# Pitfalls in PA

李晗

2025-12-02

南京大学 计算机学院

# 目录

PA2 ..... 2
Pitfalls ..... 3
Difftest ..... 10
Lab2 / PA2+ ..... 13
性能 ..... 14

# 目录

PA2 ..... 2
Pitfalls ..... 3
Difftest ..... 10
Lab2 / PA2+ ..... 13
性能 ..... 14

## Pitfalls

部分 AM kernels 需要大家实现 RISC-V M 扩展，一共也才 8 条指令。

都已经在 RV32I 实现过 30 多条指令了，想必很简单吧……

| RV32M Standard Extension | | | | | | |
|---|---|---|---|---|---|---|
| 0000001 | rs2 | rs1 | 000 | rd | 0110011 | MUL |
| 0000001 | rs2 | rs1 | 001 | rd | 0110011 | MULH |
| 0000001 | rs2 | rs1 | 010 | rd | 0110011 | MULHSU |
| 0000001 | rs2 | rs1 | 011 | rd | 0110011 | MULHU |
| 0000001 | rs2 | rs1 | 100 | rd | 0110011 | DIV |
| 0000001 | rs2 | rs1 | 101 | rd | 0110011 | DIVU |
| 0000001 | rs2 | rs1 | 110 | rd | 0110011 | REM |
| 0000001 | rs2 | rs1 | 111 | rd | 0110011 | REMU |

# Pitfalls

```
INSTPAT("...", mul    , R, R(rd) = src1 * src2);
INSTPAT("...", mulh   , R, R(rd) = (SEXT(src1, 32) * SEXT(src2, 32)) >> 32);
INSTPAT("...", divu   , R, R(rd) = src1 / src2);
INSTPAT("...", remu   , R, R(rd) = src1 % src2);
```

这个实现正确吗?

## Pitfalls

M 扩展的手册原文一共不到两页，非常清晰地指明了除法的行为，甚至列了一个表格：

> The semantics for division by zero and division overflow are summarized in Table 11.
>
> The quotient of division by zero has all bits set, and the remainder of division by zero equals the dividend. Signed division overflow occurs only when the most-negative integer is divided by $-1$. The quotient of a signed division with overflow is equal to the dividend, and the remainder is zero. Unsigned division overflow cannot occur.

| Condition | Dividend | Divisor | DIVU[W] | REMU[W] | DIV[W] | REM[W] |
|---|---|---|---|---|---|---|
| Division by zero | $x$ | $0$ | $2^L - 1$ | $x$ | $-1$ | $x$ |
| Overflow (signed only) | $2^L - 1$ | $-1$ | - | - | $-2^L - 1$ | $0$ |

## Pitfalls

手册里的细节：

- 乘法操作数和结果的有无符号。
- 在 32 位上计算 64 结果，即同时需要 MUL 和 MULH 的结果时，两条指令的发射顺序。
- 为什么我们需要 MULHSU 这个操作数符号混合、看起来非常奇怪的指令？

  为什么没有 DIVSU？
- 有符号和无符号除法无法整除时，结果如何取整？
- 为什么没有像 x86 一样在被零除时抛出异常？

# Pitfalls

```
INSTPAT("...", divu   , R, R(rd) = src1 / src2);
// __am_timer_uptime
uint64_t current_time = *(volatile uint64_t *)(RTC_ADDR);
uint64_t current_time = ((uint64_t)inl(RTC_HIGH_ADDR) << 32)
                      + (uint64_t)inl(RTC_ADDR);
```

RV32I 一次只能读取 32 位的时钟计数器，以微秒计数一共只能表示大约 71 分钟。

| | t=19 | t=20 |
|---|---|---|
| 高位 | 1 | 1 |
| 低位 | | 0 |

| | t=19 | t=20 |
|---|---|---|
| 高位 | | 2 |
| 低位 | 9 | 9 |

无论先读取高位还是低位，都有可能在读取过程中低位溢出，导致读取到的值不正确。
实际上因为启动时计数器不一定为 0，一个运行 1 分钟的程序有 1/71 的概率出现溢出……

# Pitfalls

```
INSTPAT("...", divu   , R, R(rd) = src1 / src2);
// __am_timer_uptime
uint64_t current_time = *(volatile uint64_t *)(RTC_ADDR);
uint64_t current_time = ((uint64_t)inl(RTC_HIGH_ADDR) << 32)
                      + (uint64_t)inl(RTC_ADDR);
```

RV32I 一次只能读取 32 位的时钟计数器，以微秒计数一共只能表示大约 71 分钟。

手册的推荐做法：

> The following code sequence will read a valid 64-bit cycle counter value into x3:x2, even if the counter overflows its lower half between reading its upper and lower halves.

```
again:
  rdcycleh     x3
  rdcycle      x2
  rdcycleh     x4
  bne          x3, x4, again
```

# PA 主线也饱含知识点

- 实现这些指令甚至都还是 PA2 主线内容的一部分
  - 想要确保完美还原 RISC-V 手册中规定的指令行为，甚至还得翻阅 C 语言标准?
- 这些缺陷都不是无理取闹，完全有可能在现实场景中触发
  - 手册同时考虑了软硬件、OS、编译器等多个层面，并且已经给出了合理的设计。

速通 PA 真的错过了很多拓宽视野、拔高思维层次的机会……

# 上次分享的 Callback

原则：一定要注意信息来源的可靠性！

其他学生的博客、GPT 都有可能犯错，尤其是考虑不全。手册才是大家最好的朋友。

RISC-V Manual (Unprivileged, Privileged, UnifiedDB, Documentation for RV64)

最经典的问题：

- PA2 中的 `mulh`，`mulhu`，`mulhsu` 是有符号还是无符号，`div`，`divu`，`rem`，`remu` 除数是 0 的行为是什么？
- PA3 中 `ecall` 抛出的异常编号到底是多少？Abstract Machine 里的 `yield` 和 `Event` 应该怎么实现？
- PA3 中 libndl 的像素格式是什么？newlib 对 `open` 等系统调用的约定是什么？

# Difftest

- Difftest 其实是一个被大家极大地低估了的工具。
- 很多同学只把 difftest 以及 ftrace、mtrace 开到 PA2，之后就再也没有重新打开过……
  - 但实际上 difftest 可以一直开到 PA4 的时钟中断前，甚至通过一些适配，可以一直开到 PA 结束。

## Difftest

- Difftest 其实是一个被大家极大地低估了的工具。
- 很多同学只把 difftest 以及 ftrace、mtrace 开到 PA2，之后就再也没有重新打开过……
  - 但实际上 difftest 可以一直开到 PA4 的时钟中断前，甚至通过一些适配，可以一直开到 PA 结束。
- 适配的工作量并没有大家想象中那么大，但是相对应的收益是非常可观的。
  - 如果在开着 difftest 的情况下，程序仍然**无法正常运行** $\Longrightarrow$ 问题一定不在 NEMU
  - 如果在开着 native 的情况下，程序**正常运行** $\Longrightarrow$ 问题一定不在 NanOS 和 Navy

在 PA3、PA4 项目模块愈加复杂的情况下，这种排除法的价值会愈加凸显。

# Difftest

未来的期末考试（但不是这一届）：

- 在你的 PA 中随机注入多个 bug，涵盖硬件和软件 😈
- 需要在 30 分钟内排除 bug，除了对比原始代码，不限制使用任何工具

对基础设施的全方位考验：

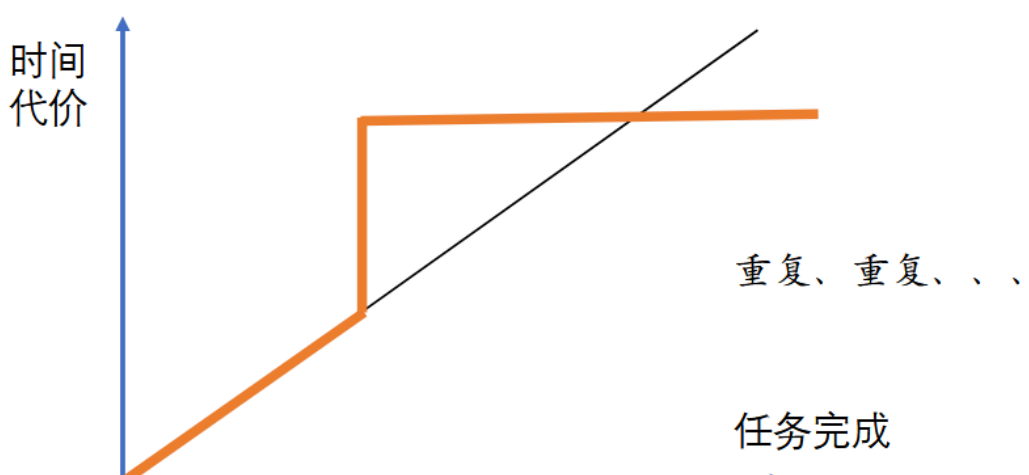
- 有 microtest $\Longrightarrow$ 光速定位问题功能
- 有 difftest $\Longrightarrow$ 光速定位第一个出错指令
- 有 sdb、ftrace、strace $\Longrightarrow$ 光速倒推问题原因

# 基础设施

基础设施是在投资，用小时间博取大时间。

尤其是现在有 AI 辅助的情况下，任何一个已经让你重复三遍以上的地方都应该自动化。

- 比如设置 `make submit` 的 TOKEN，生成 compiledb，运行回归测试，开关 difftest……



# 目录

PA2 ..... 2
Pitfalls ..... 3
Difftest ..... 10
Lab2 / PA2+ ..... 13
性能 ..... 14

# Lab2 情况反馈

- 代码量非常小，给大家轻松一下。多数同学交了就是满分。
  - 第一部分代码量小于 10 行。
  - 第二部分有现成的文章讲解和参考实现。
- 但是非常建议大家真正上手运行一次 perf
  - 第一部分的优化点在火焰图上非常明确。
  - 思考题：perf 的原理是什么？如何在 NEMU 内运行 perf 找到第二部分的优化点？

# 性能优化近似玄学

很多大家想象中能巨幅提升性能的点在工程实践中反而没有很好的效果?

- 都说 KMP 算法效率高，现实中真没看到哪些领域用到的最多?
- 为什么很多语言的实现里面的 Lexer 都没有使用 DFA?
- 内存为什么不能设计成二维寻址?

# 性能优化近似玄学

很多大家想象中能巨幅提升性能的点在工程实践中反而没有很好的效果？

- 都说 KMP 算法效率高，现实中真没看到哪些领域用到的最多？
- 为什么很多语言的实现里面的 Lexer 都没有使用 DFA？
- 内存为什么不能设计成二维寻址？

原因：软硬件左脚踩右脚的螺旋上升

- 硬件设计者根据程序的时空局部性，设计了复杂的缓存来优化程序性能。
- 程序设计者根据硬件特性，调整了数据结构和算法来达成时空局部性。

抽象不再是无本万利的好事，你需要知道部分硬件的设计细节，才能写出高性能的程序。

# 性能优化不是玄学

优化需要综合利用数据特征和硬件特性，让**性能指标**最优化。

已知算法 A 的平均复杂度是 $O(n)$，算法 B 的最差复杂度是 $O(\sqrt{n})$，选择算法 A 还是 B?

# 性能优化不是玄学

优化需要综合利用数据特征和硬件特性，让**性能指标**最优化。

已知算法 A 的平均复杂度是 $O(n)$，算法 B 的最差复杂度是 $O(\sqrt{n})$，选择算法 A 还是 B?

- 取决于你优化目标是期望时间、99% 分位时间还是最坏时间。
- 取决于在某个**真实**的机器上、在**真实**数据分布下，各个算法的表现。

| | 算法 A | 算法 B | 出现概率 |
|---|---|---|---|
| n=10 | 1s | 3s | 90% |
| n=100 | 10s | 5s | 10% |

# 性能优化不是玄学

优化需要综合利用数据特征和硬件特性，让**性能指标**最优化。

已知算法 A 的平均复杂度是 $O(n)$，算法 B 的最差复杂度是 $O(\sqrt{n})$，选择算法 A 还是 B?

- 取决于你优化目标是期望时间、99% 分位时间还是最坏时间。
- 取决于在某个**真实**的机器上、在**真实**数据分布下，各个算法的表现。

| | 算法 A | 算法 B | 出现概率 |
|---|---|---|---|
| n=10 | 1s | 3s | 90% |
| n=100 | 10s | 5s | 10% |

我全都要！让程序自动根据数据分布选择算法!

- 恭喜你，发明了混合排序算法 (C++) 和 Pattern-defeating quicksort (Rust)。

# 性能优化不是玄学

以后遇到性能问题的时候，请好好抑制住随机调优的冲动。

- 制定一个明确且**合理**的性能指标
  - 不光是时间，还有空间，甚至能耗……
  - 一些任务甚至允许你牺牲部分“正确性”，例如视频编解码
- 收集**真实**的数据分布
  - 尽可能覆盖所有情况，但也不要只看极端情况
  - 不然就是跑分没输过，体验没赢过
- 用**真实**机器上收集的实验数据指导优化
  - 不要盲目相信“理论上”“书上说”“网上说”“我觉得”

# 总结

多读书、多看手册、多看代码

- 阅读长文章的能力是基本功，不要被快餐文化绑架
- 完整的知识体系 ≫ 零散的片段，原始文档 ≫ 转载

基础设施是大家的伙伴

- 投资时间搭建基础设施，在未来节省大量重复劳动
- 自动测试、调试工具、性能分析……

性能优化不是玄学

- 用科学的方法指导优化，不再盲目调优