一种基于微基准程序和理想上限的处理器性能分析方法

马 可[1,2], 章隆兵[2]
(1. 中国科学技术大学计算机科学技术系, 安徽合肥 230027; 2. 中国科学院计算技术研究所, 北京 100080)

摘 要: 随着现代高性能通用处理器结构的不断发展, 处理器的性能分析已经变得越来越困难. 基于大工作负载和单纯依靠模拟器的性能分析方法复杂度高, 且难以直观地反映微体系结构特征. 本文针对超标量处理器的特点, 提出一种新的处理器性能分析方法, 具体包括: 设计一个微基准程序集 Godson-Microbench, 并提出相应的理想性能上限计算公式. 这种方法扩充了过去基于约束的性能分析方法, 可以更加完整地评估流水线性能并有效地发现性能瓶颈. 本文使用这种方法分析比较了龙芯 2 号处理器和 Alpha21264 处理器, 并依此改进了龙芯 2 号的结构, 使得微基准程序的平均性能提高了 13.8%, SPEC 程序集的 IPC 提高了 28.8%. 本文提出的性能分析方法在龙芯 2 号的结构优化工作中发挥了重要作用.

关键词: 性能分析; 微基准程序; 理想上限; 龙芯 2 号处理器; Alpha21264 处理器
中图分类号: TP332 文献标识码: A 文章编号: 0372-2112 (2008) 02-0350-08

A Microprocessor Performance Analysis Approach Based on Microbench and Ideal Bound

MA Ke[1,2], ZHANG Long-bing[2]
(1. *Department of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, China;*
2. *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, China*)

Abstract: The performance analysis process of microprocessor has become more and more difficult with the rapid development of modern processor architecture. The traditional performance analysis approach based on macro benchmarks and simulator is complex and time-consuming. Moreover, it can not analyze the micro-architectural characteristics systematically. In this paper, a novel performance analysis approach based on micro-benchmark (Godson-Microbench) and ideal-bound computation formulas is proposed. This approach extends the bounds based performance analysis approach. Compared with the traditional approach, this approach can be used to analyze the pipeline efficiency and identify the performance bottleneck. With this approach, the micro-architecture of the Godson-2 processor and Alpha21264 processor are analyzed and compared. On the basis of this work, we improve the architecture of Godson-2, resulting a 13.8% increase in performance of the micro-benchmarks and 28.8% increase in IPC of the SPEC CPU2000. This performance analysis approach is a complement of the cycle-accurate simulation and plays an important role in the architectural optimization process of Godson-2.

Key words: performance analysis; micro-benchmark; ideal bound; Godson-2 processor; Alpha21264 processor

1 引言

随着超标量处理器结构的不断发展, 处理器的性能分析工作已经变得越来越复杂. 这一方面是由于对新的结构缺少细致而又系统的分析方法, 另一方面也是因为微体系结构部件之间存在着紧密的联系, 各部件综合地作用于处理器性能[1]. Pentium 4 的性能分析使用了来自 SPEC CPU2000 的 550 组踪迹[2], 在结构级模拟器上完整运行一组 Reference 规模的踪迹需要数十小时, 而分析工作又需要性能调试、结构设计、程序分析、编译等多个团队的协作进行, 所以周期长且复杂度高.

性能分析工作的核心是工作负载的设计以及处理器性能模型的建立. 大数据集驱动的实际工作负载(如 SPEC CPU[3]、IBS[4] 和 SPLASH[5] 等) 可以用于评估处理

收稿日期: 2007-03-09; 修回日期: 2007-06-06
基金项目: 国家杰出青年基金(No. 60325205); 国家自然科学基金(No. 60673146); 国家 863 高技术发展计划(No. 2006AA010201), 国家 973 重点基础研究计划(No. 2005CB321600); 中科院计算所知识创新课题(No. 20066012); 北京市自然科学基金(No. 4072024)

器在真实应用中的性能, 数值计算的基准程序(如 Livermore Fortran Kernels[6]、Whetstone[7]、Dhrystone[8] 和 Linpack[9]等) 可用于反映处理器的定点和浮点运算能力. 然而这些程序都无法直观反映各流水级对性能独立的影响. 结构级模拟器可以模拟处理器的结构细节, 统计性能分析数据. 然而单纯基于模拟的性能分析方法不够系统化, 需要丰富的相关经验才能够从统计数据中分析出结构存在的问题.

微基准程序通过循环执行使得流水线达到稳定状态, 进而独立评估各部件的性能. 本文定义处理器性能的理想上限为: 在达到结构设计目标和不考虑具体实现细节的情况下, 处理器所能够达到的性能[11]. 这里的处理器结构设计目标包括: (1) 指令执行流水化; (2) 各种队列的项数无限多; (3) 除 RAW 之外的其他相关都通过寄存器重命名等技术完全消除.

本文针对超标量处理器的特点, 提出了一种基于微基准程序和理想上限的性能分析方法, 具体包括: 设计了微基准程序 Godsorr Microbench, 并建立了计算理想性能上限的公式. 采用这种方法, 本文分析了龙芯 2 号和 Alpha21264 的性能, 并提出了结构改进的措施. 测试结果表明微基准程序在龙芯 2 号上的平均性能提高了 13.8%, SPEC CPU2000 的 IPC 提高了 28.8%.

2　相关工作

Mangione-Smith 等人[10] 基于约束的方法来分析 IBM RS/6000 处理器上 Livermore Fortran Kernels[6] 的 CPI, 并以此来优化 Fortran 编译器. 他们利用访存和浮点部件延迟及带宽的信息, 为有可能成为性能瓶颈的部件估算一次循环所需要的总延迟. 通过分析理想与实际性能之间差距产生的原因, 他们使用循环展开和指令调度等技术生成了更加优化的代码. 在此基础上, Bose[11] 等人提出了一种基于约束的性能模型. 他们使用这一模型来分析一组循环结构的浮点程序在 POWER 系列处理器上的性能, 从而发现硬件实现中未达到设计目标的性能缺陷, 并调整其结构. Bose 等人的工作依然着重于调整处理器的浮点性能, 主要分析了受限资源对性能的影响.

Desikan[12] 为 Alpha 21264 处理器编写了一个周期精确的 Sim-Alpha 模拟器. 为了更加有效的验证这一模拟器, 他编写了一组微基准程序来比较其在 Sim-Alpha 和 Compaq DS-10L 工作站上运行的结果, 以发现模拟器中存在的误差.

Singhal 等人[2]介绍了 Pentium 4 处理器的性能分析流程. Intel 使用了内部开发的微基准程序来调整 Pentium 4 的各种结构参数, 这些程序的代码并未公开. Intel 同时使用了 Lmbench[13] 来测试 Cache 延迟. Lmbench 是一组用于测试操作系统性能的程序集, 可以测试处理器和内存、网络、文件系统及磁盘之间数据传输的带宽和延迟.

Mangione-Smith 等人的工作主要分析了浮点访存和运算指令, 本文对龙芯 2 号的分支预测、定点和浮点运算以及访存性能都进行了分析. 对于并行度较低的微基准程序, 本文建立了根据依赖链计算理想性能上限的公式. 本文以 Desikan 编写的微基准程序为基础, 建立了一组微基准程序 Godsorr Microbench, 与 Lmbench 等现有的基准程序相比, Godsorr-Microbench 更着重于测试处理器微体系结构部分对性能的影响.

3　Godsorr Microbench 的设计

典型超标量处理器的流水线可以分为前端和后端, 前端负责取指、译码和寄存器重命名等工作, 后端负责指令发射、执行和写回等工作. 表 1 比较了五款现代通用处理器的结构, 根据各部件对流水线效率的影响, 我们基于如下考虑设计 Godsorr Microbench: (1) 影响前端流水线效率的主要因素是分支预测器的性能, 我们针对常用的分支预测技术设计了 C-Benchmark. (2) 后端流水线中, 指令执行部分反映了处理器的运算能力. 我们设计了 E-Benchmark 来分析功能部件资源的数量及延迟. (3) 由于处理器核心与内存的频率差距不断加大, 访存部分已经成为影响性能的主要因素. 指令 Cache 会影响前端的取指效率, Cache 的组织结构以及内存控制器的设计会影响后端执行访存指令的效率. 我们设计了 M-Benchmark 用于测试 Cache 对处理器性能的影响. 这三组程序构成了 Godsorr Microbench. 由于处理器的性能优化包括处理器结构改进和编译优化等多个方面, 同时考虑到可移植性, 我们使用 C 语言编写 Godsorr-Microbench.

表 1　现代通用处理器微体系结构比较

	Microarchitecture	Alpha21264	Power 5	MIPSR10000	Pentium 4	Opteron
控制部分	BHT(Branch History Table)	Local History Table 1,024×10 + Local Prediction 1,024×3 + Global Prediction 4,096×2 + Choice Prediction 4,096×2	Local Predictor 16K-entry Global Predictor 16K-entry Selector Table 16K-entry	512-entry x 2-bit	4K-entry	16K-entry x 2-bit
	BTB(Branch Target Buffer)	1K Line/Way Predictor	32-entry	-	4K-entry	2K-entry
	RAS(Return Address Stack)	32-entry	32-entry	4-entry	-	12-entry

执行部分	FXU(Fix-Point Unit)	4	2	2	3	3
	AGU(Address Generate Unit)	—	—	—	—	3
	LSU(Load-Store Unit)	2	2	1	2	2
	FPU(Floating-Point Unit)	2	2	2	2	3
	BRU(Branch Unit)	—	1	—	—	—
	Issue Width	6	5	5	6	8
访存部分	L1 Instruction Cache	64K 2-way 64-byte line	64K 1-way 128-byte line	32K 2-way 64-byte line	12K uops Trace Cache	64K 2-way 64-byte-block
	L1 Data Cache	64K 2-way 64-byte line	32K 2-way 128-byte line	32K 2-way 64-byte line	8K 4-way	64K 2-way 64-byte-block
	L2 Cache	2M 1-way	1.5M 8-way	512K~16M 2-way	256K 8-way	1M 16-way
	L1 Load-to-use Latency	3	4	2	2	3

3.1 E-Benchmark

E-I 和 E-F(图 1)分别由一系列彼此独立的整数和浮点加法运算组成. E-D_n 包含 6 个程序,这些程序由一系列前后相关的算术指令构成,n 表示前后相关的两条指令之间的距离. 这组程序用于测试功能部件分配策略和定点运算指令的延迟. E-IM 和 E-MA 分别由一系列彼此独立的乘法运算和前后相关的浮点乘加运算构成,用于测试定点乘法和浮点乘加部件的效率.

```
int i,k,l,m,n;
for (i){
k = k + i;
l = l - i;
m = m + i;
n = n - i;
}
```
(a)E-I

```
float k,l,m,n;
for (i){
k = k + i;
l = l - i;
m = m + i;
n = n - i;
}
```
(b)E-F

```
int i,k,l,m,n;
for (i){
k = n + i;
l = k + i;
m = l + i;
n = m + i;
}
```
(c)E-D_1

```
int i,k,l,m,n;
for (i){
k = k * i;
l = l * i;
m = m * i;
n = n * i;
}
```
(d)E-IM

```
float k,l,m,n;
for (i){
k = k * i;
k = k + l;
m = m * i;
n = m - n;
}
```
(e)E-MA

图 1 E-Benchmark 代码示例

3.2 C-Benchmark

C-C(见图 2)包含一条 if... else... 语句,每次循环交替执行两个分支. C-R 的循环体中包含一个深度为 1,000 的递归. C-S_n 包含 3 个程序,其中每个程序都具有一个 switch... case... 语句的循环体. 这里 n 是指在 switch 语句的反复执行中,每一个 case 语句块被连续执行的次数,用于测试 BTB 的性能. C-CO 程序可看作是 C-C 和 C-S 的结合,在 C-C 的两个分支里分别放置了 C-S2 和 C-S3.

```
int i,j,p,r;
for(i=0;i<1000;i++)
{
  j = i % 2;
  if (j == 0)
      p++;
  else
      r++;
}
```
(a)C-C

```
static int k = 0;
int i = 1000;
for (j=0;j<1000;j++){
  func(i,j);
}
void func(int l,int j)
{
  k = j + l;
  if (l == 0)
    return;
  else
    func(l-1, j);
}
```
(b)C-R

```
for (i=0;i<1000;i++) {
    j=i%10;
    switch (j) {
    case 0:
        k++;break;
    case 1:
        l++;break;
    ......
    case 9:
        n --;
    }
}
```
(c)C-S_1

```
if (i%2 == 0) {
  j++; j%=4;
  switch (j) {
    case 0,1:
      k++; break;
    case 2,3:
      l++;}
}else{
  m++; m%=6;
  switch (m) {
    case 0,1,2:
      n++; break;
    case 3,4,5:
      o++;}
}
```
(d)C-CO

图 2 C-Benchmark 代码示例

3.3 M-Benchmark

M-D 包含一系列前后相关的 Load 指令,每次取回的值做为下一次取数的地址. 图 3 中 $a[i]$ 数组中的每一个元素是指向下一元素的指针,核心循环中每次取得数组中的一个元素,并以此为地址取下一个元素,从而依次取回所有元素. 这个程序用于测试 Load-to-use(定点取数)延迟. 为避免一级 Cache 失效,核心循环的次数应为 $S_{L1\text{-}cache}/(Way_{L1\text{-}cache} \times \text{sizeof(int)})$(参数见表 3). M-DF 用于测试浮点取数延迟. M-L2 用于测试二级 Cache 的访问延迟. M-M 用于测试访问内存的延迟. M-IP 用来测试指令 Cache 的失效开销及预取能力. 这个程序包含 32K 条前后不相关的算术指令,占用 128KB 空间,在小于或等于 64KB 的指令 Cache 中将会频繁发生失效.

```
for(i=0;i<4095;i++)
a[i]=(int *)&a[i+1];
for (i=0;i<1000;i++) {
b=(int **)a[0];
for(k=1;k<4095;k++){
b=(int **) b[1];
  }
}
```
(a)M-D

```
for(i=0;i<131071;i++)
  a[i]=(int *)&a[i+1];
for (i=0;i<1000;i++) {
  b=(int **)a[0];
for(k=1;k<16383;k++){
      b=(int **) b[7];
  }
}
```
(b)M-L2

```
for(i=0;i<524287;i++)
a[i]=(int *)&a[i+1];
for (i=0;i<1000;i++) {
b=(int **)a[0];
for(k=1;k<65535;k++){
b=(int **) b[7];
  }
}
```
(c)M-M

```
for (i){
  k = k + i;
  l = l - i;
  m = m + i;
  n = n - i;
  o = o + j;
  p = p - j;
  q = q + j;
  r = r - j;
  ......
}
```
(d)M-IP

图 3 M-Benchmark 代码示例

4 理想上限分析公式

由于 Godson-Microbench 中各个程序的循环体代码都较为规则,所以在分析其理想 IPC 上限时,可以使用式(1)、(2)[11]:

$$CPI = cpI / N \tag{1}$$

$$cpI_{ideal} = \max(cpI_{fetch\text{-}bound}, cpI_{dispatch\text{-}bound}, cpI_{lsu\text{-}issue\text{-}bound}, cpI_{fxu\text{-}issue\text{-}bound}, cpI_{fpu\text{-}issue\text{-}bound}, cpI_{retire\text{-}bound}, \cdots) \tag{2}$$

其中,$cpI_{fetch\text{-}bound} = N / W_{fetch}$,$cpI_{dispatch\text{-}bound} = N / W_{dispatch}$,$cpI_{retire\text{-}bound} = N / W_{retire}$,$cpI_{lsu\text{-}issue\text{-}bound} = (N_L + N_S) / N_{lsu}$,$cpI_{fxu\text{-}issue\text{-}bound} = N_I / N_{fxu}$,$cpI_{fpu\text{-}issue\text{-}bound} = N_F / N_{fpu}$

这里 cpI 表示一次循环所需拍数. 式(2)反映出 cpI 的理想上限由各个流水级 cpI 上限的最大值决定. 其中 W_{fetch}、$W_{dispatch}$ 和 W_{retire} 分别是取指、发射和提交宽度,N_{lsu},N_{fxu} 和 N_{fpu} 分别代表访存,定点和浮点部件个数. N_L,N_S,N_I 和 N_F 分别表示 Load,Store,定点和浮点指令

数.

上述公式应用的前提是各指令之间不存在相关. 对于存在大量数据相关的程序可使用公式(3):

$$IPC = \frac{N}{\max_i\left(\sum_{j=1}^{n} L_j^i\right)} \tag{3}$$

若前后相关的一系列指令组成一条依赖链, 那么 n_i 表示循环中存在的第 i 条依赖链上的指令数, L_j^i 表示第 i 条依赖链上第 j 条指令的延迟. E-D_n 的理想 IPC 可以使用此公式计算.

如果在乱序执行的过程中发生了分支误预测或 Cache 失效, 处理器必须暂停流水线, 等待失效事件被处理后再继续执行. 研究人员指出[14] 失效事件间的重叠可以被忽略, 所以 *CPI* 由理想情况下可以达到的 CPI_{ideal}、分支误预测的开销 CPI_{brmisp}、数据 Cache 失效的开销 $CPI_{dcachemiss}$ 和指令 Cache 失效的开销 $CPI_{icachemiss}$ 这四个部分组成, 即

$$CPI = CPI_{ideal} + CPI_{brmisp} + CPI_{dcachemiss} + CPI_{icachemiss} \tag{4}$$

C-Sn 的理想 IPC 可以根据式(4) 来计算. C-S1 的核心循环包含三个部分. 第一部分用于计算 switch 语句跳转的目标地址. 由于前后指令之间都具有相关性, 根据式(3) 可知其 IPC 为 1. 第二部分包含一条间接跳转指令(switch) 和跳转表中的一项(case), 由于间接跳转指令每次的目标地址都和上次不同, 无法从 BTB 中得到正确预测, 这一部分代码的执行开销就是分支误预测的开销 C_{brmisp}. 第三部分也包含若干条前后相关的指令, IPC 近似为 1. 所以

$$IPC = \frac{N}{cpI} \leqslant \frac{N}{N + C_{brmisp}}$$

C-S_2 和 C-S_3 可用于测试 BTB 中饱和计数器的初始值(V_{init}) 和阈值($V_{threshold}$). 当 BTB 预测错误时, 饱和计数器的值减 1, 当这一值小于或等于 $V_{threshold}$ 时, 相应的 BTB 表项可以被更新, 更新后饱和计数器的值被设为 V_{init}. C-S_n 程序中的间接跳转指令(switch) 连续 n 次具有相同的目标地址. 设饱和计数器的最大值为 V_{max}, n 次分支预测中正确的次数为 $n_{correct}$, 则当分支目标地址改变时, 饱和计数器的值是 $V_{mispredict} = \min(V_{max}, V_{init} + n_{correct})$. 这样分支预测错误 $\max(V_{mispredict} - V_{threshold} + 1, 1)$ 次后, 饱和计数器的值才会被更新. 所以

$$\begin{aligned} n_{correct} &= n - \max(V_{mispredict} - V_{threshold} + 1, 1) \\ &= n - \max(\min(V_{max}, V_{init} + n_{correct}) - V_{threshold} + 1, 1) \end{aligned}$$

对于 C-S_n, V_{init} 越小, $V_{threshold}$ 越大, 分支误预测率就会越小. 当 n 较大时, 分支误预测率仅和 $V_{threshold}$ 相关.

M-Benchmark 的理想 IPC 可以根据式(4) 和不同访存层次的延迟来计算:

$IPC_{M-D} = N / L_{L1\text{-}data\,cache}$, $IPC_{M-L2} = N / L_{L2-cache}$,

$IPC_{M\text{-}M} = N / L_{memory}$

$$\begin{aligned} IPC_{M\text{-}IP} &= 1/CPI = 1/(CPI_{ideal} + CPI_{icachemiss}) \\ &= 1/(1/W_{fetch} + C_{icachemiss}/(S_{icache\,black} \cdot (N_{prgfetch} + 1)/L_{inst})) \end{aligned}$$

其中 $L_{L1\text{-}DATA\text{-}cache}$, $L_{L2\text{-}CACHE}$ L_{memory} 分别代表一二级 Cache 和访存的延迟, $S_{icache\,black}$ 是指令 Cache 的块大小, $N_{prgfetch}$ 是指令 Cache 预取的块数, L_{inst} 是指令长度.

表 2 列出了各个 *Godson-Microbench* 的理想上限分析公式.

表 2　Godson-Microbench 理想上限分析公式

Bench	Ideal Performance Computation Formula	Bench	Ideal Performance Computation Formula
E-I	$CPI = \max(cpI_{fetch-bound}, cpI_{dispatch-bound}, cpI_{fxu\,issue-bound}, cpI_{retire-bound})/N = \max(\lceil 1/W_{fetch}\rceil, \lceil 1/W_{dispatch}\rceil, \lceil N_I/N \cdot N_{fxu}\rceil, \lceil 1/W_{retire}\rceil)$	C-R	$CPI = \max(\lceil N/W_{fetch}\rceil, \lceil N/W_{dispatch}\rceil, \lceil N_{immejmp} \cdot L_{immejmp}/N_{bur} + N_{indirjmp} \cdot L_{indirjmp}/N_{bru}\rceil, \lceil N/W_{retire}\rceil)/N$
E-F	$CPI = \max(\lceil N/W_{fetch}\rceil, \lceil N/W_{dispatch}\rceil, \lceil N_F/N_{fpu}\rceil, \lceil N/W_{retire}\rceil)/N$	C-S1	$CPI \approx \frac{N + C_{brmisp}}{N}$
E-D1	$CPI = 1/IPC = \frac{\max_i\left(\sum_{j=1}^{n_i} L_j^i\right)}{N} = \frac{\sum_{j=1}^{N} L_{add}}{N} = L_{add}$	C-Sn	$CPI \geqslant \frac{cpI_{part1} + cpI_{part2} + cpI_{part3}}{N} = (\lceil (N_{part1} + N_{part3})/\min(n_{correct}, W_{fetch})\rceil + \lceil (C_{brmisp} \cdot (n - n_{correct}) + L_{indirectjmp} \cdot n_{correct})/n\rceil)/N$
E-Dn	$CPI = \max(\lceil N/N_{fxu}\rceil N/n \cdot L_{add})/N = \max(\lceil 1/N_{fxu}\rceil L_{add}/n)$	M-D	$CPI = L_{L1\text{-}dtatcache}/N$
E-IM	$CPI = \max(\lceil N_{fpmul}/N_{multfpu}, N_I/N_{fxu}\rceil)/N$	M-L2	$CPI = L_{L2\text{-}ache}/N$
E-MA	$CPI = \max(\lceil N_{fpadd}/N_{addfpu}\rceil, \lceil N_{fpmul}/N_{multfpu}\rceil)/N$	M-M	$CPI = L_{memory}/N$
G-C	$CPI = \max(\lceil N/W_{fetch}\rceil, \lceil N_I/N_{fxu}\rceil)/N$	M-IP	$CPI = CPI_{ideal} + CPI_{icachemiss} = 1/W_{fetch} + C_{icachemiss}/(S_{icache\,black} \cdot (N_{prgfetch} + 1)/L_{inst})$

5　性能测试及分析

基于微基准程序和理想上限公式的性能分析流程如图 4 所示. 首先分析微基准程序的特征, 得到各种指令的条数和相关关系, 分支转移模式和访存行为. 然后分析处理器结构得到各流水级宽度, 功能部件个数, 分支预测器结构以及 Cache 延迟等参数. 再使用分析公式计算理想性能. 另一方面, 微基准程序作用于模拟器可得到实际性能. 最后通过比较理想和实际性能可以发现结构设计中存在的问题.

本文分别在 Sim-Alpha[12] 和 ICT-Godson 模拟器[15] 上运行 Godson-Microbench, 并统计性能数据. Sim-Alpha

模拟了 Alpha 21264[16] 的结构细节, 并与实际硬件进行了对比校验[17]. 龙芯 2 号[18] 是中科院计算所研制的高性能通用处理器, 具有 64 位 4 发射的超标量结构, 实现了寄存器重命名、分支预测和动态调度等乱序执行技术, 以及非阻塞 Cache 和 Load-speculation 等动态存储访问机制. ICT-Godson 是龙芯 2 号的信号级模拟器. Sim-Alpha 和 ICT-Godson 都提供性能统计功能.

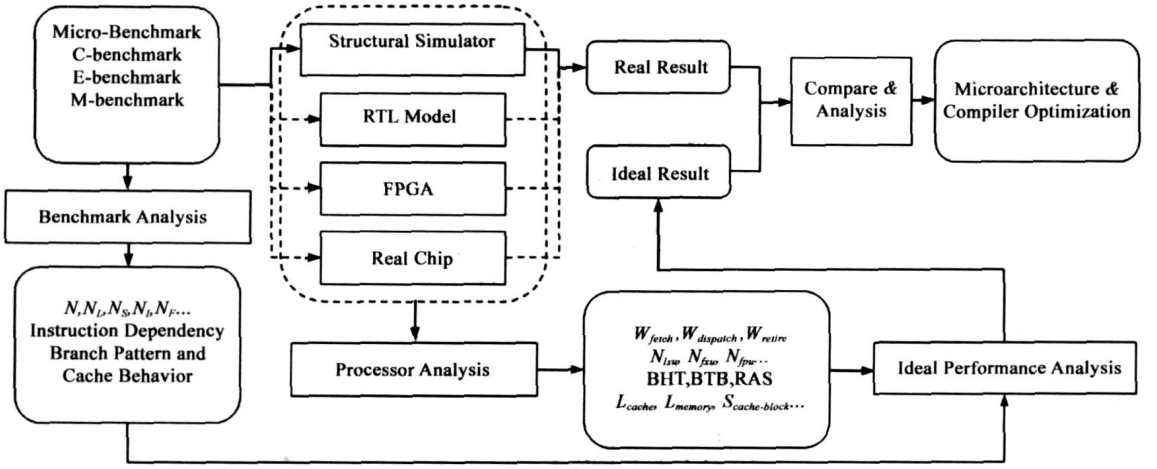

图 4 基于微基准程序和理想上限公式的性能分析流程

表 3 Sim-Alpha 和 ICT-Godson 配置参数

	Sim-Alpha	ICT-Godson		Sim-Alpha	ICT-Godson
取指宽度(W_{fetch})	4	4	发射宽度($W_{dispatch}$)	6	5
提交宽度(W_{retire})	11	4	定点功能部件数(N_{fxu})	4	2
浮点功能部件数(N_{fpu})	2	2	访存功能部件数(N_{lsu})	2	1
分支功能部件数(N_{bru})	1	1	加法指令延迟(L_{add})	1	1
直接跳转指令延迟($L_{immejmp}$)	4	1	间接跳转指令延迟($L_{indirjmp}$)	7	1
BTB	V_{init}= 0, $V_{threshold}$= 2	V_{init}= 3, $V_{threshold}$= 0			
一级 Cache(数据和指令 Cache)					
大小($S_{L1\text{-}cache}$)	64KB	64KB	块大小($S_{L1\text{-}block}$)	64bytes	32bytes
相联度($Way_{L1\text{-}cache}$)	2	4	取数延迟($L_{L1\text{-}datacache}$)	3	4
二级 Cache					
大小($S_{L2\text{-}cache}$)	2MB	1MB	块大小($S_{L2\text{-}block}$)	64byte	32byte
相联度($Way_{L2\text{-}cache}$)	1	2	取数延迟($L_{L2\text{-}cache}$)	15	11
DRAM 模块					
预充电延迟(T_{rp})	1	3	列选中延迟(CL)	1	3
行选中延迟(T_{rcd})	1	3	一次读取字节数(S_{data})	64	32
一次传输数据的宽度(W)	16	8	DRAM 的倍频(M_{clock})	6	6
传输速率(R_{data})	1	1	页访问策略	Open Page	Closed Page
请求响应延迟(L_{req})	2	2	数据读延迟(L_{return})	2	2

表 3 列出了测试过程中模拟器的参数配置. 表 4 列 出了实测结果和理想上限分析结果.

表 4 Godson-Microbench 的测试及分析结果

Godson MicroBench	Sim-Alpha				ICT-Godson			
	Real IPC	Program Characteristic	Ideal IPC	改进策略	Real IPC	Program Characteristic	Ideal IPC	改进策略
E-I	3.96	N= 8, N_I= 8	4		1.99	N= 8, N_I= 8	2	
E-F	1.00	N= 8, N_F= 8	1		1.01	N= 8, N_F= 8	1	
E-D1	1.04	N= 8	1		1.03	N= 8	1	
E-D2	2.02	N= 8	2		1.99	N= 8	2	

E-D3	2.52	N= 24	3	功能部件动态分配策略	2.00	N= 24	2	
E-D4	3.04	N= 8	4		1.99	N= 8	2	
E-D5	3.36	N= 40	4		2.00	N= 40	2	
E-D6	3.23	N= 24	4		2.00	N= 24	2	
E-IM	1	N= 8, N_{fpmult}= 8	1		1.33	N= 2, N_{fpmult}= 1	2	-m4010 选项/空操作提前写回技术
E-MA	1.99	N= 8, N_{fpadd}= 4, N_{fpmul}= 4	1		1.99	N= 8, N_{fpadd}= 4, N_{fpmul}= 4	2	
C-C	1.87	N= 7.5	1.89		1.05	N= 21	1.96	Likely 分支预测技术/功能部件动态分配策略
C-R	1.00	N= 11	1	分支执行流水化	1.70	N= 12	2	
C-S1	0.73	N= 30, N_{part1}= 27, N_{part3}= 3	0.75	分支提早恢复机制	0.77	N= 27, N_{part1}= 17, N_{part3}= 6	0.79	
C-S2	1.07	N= 30	1.20		0.78	N= 27	1.23	BTB 更新策略
C-S3	1.05	N= 30	1.07		0.93	N= 27	0.96	
C-CO	2.00	N= 42	2		1.16	N= 19.25	1.14	局部分支预测技术
M-D	1.33	N= 4	1.33		0.75	N= 3	0.75	
M-DF	1.74	N= 7	1.75		1.25	N= 5	1.25	
M-L2	0.27	N= 4	0.27		0.28	N= 3	0.27	
M-M	0.05	N= 4	0.05		0.02	N= 3	0.02	北桥配置优化/动态 page 管理策略
M-IP	1.77	N= 80	2.35		1.04	N= 80	1	

6　性能数据比较与结构优化

通过比较 Godsorr Microbench 在龙芯 2 号上的理想和真实性能, 我们发现可以改变编译选项来改进代码以提高执行效率, 并改进了 BTB 更新策略, 优化了内存控制器配置, 提出了空操作提前写回技术、Likely 分支预测技术、功能部件动态分配策略. 这些策略使得 Godsorr Microbench 在龙芯 2 号上的 IPC 平均提高了 8.1% (图 5), 性能平均提高了 13.8% (图 6).

(1) 空操作提前写回技术

E-IM 的理想 IPC 高于实测值, 这是由于早期的 MIPS 处理器采用静态流水, 编译器会在相邻的 Mflo 和 Mult 指令之间插入两条 Nop 指令. 然而龙芯 2 号采用了动态流水线技术, 所以无需插入空操作. 使用-m4010 选项可以控制编译器不产生这两条 Nop 指令. 此外, 由于 Nop 指令并不需要被执行, 而龙芯 2 号却将 Nop 指令发射至定点功能部件, 所以浪费了执行周期. 我们使用一种称为"指令提前写回"的技术在译码阶段对 Nop 指令加以特殊标记, 并在重命名阶段将其直接写回到 Reorder Buffer, 从而提高 Nop 指令的执行效率. 这两种方法都可以将 E-IM 的 IPC 提高为 2, 执行时间减少了 33.33%.

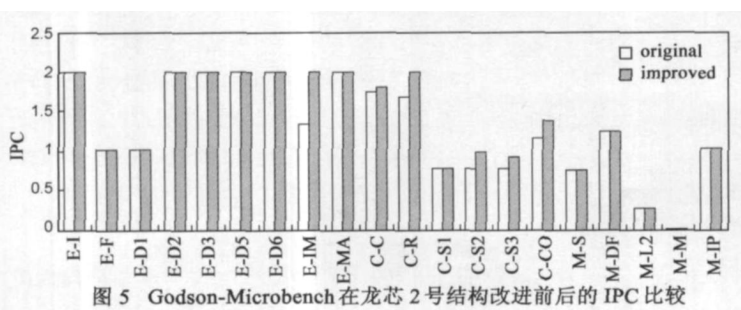

图 5　Godson-Microbench 在龙芯 2 号结构改进前后的 IPC 比较

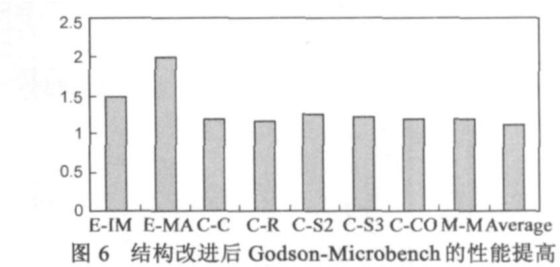

图 6　结构改进后 Godson-Microbench 的性能提高

(2) Likely 分支预测技术和功能部件动态分配策略

C-C 在龙芯 2 号上的分支误预测率较高. 这是由于 C-C 根据迭代变量的奇偶性来决定是否转移, 故指令 4 (见图 7*b*) 跳转的概率为 50%. 然而 GCC 默认情况下会编译出 Branch Likely 指令[19], 龙芯 2 号对于 Branch Likely 指令都猜测跳转, 会较为严重的影响到分支预测正确率. 解决这一问题有两个途径:

(*a*) 编译时加入-mno-branch-likely 选项, 编译器将不产生 Branch Likely 指令, 实测得到的 IPC 为 1.75.

(*b*) 修改分支预测器, 对 Likely 指令也进行分支预测. 修改后 C-C 的 IPC 提高为 1.80.

由于龙芯 2 号在重命名阶段根据上一次分配的功能部件来决定本次要分配的功能部件, 然而最近被分配未必就是最近有指令在使用, 所以在执行 C-C 程序时会出现指令就绪却无法发射的情况. C-C 循环开始处计算 $i/2$ 的几条相关指令被交错分配到 ALU1 和 ALU2 两个运算部件, 但这几条指令必须串行执行, 所以分配策略失去了其本来的依据. 由于分支指令必须由 ALU1 来执

```
# for (i=0;i<1000;i++) {
#     j = i%2;
loop: and   t1,0x1,t3      1
      blt   t1, L2         2
      unop                 3
      unop                 4
#     if (j==0)
      bne   t3,L0          5
#         p++;
      addl  a0,0x1,a0      6
#     else
      br    L1             7
      unop                 8
#         r++;
L0:   addl  s0,0x1,s0      9
L1:   addl  t1,0x1,t1      10
      cmple t1,t2,t0       11
# }
      bne   t0,loop        12
```

(a)

```
# for (i=0;i<1000;i++) {
#     j = i%2;
loop:   addu   v0,v1,v0      1
        sra    v0,v0,0x1     2
        sll    v0,v0,0x1     3
#   if (j<>0)
        bnel   v1,v0,L0      4
#       r++;
        addiu  s0,s0,1       5
#   else
#       p++;
        addiu  a0,a0,1       6
# i++)
L0:     addiu  v1,v1,1       7
# i<1000;
        slt    v0,a1,v1      8
# }
        beqz   v0,loop       9
        srl    v0,v1,0x1f    10
```

(b)

图 7 C-C 核心循环的汇编代码 (a)Alpha 体系结构；(b)MIPS 体系结构

行, 所以如果将这些运算指令都分配到 ALU2, 就可以更加高效的执行 G-C. 我们依此提出了“功能部件动态分配策略”, 根据发射队列中功能部件已经被指定的指令条数来决定每条指令应分配到的功能部件. 这样 G-C 程序的性能可以提高 9%.

(3) 改进的 BTB 更新策略

G-S_2 和 G-S_3 的实测 IPC 低于理想值, 这是由于其分支误预测率很高. 经过分析可知, 龙芯 2 号的 BTB 更新策略仅适合于跳转指令的目标地址出现某一值的概率较大的程序. BTB 的 V_{init} 和 $V_{threshold}$ 分别被设置为 3 和 0, 使得 G-S_2 和 G-S_3 中所有的 Jr 指令都无法得到正确的分支预测. 我们令 $V_{threshold}=2$, $V_{init}=0$, 则 G-S_2 和 G-S_3 的 IPC 分别提高到 1.00 和 0.93.

(4) 局部与全局分支预测技术

龙芯 2 号使用 GShare[20] 算法进行分支预测. 这种预测机制对于 G-CO 中 switch 语句处的分支预测正确率不高. 这两处分支自身的转移模式很有规律, 但全局分支历史却很复杂. switch 语句的转移情况见表 5(a), 由于两次循环中 switch(j) 和 switch(m) 各被执行一次, 所以在表中以 $i/2$ 作为索引.

由于 G-CO 一次循环中的分支指令平均为 5 条, 而龙芯 2 号中最近的 9 次分支转移情况会影响分支指令在 BHT 中的索引, 所以 switch 语句处的分支指令会受到前面两次循环中的分支执行情况的影响. 表 5(b) 列出了 switch(m) 处的全局分支历史, 表中按照分支历史分组, 同组中的 switch(m) 最近两次分支历史相同. 可见具有相同分支历史时, 3 次执行 switch(m) 语句是否转移却并不一致. 由于龙芯 2 号中 BHT 表项含有一个 2 位的饱和计数器, 所以三次分支预测中将会有两次正确, 误预测率为 33.3%. 我们可以使用局部历史表和全局预测表[16] 分别考虑局部和全局分支历史对分支的影响, 从而使用较小的硬件代价对 G-CO 做出更好的预测. 在使用这一分支预测技术后, G-CO 的 IPC 可以提高为 1.41, 运行时间减少了 9.74%.

表 5 G-CO 中 switch 语句分支转移情况

(a) switch(j) 和 switch(m) 的转移情况;

(b) switch(m) 的转移情况及分支历史

i/2	switch(j)	switch(m)
0	1	1
1	1	1
2	0	1
3	0	0
4	1	0
5	1	0
6	0	1
7	0	1
8	1	1
9	1	0
10	0	0
11	0	0

(a)

i/2	分支历史	是否转移
0	0001	1
4	1001	0
5	0101	0
1	0111	1
8	1011	1
9	1111	0
2	1110	1
3	1010	0
7	0010	1
6	0100	1
10	1100	0
11	0000	0

(b)

(5) 北桥配置优化和动态 Page 管理策略

龙芯 2 号通过 SysAD 总线访问北桥并控制 DRAM 的读写. DRAM 的访问延迟仅为 14 拍, 但访存延迟为 161 拍, M-M 的 IPC 仅为 0.019. 访存效率不够理想一方面是由于访存请求需要经过北桥的缓冲和仲裁; 另一方面是内存控制器的配置未达到最优化. 我们可以在操作系统初始化时打开 Open Page 模式, 并降低相关延迟和 DRAM 的刷新频率, 此时 M-M 的 IPC 提高到 0.023. 为进一步提高访存性能, 龙芯 2E 在片上集成了内存控制器, 同时使用了内存设备的动态 Page 管理策略, 极大地提高了性能.

6.1 性能改进效果

为验证改进策略对真实程序的影响, 我们分别在改进前后的模拟器上运行了 Reference 规模的 SPEC CPU2000 程序集*, IPC 平均提高 28.84%(见图 8). 由于访存部分是龙芯 2 号的瓶颈, 访存优化对性能的影响非常明显(8.29%), Likely 分支预测技术也将 IPC 提高了 3.73%. 改进的 BTB 更新策略将 IPC 提高了 14.04%, 说明原有的 BTB 饱和计数器设置并不合理. 功能部件动态分配策略对性能的影响较小, 这是由于 Cache 失效和分支误预测掩盖了对分配策略优化的效果.

7 结论和未来工作

本文使用 Godsorr-Microbench 和理想性能上限公式评估了龙芯 2 号各个部分的性能, 发现了结构中存在的性能瓶颈. 与传统的单纯基于模拟的处理器性能分析方法相比, 本文提出的性能分析方法能够独立考察处理器各个部件对微基准程序性能的影响, 从而有效地发现优化结构部件的方法, 降低性能分析的复杂度.

* 模拟时采用先快速功能模拟 20 亿条指令以跳过程序初始化阶段, 再详细模拟 5 亿条指令的方法.

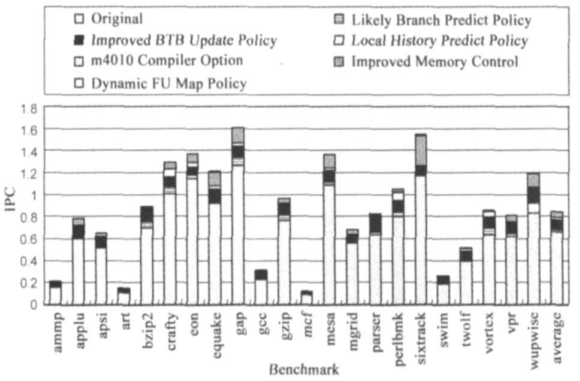

图 8　性能优化对 SPEC CPU2000 IPC 的影响

基于微基准程序和理想上限的性能分析方法不仅可用于周期精确的模拟器, 也可以用于处理器的 RTL 仿真平台和实际硬件, 与硬件性能计数器相结合, 发现物理实现中未能达到性能设计预期的部分. 本文提出的性能分析方法可以进一步编写成为理想性能评估工具, 作为龙芯处理器性能分析平台的一个组成部分, 辅助设计人员更加快捷而准确地完成性能分析工作.

参考文献:

[1] P Bose, T M Conte. Performance analysis and its impact on design[J]. IEEE Computer, 1998, 31(5): 41- 49.

[2] R Singhal, K S Venkatraman, E R Cohn. Performance analysis and validation of the Intel Pentium 4 processor on 90nm technology[J]. Intel Technology Journal, 2004, 8(1): 1- 17.

[3] The Standard Performance Evaluation Corporation. SPEC CPU 2000[S/OL]. http://www.spec.org/cpu2000/, 2005-11-01/2006-08-18.

[4] R Uhlig, D Nagle, T Mudge, S Sechrest, J Emer. Instruction fetching: coping with code bloat[A]. Proceedings of the 22nd Annual International Symposium on Computer Architecture [C]. Santa Margherita Ligure, Italy: ACM Press, 1995. 23(2): 345- 356.

[5] J P Singh, W Weber, A Gupta. Splash: Stanford parallel applications for shared memory[J]. ACM Computer Architecture News, March 1992, 20(1): 5- 44.

[6] F H McMahon. The Livermore Fortran Kernels: a computer test of the numerical performance range[R]. California: Lawrence Livermore National Laboratory, 1986. UCRL-53745.

[7] H J Curnow, B A Wichmann. A synthetic benchmark[J]. Computer Journal, 1976, 19(1): 43- 49.

[8] R P Weicker. Dhrystone benchmark program[J]. Communications of the ACM, 1984, 27(10): 1013- 1030.

[9] J J Dongarra. Performance of various computers using standard linear algebra software in a Fortran environment[J]. ACM SIGARCH Computer Architecture News, 1983, 11(5): 22- 27.

[10] W Mangione-Smith, T P Shih, S G Abraham, E S Davidson. Approaching a machine-application bound in delivered performance on scientific code[A]. Proceedings of the IEEE[C]. New York: ETATS-UNIS, 1993. 81(8): 1166- 1178.

[11] P Bose, S Kim, F O' Connell, W A Ciarfella. Bounds-based loop performance analysis: application to validation and tuning [A]. Proceedings of IEEE International Performance, Computing and Communication Conference[C]. Tempe/Phoenix, AZ, US: IEEE Press, 1998. 178- 184.

[12] R Desikan, D Burger, S Keckler, T Austin. Sim-alpha: a validated execution driven alpha 21264 simulator[R]. Austin: Department of Computer Sciences, University of Texas, Technical Report TR-01-23, 2001.

[13] L W McVoy and C Staelin. Lmbench: portable tools for performance analysis[A]. Proceedings of USENIX 1996 Annual Technical Conference[C]. San Diego, CA: USENIX Association, 1996. 279- 294.

[14] T S Karkhanis, J E Smith. A first-order superscalar processor model[A]. Proceedings of the 31st Annual International Symposium on Computer Architecture[C]. Munich, Germany: IEEE Computer Society, 2004. 338- 349.

[15] F X Zhang. Performance analysis, Optimizations of Microprocessors[D]. Beijing: Institute of Computing Technology, Chinese Academy of Sciences, 2006.

[16] SimpleScalar LLC. The Simplescalar Tool Set[EB/OL]. http://www.simplescalar.com, 1997-07-10/2006-08-18.

[17] R E Kessler. The Alpha 21264 microprocessor[J]. IEEE Micro, 1999, 19(2): 24-36.

[18] W W Hu, F X Zhang, Z S Li. Microarchitecture of the Godson-2 processor[J]. Journal of Computer Science and Technology, 2005, 20(2): 243- 249.

[19] D Sweetman. See Mips Run[M]. San Francisco: Morgan Kaufmann Publishers, 1999.

[20] S McFaring. Combining branch predictors[R]. Sydney, Australia: Digital Equipment Corporation, Western Research Lab, June 1993. Technical Report TN-36.

作者简介:

马　可　男, 1980 年生于湖北, 中国科学技术大学计算机科学技术系博士研究生. 研究方向为计算机系统结构、微处理器性能分析与结构设计. E-mail: kma@ict.ac.cn

章隆兵　男, 1974 年生于安徽, 博士, 副研究员. 研究方向为微处理器设计, 系统结构, 机群计算等.