1、

MIPS：

#include <stdio.h>

#include <time.h>

int **main**() {

**clock\_t** start, end;

    double cpu\_time\_used;

    int sum = 0;

    start = **clock**();

    for(int i = 0; i < 1000000; i++) {

        sum += i;

    }

    end = **clock**();

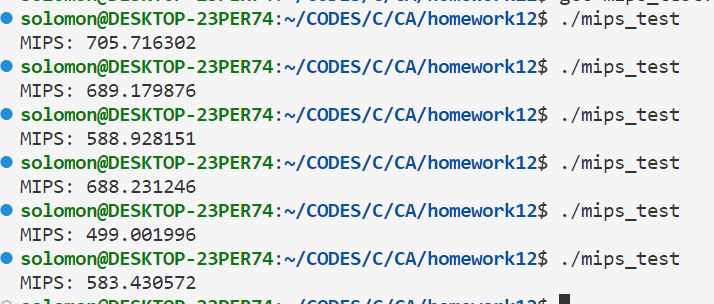
    cpu\_time\_used = ((double) (end - start)) / **CLOCKS\_PER\_SEC**;

**printf**("MIPS: %f\n", 1000000.0 / cpu\_time\_used / 1000000.0);

    return 0;

}

运行结果：



MFLOPS:

#include <stdio.h>

#include <time.h>

int **main**() {

**clock\_t** start, end;

    double cpu\_time\_used;

    double sum = 0.0;

    start = **clock**();

    for(int i = 0; i < 1000000; i++) {

        sum += (double)i \* 0.1;

    }

    end = **clock**();

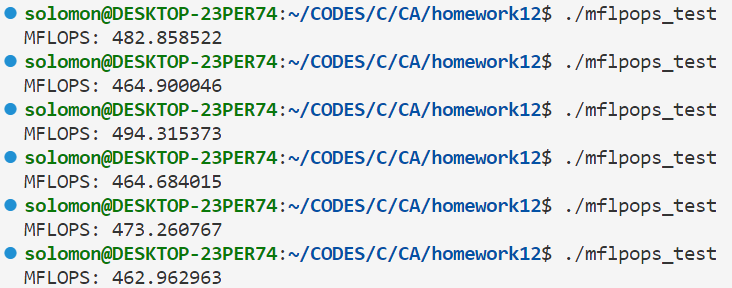
    cpu\_time\_used = ((double) (end - start)) / **CLOCKS\_PER\_SEC**;

**printf**("MFLOPS: %f\n", 1000000.0 / cpu\_time\_used / 1000000.0);

    return 0;

}

运行结果：



2、

（1）、

solomon@DESKTOP-23PER74:~/CODES/C/CA/homework12$ gcc -O3 -mtune=native -march=native -fopenmp -DN=200000000 -DNTIMES=100 stream\_test.c -o stream\_test

solomon@DESKTOP-23PER74:~/CODES/C/CA/homework12$ ./stream\_test

-------------------------------------------------------------

STREAM version $Revision: 5.10 $

-------------------------------------------------------------

This system uses 8 bytes per array element.

-------------------------------------------------------------

\*\*\*\*\* WARNING: \*\*\*\*\*\*

It appears that you set the preprocessor variable N when compiling this code.

This version of the code uses the preprocesor variable STREAM\_ARRAY\_SIZE to control the array size

Reverting to default value of STREAM\_ARRAY\_SIZE=10000000

\*\*\*\*\* WARNING: \*\*\*\*\*\*

Array size = 10000000 (elements), Offset = 0 (elements)

Memory per array = 76.3 MiB (= 0.1 GiB).

Total memory required = 228.9 MiB (= 0.2 GiB).

Each kernel will be executed 100 times.

The \*best\* time for each kernel (excluding the first iteration)

will be used to compute the reported bandwidth.

-------------------------------------------------------------

Number of Threads requested = 16

Number of Threads counted = 16

-------------------------------------------------------------

Your clock granularity/precision appears to be 1 microseconds.

Each test below will take on the order of 8384 microseconds.

(= 8384 clock ticks)

Increase the size of the arrays if this shows that

you are not getting at least 20 clock ticks per test.

-------------------------------------------------------------

WARNING -- The above is only a rough guideline.

For best results, please be sure you know the

precision of your system timer.

-------------------------------------------------------------

Function Best Rate MB/s Avg time Min time Max time

Copy: 18590.5 0.005066 0.004146 0.010021

Scale: 24316.6 0.007505 0.006580 0.016542

Add: 24648.8 0.010660 0.009737 0.014932

Triad: 25561.4 0.010364 0.009389 0.014550

-------------------------------------------------------------

Solution Validates: avg error less than 1.000000e-13 on all three arrays

测试程序共四种测试操作：

* Copy 是复制操作，即从内存单元中读取一个数，并复制到其他内存单元中，两次访问内存操作
* Scale 是乘法操作，即从内存单元中读取一个数，与常数相乘，得到的记过存到其他内存单元，两次访问内存操作
* Add 是加法操作，从两个内存单元中分别读取两个数，将其进行加法操作后，得到的结果写入另一个内存单元中，3次访问内存操作
* Triad 是前面三种的结合，先从内存中读取一个数，与一个常数相乘得到一个乘积，然后从另一个内存单元中读取一个数与刚才乘积结果相加，得到的结果写入内存。3次访问内存操作

观察测试结果，一般的规律是Add > Triad > Copy > Scale。一次Add操作需要访问三次内存（两个读操作，一个写操作），Triad操作也需要三次访问内存， Copy和Scale操作需要两次访问内存。单位操作内，访问内存次数越多，越能够掩盖访存延迟，带宽越大。

（2）、

| **电源设置** | **CPU 频率** | **Best Rate MB/s** | | | |
| --- | --- | --- | --- | --- | --- |
| **Copy** | **Scale** | **Add** | **Triad** |
| 最佳性能 | 3.5 GHz | 17906.9 | 22857.2 | 23387.2 | 24117. |
| 平衡模式 | 2.2 GHz | 15678.4 | 18473.9 | 19764.6 | 19876.9 |
| 节电模式 | 1.3 GHz | 11193.4 | 14153.7 | 16998.2 | 15797.5 |

通过数据可以简单得出结论，CPU 频率与内存带宽存在一个正相关的关系。

（3）、

从双精度修改为单精度，将代码中的宏定义  修改为 #define STREAM\_TYPE float，重新编译测试，得到如下结果：

Function Best Rate MB/s Avg time Min time Max time

Copy: 20004.8 0.002944 0.002105 0.007034

Scale: 26845.7 0.004049 0.002980 0.010338

Add: 26026.0 0.005397 0.004611 0.010972

Triad: 27810.6 0.005182 0.004315 0.009092

可以看出，单精度测试时，平均时间约为双精度的一半，因为测试的数据减少了一半。而测得的实际带宽几乎不变。

3、

libquantum程序的重要测试对象是SIMD （单指令流多数据流）性能，对微结构的压力主要是对平行处理微元的测试，模拟量子计算机的 C 语言库，其需要模拟大量量子比特的状态变化，这些过程在量子计算机上是并行的，但在 libquantum 中需要串行完成。

查看 spec.org 网站上的测试结果，使用 icc 的测试得分要比其他的高得多。根据 462.libquantum 的特点，icc 可能采用了将串行程序转换为并行的技术，具有高并行、高向量化的特点。通常会优化代码以利用 Intel 处理器的并行性能。这可能包括将串行代码转换为并行代码， icc 还可能通过向量化等技术提高性能。

4、

4. 在linux-windows双系统电脑上运行perf测试工具，结果如下：

(1)Bubble:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void **bubble\_sort**(int arr[], int n)

{

 int i, j, temp;

 for (i = 0; i < n - 1; i++)

 {

     for (j = 0; j < n - i - 1; j++)

     {

         if (arr[j] > arr[j + 1])

         {

             temp = arr[j];

             arr[j] = arr[j + 1];

             arr[j + 1] = temp;

         }

     }

 }

}

int **main**()

{

 int arr[100000];

 int n = 100000;

**srand**(**time**(**NULL**));

 for (int i = 0; i < n; i++)

 {

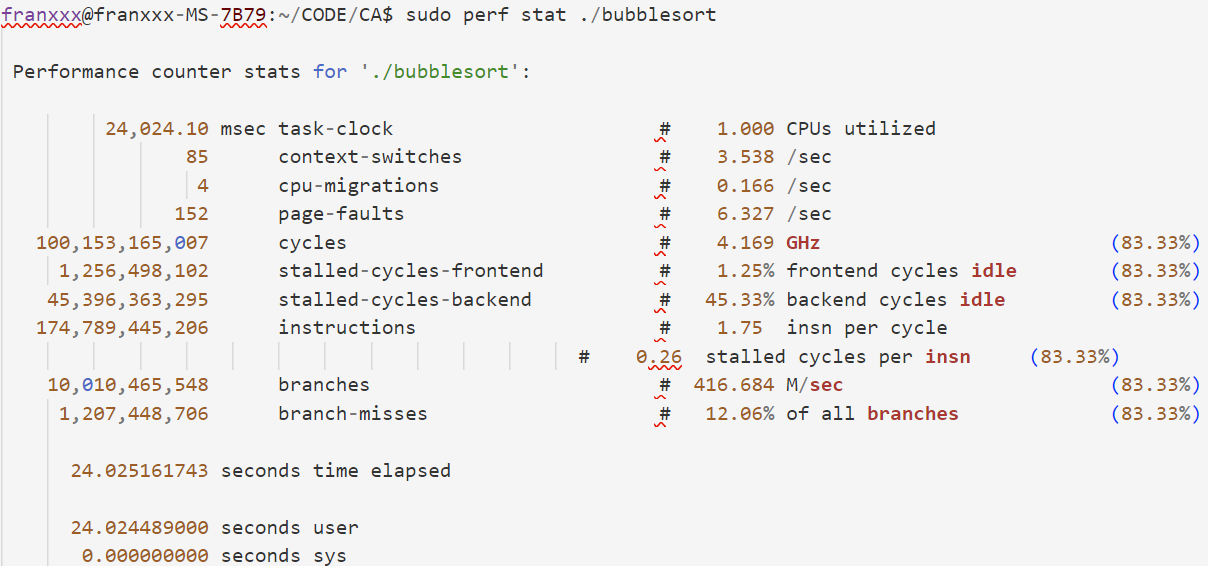
     arr[i] = **rand**() % 100000;

 }

**bubble\_sort**(arr, n);

 return 0;

}



(2)Shell:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void **shellSort**(int arr[], int n)

{

   int gap;

*// 初始步长为数组长度的一半*

   gap = n / 2;

*// 对每个子数组进行插入排序*

   while (gap > 0)

   {

       for (int i = gap; i < n; i++)

       {

           int temp = arr[i];

           int j;

*// 插入排序*

           for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

           {

               arr[j] = arr[j - gap];

           }

           arr[j] = temp;

       }

*// 更新步长*

       gap /= 2;

   }

}

int **main**()

{

   int arr[10000];

*// 生成随机数组*

**srand**(**time**(**NULL**));

   for (int i = 0; i < 10000; i++)

   {

       arr[i] = **rand**() % 10000;

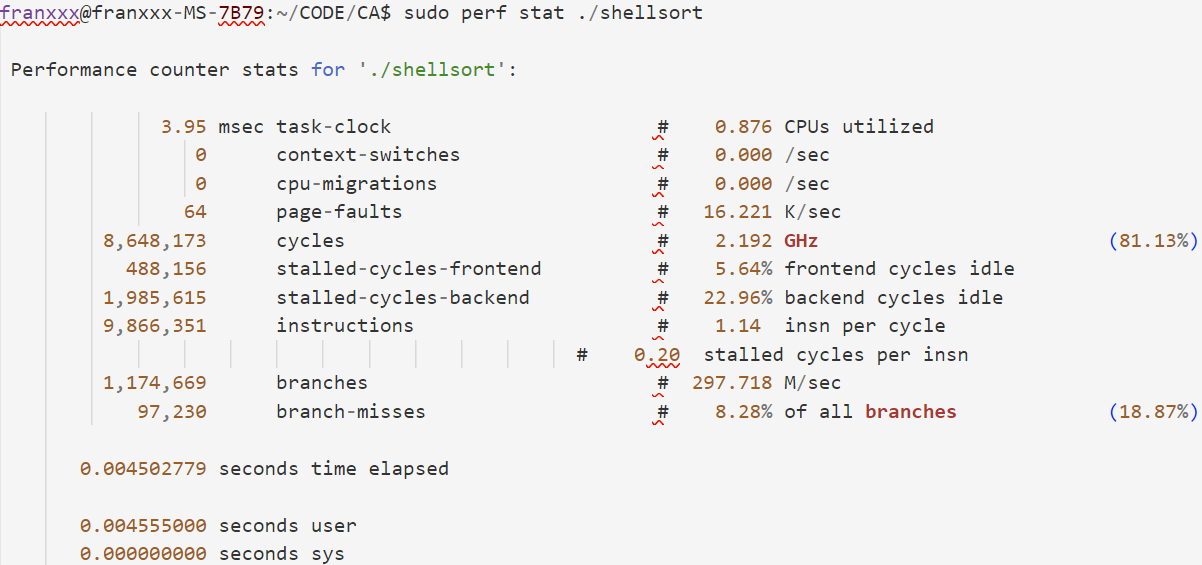
   }

*// 对数组进行排序*

**shellSort**(arr, 10000);

   return 0;

}



(3)

qsort(Quicksort):

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void **quick\_sort**(int arr[], int left, int right)

{

   int i, j, temp, pivot;

   if (left < right)

   {

       i = left;

       j = right;

       pivot = arr[left];

       while (i < j)

       {

           while (i < j && arr[j] > pivot)

           {

               j--;

           }

           arr[i] = arr[j];

           while (i < j && arr[i] < pivot)

           {

               i++;

           }

           arr[j] = arr[i];

       }

       arr[i] = pivot;

**quick\_sort**(arr, left, i - 1);

**quick\_sort**(arr, i + 1, right);

   }

}

int **main**()

{

   int arr[10000];

   int n = 10000;

**srand**(**time**(**NULL**));

   for (int i = 0; i < n; i++)

   {

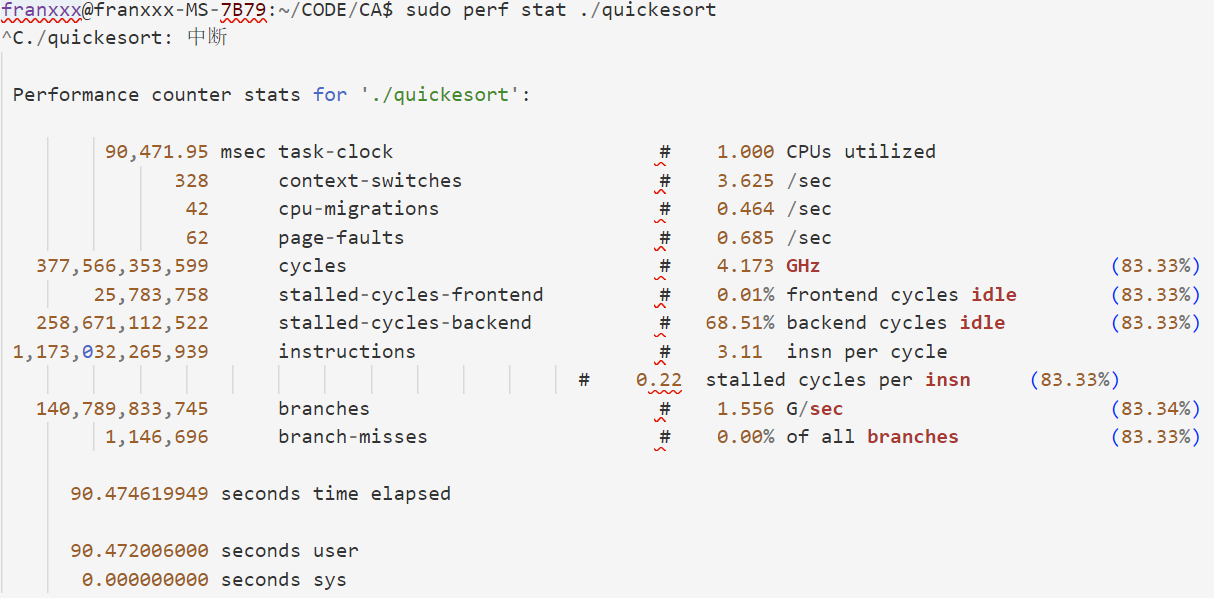
       arr[i] = **rand**() % 100000;

   }

**quick\_sort**(arr, 0, n - 1);

   return 0;

}



通过比较三个结果，IPC依次递减，转移失败率依次递增。快速排序通常具有最高的吞吐率，因为它的平均时间复杂度为O(n log n)，而冒泡排序和希尔排序的时间复杂度较高。冒泡排序和希尔排序可能更依赖于转移猜测，因为它们包含更多的内部循环。而快速排序的分区过程则更可能导致不均匀的分支，这导致转移猜测出错反而更多。但是快速排序因为排序数目较多，随机数组执行较慢，未跑出结果。

5、测试如下

gcc -pg -o test\_gprof linpack.c -lm

solomon@DESKTOP-23PER74:~/CODES/C/CA/homework12$ ./test\_gprof

15 December 2023 10:02:47 PM

LINPACK\_BENCH

C version

The LINPACK benchmark.

Language: C

Datatype: Double precision real

Matrix order N = 1000

Leading matrix dimension LDA = 1001

Norm. Resid Resid MACHEP X[1] X[N]

6.491510 0.000000 2.220446e-16 1.000000 1.000000

Factor Solve Total MFLOPS Unit Cray-Ratio

0.539980 0.000000 0.539980 1238.317469 0.001615 9.642500

LINPACK\_BENCH

Normal end of execution.

15 December 2023 10:02:47 PM、

solomon@DESKTOP-23PER74:~/CODES/C/CA/homework12$ gprof ./test\_gprof gmon.out

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls ms/call ms/call name

94.87 0.54 0.54 501499 0.00 0.00 daxpy

5.27 0.57 0.03 main

0.00 0.57 0.00 2000000 0.00 0.00 r8\_random

0.00 0.57 0.00 999 0.00 0.00 dscal

0.00 0.57 0.00 999 0.00 0.00 idamax

0.00 0.57 0.00 4 0.00 0.00 cpu\_time

0.00 0.57 0.00 2 0.00 0.00 r8mat\_gen

0.00 0.57 0.00 2 0.00 0.00 timestamp

0.00 0.57 0.00 1 0.00 538.62 dgefa

0.00 0.57 0.00 1 0.00 2.16 dgesl

% the percentage of the total running time of the

time program used by this function.

cumulative a running sum of the number of seconds accounted

seconds for by this function and those listed above it.

self the number of seconds accounted for by this

seconds function alone. This is the major sort for this

listing.

calls the number of times this function was invoked, if

this function is profiled, else blank.

self the average number of milliseconds spent in this

ms/call function per call, if this function is profiled,

else blank.

total the average number of milliseconds spent in this

ms/call function and its descendents per call, if this

function is profiled, else blank.

name the name of the function. This is the minor sort

for this listing. The index shows the location of

the function in the gprof listing. If the index is

in parenthesis it shows where it would appear in

the gprof listing if it were to be printed.

Copyright (C) 2012-2020 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,

are permitted in any medium without royalty provided the copyright

notice and this notice are preserved.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 1.75% of 0.57 seconds

index % time self children called name

<spontaneous>

[1] 100.0 0.03 0.54 main [1]

0.00 0.54 1/1 dgefa [3]

0.00 0.00 1/1 dgesl [4]

0.00 0.00 4/4 cpu\_time [8]

0.00 0.00 2/2 timestamp [10]

0.00 0.00 2/2 r8mat\_gen [9]

-----------------------------------------------

0.00 0.00 1999/501499 dgesl [4]

0.54 0.00 499500/501499 dgefa [3]

[2] 94.7 0.54 0.00 501499 daxpy [2]

-----------------------------------------------

0.00 0.54 1/1 main [1]

[3] 94.4 0.00 0.54 1 dgefa [3]

0.54 0.00 499500/501499 daxpy [2]

0.00 0.00 999/999 idamax [7]

0.00 0.00 999/999 dscal [6]

-----------------------------------------------

0.00 0.00 1/1 main [1]

[4] 0.4 0.00 0.00 1 dgesl [4]

0.00 0.00 1999/501499 daxpy [2]

-----------------------------------------------

0.00 0.00 2000000/2000000 r8mat\_gen [9]

[5] 0.0 0.00 0.00 2000000 r8\_random [5]

-----------------------------------------------

0.00 0.00 999/999 dgefa [3]

[6] 0.0 0.00 0.00 999 dscal [6]

-----------------------------------------------

0.00 0.00 999/999 dgefa [3]

[7] 0.0 0.00 0.00 999 idamax [7]

-----------------------------------------------

0.00 0.00 4/4 main [1]

[8] 0.0 0.00 0.00 4 cpu\_time [8]

-----------------------------------------------

0.00 0.00 2/2 main [1]

[9] 0.0 0.00 0.00 2 r8mat\_gen [9]

0.00 0.00 2000000/2000000 r8\_random [5]

-----------------------------------------------

0.00 0.00 2/2 main [1]

[10] 0.0 0.00 0.00 2 timestamp [10]

-----------------------------------------------

……………

第一张表是各个函数的执行和性能报告，根据第一张表现的内容，lnpack测试程序的热点函数为daxpy和dgefa

6、

Host OS Mhz L1 $ L2 $ Main mem Rand mem Guesses

--------- ------------- --- ---- ---- -------- -------- -------DESKTOP-2 Linux 5.10.16 4537 1.1020 1.7020 9.452000 130.2

make[1]: Leaving directory '/home/solomon/CODES/C/CA/homework12/lm-test/lmbench-3.0-a9/results'

单位为ns。Main mem对应L3 cache。

7、

SimpleScalar 模拟器安装不成功，也无法在机器上进行测试，这个问题无法解决。周围也鲜有同学做出，不知道要如何完成这个测试……

8 、桌面基准测试所面向的测试对象主要为通用计算机系统。而嵌入式基准测试的对象为嵌入式计算机系统。嵌入式系统通常是面向特定应用的，其 Soc 的微结构要比一般的桌面 CPU 简单很多。Cache、乱序执行、重命名寄存器等桌面 CPU 常见的技术很少会在资源紧张的嵌入式系统上应用。嵌入式基准测试更注重于机器在某一特定领域上的功能。而桌面基准测试更注重全面性和通用性。其对功耗要求更高，嵌入式基准测试程序可能会专门对功耗进行测试。

9.、ARM CotexA-57

**Table 11-5 Common Event Identification Register 0 bit assignments**

| **Bit** | **Name** | **Event number** | **Value** | **Event implemented if bit set to 1 or not implemented if bit set to 0** |
| --- | --- | --- | --- | --- |
| [31] | - | 0x1F | 0 | Reserved, *RES0*. |
| [30] | CH | 0x1E | 1 | Chain.[a](http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488h/way1382543397991.html" \l "fntarg_a) An odd-numbered counter increments when an overflow occurs on the preceding even-numbered counter. For even-numbered counters, does not count. |
| [29] | BC | 0x1D | 1 | Bus cycle. |
| [28] | TW | 0x1C | 1 | TTBR write, architecturally executed, condition check pass - write to translation table base. |
| [27] | IS | 0x1B | 1 | Instruction speculatively executed. |
| [26] | ME | 0x1A | 1 | Local memory error. |
| [25] | BA | 0x19 | 1 | Bus access. |
| [24] | DC2W | 0x18 | 1 | Level 2 data cache Write-Back. |
| [23] | DC2R | 0x17 | 1 | Level 2 data cache refill. |
| [22] | DC2A | 0x16 | 1 | Level 2 data cache access. |
| [21] | DC1W | 0x15 | 1 | Level 1 data cache Write-Back. |
| [20] | IC1A | 0x14 | 1 | Level 1 instruction cache access. |
| [19] | MA | 0x13 | 1 | Data memory access. |
| [18] | BP | 0x12 | 1 | Predictable branch speculatively executed. |
| [17] | CC | 0x11 | 1 | Cycle. |
| [16] | BM | 0x10 | 1 | Mispredicted or not predicted branch speculatively executed. |
| [15] | UL | 0x0F | 0 | Instruction architecturally executed, condition check pass - unaligned load or store. |
| [14] | BR | 0x0E | 0 | Instruction architecturally executed, condition check pass - procedure return. |
| [13] | BI | 0x0D | 0 | Instruction architecturally executed - immediate branch. |
| [12] | PW | 0x0C | 0 | Instruction architecturally executed, condition check pass - software change of the PC. |
| [11] | CW | 0x0B | 1 | Instruction architecturally executed, condition check pass - write to CONTEXTIDR. |
| [10] | ER | 0x0A | 1 | Instruction architecturally executed, condition check pass - exception return. |
| [9] | ET | 0x09 | 1 | Exception taken. |
| [8] | IA | 0x08 | 1 | Instruction architecturally executed. |
| [7] | ST | 0x07 | 0 | Instruction architecturally executed, condition check pass - store. |
| [6] | LD | 0x06 | 0 | Instruction architecturally executed, condition check pass - load. |
| [5] | DT1R | 0x05 | 1 | Level 1 data TLB refill.This event is implemented. |
| [4] | DC1A | 0x04 | 1 | Level 1 data cache access. |
| [3] | DC1R | 0x03 | 1 | Level 1 data cache refill. |
| [2] | IT1R | 0x02 | 1 | Level 1 instruction TLB refill. |
| [1] | IC1R | 0x01 | 1 | Level 1 instruction cache refill. |
| [0] | SI | 0x00 | 1 | Instruction architecturally executed, condition check pass - software increment. |

10、

模拟建模是一种基于计算机模型的方法，可以通过模拟系统的行为来预测其性能。相比之下，性能测量是一种基于实验的方法，可以直接测量系统的性能。模拟建模不需要实际构建系统或进行实验，可以更容易地进行参数化分析，以便更好地了解系统的行为。但它无法准确地模拟系统的行为。

性能测量可以提供更准确的结果，但是，性能测量需要更多的时间和资源。

11、

SimPoint的基本原理是将程序切成片段，每个片段有自己的特征，然后对这些片段进行聚类。它认为一个聚类中的片段是相似的，用聚类最靠近中心点或其他指标来选取最有代表性的片段。一个聚类中的片段越多，这个片段的权重就越大。

SimPoint 用基本块向量 (BBV) 作为片段的特征，BBV 是一个向量，这个向量的每一维表示某程序运行时进入一个基本块的次数。BBV 表示了片段的特性，比如一个片段总是调用一个函数，那么这个函数的第一个基本块对应的维的数值就会比较大。

SimPoint 的优点在于它可以减少模拟建模的时间，可以通过聚类的方法来减少需要模拟的片段数量，这样模拟仿真的程序是采样的片段而非原来的整个程序，所以这种采样方法能大大减少模拟建模时间，从而提高模拟的效率。

12、

模拟器和真实的机器之间的校准，常见的方法是使用基准测试套件来比较模拟器和真实机器的性能。基准测试套件评估计算机系统的性能，在相同的参数设置下，模拟器和真实机器的微测试程序运行结果越接近，则准确率越高，从而比较模拟器和真实机器的性能。

另一种方法是使用统计方法来比较模拟器和真实机器的性能。这些方法通常涉及使用回归分析或其他统计技术来比较模拟器和真实机器的性能。校准的评价指标通常包括平均误差率、平均绝对误差、平均相对误差等。这些指标可以帮助评估模拟器和真实机器之间的差异，并确定校准的质量。

13、