计算机体系结构gem5实验二实验报告

PB21111681 朱炜荣

一、在不同的配置下运行benchmark

应对不同的配置，我选择使用的是gem5自带的se.py配置文件，通过阅读se.py --h中对于不同配置定义的方法，我们可以得到下面为定义各个计算机系统配置使用的命令行

1.由于各个issue-width默认值即为8，所以并没有在命令行参数中得到体现

```
build/X86/gem5.opt configs/deprecated/example/se.py --cmd=../LAB/lab2/lab2-benchmark/xx --cpu-type=DerivO3CPU --mem-type=DDR3_1600_8x8 --cpu-clock=1GHz  --caches --l1d_size=64kB --l1i_size=64kB
```

2.修改cpu的类型为MinorCPU

```
build/X86/gem5.opt configs/deprecated/example/se.py --cmd=../LAB/lab2/lab2-benchmark/xx --cpu-type=X86MinorCPU --mem-type=DDR3_1600_8x8 --cpu-clock=1GHz  --caches --l1d_size=64kB --l1i_size=64kB
```

3.由于issue-width无法直接通过参数修改，所以这里的指令和第一条是一样的，还需要进入se.py中手动修改issue-width

```
build/X86/gem5.opt configs/deprecated/example/se.py --cmd=../LAB/lab2/lab2-benchmark/xx --cpu-type=DerivO3CPU --mem-type=DDR3_1600_8x8 --cpu-clock=1GHz  --caches --l1d_size=64kB --l1i_size=64kB
```

4.修改cpu-clock为4GHz

```
build/X86/gem5.opt configs/deprecated/example/se.py --cmd=../LAB/lab2/lab2-benchmark/xx --cpu-type=DerivO3CPU --mem-type=DDR3_1600_8x8 --cpu-clock=4GHz  --caches --l1d_size=64kB --l1i_size=64kB
```

5.从5-7的不同区别为增加l2cache的大小

```
build/X86/gem5.opt configs/deprecated/example/se.py --cmd=../LAB/lab2/lab2-benchmark/xx --cpu-type=DerivO3CPU --mem-type=DDR3_1600_8x8 --cpu-clock=1GHz  --caches --l1d_size=64kB --l1i_size=64kB --l2cache --l2_size=256kB
```

6.大小为2MB

```
build/X86/gem5.opt configs/deprecated/example/se.py --cmd=../LAB/lab2/lab2-benchmark/xx --cpu-type=DerivO3CPU --mem-type=DDR3_1600_8x8 --cpu-clock=1GHz  --caches --l1d_size=64kB --l1i_size=64kB --l2cache --l2_size=2MB
```

7.大小为16MB

```
build/X86/gem5.opt configs/deprecated/example/se.py --cmd=../LAB/lab2/lab2-benchmark/xx --cpu-type=DerivO3CPU --mem-type=DDR3_1600_8x8 --cpu-clock=1GHz  --caches --l1d_size=64kB --l1i_size=64kB --l2cache --l2_size=16MB
```

二、问题回答

1. 在运行程序生成的各个文件中有很多指标都可以或多或少的表示系统性能，比如CPI代表了每条指令使用的周期数、IPC代表了每个周期的指令数、numCycles代表了模拟的CPU周期数。都可以或多或少代表性能，但是均不够全面。

   但是对于同一个程序，最直观的想法就是看谁运行的快，也就是文件中最开头的simSeconds。对于同样的一个程序，不同体系结构谁正确运行的快谁的性能就更好。在后续检查实验的过程中，我发现使用模拟时间可能精度是不够的，还可以参考着时钟周期数来搭配评估。

   （由于lfsr编译中优化会出现错误，所以我选择的做法是直接将if语句进行了删除，如下是修改过后重新编译的模拟情况）

2. 各个配置的benchmark性能表示如下：

	lfsr	merge	mm	sieve	spvm
初始配置	0.000033	0.001771	0.003554	0.053443	0.027954
MinorCPU	0.000059	0.004669	0.020403	0.055777	0.121964
issueWidth=2	0.000035	0.003072	0.007384	0.054133	0.028466
cpu-clock=4GHz	0.000026	0.000487	0.001484	0.050494	0.026994
l2Cache=256kB	0.000050	0.001808	0.003390	0.081515	0.025423
l2Cache=2MB	0.000050	0.001808	0.003390	0.025131	0.017636
l2Cache=16MB	0.000050	0.001808	0.003390	0.025131	0.012839

	lfsr	merge	mm	sieve	spvm
初始配置	33109	1771107	3553945	53442902	27953951
MinorCPU	58569	4669478	20402759	55776861	121963739
issueWidth=2	35317	3072408	7384250	54132794	28465545
cpu-clock=4GHz	104433	1948000	5936423	201975052	107977360
l2Cache=256kB	50170	1808167	3390479	81515465	25423037
l2Cache=2MB	50170	1808167	3390479	25130921	17636255
l2Cache=16MB	50170	1808167	3390479	25130921	12838773

   我们可以发现merge收益于删除L2Cache，我认为是因为这里涉及到的数据数量比较小，没有怎么发生miss的情况，所以添加一个L2Cache反而增加了L1Cache获取数据的时间。在L2Cache不够大时，sieve也会收益于删除L2Cache。

3. 以下分别为各个概念的理解

   1. 内存规则性（Memory Regularity）： 内存规则性指的是程序在访问内存时的规律性。如果程序的内存访问模式具有规则性，意味着它们以可预测的模式访问内存位置。这有助于提高内

存访问的效率和并行性。这个规则性主要影响了内存访问相关中Cache相关的部分，更加规律有序的访问内存可以降低Cache的miss概率提高整体性能。

2. 控制规则性（Control Regularity）： 控制规则性涉及到程序的执行流程的规律性。如果程序在执行时具有控制规则性，意味着它的控制流程在不同的执行实例中是相似的或者是可预测的。主要影响了系统中有关分支预测的部分，如果控制指令更加规律那么我们的分支预测会更加成功，减少了流水线中的冲刷或停顿，从而增加了流水线的运行效率。
3. 内存局部性（Memory Locality）： 内存局部性是指程序在访问内存时倾向于访问附近的内存位置的倾向。内存局部性可以分为两种类型：时间局部性和空间局部性。时间局部性指的是程序在一段时间内倾向于多次访问相同的内存位置，而空间局部性指的是程序在访问一个内存位置时倾向于同时访问附近的内存位置。该部分同样影响了Cache的命中率，从而影响整体的性能。

4. 下面分别展示各个属性的指标
    1. memory regularity：这个属性可以用`system.mem_ctrls.dram.pageHitRate`来表示，页表的命中率可以侧面体现了
    2. control regularity：这个属性可以通过`system.cpu.icache.overallMissRate::total`来评估。如果这个比例很小，那么说明程序的控制流比较规则，因为整体iCache的访问命中率比较高，即每条指令都在顺序执行，也就是指令大多都能在iCache中找到。
    3. memory locality：这个属性可以使用`system.cpu.dcache.ReadReq.missRate::total`和`system.cpu.dcache.WriteReq.missRate::total`的比例之和来表示，因为如果一个程序的内存访问模式具有很好的局部性，那么它的缓存失效率应该会很低。
5. 对于不同的benchmark，不同的瓶颈如下：

Memory Regularity	**lfsr**	**merge**	**mm**	**sieve**	**spvm**
初始配置	74.29	74.72	87.39	88.79	58.24
MinorCPU	75.86	53.20	83.99	88.97	54.20
issueWidth=2	73.12	63.56	86.06	88.79	58.14
cpu-clock=4GHz	74.39	84.82	88.88	88.82	58.33
l2Cache=256kB	73.25	72.62	84.96	83.13	54.61
l2Cache=2MB	73.25	72.62	84.96	92.06	90.02
l2Cache=16MB	73.25	72.62	84.96	92.06	88.18

Control Regularity	**lfsr**	**merge**	**mm**	**sieve**	**spvm**
初始配置	0.252864	0.001274	0.000695	0.003194	0.000175
MinorCPU	0.153110	0.000855	0.000405	0.000084	0.000087
issueWidth=2	0.226836	0.000957	0.000694	0.003215	0.000174
cpu-clock=4GHz	0.259526	0.001330	0.000741	0.003304	0.000183
l2Cache=256kB	0.255382	0.001303	0.000713	0.003251	0.000181
l2Cache=2MB	0.255382	0.001303	0.000713	0.003262	0.000180
l2Cache=16MB	0.255382	0.001303	0.000713	0.003262	0.000181

Memory Locality	lfsr	merge	mm	sieve	spvm
初始配置	0.156858	0.005394	0.055531	0.896312	0.453299
MinorCPU	0.192461	0.007225	0.021697	0.553533	0.226950
issueWidth=2	0.184143	0.007717	0.055439	0.897851	0.465186
cpu-clock=4GHz	0.159495	0.006272	0.056590	0.896259	0.475359
l2Cache=256kB	0.157856	0.005581	0.054498	0.895357	0.525019
l2Cache=2MB	0.157856	0.005581	0.054498	0.894970	0.546258
l2Cache=16MB	0.157856	0.005581	0.054498	0.894970	0.468749

	Memory regularity	Control regularity	Memory locality
lfsr	高	低	高
merge	低	高	高
mm	高	高	高
sieve	低	高	低
spvm	高	高	高

1. lfsr这个程序实现了一个线性反馈移位寄存器的循环。其由于逻辑较为简单，整体性能的瓶颈在于给定的cpu时间，频率上升带给运行时间降低的效果最好。
2. merge归并排序与1类似，逻辑较为简单，所以对于cpu时间比较敏感，同时由于程序的并行性较好，所以issue-width的改变对其影响也不小。
3. mm实现了一个矩阵乘法，瓶颈情况与2较为类似，也为cpu时间和issue-width。
4. sieve是素数筛的程序，瓶颈在于Cache的大小，因为其逻辑底层涉及到数组的各种遍历，对于读取的数据范围比较大，所以对于大容量的L2Cache可以有效提升系统性能。
5. spvm稀疏矩阵向量乘法的程序中，情况与4类似，系统的整体瓶颈在于Cache的大小。

6. 这里我选择的是mm作为分析程序

    1. 首先是应用程序增强

```
for (jj = 0; jj < row_size; jj += block_size){
        for (kk = 0; kk < row_size; kk += block_size){
            for ( i = 0; i < row_size; ++i){
                for (k = 0; k < block_size; ++k){
                    i_row = i * row_size;
                    k_row = (k  + kk) * row_size;
                    for (j = 0; j < block_size; ++j){
                        prod[i_row + j + jj] += m1[i_row + k + kk] *
m2[k_row + j + jj];
                    }
                }
            }
        }
    }
```

    此处可以进行一些优化，比如将i_row + jj 的过程提到对j的for循环外边来，可以减少重复计算量。或者使用矩阵分割乘法来降低时间复杂度。

2. ISA 增强：可以直接在指令集架构中扩充加入有关向量操作的指令，可以一次执行多个相同类型的操作，提高计算效率。从而降低指令条数。
3. 微体系结构增强：通过比较各个配置之间的性能，我们发现无脑增大l2cache的内存无法产生过大影响，所以可以适量增加l1cache（尤其是Dcache）来提高系统性能