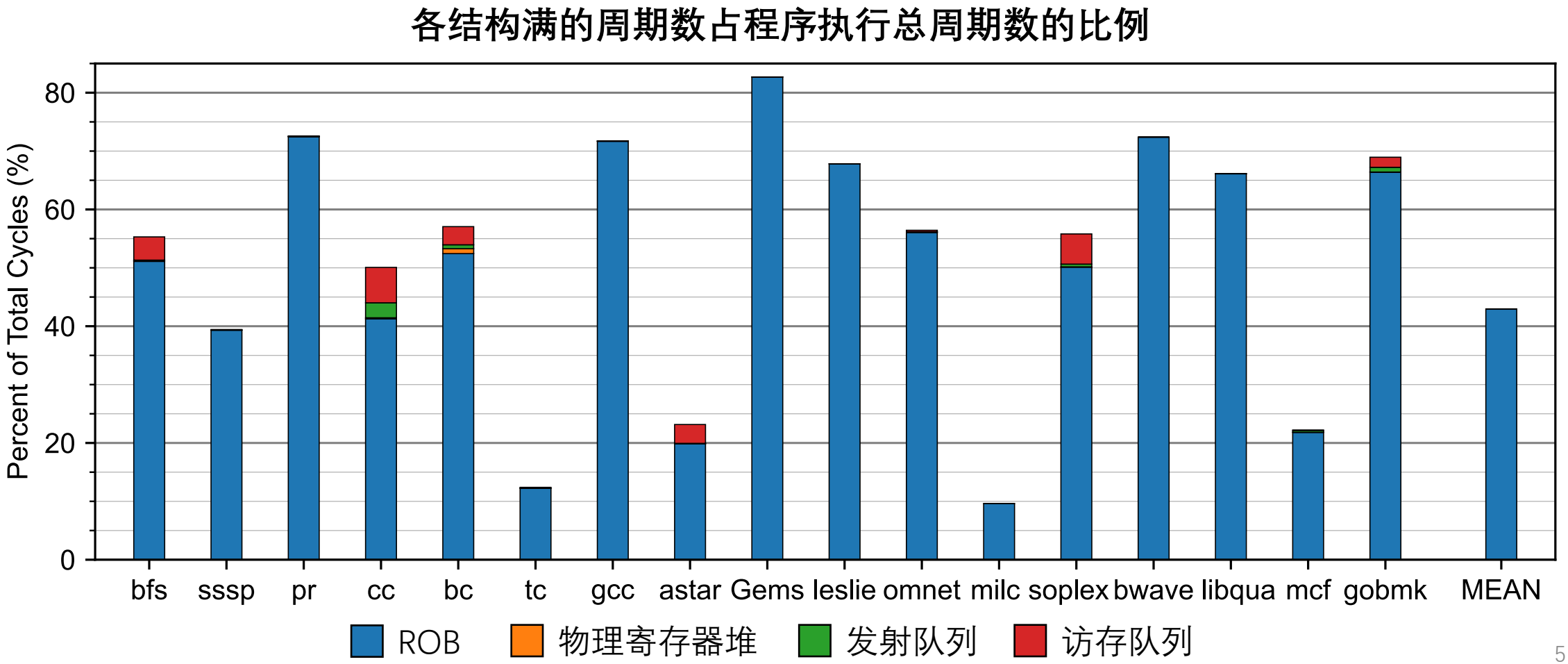
* Sudhanshu Shukla et al., *Register File Prefetching*, in ISCA, 2022

1. 研究背景

参数: gem5 O3CPU
W 4, PRF 128, ROB 128
L1 32K, L2 256K, L3 1M

全窗口停顿

- 对于一组访存密集型程序，全窗口停顿平均占43%的执行时间，其中ROB满占比最大

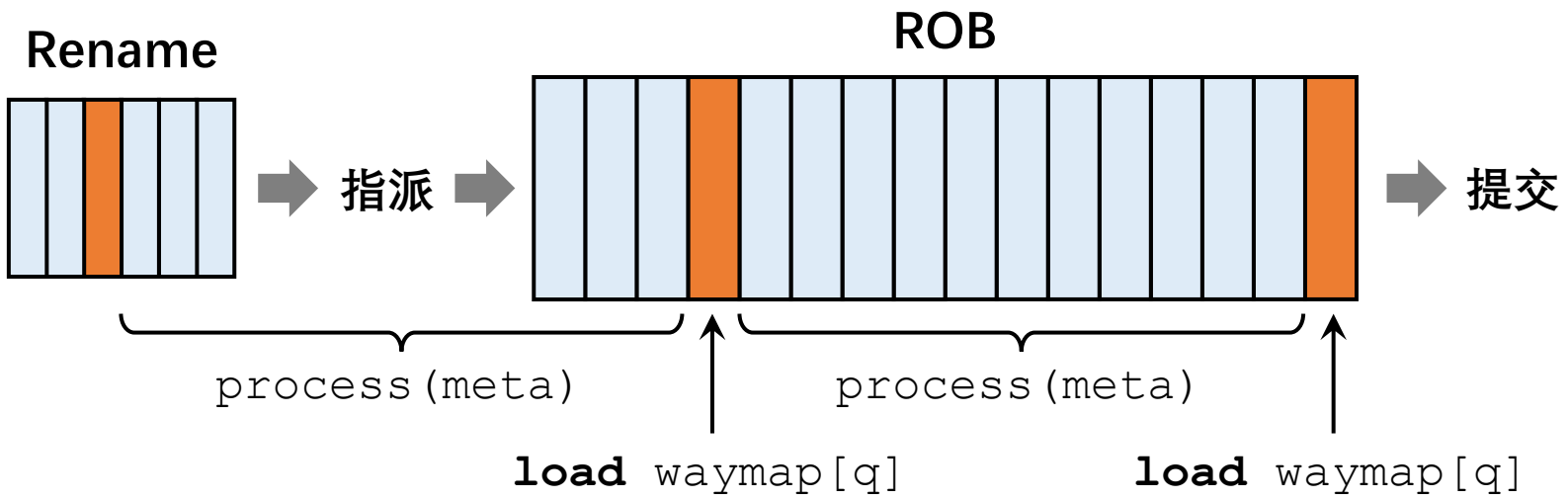


示例：全窗口停顿对预取的影响

- 考虑473.astar中一个访存密集的片段

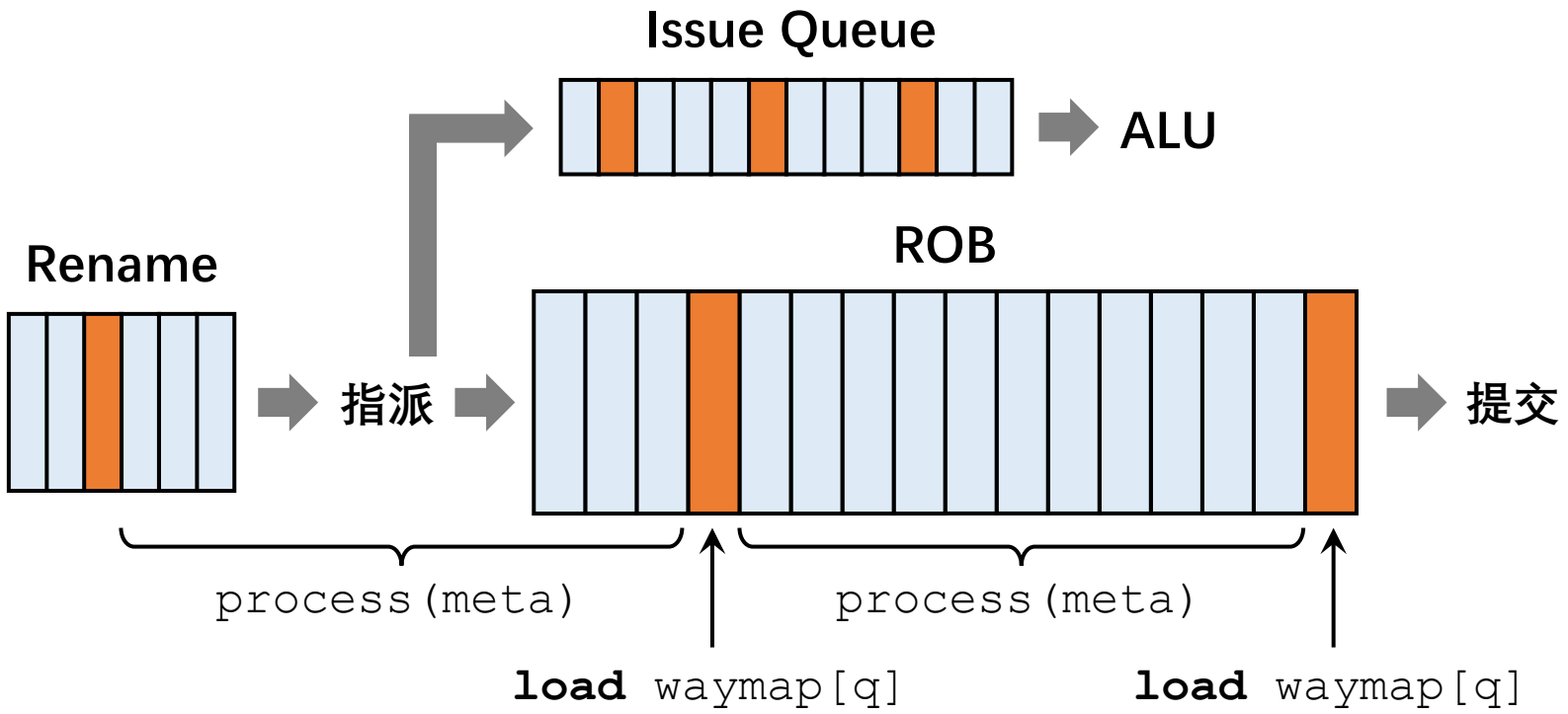
```
1 func makebound(bound1: List[int])
2   for p in bound1 do
3     for q in 8 points around p do
4       meta := waymap[q]
5       process(meta)
6     done
7   done
8 endfunc
```

- p范围很大，且无规律，导致第4行发生miss
- 第4行的miss使ROB满，阻塞新的指令指派与执行，亦无法发出新的访存



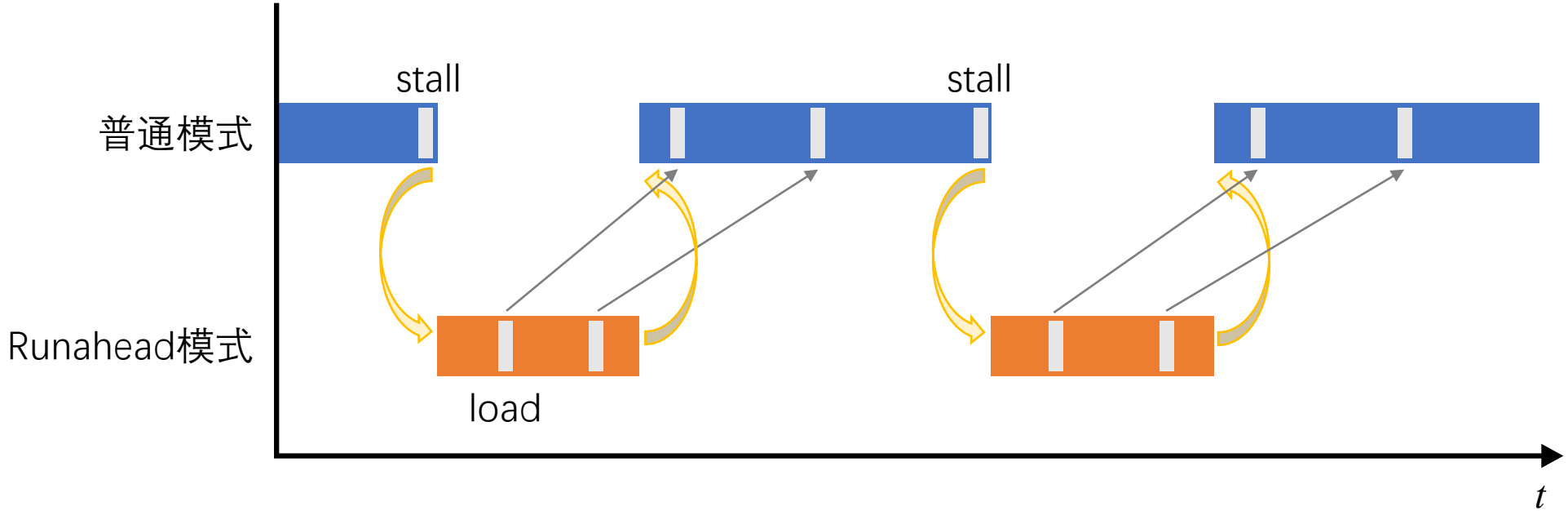
示例：全窗口停顿对预取的影响

- ROB满时**无法指派**新的指令，于是**无法执行**新的指令
- 但此时**其他组件**（物理寄存器堆、发射队列、执行单元等）**仍有空闲**
- 如果能**忽略ROB**，继续指派和执行新的指令，就可以实现预取



超前执行（Runahead Execution）

- 使用同一个硬件线程，分为**正常模式**和**Runahead模式**
- **发生ROB满时** → 保存体系结构状态，进入Runahead模式，继续指派、发射、执行指令
- **阻塞ROB的指令完成时** → 恢复体系结构状态，返回正常模式
- Runahead过程中的**load指令**自然进行预取



北京大学 PEKING UNIVERSITY

超前执行相关论文

- Pre-executing Under Miss (ICS'97)：在**按序五级**流水线上做Runahead
- Runahead Execution (HPCA'03)：首篇在**超标量乱序**处理器上做Runahead
- Techniques for RE Engines (ISCA'05)：改进Runahead性能的一些技巧
- Runahead Buffer (MICRO'15)：使用**Load Slice**代替执行所有指令
- Continuous Runahead (MICRO'16)：使用**额外的引擎**执行Runahead模式
- Precise Runahead Execution (HPCA'20)：改进了**Load Slice寻找**和**进出Runahead模式**的机制
- Vector Runahead (ISCA'21)：在Runahead模式下使用**向量指令**
- Register Flush-free RE (SBAC-PAD'21)：退出Runahead模式时**保留寄存器**的方法
- Reliability-Aware Runahead (HPCA'22)：研究Runahead过程中的**侧信道**问题

北京大学
PEKING UNIVERSITY

2.
Precise Runahead Execution

优势 / 模式切换 / Load Slice / 寄存器释放

精确超前执行（Precise Runahead Execution，PRE）

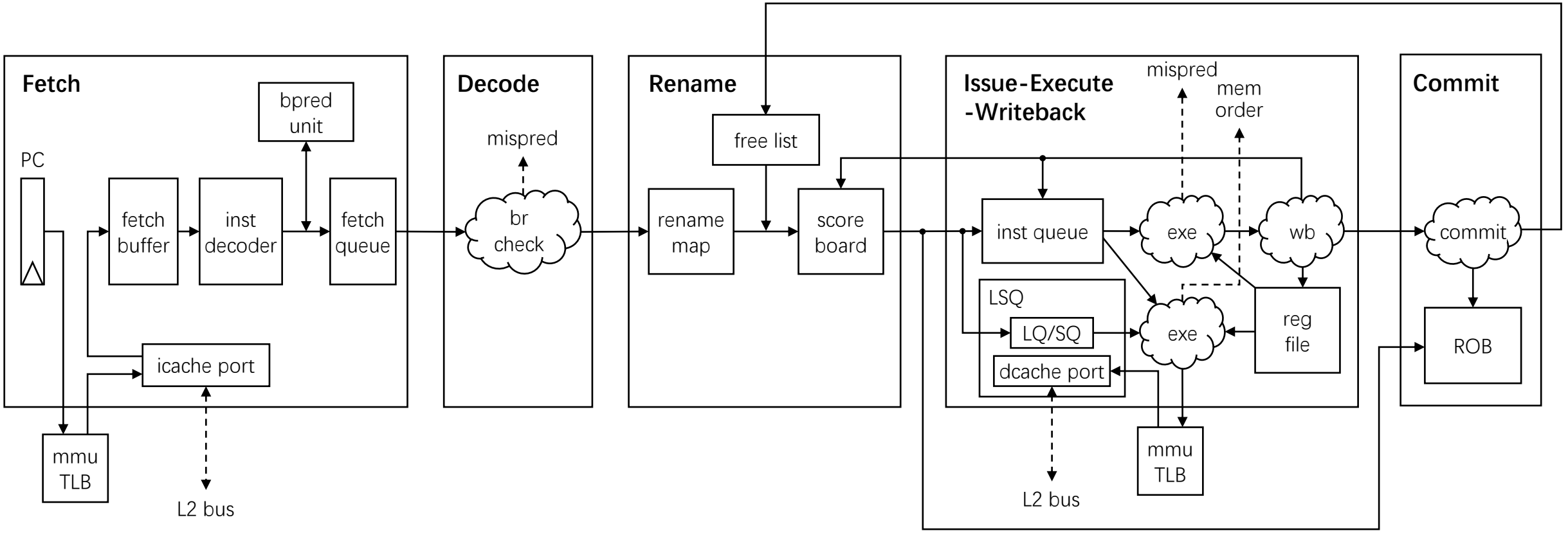
- PRE是标量处理器上最领先的超前执行方法
- 三个“**精确**”之处：不需清空流水线，不需执行所有指令，可释放寄存器
- 有很好的硬件可实现性

在gem5模拟器上实现PRE

- PRE直接在流水线上修改，比在cache上修改的BOP、SPP更复杂
- PRE还没有人在硬件上验证，我们直接做硬件有风险
- PRE的设计不成熟，在模拟器上可以更好地完善设计

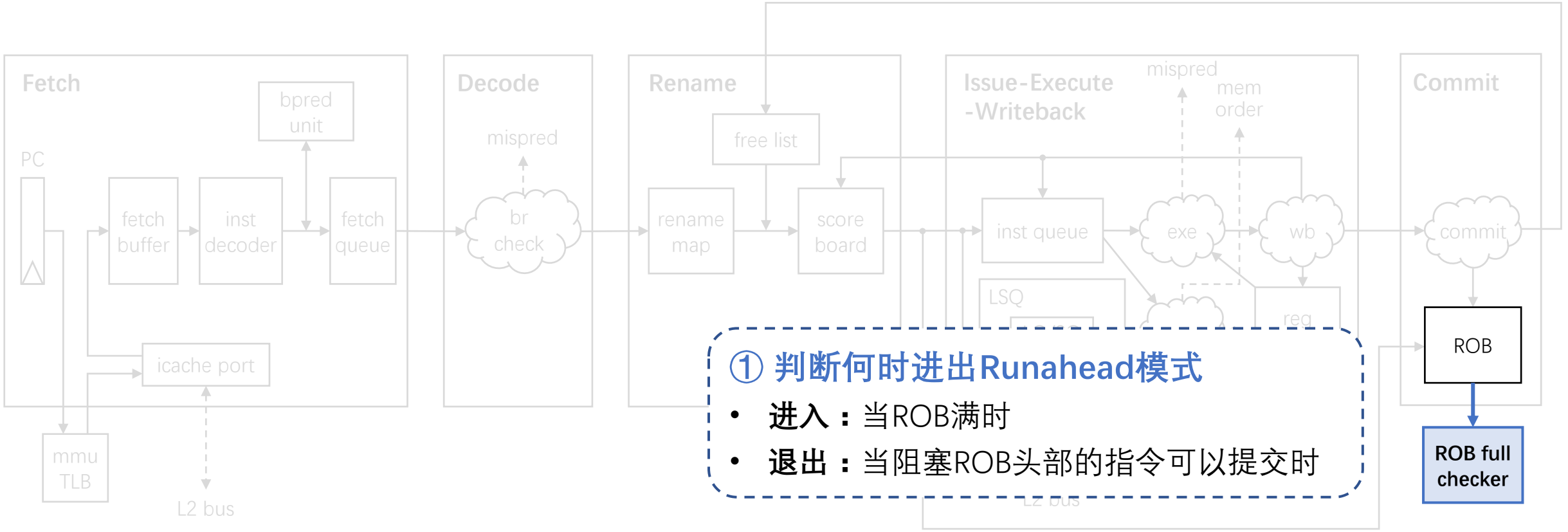
2. Precise Runahead Execution

PRE的工作原理



北京大学 PEKING UNIVERSITY

PRE的工作原理

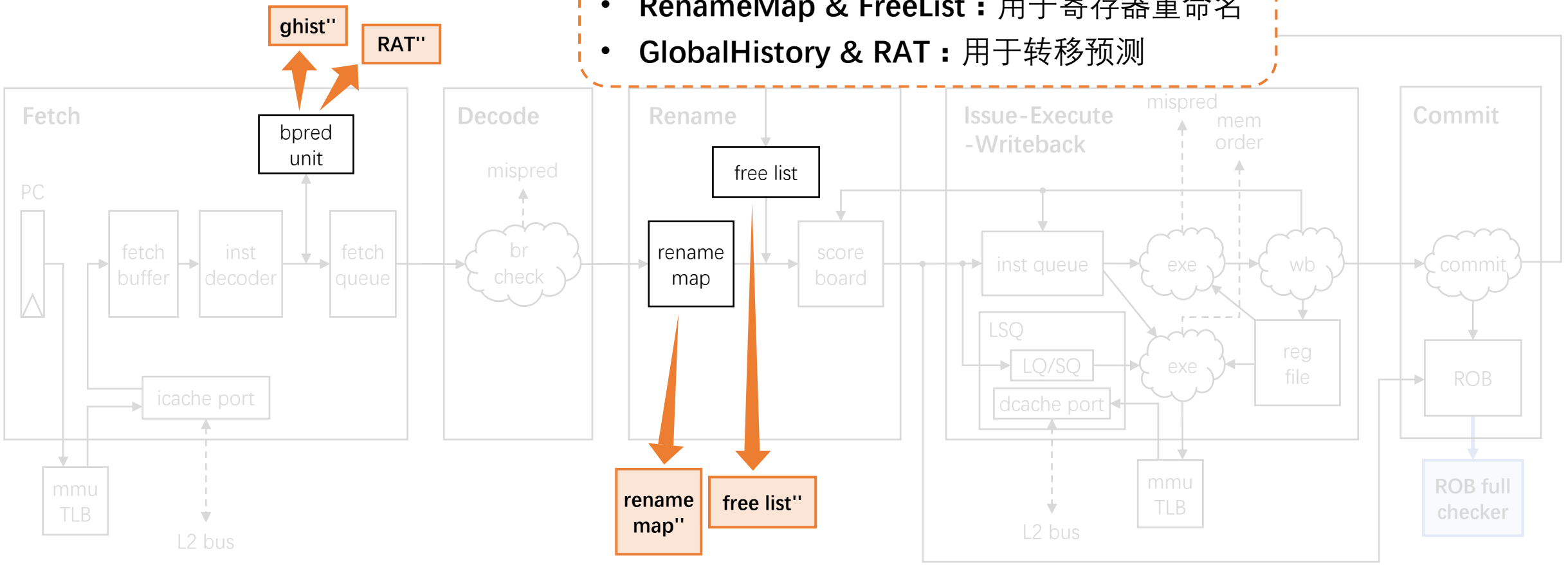


2. Precise Runahead Execution

PRE的工作原理

② 进入Rnahead模式时要保存什么内容

- **RenameMap & FreeList：**用于寄存器重命名
- **GlobalHistory & RAT：**用于转移预测

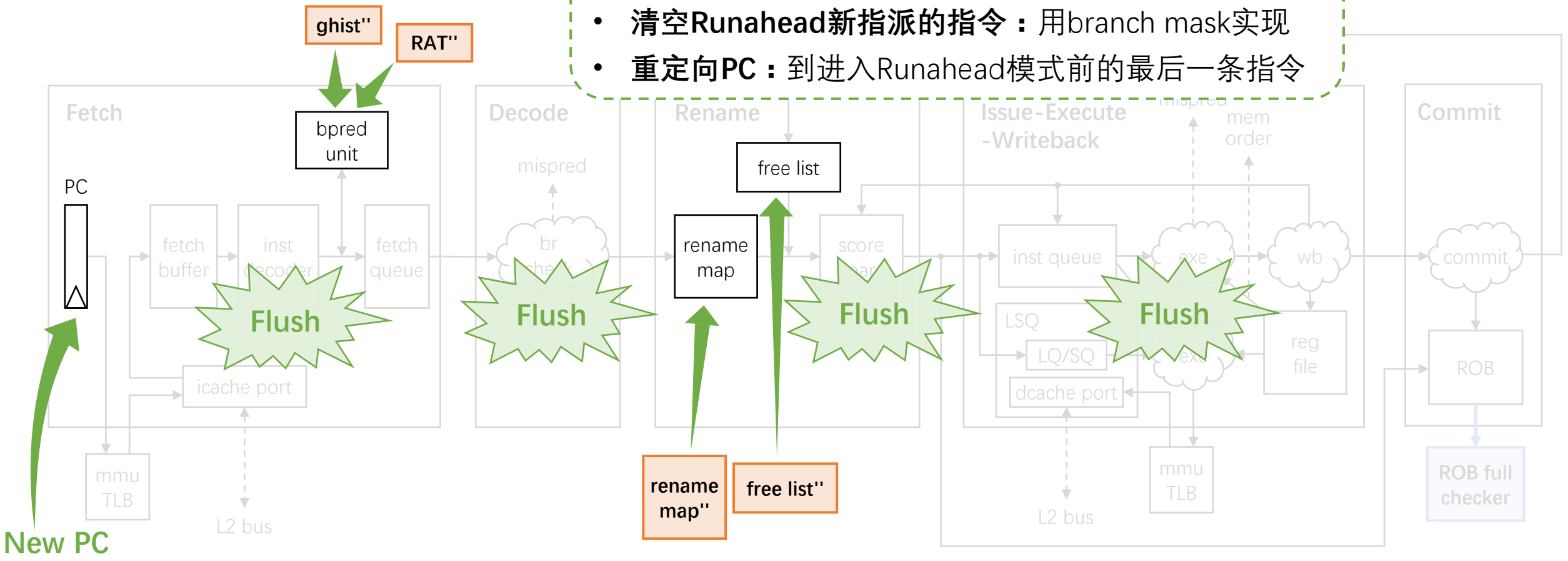


2. Precise Runahead Execution

PRE的工作原理

③ 如何退出Rnahead模式

- **恢复体系结构信息：**恢复上一步保存的信息
- **清空Runahead新指派的指令：**用branch mask实现
- **重定向PC：**到进入Runahead模式前的最后一条指令



类比

- ✓ 进入Runahead = 进入分支指令的其中一个方向
- ✓ 退出Runahead = 分支预测错误

北京大学 PEKING UNIVERSITY

Load Slice

- 在Runahead模式下执行所有指令是浪费的，很多指令对load无直接作用
- 在一个闭包空间内，load指令与计算load地址所需的所有指令称为**Load Slice**

```
1 func makebound(bound1: List[int])
2   for p in bound1 do
3     for q in 8 points around p do
4       meta := waymap[q]
5       process(meta)
6     done
7   done
8 endfunc
```

```
loop:
  li    a1, $waymap
  add   a2, a0, a1
  load  a3, (a2)
  xxx   xxx
  xxx   xxx
  xxx   xxx
  xxx   xxx
  xxx   xxx
  xxx   xxx
  addi  a0, a0, 1
  bne   a0, 8, loop
```

Load Slice

- **如何寻找Load Slice：**使用寄存器的传递关系，迭代式地寻找Load Slice
  - 用 **last_producer** 域记录寄存器的来源指令

```
loop:
  li     a1, $waymap
  add    a2, a0, a1
  load   a3, (a2)
  xxx    xxx
  xxx    xxx
  xxx    xxx
  xxx    xxx
  xxx    xxx
  xxx    xxx
  addi   a0, a0, 1
  bne    a0, 8, loop
```

首先将发生LLC miss的**load**指令加入Load Slice

Load Slice

- **如何寻找Load Slice：**使用寄存器的传递关系，迭代式地寻找Load Slice
  - 用 **last_producer** 域记录寄存器的来源指令

```
loop:
  li     a1, $waymap
  add    a2, a0, a1
  load   a3, (a2)
  xxx    xxx
  xxx    xxx
  xxx    xxx
  xxx    xxx
  xxx    xxx
  xxx    xxx
  addi   a0, a0, 1
  bne    a0, 8, loop
```

循环进行到下一周期时，发现**a2**来源于add指令

将add指令加入Load Slice

北京大学 PEKING UNIVERSITY

Load Slice

- **如何寻找Load Slice：**使用寄存器的传递关系，迭代式地寻找Load Slice
  - 用 **last_producer** 域记录寄存器的来源指令

```
loop:
  li     a1, $waymap
  add    a2, a0, a1
  load   a3, (a2)
  xxx    xxx
  xxx    xxx
  xxx    xxx
  xxx    xxx
  xxx    xxx
  xxx    xxx
  addi   a0, a0, 1
  bne    a0, 8, loop
```

循环周期进行到下一周期时，发现**a0**来源于addi指令，**a1**来源于li指令

将addi和li加入Load Slice

Load Slice

- **如何寻找Load Slice：**使用寄存器的传递关系，迭代式地寻找Load Slice
  - 用 **last_producer** 域记录寄存器的来源指令

```
loop:
  li    a1, $waymap
  add   a2, a0, a1
  load  a3, (a2)
  xxx   xxx
  xxx   xxx
  xxx   xxx
  xxx   xxx
  xxx   xxx
  xxx   xxx
  addi  a0, a0, 1
  bne   a0, 8, loop
```

之后的循环再也无法找到新的未确认来源的寄存器

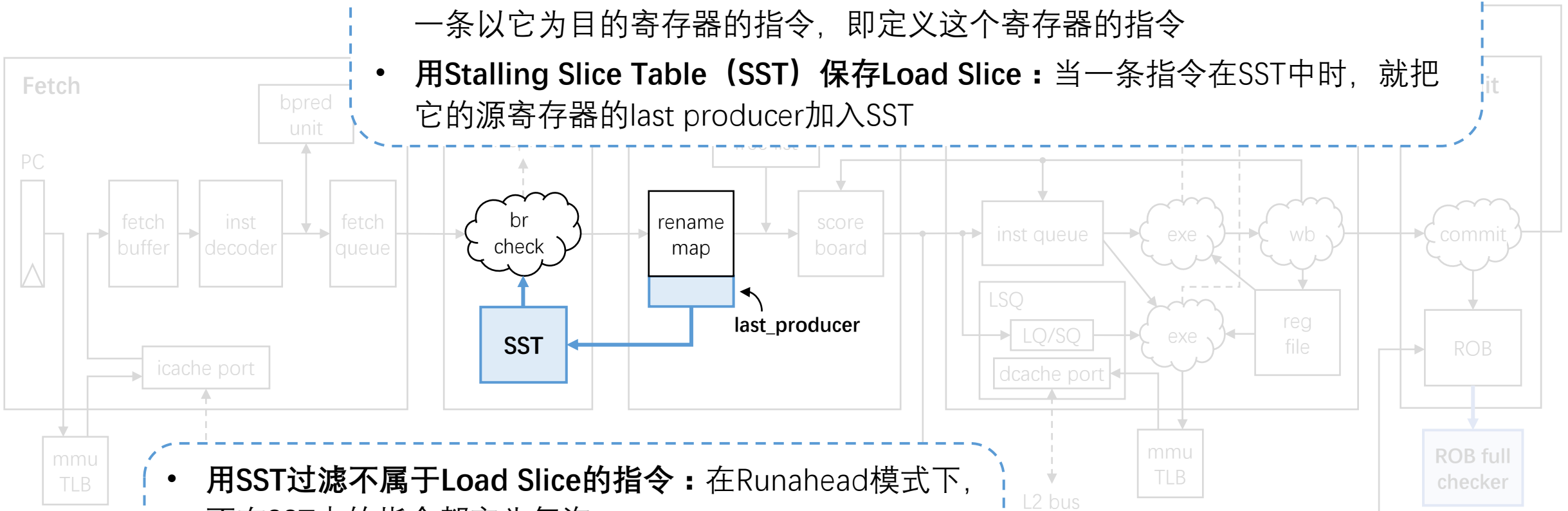
寻找Load Slice完成

北京大学 PEKING UNIVERSITY

PRE的工作原理

④ 寻找和使用Load Slice

- **用last_producer域记录寄存器依赖关系：**对于每个体系结构寄存器，记录上一条以它为目的寄存器的指令，即定义这个寄存器的指令
- **用Stalling Slice Table（SST）保存Load Slice：**当一条指令在SST中时，就把它的源寄存器的last producer加入SST



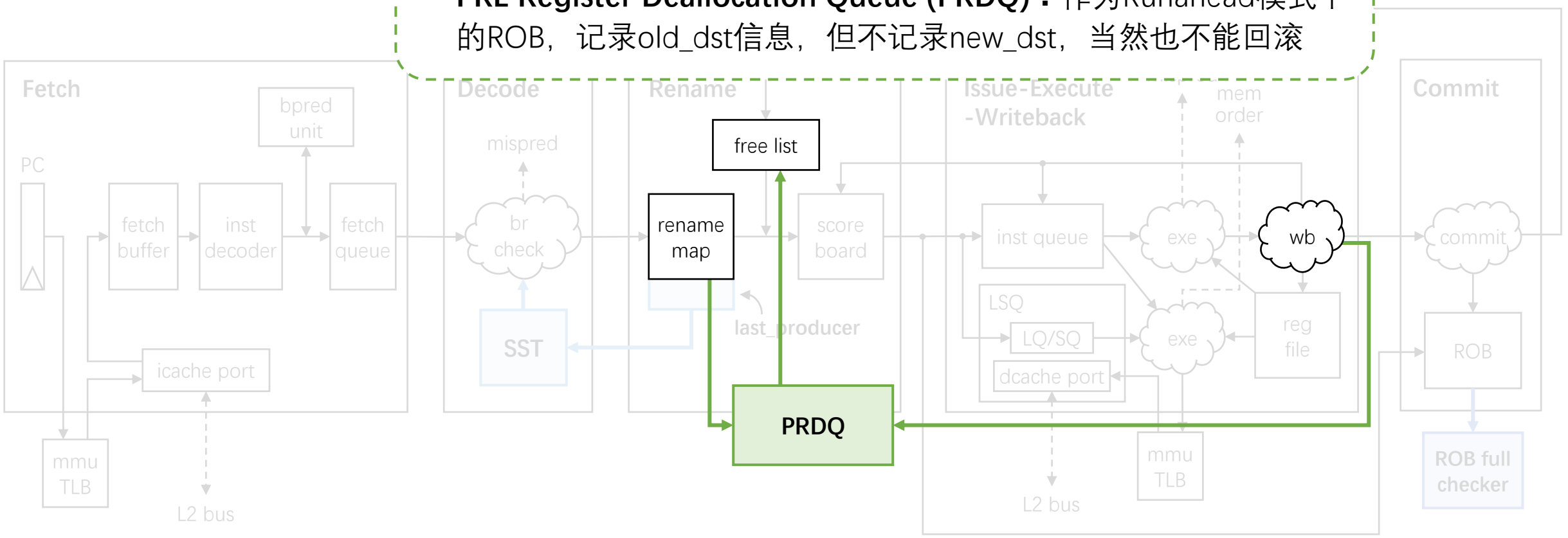
- **用SST过滤不属于Load Slice的指令：**在Runahead模式下，不在SST中的指令都变为气泡
- **关于转移指令：**不会被加入SST，故不会被执行，完全依赖转移预测器的结果

北京大学 PEKING UNIVERSITY

PRE的工作原理

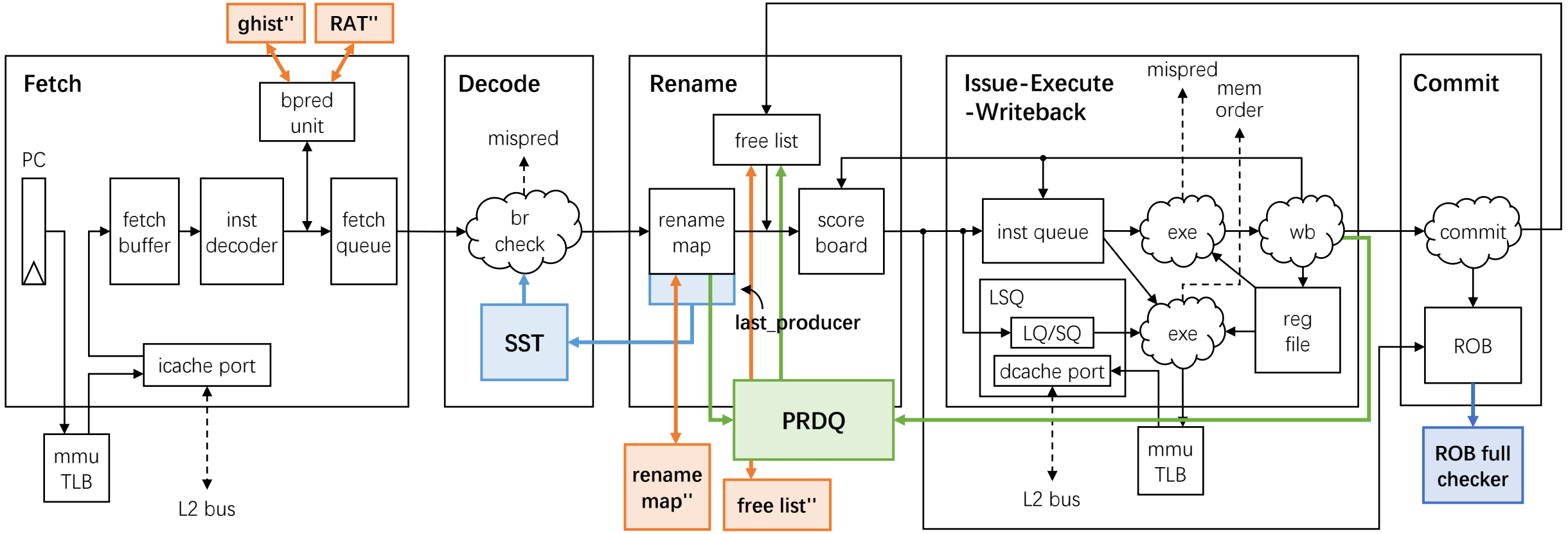
⑤ Runahead模式下的寄存器释放

- **PRE Register Deallocation Queue (PRDQ)：**作为Runahead模式下的ROB，记录old_dst信息，但不记录new_dst，当然也不能回滚



北京大学 PEKING UNIVERSITY

PRE的工作原理



总结

- ✓ 主要新增存储为SST、last_producer域、PRDQ
- ✓ 进出Runahead模式、RenameMap等可以沿用处理分支指令的机制

北京大学
PEKING UNIVERSITY

3.
实验结果与分析

论文复现 / 影响因素 / 传统预取器对比

北京大学 PEKING UNIVERSITY

处理器配置

Freqency	2.66 GHz
Pipeline width	4
ROB size	128
Issue queue size	92
Load queue size	32
Store queue size	32
Branch predictor	8 KB TAGE-SC-L
Register file	168 int, 168 fp
SST size	128
PRDQ size	192
L1 D-cache	32KB, assoc 4, 2 cyc
L2 cache	256KB, assoc 8, 8 cyc
LLC cache	1MB, assoc 16, 30 cyc
Memory	DDR3-1600, 800 MHz

类似于MegaBOOM的处理器

← 原文为192，由于原文用x86有uop膨胀效应，本文用RISC-V时调小了ROB，否则得不到足够的ROB满的机会

} 原文为64，过大且没有意义，本文将其改小

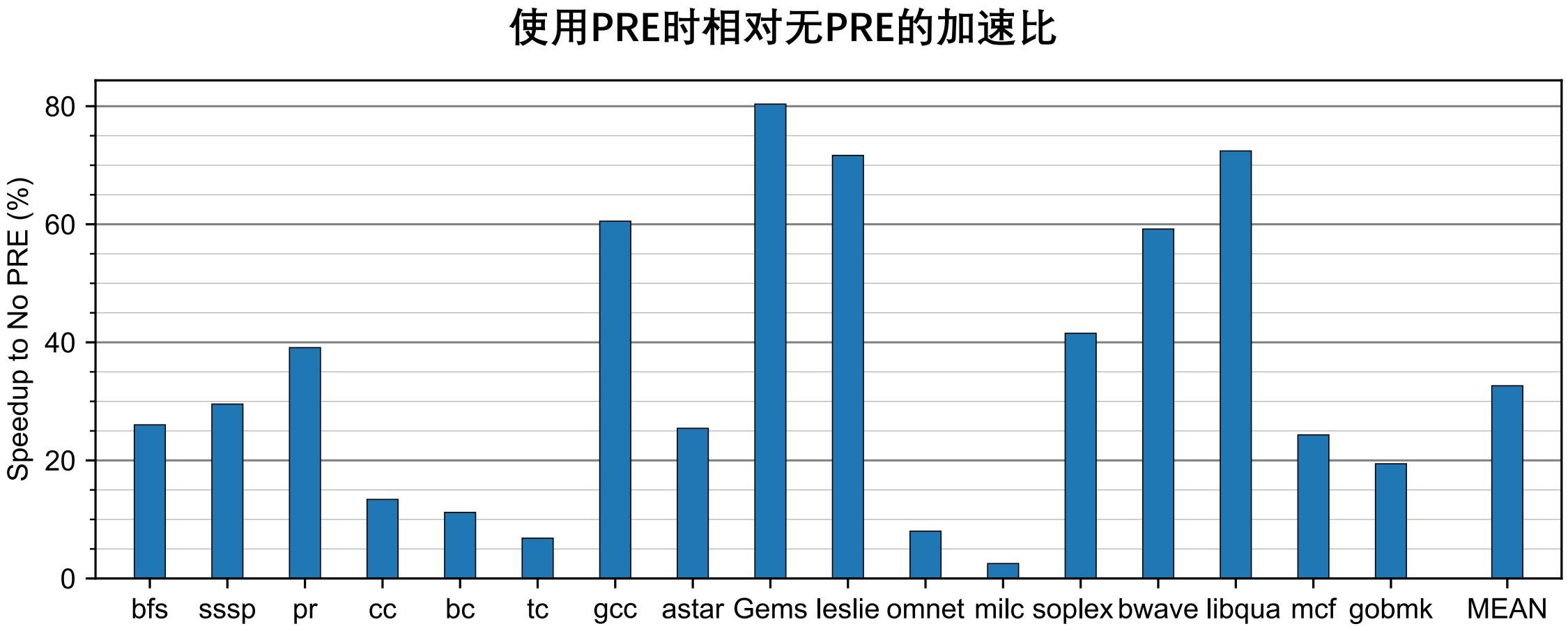
PRE的硬件开销

- **朴素实现**：7.5KB
- **高效实现**（哈希PC）：3.3KB
- **最简实现**（借用Rename Map和Free List备份）：2.9KB

3. 实验结果与分析

✓ PRE相比无预取可以带来平均32.6%的加速

论文复现



3. 实验结果与分析

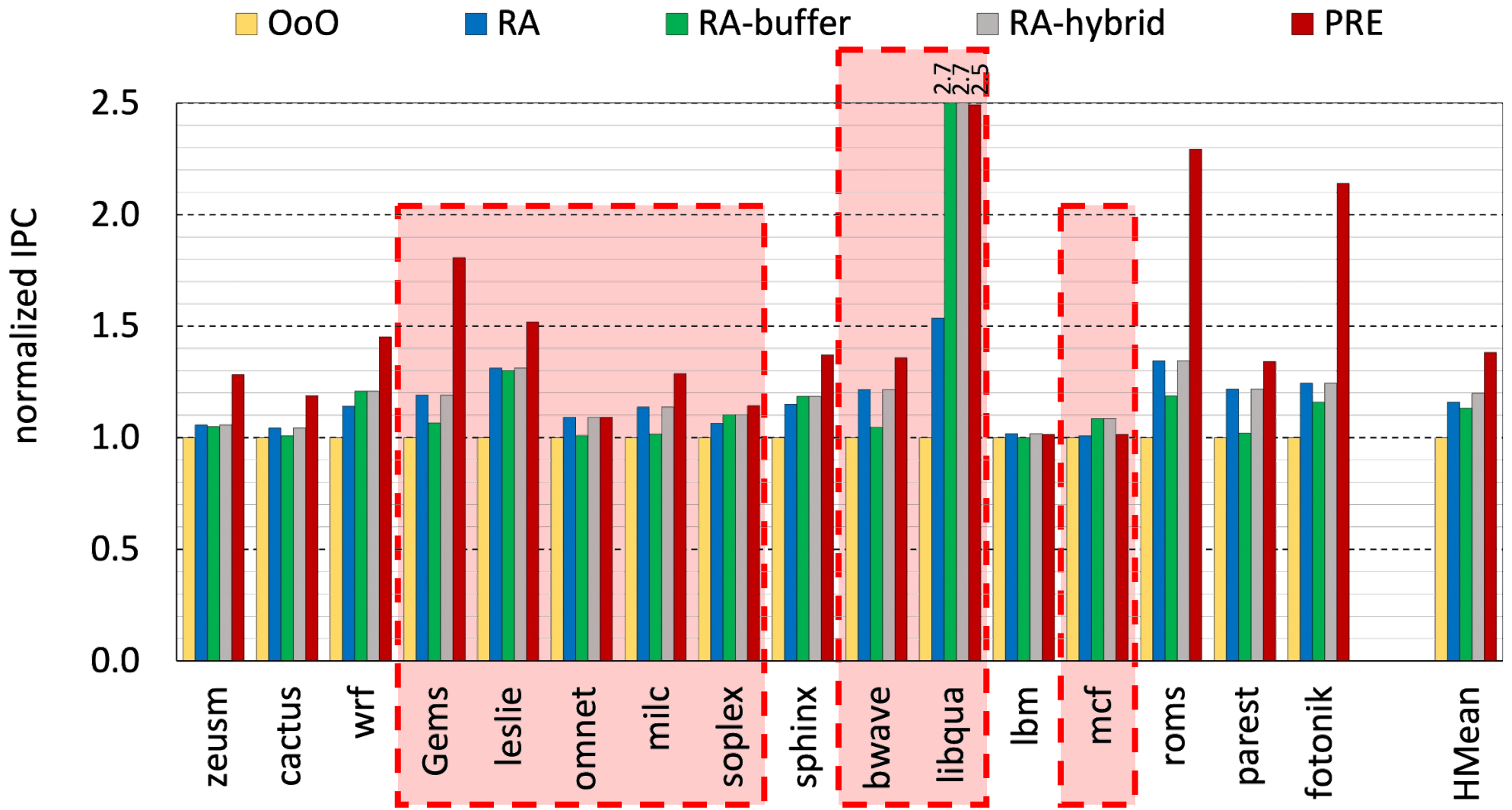
✓ 原文的平均加速比为39%，与复现结果相近

✓ 原文各程序的相对加速比与复现结果相近

北京大学 PEKING UNIVERSITY

论文复现

原文：PRE相比于无预取的归一化IPC

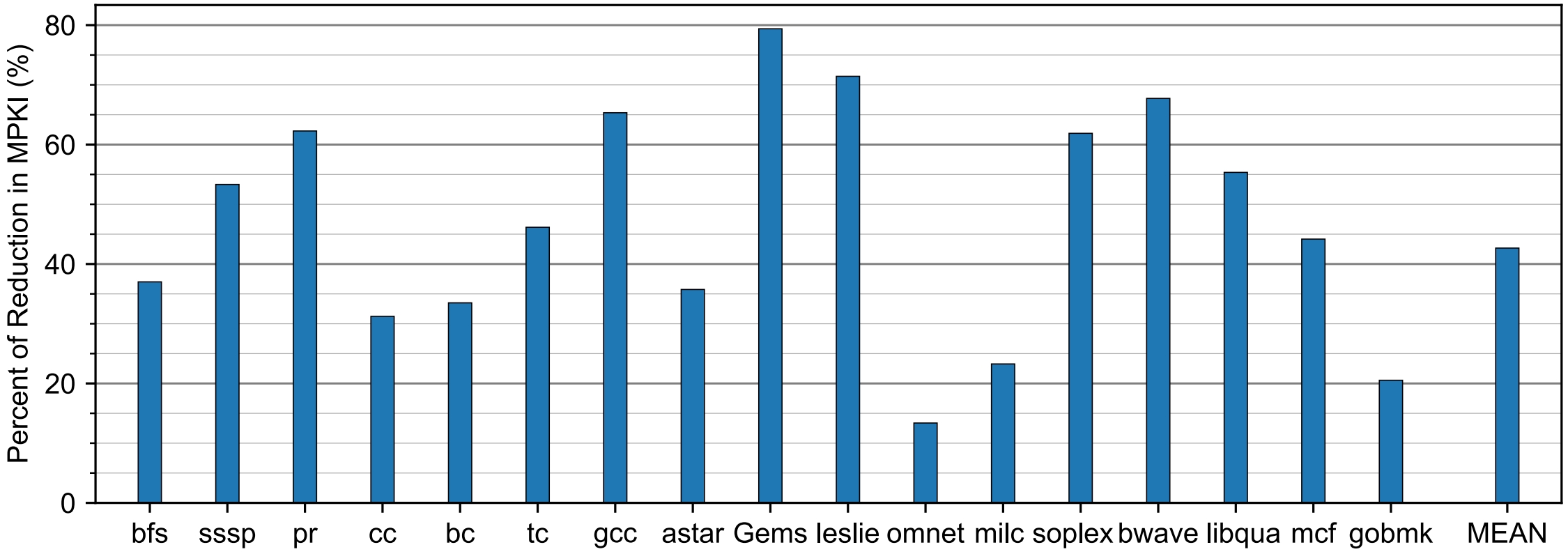


3. 实验结果与分析

- ✓ PRE显著降低了MPKI（平均42.7%）
- ✓ 与原文相近（49%）

结果解释

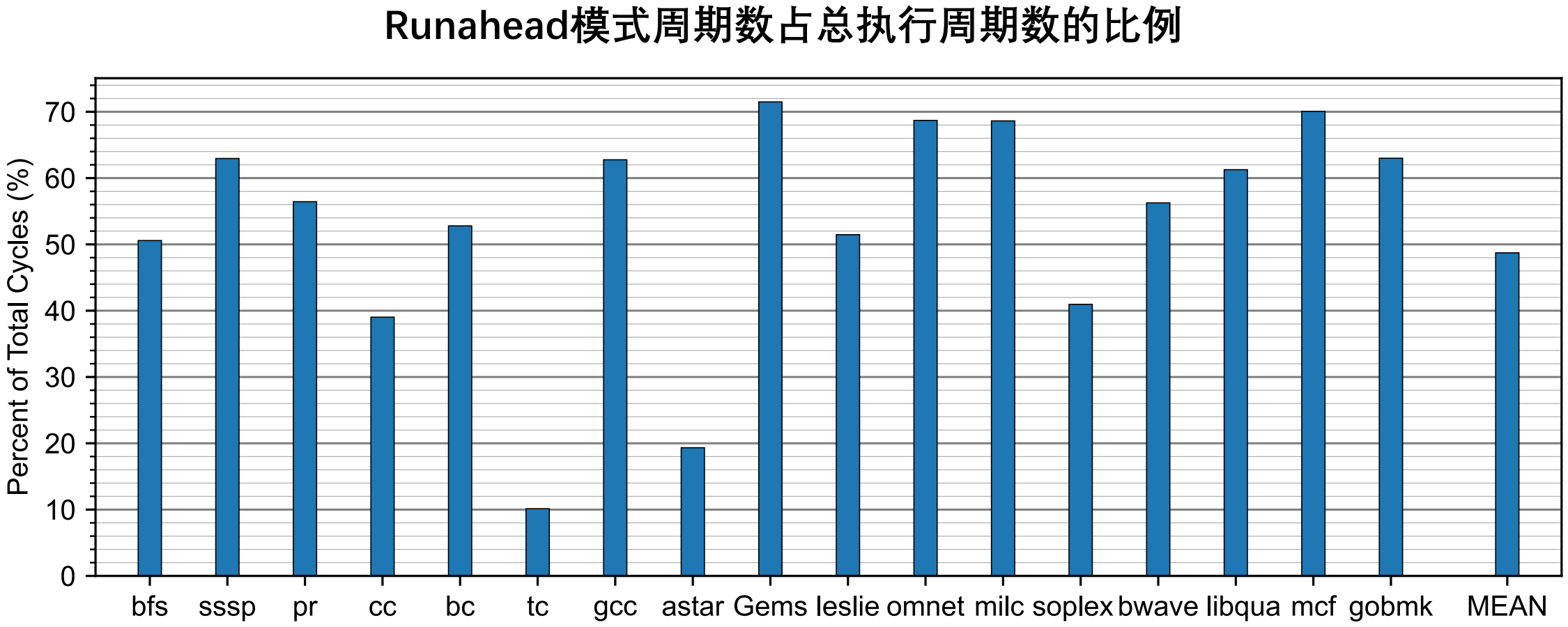
使用PRE时降低LLC Miss（以MPKI计）的比例



3. 实验结果与分析

- ✓ PRE发挥作用的前提是有机会进入Runahead模式
- ✓ 在访存密集型程序中有大量进入Runahead模式的机会（48.6%）

结果解释

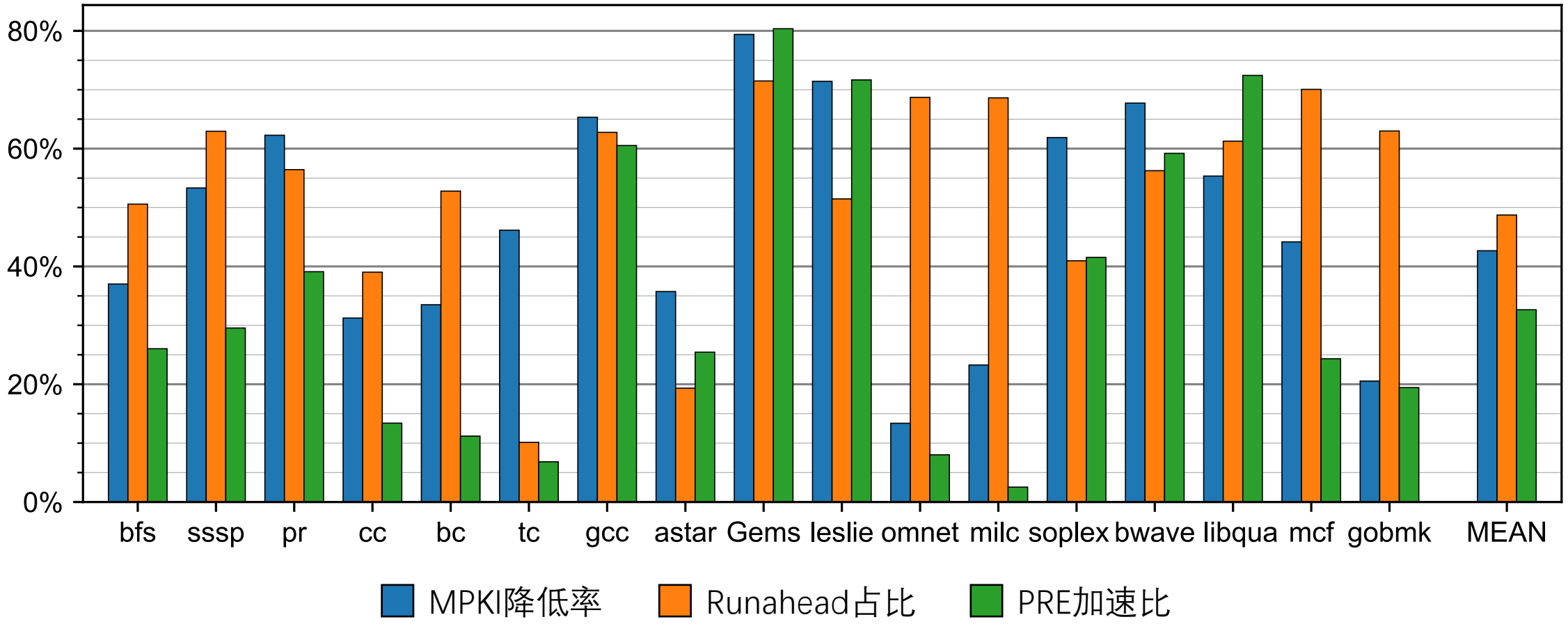


3. 实验结果与分析

✓ 更高的MPKI降低率、更高的Runahead占比意味着更高的加速比

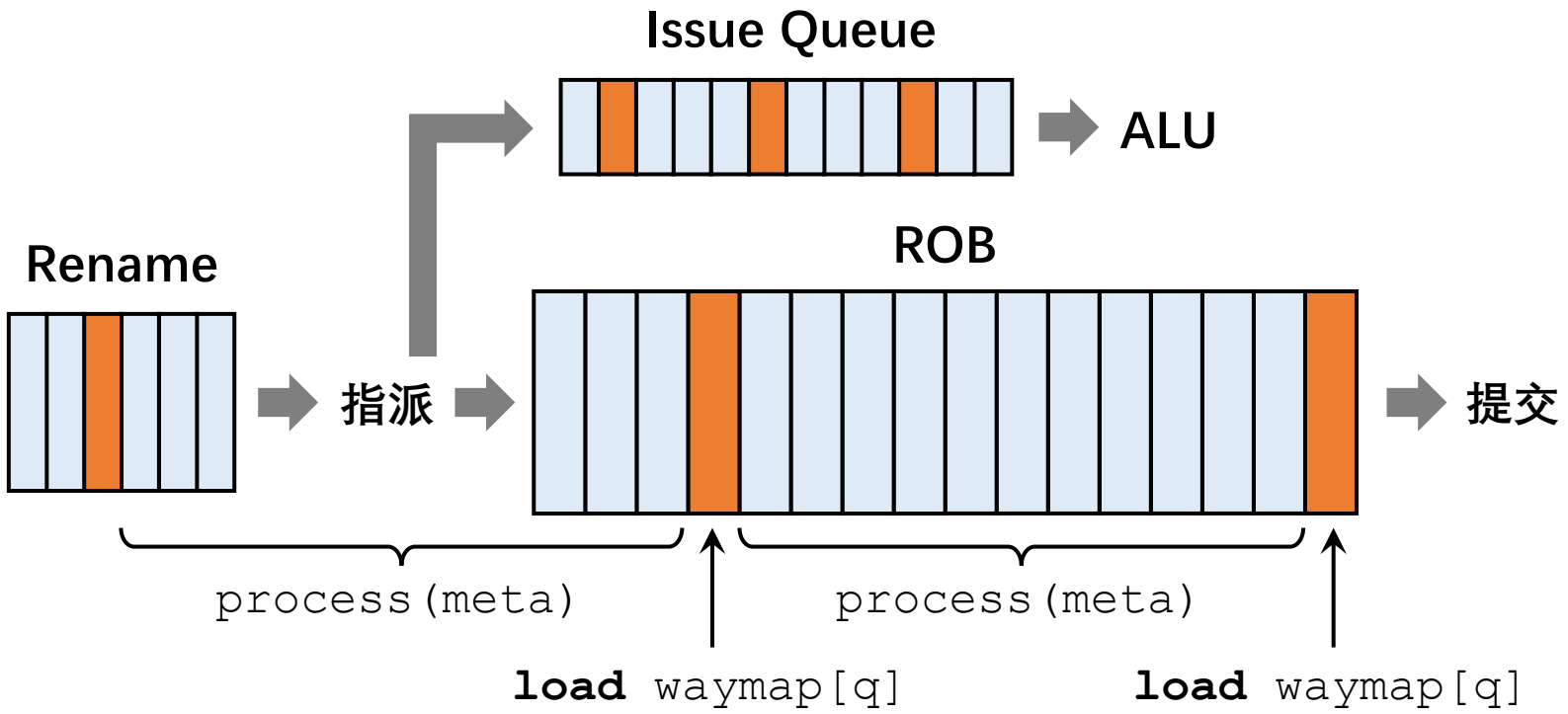
结果解释

组合图：MPKI降低-Runahead占比-PRE加速比



影响PRE性能的因素

- 主要有三个
  - 物理寄存器数量
  - 发射队列大小
  - ROB大小



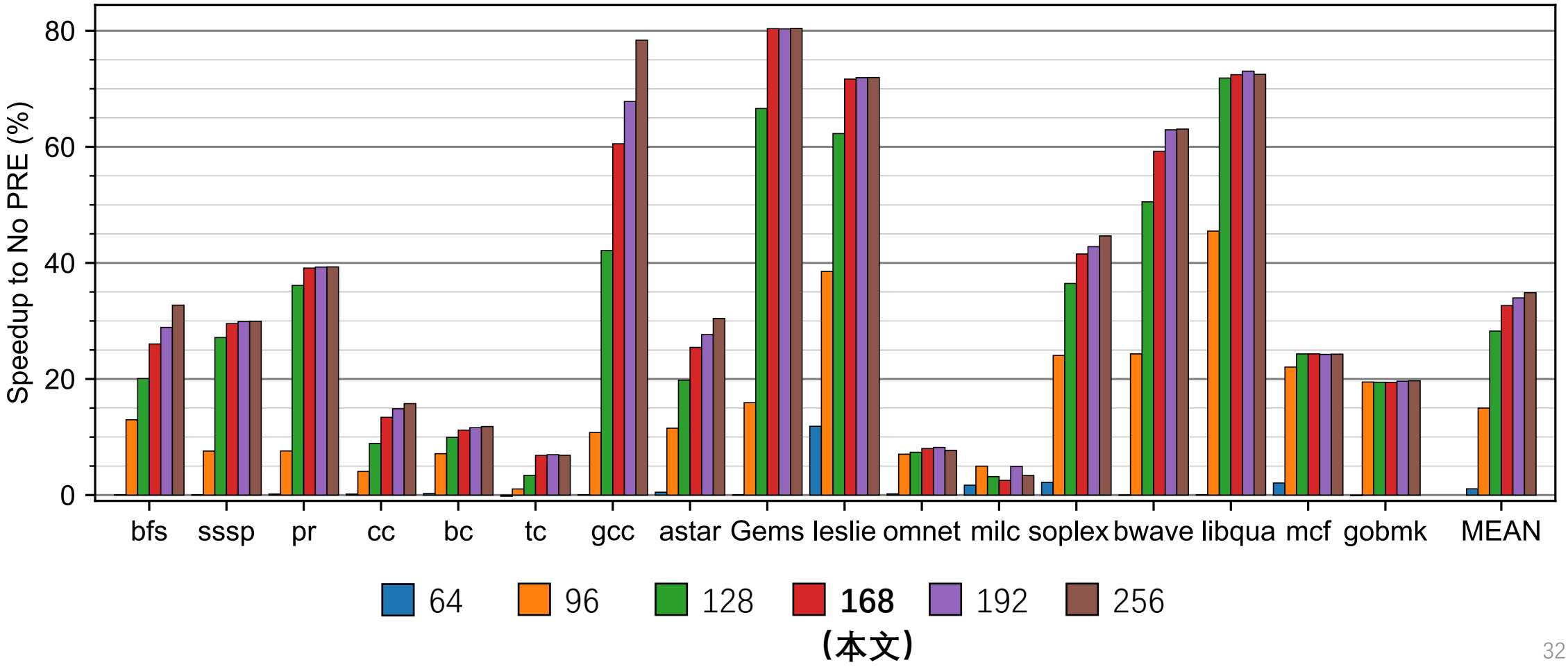
3. 实验结果与分析

✓ 增加物理寄存器可以提高RPE的加速比

✓ Runahead过程会分配物理寄存器，且可能不释放，只能增加数量

影响PRE性能的因素

不同物理寄存器数量下PRE相对于无PRE的加速比



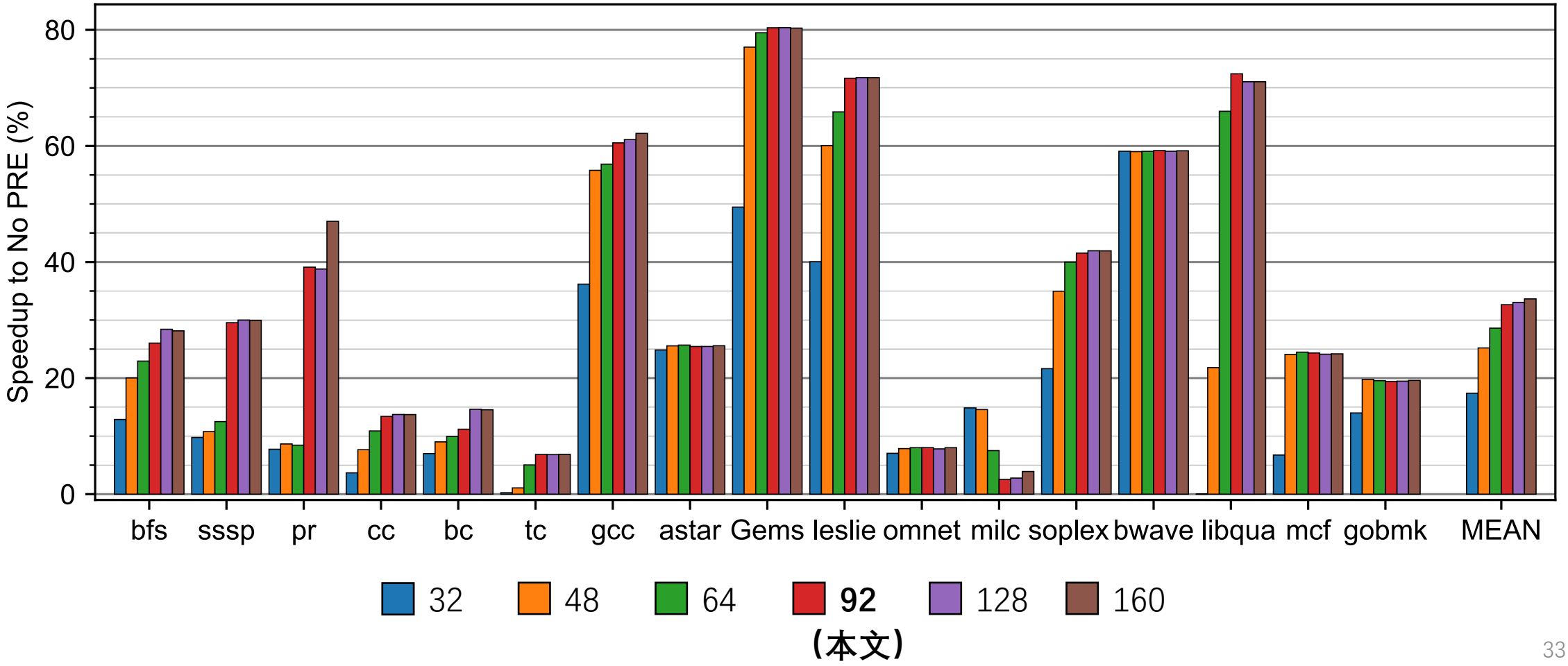
3. 实验结果与分析

✓ 增大发射队列可以提高RPE的加速比，但变化没物理寄存器显著

✓ Runahead过程同样会占用发射队列，但仅限一串有依赖的load指令

影响PRE性能的因素

不同发射队列大小下PRE相对于无PRE的加速比

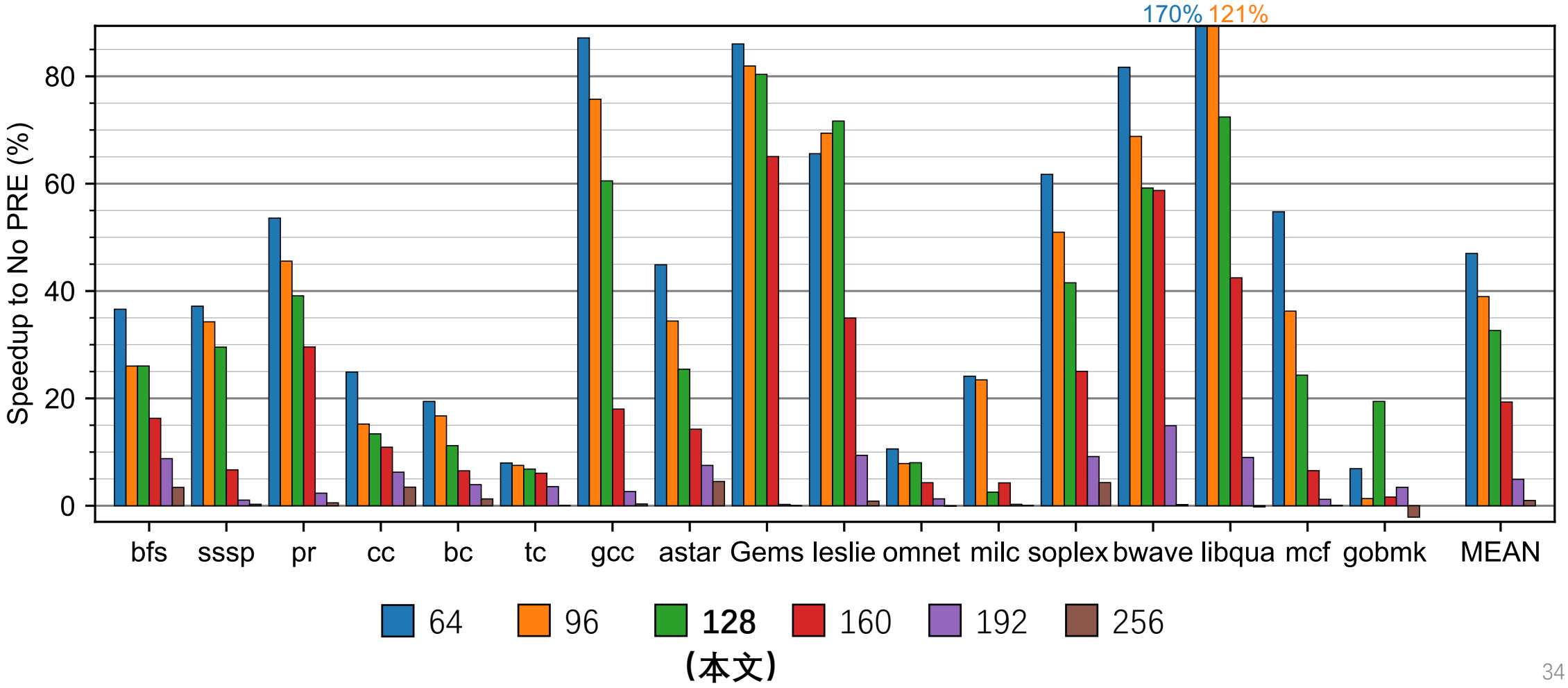


3. 实验结果与分析

- ✓ 增大ROB会降低PRE的加速比，因为减少了进入Runahead模式的机会
- ✓ 反映出PRE的本质是虚拟增加了ROB的大小

影响PRE性能的因素

不同ROB大小下PRE相对无PRE的加速比

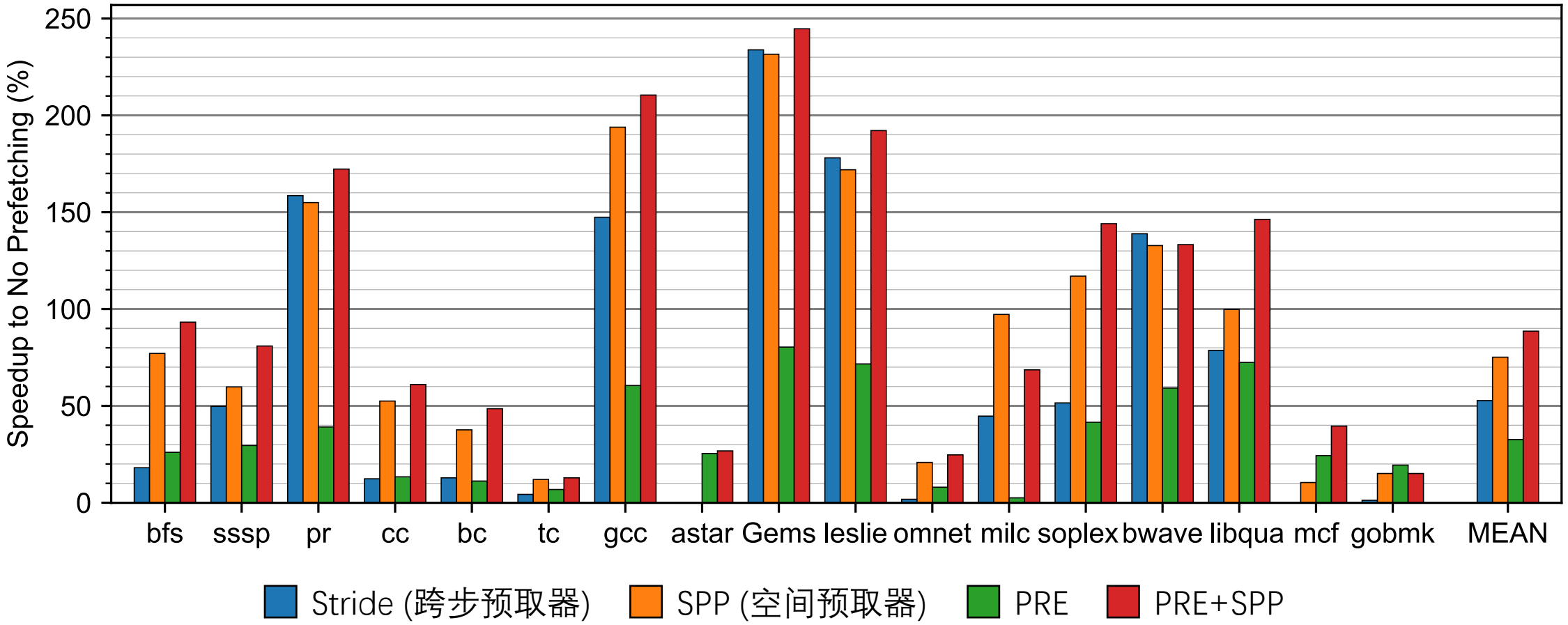


3. 实验结果与分析

- ✓ PRE单独使用时不如传统预取器
- ✓ PRE与SPP互补，比单独使用SPP提高17.9%

与传统预取器对比

不同预取方案的加速比



北京大学
PEKING UNIVERSITY

4.
结论与展望

结论 / 改进思路

北京大学 PEKING UNIVERSITY

结论

- PRE可以带来真实的**性能提升**
  - PRE可以带来平均32.6%的性能提升
- PRE的性能受**处理器参数**的影响
  - 与物理寄存器数和发射队列大小正相关，与ROB大小负相关
  - 但不应为了PRE而修改处理器参数，应以参数平衡为原则，PRE只是对现有的补充
- PRE的**适用性**不如传统预取器，但与传统预取器**互补**
  - PRE只能在ROB满时发挥作用，而传统预取器可以随时预取
  - 但PRE可以预取地址无规律的访存

改进思路

- 提高PRE性能的两个关键： “**跑得远**” “**跑得准**”

✓ **跑得远：**在有限的Runahead时间内发出更多预取

- 缓解资源耗尽问题：寄存器提前释放
- 提高前端取指效率：使用uop buffer

✓ **跑得准：**在Runahead过程中保持高转移预测准确率

- 选择性地执行一部分转移指令

- PRE存在性能上限，跳出PRE的框架可能是更好的思路

谢谢