# 论文阅读报告

论文：Performance Analysis Using the MIPS R10000 Performance Counters. Marco Zagha, Brond Larson, Steve Turner, Marty Itzkowitz.

## 一、 目的

### 1.1 解决的问题

本论文介绍了一种在 MIPS R10000 平台上分析程序和底层结构之间的联系的方法，用于帮助程序开发者调试程序、提升性能。

对于性能分析存在以下的要求：

1、 不会扰乱程序执行。

2、 不会造成延迟。

3、 适应动态超标量的特性。

4、 具有较强的存储器系统行为分析能力。

R10000 通过使用硬件计数器统计不同类型的事件的性能分析方法来达到这些要求，事件包括缓存未命中、缓存数据的一致性处理和分支预测错误等。

### 1.2 面临的挑战

相比于同类型的设计实现（Cray），在 R10000 平台上存在一定的限制：

1、 较小的空间预算。

2、 支持任意二进制文件的程序级事件分析。

3、 不需要明确的重编译或者重链接。

4、 不能扰乱内存地址。

针对以上的限制，R10000 通过这些方面来解决：硬件综合设计、操作系统接口函数、性能分析工具。以下对其分点阐述。

## 二、 实现方案

### 2.1 硬件综合设计

R10000 硬件实现了两个 32 位计数器，任何一个进行配置后可以用于监视一个事件行为。每一个计数器对应 16 种不同的事件类型，两个计数器总共可以监视 30 个事件类型（存在两个事件可以被两个计数器同时监视）。

当计数器溢出时，会触发内核级中断。并且可以设置计数器的初始值，使其具有统计事件周期性的能力。

并且该结构的电路设计未引入关键时序路径，不会影响程序的性能。

实现过程中的主要部分：

1、 选择合适的事件：简单且相互独立的事件集，能够精细且简洁易懂的描述系统行为（动态行为），并且对处理器的整体性能影响较小。

2、 在芯片上布线以获取对应事件的片上信号，由于信号分散在各个逻辑块上，因此需

要几条较长的线路。

3、 验证计数器的正确性。

## 2.2 操作系统接口函数

操作系统这一层次将硬件计数器虚拟化，并在用户模式中建立了一个灵活且共享的抽象。在牺牲了一定效率的基础上，提供了具有较高实用性和结果可读性的接口。

具体来说，操作系统内核维持着 32 个不同的 64 位虚拟计数器（可以同时监视所有的事件），并通过程序接口提供给开发人员。也就是说，用户程序可以为每个硬件计数器指定 16 个事件，而实现方法就是通过开关或者时间复用 n 路信号，缩短信号的处理时间至 1/n，完成指定在同一计数器的事件的计数。

接口函数允许开发人员为监视的事件指定频率，内核会以此为依据向程序传递相应的用户信号。

## 2.3 性能分析工具

能够提供 R10000 计数器数据的工具有：perfex 和 SpeedShop。以下分别对其进行讲解。

a) perfex

一个命令行接口，能够提供程序级的事件信息（一个程序中共发生多少次指定事件），通过该工具可以确定应用中是哪个处理器事件造成了性能问题。

b) SpeedShop

该工具的性能统计分析是基于硬件计数器的溢出来实现的。当特定的计数器溢出时，就会记录当前的程序计数器，基于此就可以找到特定事件集中的程序行，得到程序行级的事件信息。

论文中将其和 PC sampling（prof：统计函数的运行时间）进行比较，可以体现出其优势。两者的不同在于 PC sampling 是基于时钟中断：固定时钟周期数，执行中断程序，记录数据。但程序的执行也是基于时钟，也就导致 PC sampling 中断间无法对调用做出响应，会产生统计误差。而基于硬件计数器溢出的 SpeedShop 不存在这种关联性，它可以响应调用事件，做出一定的调整。

不仅如此，SpeedShop 还能对其他 PC sampling 无法接触到的事件进行统计，比如缓存未命中。但在这种情况下，应用中的程序结构可能会和计数器的中断周期数产生关联，导致统计不准确。因此 R10000 为每个计数器提供了两套基本预置数，可用于保证统计数据的有效性。

# 三、 总结

由上述三个层次内容的整合，给予了开发人员观察程序动态行为和程序执行过程中资源消耗情况的机会。将性能分析方法从原本的只能以汇编语言为主，提升到应用程序的层次，甚至能够具体到程序中的某个部分。

最终该结构呈现在程序开发人员面前的只有 perfex 和 SpeedShop 两个性能分析工具，本文最后列举几个以这两个工具为基础的性能分析案例。

# 四、 性能分析案例

**----该部分存在比较多的细节内容和我自己的理解（可能存在问题。）。**

## 4.1 缓存未命中优化

首先通过 perfex 获取在各种硬件单元的资源消耗情况，数据统计在图 1 中体现。可见程序的运行时间主要消耗在处理二级数据缓存未命中事件中。

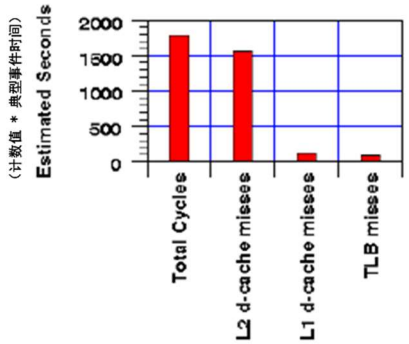



图 1.

然后使用 SpeedShop 针对二级数据缓存未命中事件，生成程序行级的数据统计，见图 2.

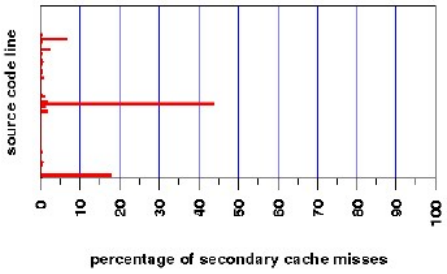



图 2.

该图体现了事件在程序中集中的位置，而且存在一条语句引发了将近 40%的二级缓存未命中事件。

对该问题语句进行分析：程序中存在一个由结构体组成的数组，该语句在一个循环结构中不断地访问数组元素（结构体）中的 int 数据类型。但一个结构体所需的存储空间大于二级缓存中一个高速缓存行的大小（128 字节），这也就导致了程序相当差的缓存空间局部性。

可以使用的改进方法就是将结构体数组中每个元素的 int 数据独立出来，构成一个单独的数组，保证程序较好的空间局部性。

## 4.2 多级处理器层次结构优化

这里使用的案例简单来说，就是从三个维度分别扫描一个 100 X 100 X 100 的网格。分析步骤：

（1）SpeedShop 统计每个维度所耗费的时钟周期数，得到结果是 X、Y、Z 分别消耗 26%、28%、45%。

（2） perfex 对各种类型事件进行统计，得到结果是存储器操作消耗了大量时间，TLB 未命中占 45%、二级数据缓存未命中占 45%、一级数据缓存未命中占 5%。

（3） SpeedShop 对上述三个存储器事件进行统计，得到每个维度上的各个面上所消耗

的平均时间。结果体现在图 3 中。

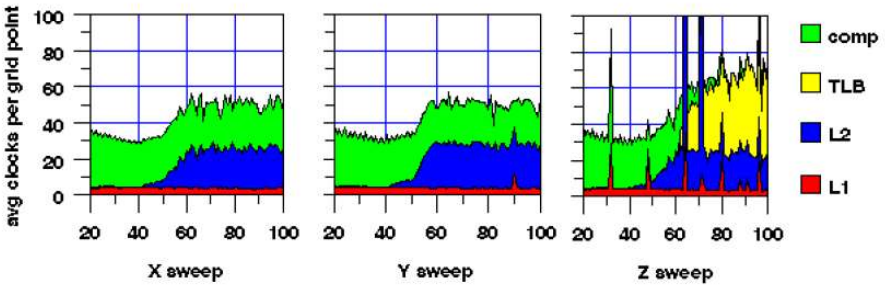



图 3.

以下将指出几个性能问题并提出针对性的优化方案：

1) 数组填补

图中的尖峰就是由缓存未命中造成的。Z 轴扫描存在较多的尖峰，是由于其内存访问步长为 100 X 100 X (data size)，造成较差的空间局部性，也就存在相当高的缓存未命中率。

考虑最坏的情况，当步长为一个较大的 2 的 x 次方（比缓存空间大），Z 方向上的数据集合都映射到同一个缓存行上（“抖动”），导致未命中率达到 100%。针对这种情况就可以填充数组，避免这种特殊情况出现。

2) 循环交换

在 Z 方向上扫描时，TLB 未命中消耗了大量时间，原因与上述相同，都是过大的内存访问步长，超过了内存中一个页的大小（16KB），也就造成一页中只有一个目标数据，当访问的次数超过 64 后（TLB 中有 64 个条目），其中就必然存在不同程度的“抖动”。

针对这种情况的优化，在延 Z 方向上扫描的三层循环中，可以将最内层循环（Z 元素），与外层循环（X、Y 元素）进行不同程度的交换（一个元素 ~ 所有元素）。其可以达到的效果就是在 Z 方向上扫描时，可以掺入部分对其他垂直方向的扫描（Z 值相同）。这样就可以提升程序的空间局部性。但由于工作集的增大，也会造成其缓存的时间局部性降低。

3) 循环融合

在原本的程序中，共存在三个三层循环分别扫描 X、Y、Z 三个方向，这里采用的优化方法是：将 X、Y 方向上的扫描合并为一个，也就变成对与 Z 轴垂直的面的扫描（顺序不变先 X 后 Y）。这样做的原因是 X 方向扫描时会在缓存中载入大部分 Y 方向扫描时所要用到的数据，这也就能明显的减少 Y 方向扫描所耗费的时间。

采用上述的优化方法之后，得到的结果如图 4 所示。

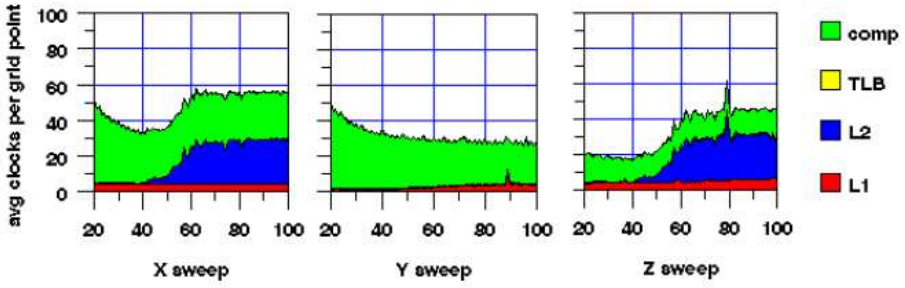



图 4.

## 4.3 多核处理器数据共享优化

这里会涉及到共享缓存的一致性处理，性能计数器使用**阻塞请求**和**无效请求**事件对其进行表示。这两个事件的含义就是在不同的处理器专用缓存中，存在同一个数据块，当数据在一个专用缓存中被修改时，就需要这两个请求事件对其他专用缓存中的相同数据块进行处理。

接下来看案例，程序在单核处理器中有较好的性能，但扩展到多核处理器后，性能反倒变差。扩展后程序的具体内容：每个处理器都存在独立的工作空间，最后会对单个处理器核内的计算结果累加。表面上，应用采用并行计算，共享程度非常低。

分析过程如下：

1. Perfex 进行程序级的性能分析，发现大部分运行时间消耗在二级缓存未命中事件上。
2. Perfex 针对二级缓存、阻塞请求和无效请求在不同数量的核的处理器中进行测试。结果见图 6.。可以发现两者存在较高的相关性，也就可以断定该程序存在很高程度的处理器核间数据共享。
3. SpeedShop 进行程序行级的性能分析，发现事件集中在累加阶段。

现在可以断定程序的问题是存在伪共享（false sharing），继续分析发现，程序中每个核的存储累加值的变量是连续分配地址空间的，也就表明多个累加变量是映射到同一个缓存行的。单个数据独立，缓存块不独立。

优化方法：将累加变量按照缓存块的大小对齐，优化结果见图 6.。

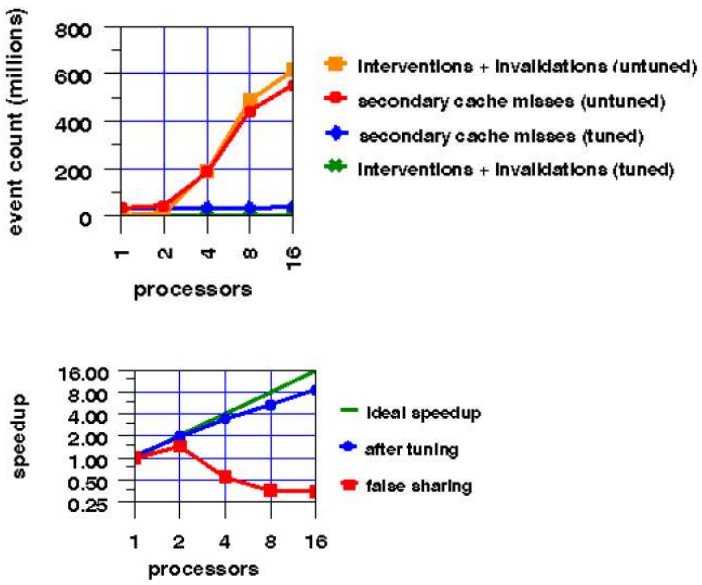



图 6.

## 4.4 分支预测优化

该案例程序是用一个 3 X 3 的滤波器对图像进行模糊处理。分析步骤如下：

1. Perfex 程序级分析，发现存在较多的分支预测错误事件，见图 7.。
2. Perfex 对分支预测失败事件进行统计，34%（接近 1/3）的概率，从这结果也就可以看出，条件分支指令的预测全部选择了分支。分支预测算法失效（两位表示分支历史），数量为 3 的循环次数，导致预测状态始终不会改变。

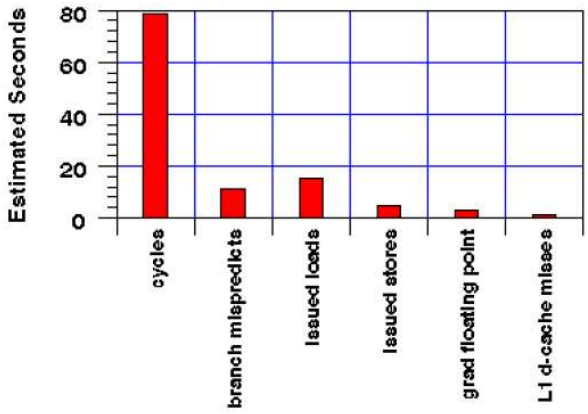
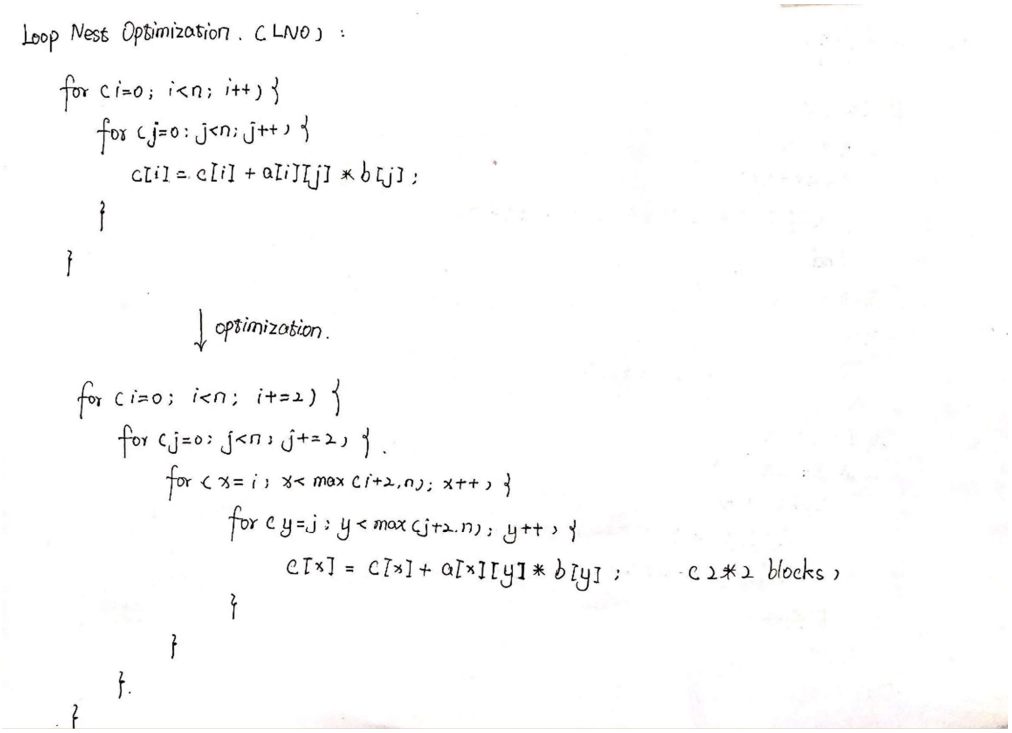



图 7.

这种情况下的优化可以交给编译器来完成，需要添加选项，使能 LNO (loop nest optimization) 和 IPA (inter-procedural analysis)。

LNO：能够交换内外层的循环，将原本的 3 X 3 尺寸转换到图像尺寸。同时“unrolls and jams”（拆解和合并）外层循环，增加一次迭代过程中处理的元素，不同元素的处理过程可能存在公共的内存引用，也就可以减少内存加载次数，说明例子见下图 8.。



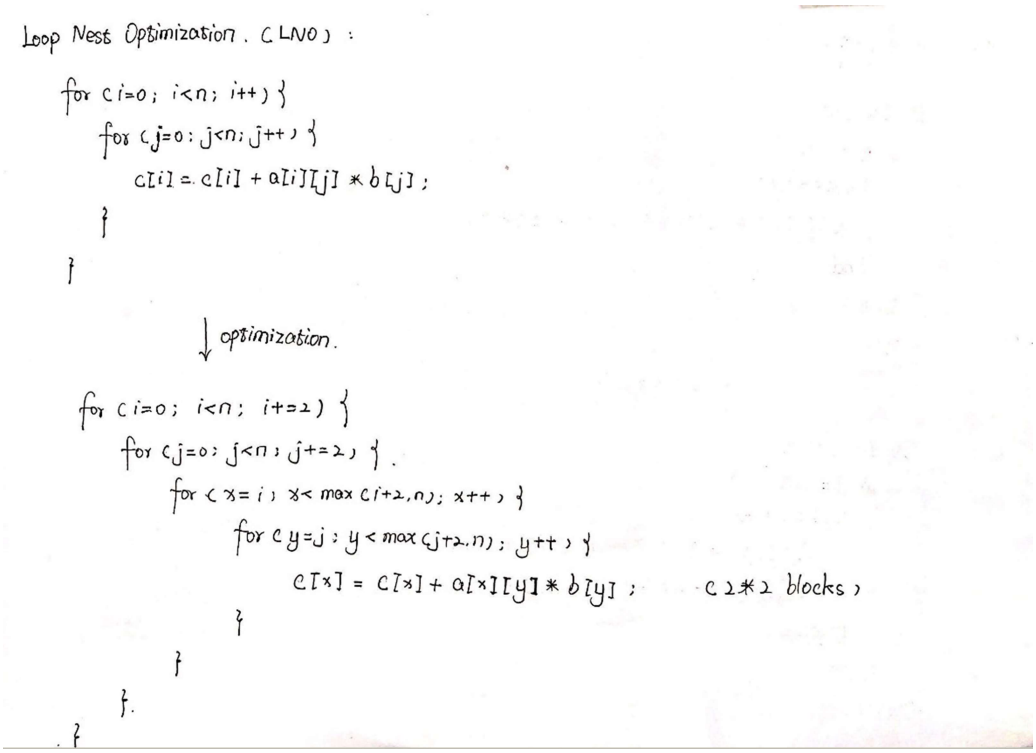



图 8.

IPA：将变量值 3 传入当前处理程序（当前处理程序能适应所有尺寸），也就给予了编译器优化的可能（原本未知尺寸，编译器无法对循环做出调整）。

最后优化结果见图 9.

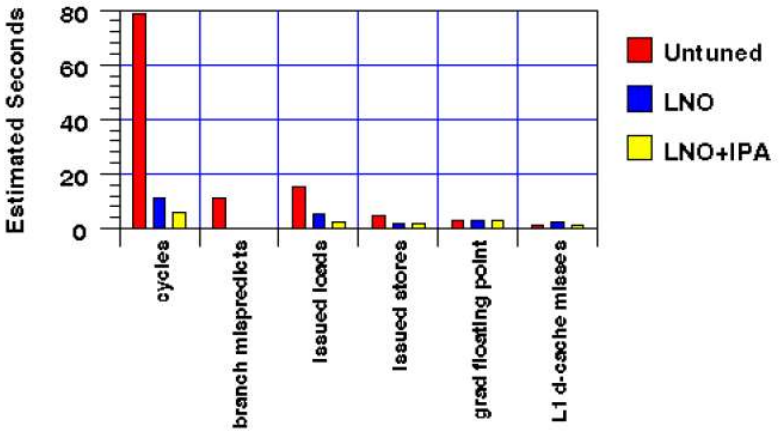



图 9.