

使用GCC编译器分析和优化程序的数据局部性

袁鹏

paul@icompile.cn

2009-10-24

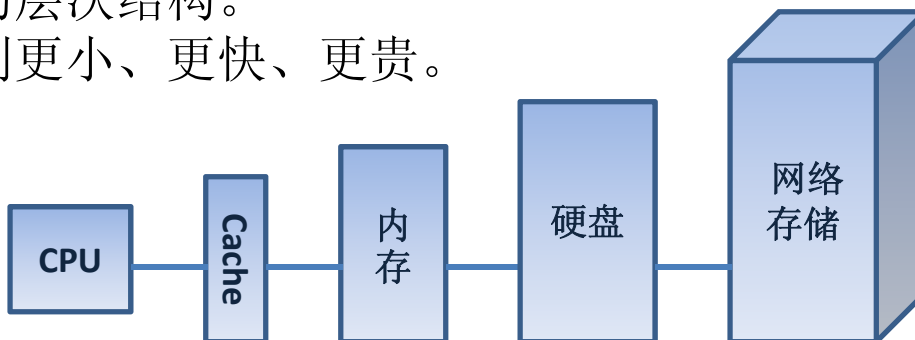


简介

- 存储层次
- 程序局部性优化方法
- GCC编译器中的数据重组优化
- 总结

存储层次

- Cache – 内存 – 硬盘 – 网络存储
 - 存储层次利用了存储技术的局部性和成本性能优势，设计了基于不同速度和大小的存储构成的层次结构。
 - 每个级别都比下一个级别更小、更快、更贵。
 - Cache: 1ns, KB/MB。
 - Memory: 100ns, GB。
 - Disk: 10ms, TB。



- Cache: 解决处理器与内存之间不断增长的性能差距
 - 处理器: 性能提升 52% (1986-2004) 20% (2004-)
 - 内存: 访问延迟每年7%性能提升。
 - 多级Cache: L1\$, L2\$, L3\$
 - Cache占处理器的面积比: Intel Core2处理器中 >50%
 - Cache中的数据以行 (Cache line) 进行组织。
 - Cache line size: 32Bytes, 64Bytes

评测存储层次性能

- OProfile是用于剖视程序行为的工具，它通过取样处理器中的硬件性能计数器来测量程序行为。
- 以AMD Opteron 270为例
 - 3发射，双核，2.0GHz。
 - L1 I\$/D\$: 64KB, 2-way, 64 bytes cache line, 2 cycles hit time。
 - L2 \$: 1MB, 16-way, 64 bytes cache line, 7 cycles hit time。
 - Exclusive mechanism for L1 and L2
- 使用方法
 - 以评测L1 D\$为例

```
opcontrol --event=RETIRED_INSTRUCTIONS:500003 --
event=DATA_CACHE_ACCESSES:500003 --
event=DATA_CACHE_REFILLS_FROM_L2_OR_SYSTEM:50003:0x1E --
event=DATA_CACHE_REFILLS_FROM_SYSTEM:50003:0x1E --image=vortex.exe
DC_isses = DC_refills_L2 + DC_refills_sys
```
 - 可以评测程序的CPI、Cache行为、TLB行为和存储带宽等，并关联到程序的源代码，确定代码中引起性能瓶颈的区域。
- 其它工具：Valgrind, VTune, Perfmon, Pin, DTrace.....

存储层次对程序性能的影响

- 程序执行时间
 - CPU time = (CPU execution clock cycles + memory stall clock cycles) * Clock cycle time
 - memory stall clock cycles = IC*(Miss_rate_L1 * Hit_time_L2 + Miss_rate_L2 * Miss_penalty_L2)
- 以程序255.vortex为例
 - 周期: 90,461,042,763, 指令数: 106,503,639,018。IPC = 1.18.
 - Cache失效
 - D\$: from_L2 366,421,984, from_system 71,454,287
 - I\$: from_L2 1,532,091,924, from_system 950,057
 - Memory stall clock cycles
 - $(366,421,984 + 153,209,192) * 7 + (71,454,287 + 950,057) * 120 = 12,325,939,512$
 - 占总执行时间比例: $12,325,939,512 / 90,461,042,763 = 13.63\%$

INT benchmarks

	Unstalled	Itlb	Icache	Branch	BE Flush	Score Board	L1Dt1b	L2Dt1b	Dcache	RSE
400.perlbench	58.42	0.29	4.05	2.12	6.50	0.11	0.46	1.89	20.39	5.75
401.bzip2	62.34	0.00	0.09	1.05	6.08	1.26	0.84	4.08	24.20	0.08
403.gcc	33.78	0.12	2.24	2.00	3.96	0.29	0.29	1.95	54.21	1.15
429.mcf	13.00	0.00	0.10	0.47	1.57	0.06	0.58	16.94	67.26	0.02
445.gobmk	55.23	0.17	11.03	5.34	10.23	0.03	0.12	0.70	13.70	3.45
456.hammer	85.88	0.00	0.05	0.33	1.72	0.00	0.03	0.00	11.97	0.02
458.sjeng	54.12	0.05	6.77	6.17	11.62	0.03	0.34	1.01	17.84	2.04
462.libquantum	18.63	0.00	0.03	0.09	0.52	0.01	0.01	0.10	80.60	0.01
464.h264ref	72.27	0.03	1.51	2.28	2.07	0.46	0.13	0.21	19.00	2.05
471.omnetpp	14.38	0.38	3.25	1.51	3.02	0.01	0.43	3.27	73.27	0.48
473.astar	34.87	0.00	0.09	2.01	7.70	0.00	0.36	5.43	49.54	0.01
483.xalancbmk	20.39	0.29	2.91	2.31	3.22	0.03	0.31	17.59	52.19	0.77
999.specrand	62.32	0.03	4.50	2.57	4.44	1.73	0.03	0.00	21.92	2.47

FP benchmarks

	Unstalled	Itlb	Icache	Branch	BE Flush	Score Board	L1Dt1b	L2Dt1b	Dcache	RSE
410.bwaves	44.08	0.00	1.34	1.73	1.21	0.00	0.00	1.10	50.48	0.06
416.gamess	68.86	0.02	3.15	1.77	2.86	0.00	0.05	0.01	17.47	5.81
433.milc	14.01	0.00	0.22	0.29	1.31	0.01	0.03	1.09	82.82	0.23
434.zeusmp	46.80	0.00	1.53	0.32	0.34	0.02	0.00	23.42	27.57	0.01
435.gromacs	64.15	0.00	0.25	0.70	3.39	0.01	0.08	0.05	30.82	0.56
436.cactusADM	69.41	0.00	0.75	0.05	0.03	0.00	0.00	1.01	28.72	0.03
437.leslie3d	44.06	0.00	0.67	0.27	0.08	0.00	0.01	2.01	52.87	0.02
444.namd	74.17	0.00	0.02	0.10	0.41	0.00	0.01	0.20	25.07	0.02
447.dealII	53.66	0.03	0.36	0.70	7.60	0.13	0.06	0.15	37.06	0.24
450.soplex	28.10	0.00	0.26	0.31	0.85	0.03	0.09	2.64	67.69	0.04
453.povray	52.66	0.13	2.94	2.05	6.77	0.00	0.40	0.00	33.97	1.09
454.calculix	50.46	0.01	0.26	0.31	0.72	0.01	0.03	0.18	47.78	0.25
459.GemsFDTD	25.88	0.00	0.38	0.07	0.09	0.00	0.05	8.67	64.84	0.02
465.tonto	57.01	0.03	2.17	1.73	1.87	0.02	0.03	1.01	33.96	2.18
470.lbm	33.64	0.00	0.18	0.03	0.03	0.00	0.01	1.00	65.11	0.01
481.wrf	63.81	0.02	1.43	0.44	0.30	0.24	0.00	1.13	32.07	0.56
482.sphinx3	30.11	0.00	0.65	1.04	1.43	0.00	0.08	1.96	64.43	0.30
999.specrand	62.32	0.03	4.50	2.57	4.44	1.73	0.03	0.00	21.92	2.47

GCC中的程序局部性优化

- 代码变换：基于依赖分析的循环变换
 - 通过改变程序中数据访问的顺序来提高局部性。
 - 循环交换、循环合并、循环分裂、循环分块等
 - tree-loop-distribution.c, tree-ssa-loop.c
- 数据预取
 - 需要硬件支持。将未来可能访问的数据提前取到cache中。
 - `for(i=0; i<N; ++i) { prefetch(a[i+pd]; sum+=A[i]; }`
 - tree-ssa-loop-prefetch.c: `-fprefetch-loop-arrays`
 - 针对数组，不支持递归数据结构。
 - 未必有收益。时效性，有用性，指令本身代价。
- 代码重组
 - 基本块重排，函数重排/分裂。
 - bb-reorder.c
- 数据重组
 - 重组程序中的数据来适应代码的访问模式。

```

/* 矩阵乘: X = Y * Z */
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j) {
    r = 0;
    for (k = 0; k < N; ++k)
      r = r + y[i][k]*z[k][j];
    x[i][j] = r;
  }

```

```

/* 分块后的矩阵乘法 */
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i = i+1)
      for (j = jj; j < min(jj+B-1,N); ++j) {
        r = 0;
        for (k = kk; k < min(kk+B-1,N); ++k)
          r = r + y[i][k]*z[k][j];
        x[i][j] += x[i][j] + r;
      }

```

重用间距是指同一元素两次相邻访问之间的不同元素个数，比如 $RD(a)=6$ 。

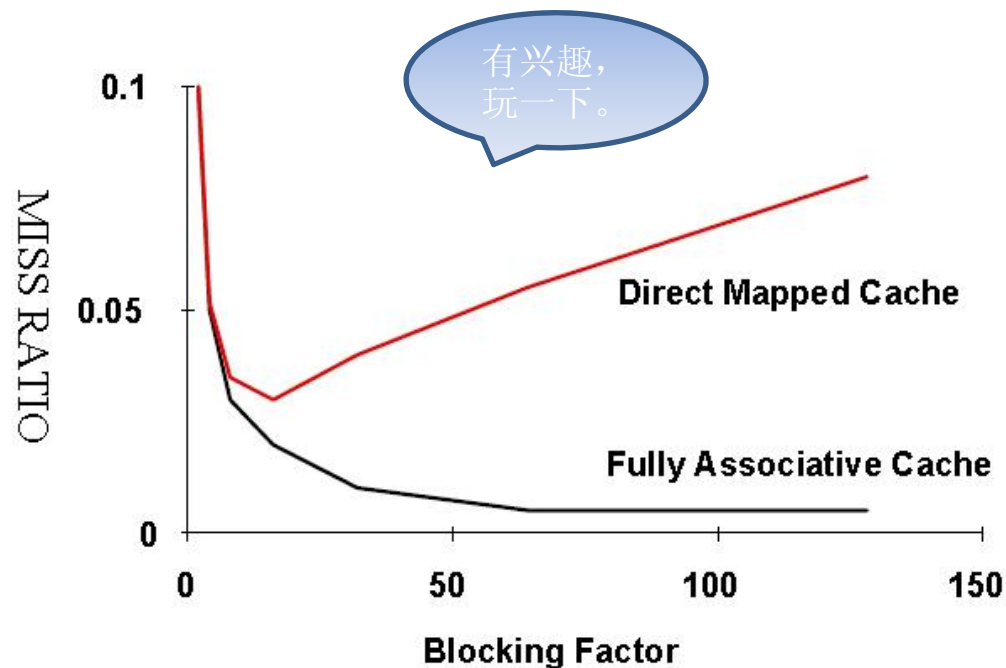
访存踪迹：“abcdefgabchiklmabc...”

分块可以减小数据重用间距（reuse distance），改善时间局部性。

<http://slo.sourceforge.net>

Kristof Beyls and Erik H. D'Hollander. Refactoring for Data Locality. *IEEE Computer*, Volume 42, Number 2, pp. 62-71, February 2009.

SLO是基于GCC的工具，分析程序运行时刻的数据重用路径，确定引起数据局部性瓶颈的代码，给出重构建议。SLO主要是针对循环结构，重构的建议包括循环的分块，合并和分裂等。



From Lam[1991]

数据重组

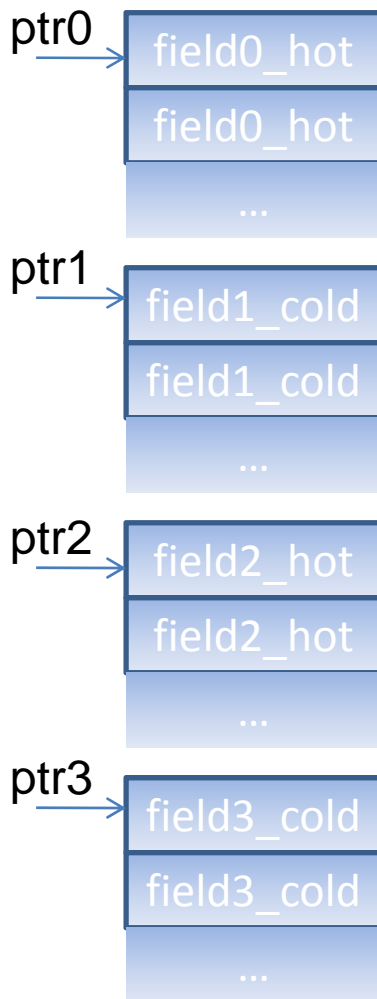
- 针对循环的代码变换局限在数据和循环密集型的科学计算类代码中，无法应用于基于指针的数据结构以及非规则的指针追溯访问。
- GCC中的已有实现：结构体全剥离（peeling）
 - ipa-struct-reorg.c[h], ipa-type-escape.c[h]
gcc -O3 -fwhole-program -combine -fipa-type-escape -fipa-struct-reorg a.c b.c
 - 初始结构体中的每个域剥离为一个新结构体。代码中对初始结构体实例和域
的访问都要进行相应的更新。
- 结构体分裂（splitting）
 - 将结构体分解成多个子结构体，并在根结构体中增加指针域与其它子结构体
相连。
- 池存储（memory pooling）
 - 针对动态分配对象的存储分配策略，将具有相同属性（比如同一种类型，具有
访问亲和性）的一组对象分配在同一个存储池中，提高数据的空间局部性。
 - 池存储提供了与标准C库函数中相似的分配函数，比如pool_alloc，pool_calloc
和pool_free等。

尝试将一个链表遍历程序中的链表结构分裂？或者剥离？ 😊

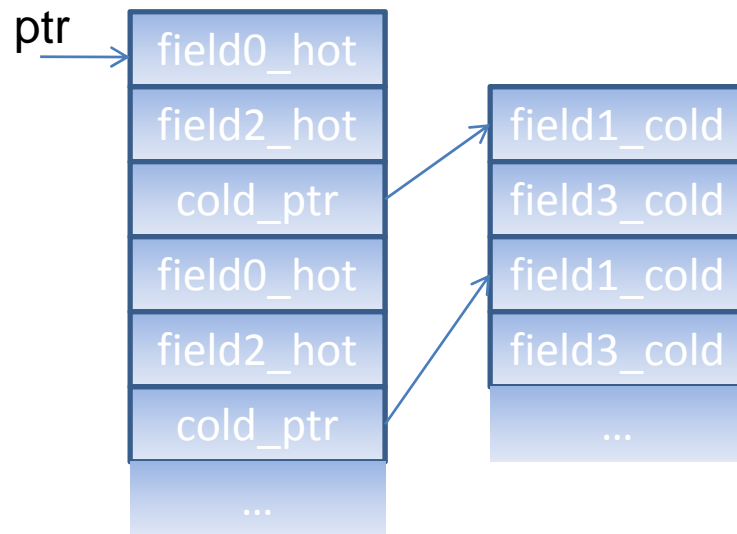
何时用剥离？
何时用分裂？



初始结构体数组



剥离



分裂

注意变换前后域访问方式的变化。比如对数组第二个元素中field3_cold的访问：

- ptr[2].field3_cold
- ptr3[2].field3_cold
- ptr[2].cold_ptr->field3_cold

```

typedef struct {
    double y;
    int reset;
} xyz;
xyz *Y;
Y = (xyz *)malloc(numf2s*sizeof(xyz));
Y[i].reset = 1;

```

初始代码

```

typedef struct {
    double y;
} xyz_1;
typedef struct {
    int reset;
} xyz_2;

xyz_1 *Y_1;
xyz_2 *Y_2;
Y_1 = (xyz_1 *)malloc(numf2s*sizeof(xyz_1));
Y_2 = (xyz_2 *)malloc(numf2s*sizeof(xyz_2));
Y_2[i].reset = 1;

```

结构体剥离

```

typedef struct {
    double y;
    xyz_cold *ptr_cold;
} xyz_hot;
typedef struct {
    int reset;
} xyz_cold;

xyz_hot *Y_1;
xyz_cold *Y_2;

Y_1 = (xyz_hot *)malloc(numf2s*sizeof(xyz_hot));
Y_2 = (xyz_cold *)malloc(numf2s*sizeof(xyz_cold));
for (i=0; i<numf2s; ++i)
    Y_1[i].ptr_cold = Y_2+i;

Y_1[i]. ptr_cold reset = 1;

```

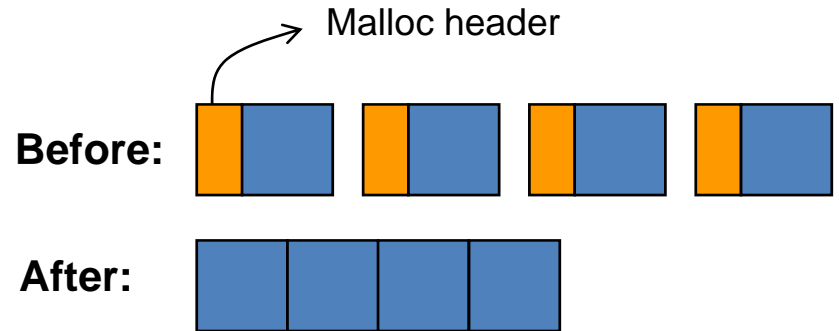
结构体分裂

池存储

```
list_node *build_list(int n) {
    list_node *new = (list_node *)
        malloc(sizeof(list_node));
    new->next = n ? makeList(n-1): NULL;
    new->data = n;
    return new;
}

int process_node(list_node X, int n)
{
    int i, sum=0;
    for (i=0; i<n; ++i) {
        sum += X->data;
        X=X->next;
    }
    return sum
}

X = build_list(100);
process_node(X, 100)
free_list (X);
```



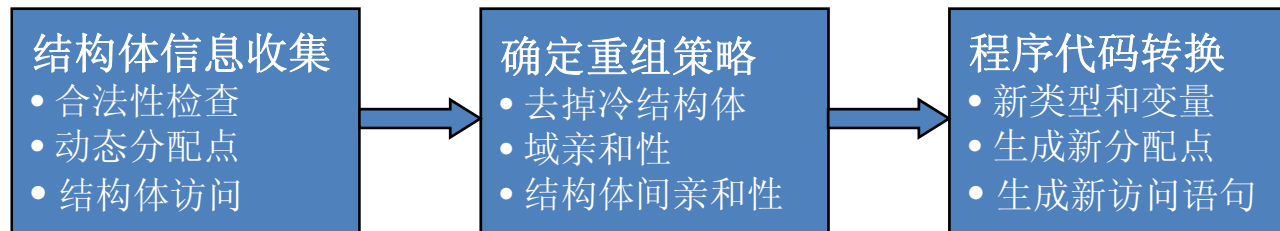
小结构体的malloc开销：4 bytes malloc header，对齐。通过池存储，将这些小结构体对象分配到连续的空间中，更好的利用空间局部性。

Apache的APR库中提供池存储函数。

```
apr_pool_t *pmain = NULL;
apr_status_t rv = apr_initialize();
rv = apr_pool_create (&pmain, NULL);
apr_palloc(pmain, sizeof(list_node));
apr_pool_destroy(pmain);
apr_terminate();
```

GCC DARE

- GCC DARE (DATA REorganization)
 - 基于数据间的亲和关系，将大结构体类型分裂为小结构体类型，并替换大结构体类型的数组为多个小结构体类型的数组，从而提高程序的数据局部性。
 - DARE的设计包含两个层次的策略。一是结构体重组策略，即如何分裂结构体类型形成多个小的结构体类型。二是数组重组策略，即如何连接基于结构体重组所形成的多个小结构体类型的数组。
 - 实现框架：扩充GCC中已有的结构体全剥离。
 - 基于过程间分析。
 - 增加数据亲和分析，作为结构体重组的依据。
 - 利用已收集的malloc信息，增加池存储（pool allocation）优化。



结构体信息收集

- `collect_structures ()`
 - 收集程序中可进行转换的结构体类型，并进行合法性检查。
 - 非法：域地址使用，类型逃逸，类型转换，用户约定。
- `collect_allocation_sites ()`
 - 收集程序中的内存分配点，包括`malloc`，`realloc`。
- `collect_data_accesses ()`
 - 结构体访问信息收集，包括域的访问次数，读/写等。
 - 在循环级别针对每个结构体记录域的访问情况。

```
for (i = 0; i < I; i++){ // count=100
  loop2 accesses a.f2;
  for (j = 0; j < J; j++) // count=500
    loop1 accesses a.f3 and a.f4;
}
for (k = 0; k < K; k++) // count=200
  loop3 accesses f1 and a.f3;
```

	a.f1	a.f2	a.f3	a.f4
loop1	0	0	500	500
Loop2	0	100	0	0
loop3	200	0	200	0

数据重组策略

- 函数is_cold_struct ()
 - 基于结构体的访问次数确定。
- 域亲和性分析
 - 基于域的亲和关系来分裂结构体，将具有高亲和度的域组成一个新的结构体。
 - 两个域亲和是指程序对它们的访问在时间上相近。那些经常相邻访问的域之间具有较高的亲和度。
 - 基于域访问信息表，合并具有相同域访问模式的循环中的数据形成亲和组。对于域集合相交的亲和组，如果它们的热度超过给定的阈值，则进行合并。下图中合并AG1和AG3。

	a.f1	a.f2	a.f3	a.f4
loop1	0	0	500	500
Loop2	0	100	0	0
loop3	200	0	200	0

	a.f1	a.f2	a.f3	a.f4
AG1			500	500
AG2		100		
AG3	200	0	200	

分裂为两个结构体：{f1, f3, f4}和{f2}。

思考：如果loop2的迭代次数很高，是否还将f2分裂出去？

程序代码转换

- 创建新类型和新全局变量
 - `create_new_types ()`
 - `create_new_global_vars ()`
- 处理单个函数
 - `do_reorg_for_func (struct cgraph_node *node)`
 - 创建新局部变量: `create_new_local_vars ()`
 - 处理函数参数: `update_parameters_for_func (node)`
`foo (struct xyz) → foo (struct xyz_new)`
 - 升级函数调用语句: `update_call_expr_for_func (node)`
 - 处理内存分配: `create_new_alloc_sites_for_func ()`
 - 对新增指针域的处理
 - 池存储: 使用apr中的pool allocation。
 - 处理结构体访问语句: `create_new_accesses_for_func ()`
- 池存储
 - 对于图、链表、树等结构密集的程序, 引入池存储, 将多个单独分配的结构体节点放入一个存储池中。
 - 主要是Olden程序。

性能评测

- 评测
 - 181.mcf 20%
 - 197.parser 9%
 - 300.twolf: 12%
 - 179.art: 60%（全剥离！）
 - Olden (9 benchmarks): ~10%
- 典型程序
 - TreeAdd: 二分树
 - Perimeter: 四分树
 - Health: 双向链表
 - 181.mcf: 三个结构体在三个函数中的访问 占90%的cache实效，集中在10行代码。
 - 300.twolf: 三个数组。两个小对象数组，一个大对象数组。

思考

- 全剥离
 - Peng Zhao等人的研究表明，与基于亲和度和基于频率的分裂相比，完全剥离可以达到最佳或接近最佳的性能。
Peng Zhao, Shimin Cui, Yaoqing Gao, Raúl Silvera and José Nelson Amaral. Forma: A Framework for Safe Automatic Array Reshaping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 30, Issue 1, November 2007.
 - 注意他们使用的评测：数组版本的OLDEN！！而不是递归数据结构版本！
- 基于热度的分裂
 - 可以减小间接寻址的开销。
 - 有时会占用存储带宽和污染数据cache。
- 基于亲和度的分裂
 - 充分利用了程序对结构体域访问的局部性特征。
 - 这种特征是全局的，并不能准确的表达一个循环中的域访问的亲和性。这种情况下也会导致数据cache的污染。
 - 将不具亲和关系的热域分解到一个新结构体中会导致地址计算的开销。
- 进一步理解亲和度：基于踪迹的计算方法
 - 热数据流方法
 - 基于重用签名的方法

总结

- 数据重组是现代编译器的一个重要优化。
 - 可以显著提升一些结构体密集程序的性能。
- 编译器的分析可以为开发者的性能调优提供建议。
 - 数据亲和分析是理解程序中数据访问关系的重要方法。
- 进一步的工作
 - 基于亲和度和基于热度的选择
 - 基于踪迹的分析：重用，热数据流
- GCC中新的LTO框架已经完成
 - 将DARE集成到LTO中。

谢谢！ 欢迎提问 😊