北京大学计算机学院/信息科学技术学院

2024-2025年秋季学期　　《编译原理》

第九章 代码优化（续）
——机器相关优化

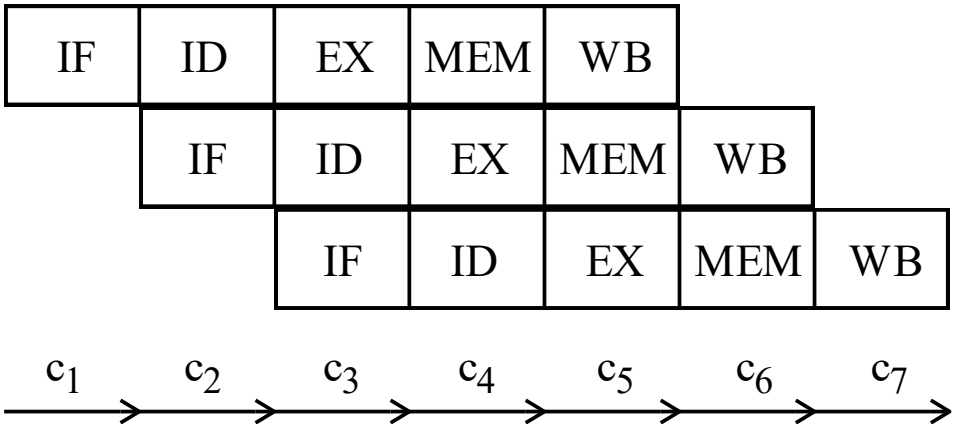
Code Optimization
Machine-Dependent Optimization

硬件机器特性

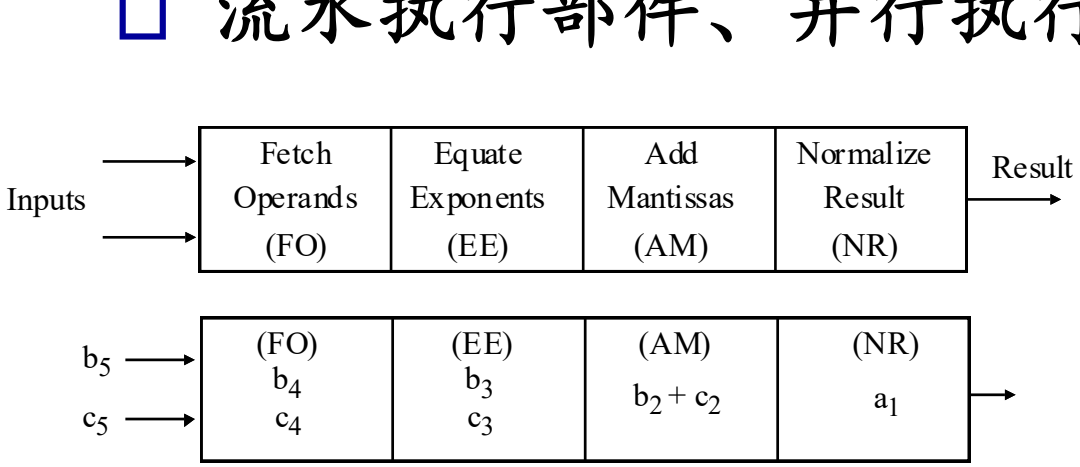
- 指令流水线
- 多个（可并行的）执行部件
  - 流水化执行部件
- 向量执行单元
- 并行处理
  - 共享内存vs分布式内存vs消息传递
- VLIW 和超标量处理器的多发射机制
- 寄存器堆
- 存储层次
- 以上特性可能叠加
  - 并行向量机
  - IA64

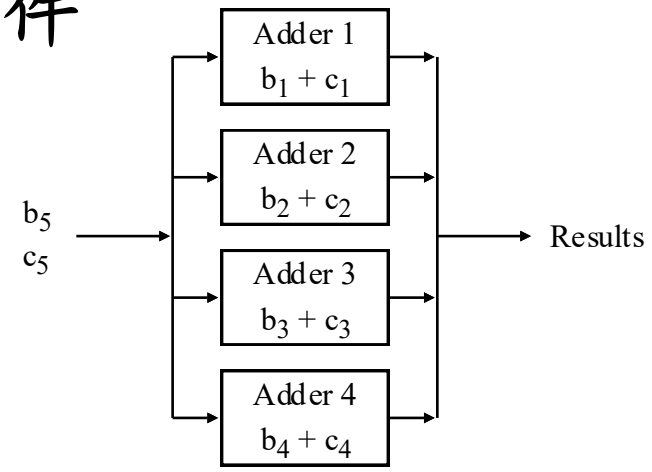
流水化的硬件

- 指令流水线



- 流水执行部件、并行执行部件





多核并行

- 减少通信（Minimizing communication）
  - 数据的排布（Data placement/layout）
- 优化通信（Optimizing communication）
  - 聚合（Aggregation）
  - 计算和通信的并行/重叠

- 可能的挑战：
  - 代码的生成和转换必须合法正确
  - 转换后得到更优的代码（如何判断收益）
- 底层代码生成（很多问题必须在低层才能决定）
  - 预取指令的插入
  - 指令的生成和调度

矩阵相乘的示例

```
DO I = 1, N
   DO J = 1, N
      C(J,I) = 0.0
      DO K = 1, N
         C(J,I) = C(J,I) + A(J,K) * B(K,I)
      ENDDO
   ENDDO
ENDDO
```

矩阵相乘：流水线机器

```
DO I = 1, N,
      DO J = 1, N, 4
         C(J,I) = 0.0        !Register 1
         C(J+1,I) = 0.0      !Register 2
         C(J+2,I) = 0.0      !Register 3
         C(J+3,I) = 0.0      !Register 4
         DO K = 1, N
            C(J,I)   = C(J,I)   + A(J,K)   * B(K,I)
            C(J+1,I) = C(J+1,I) + A(J+1,K) * B(K,I)
            C(J+2,I) = C(J+2,I) + A(J+2,K) * B(K,I)
            C(J+3,I) = C(J+3,I) + A(J+3,K) * B(K,I)
         ENDDO
      ENDDO
   ENDDO
```

矩阵相乘：向量机器

```
DO I = 1, N
   DO J = 1, N, 64
      DO JJ = 0,63
         C(JJ,I) = 0.0
      ENDDO
      DO K = 1, N
         DO JJ = 0,63
            C(J,I) = C(J,I) + A(J,K) * B(K,I)
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

矩阵相乘：向量机器

```
DO I = 1, N
   DO J = 1, N, 64
      C(J:J+63,I) = 0.0
      DO K = 1, N
         C(J:J+63,I) = C(J:J+63,I) + A(J:J+63,K)
*B(K,I)

      ENDDO
   ENDDO
ENDDO
```

矩阵相乘：向量SMP机器

```
PARALLEL DO I = 1, N
   DO J = 1, N, 64
      C(J:J+63,I) = 0.0
      DO K = 1, N
         C(J:J+63,I) = C(J:J+63,I) + A(J:J+63,K)*B(K,I)
      ENDDO
   ENDDO
ENDDO
```

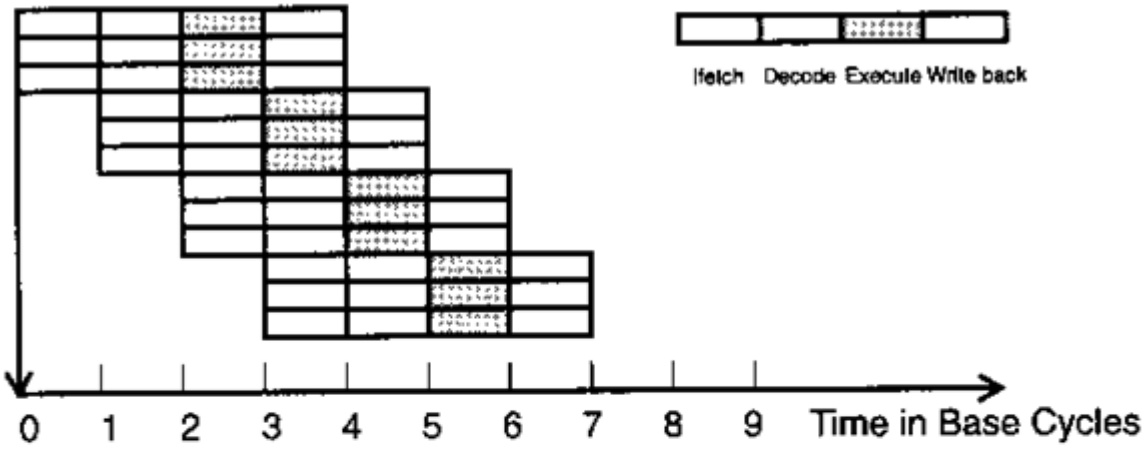
矩阵相乘：cache优化

```
DO I = 1, N, S
    DO J = 1, N, S
        DO p = I, I+S-1
            DO q = J, J+S-1
                C(q,p) = 0.0
            ENDDO
        ENDDO
        DO K = 1, N, T
            DO p = I, I+S-1
                DO q = J, J+S-1
                    DO r = K, K+T-1
                        C(q,p) = C(q,p) + A(q,r) * B(r,p)
                    ENDDO
                ENDDO
            ENDDO
        ENDDO
    ENDDO
ENDDO
```

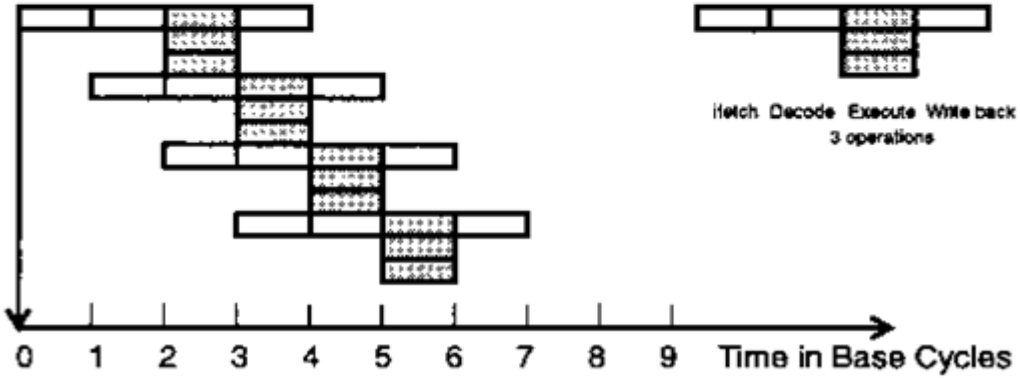
矩阵相乘：分布式共享存储机器

```
PARALLEL DO I = 1, N
    PARALLEL DO J = 1, N
        C(J,I) = 0.0
    ENDDO
ENDDO
PARALLEL DO I = 1, N, S
    PARALLEL DO J = 1, N, S
        DO K = 1, N, T
            DO p = I, I+S-1
                DO q = J, J+S-1
                    DO r = K, K+T-1
                        C(q,p) = C(q,p) + A(q,r) * B(r,p)
                    ENDDO
                ENDDO
            ENDDO
        ENDDO
    ENDDO
ENDDO
```

超标量vs超长指令字VLIW



A superscalar processor of degree m = 3.



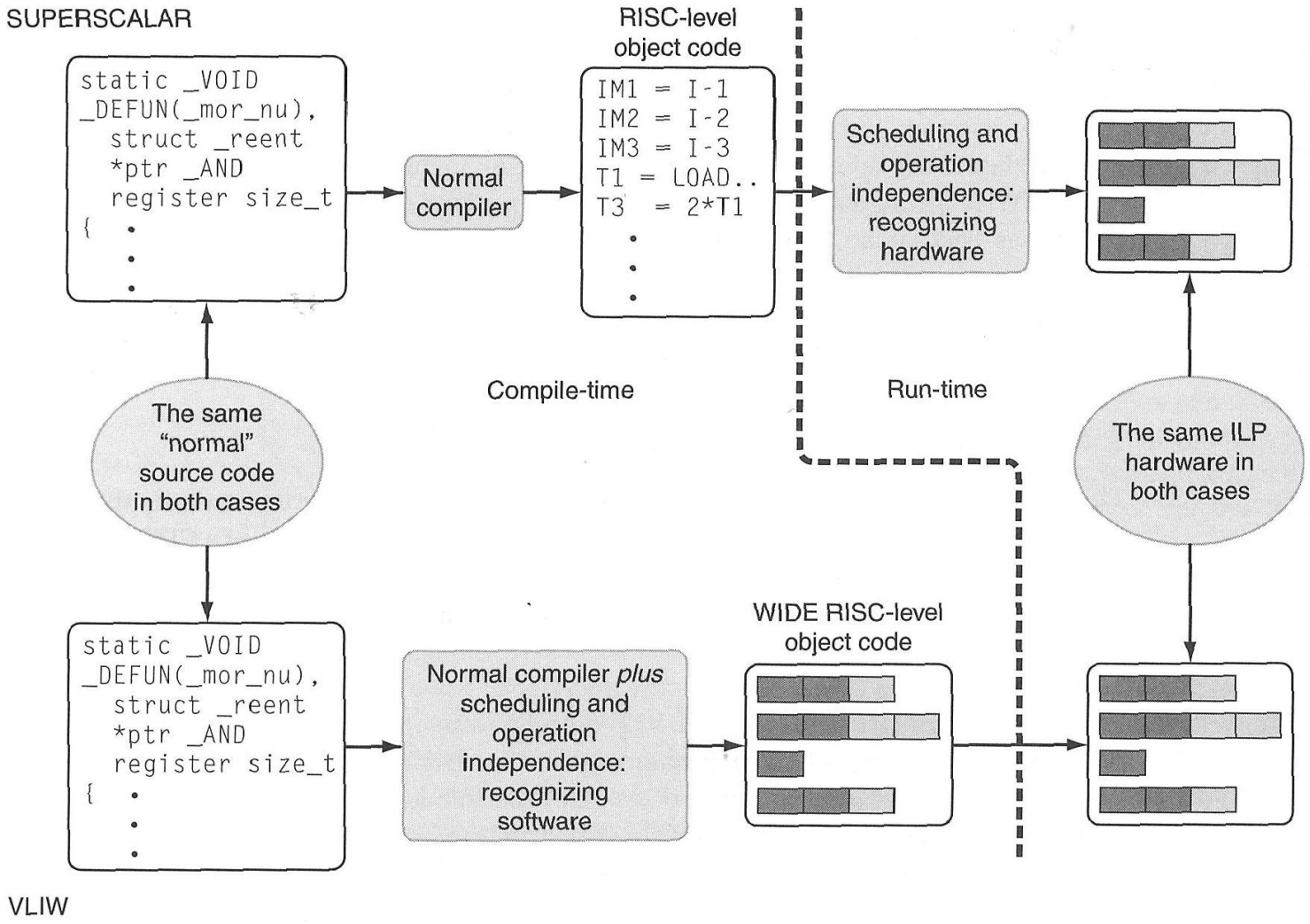
VLIW execution with degree $m = 3$

动态指令调度

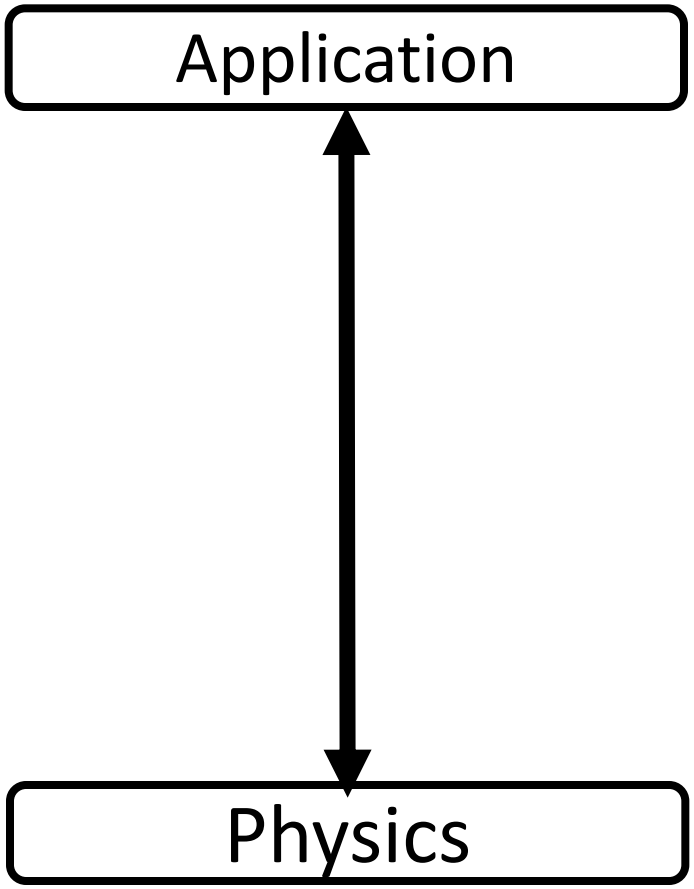
Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1					3
	I4			I3	I1	I2	4
I5	I6			I4			5
	I6		I5		I3	I4	6
			I6				7
					I5	I6	8

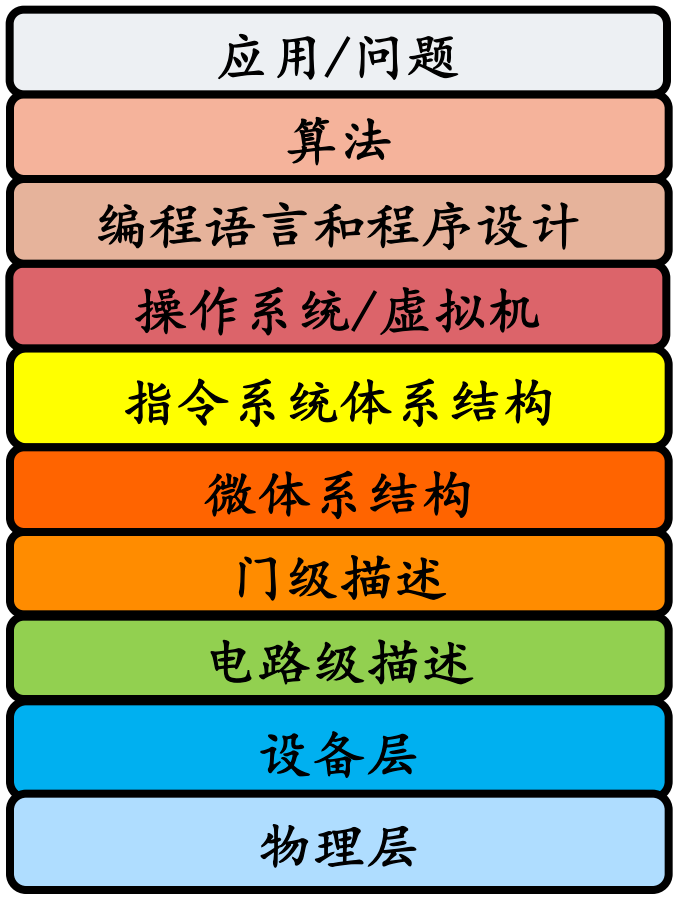
Decode		Window	Execute			Write		Cycle
I1	I2							1
I3	I4	*I1,I2*	I1	I2				2
I5	I6	*I3,I4*	I1		I3	I2		3
		I4,I5,I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

超标量vs超长指令字VLIW



计算机系统设计中的层次观





UCB EECS Courses

CS170：算法

CS164：编程语言和编译器

CS162：操作系统和系统编程

CS152：计算机系统结构

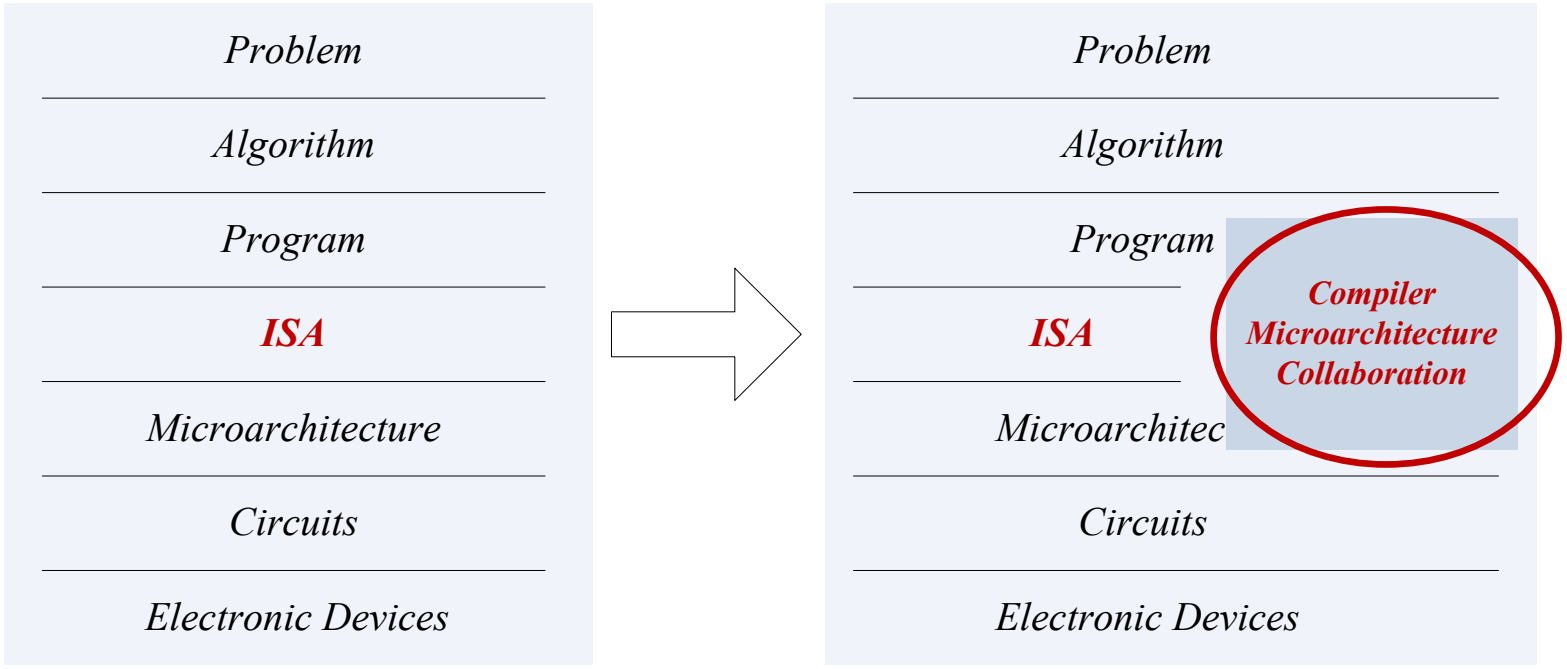
CS150：数字系统

EE141：集成电路

EE143：制造工艺

Gap too large to bridge in one step

跨层次的协作优化

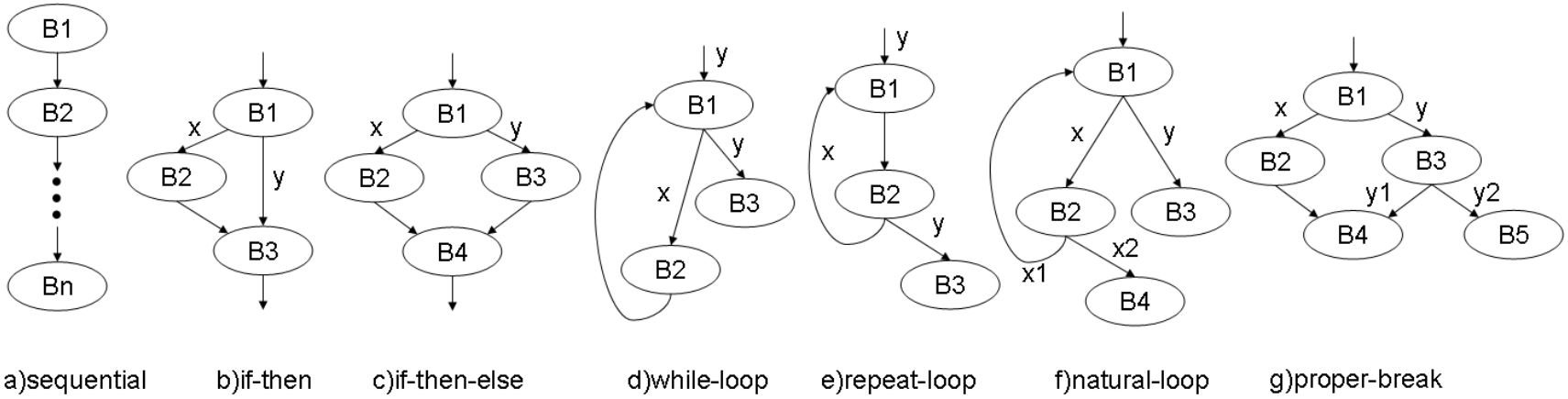


(a) Traditional layer

(b) Improved layers

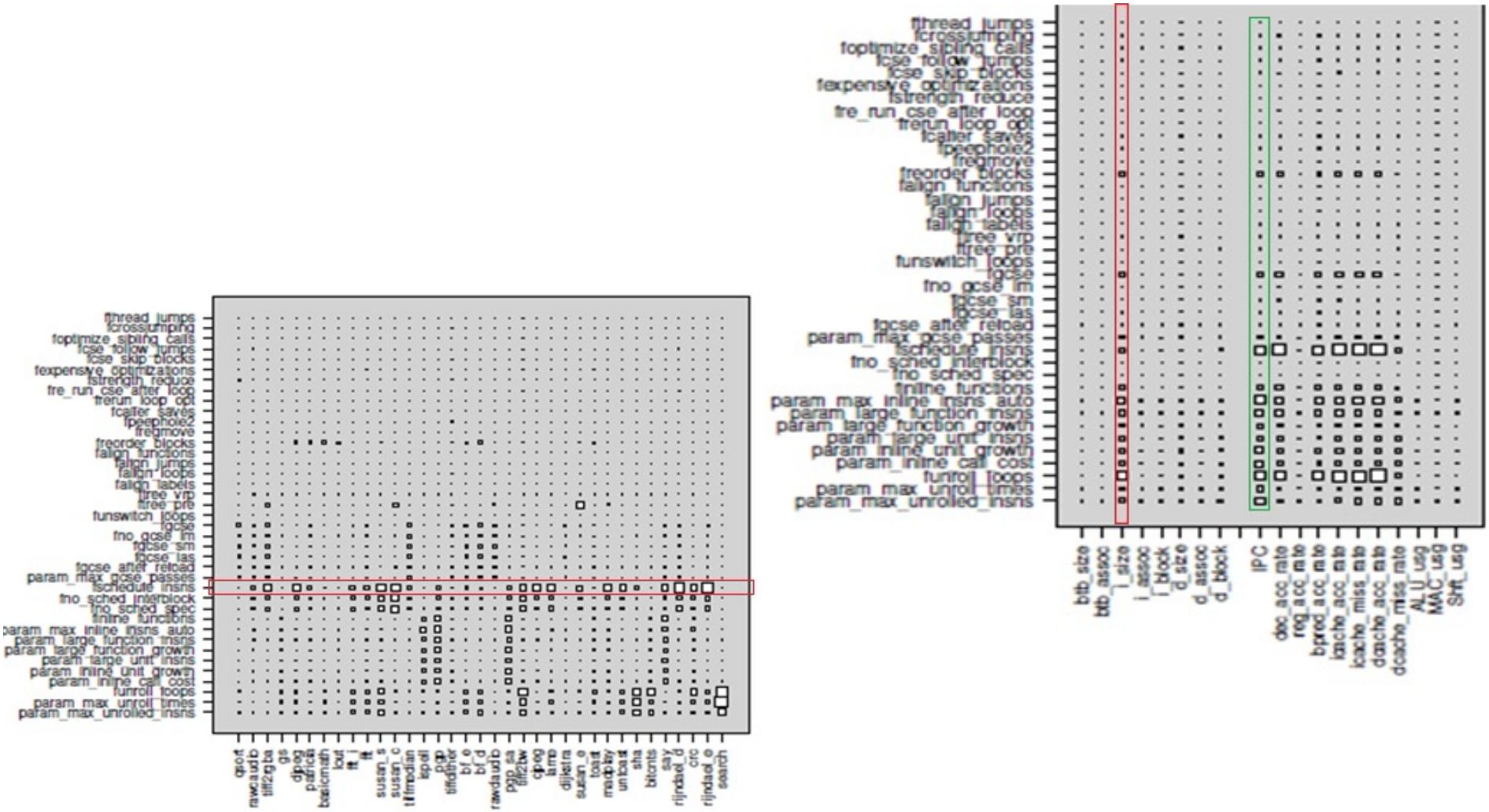
How to feed compiler-guided information to the hardware?

程序的结构特性



	静态特征	特征描述
1	SS_No.	典型子结构唯一标识
2	Edge_No.	典型子结构中控制流边的唯一标识
3	I_last_of_head	该边首基本块最后一条指令的操作码
4	Br_direction	该边首基本块最后一条指令的跳转方向
5	I_pre_last	该边首基本块最后一条指令的前一条指令的操作码

程序特性、架构特性对性能的影响



例：基于转移历史栈的转移预测器优化

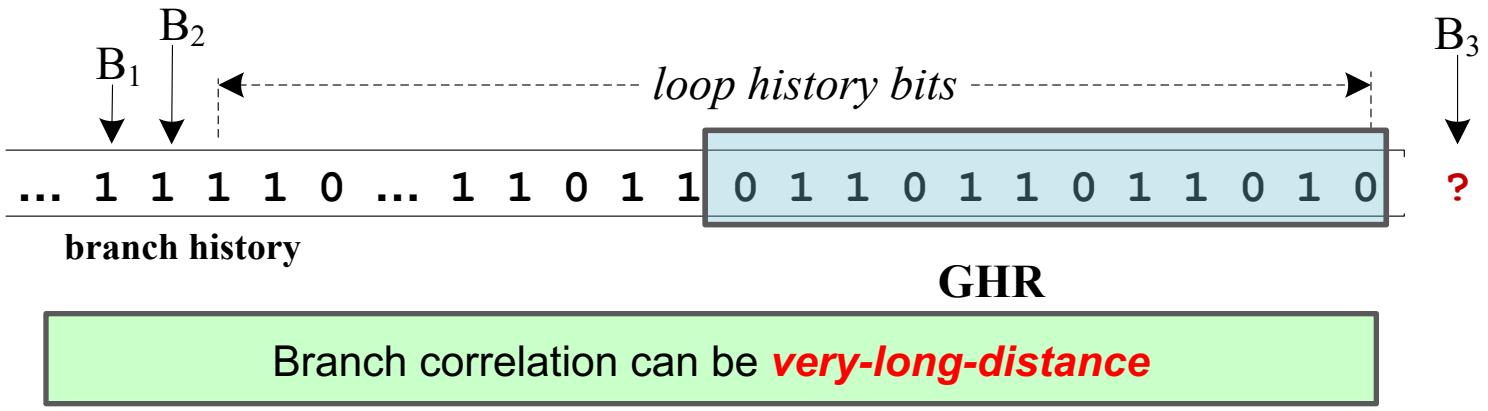
```
If (a==2)     //B1
    a=0;
If (b==2)     //B2
    b=0;
If (a!=b)     //B3
```

Expected source code

```
If (a==2)     //B1
    a=0;
If (b==2)     //B2
    b=0;
for(i=0;i<N;i++)
  …
If (a!=b)     //B3
```

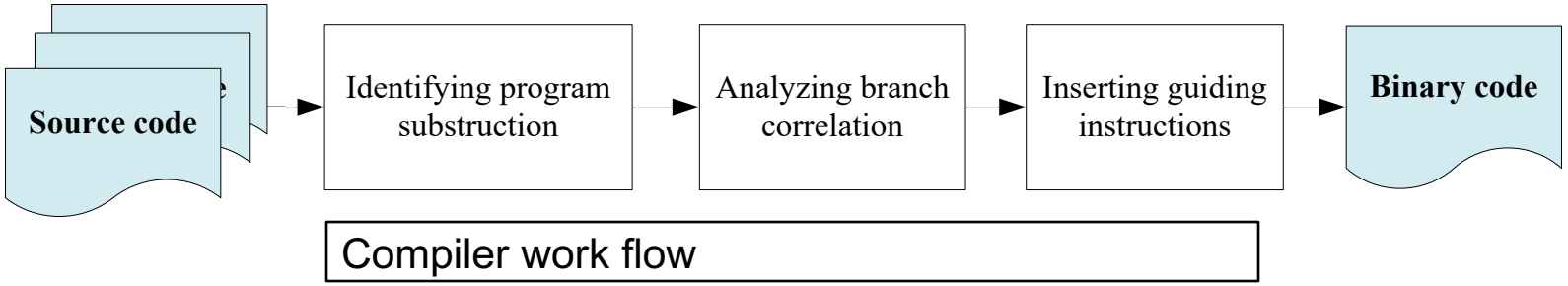
```
If (a==2)     //B1
    a=0;
If (b==2)     //B2
    b=0;
call fun1()
  …
If (a!=b)     //B3
```

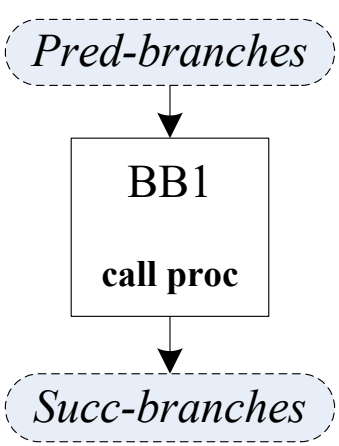
Unexpected source code

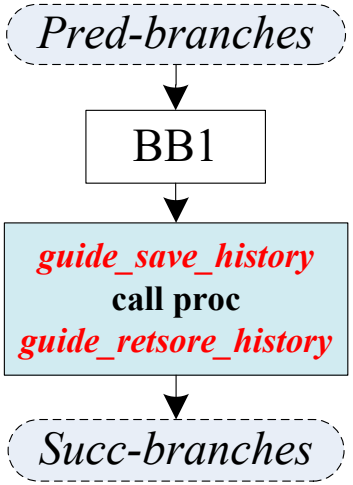


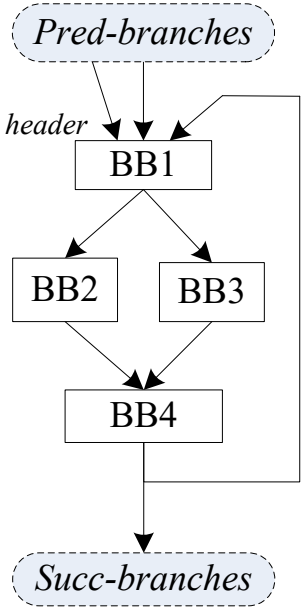
Branch correlation can be ***very-long-distance***

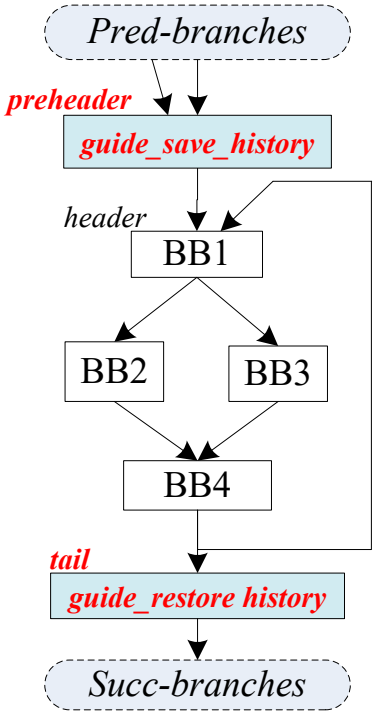
例：基于转移历史栈的转移预测器优化



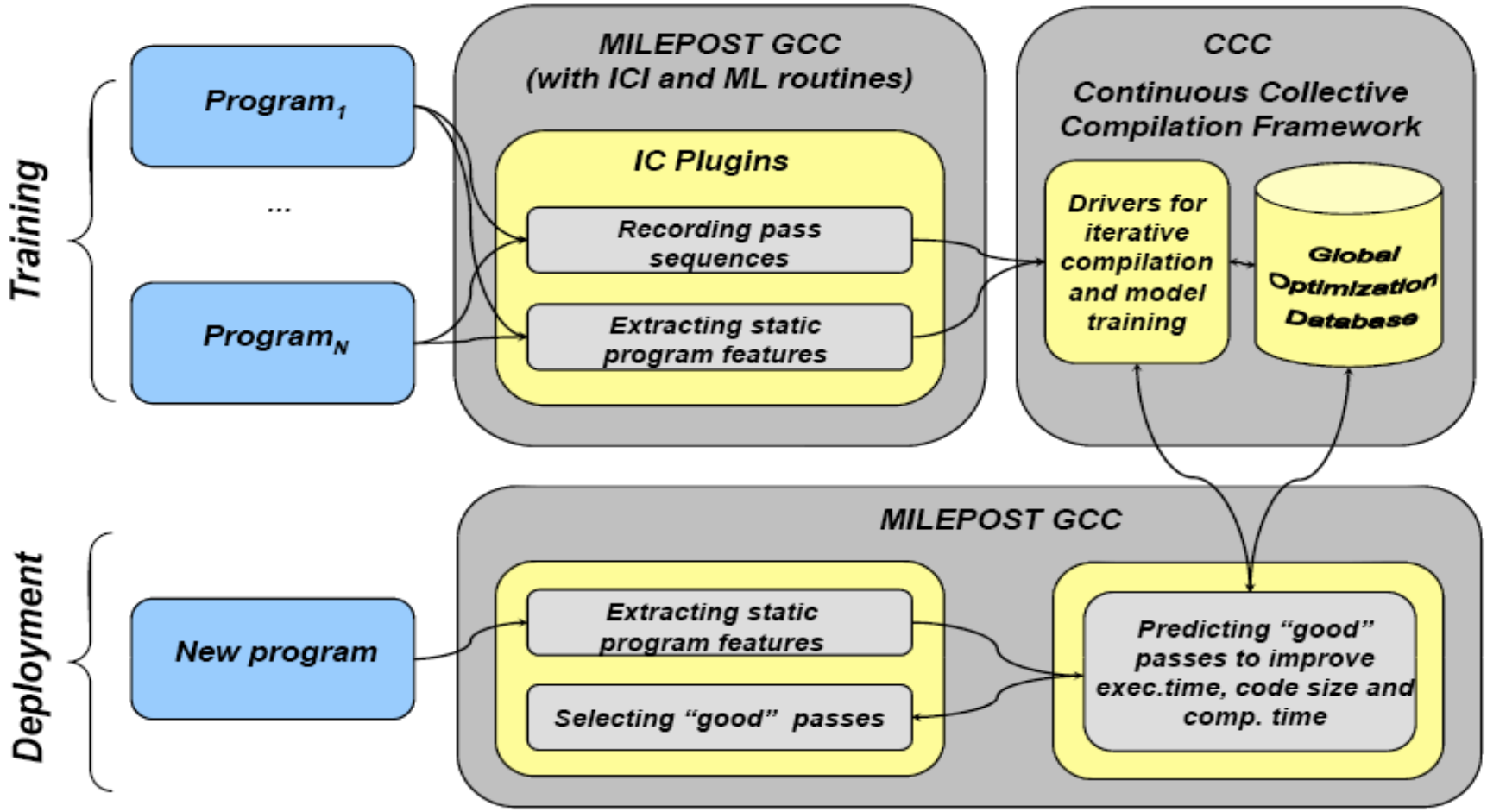






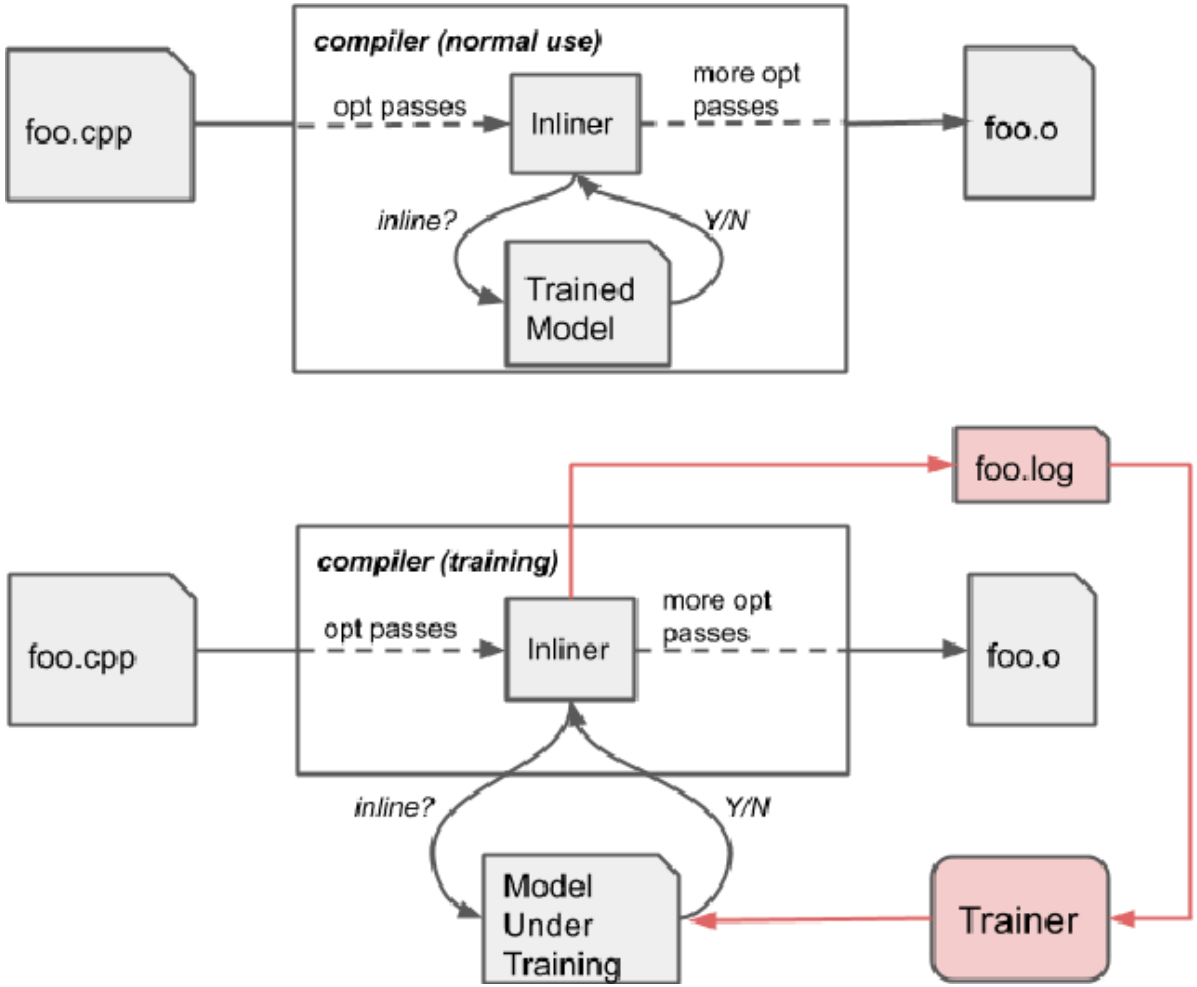


MILEPOST GCC Framework



Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle. ***MILEPOST GCC: machine learning based research compiler.*** *Proceedings of the GCC Developers' Summit, Ottawa, Canada, June 2008* ***(based on CGO'06 and HiPEAC'05 papers)***

MLGO：机器学习指导的编译优化

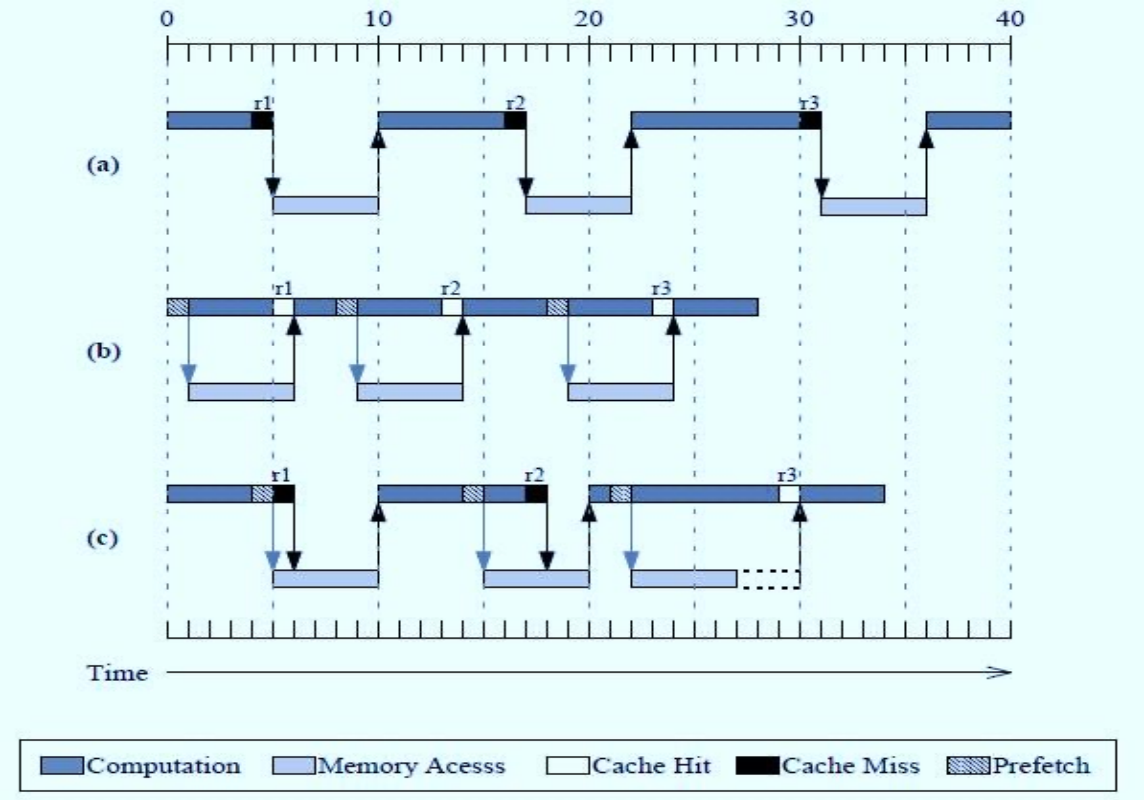


Review：基于仿射变换的若干应用

- 分布式内存计算机中数据的分配问题。（使用投影的方式决定哪个处理器使用数组的哪一部分）
- 多发射处理器中对关键资源的分配和使用。
- 对向量操作、SIMD指令的编译支持。
- 面向数据局部性的排布优化和预取优化。

数据预取及效果示意

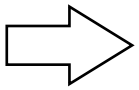
- 提前对未来预期访问的数据发出请求
- 计算和访存时间重叠



软件数据预取

- 由程序员/编译器调度预取指令
- 一般预取整个cache行，利用其空间局部性
- 无（功能语义上的）副作用
- 可以（但不是必须）和硬件预取相结合

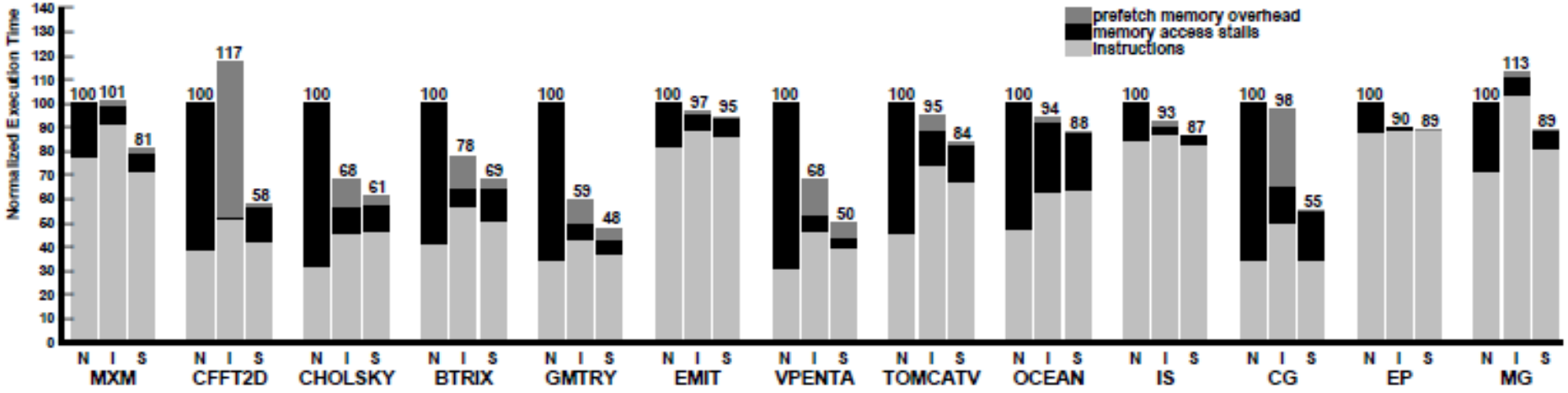
```
for (i = 0; i < 100; i++)
    x[i] = c * x[i];
```

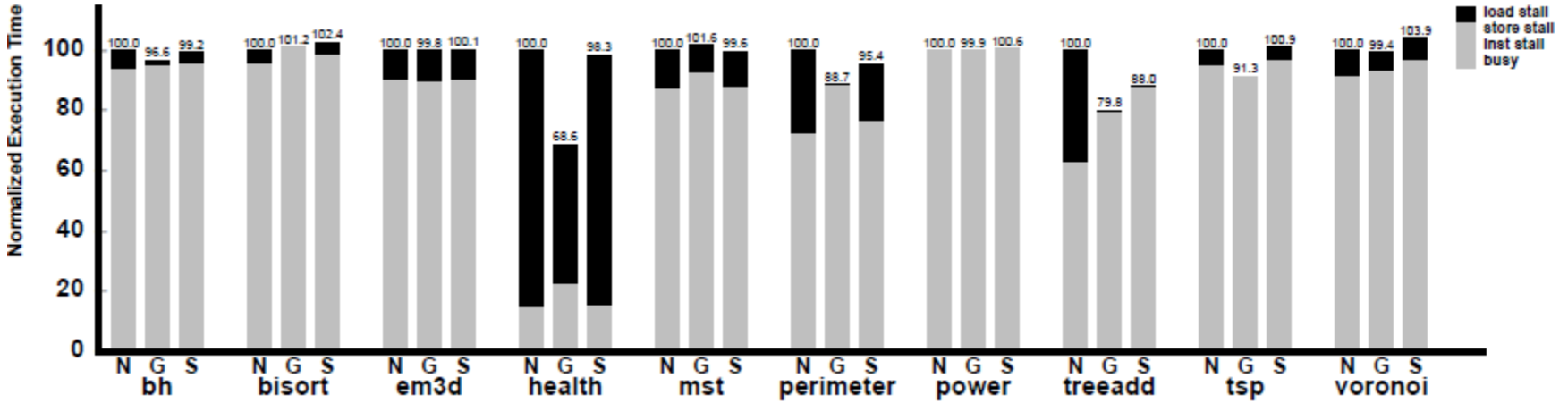


```
for (i = 0; i < 100; i++) {
    prefetch(x[i+k]);
    x[i] = c * x[i];
}
```

Where *k* depends on
(1) the miss penalty and
(2) the time it takes to execute an iteration *assuming hits*

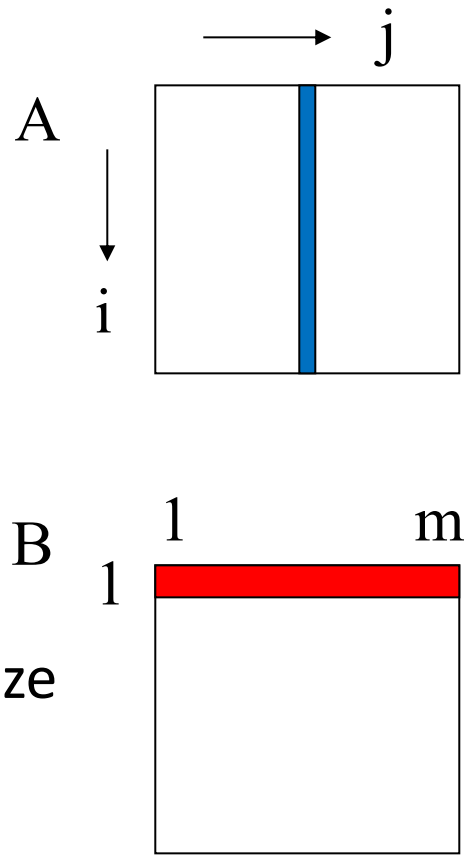
数据预取效果


Normalized Execution Time
prefetch memory overhead
memory access stalls
instructions
MXM
CFFT2D
CHOLSKY
BTRIX
GMTRY
EMIT
VPENTA
TOMCATV
OCEAN
IS
CG
EP
MG


Normalized Execution Time
load stall
store stall
Inst stall
busy
bh
bisort
em3d
health
mst
perimeter
power
treeadd
tsp
voronoi

插入预取代码示例

```
do j = 1, n
  do i = 1, m
    A(i,j) = B(1,i) + B(1,i+1)
    if (iand(i,7) == 0)
      prefetch (A(i+k,j))
    if (j == 1)
      prefetch (B(1,i+t))
   enddo
enddo
```

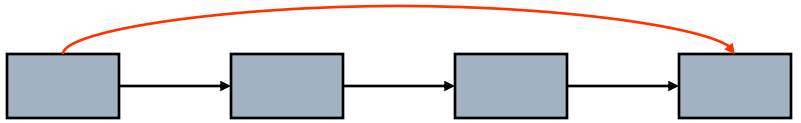


Assumed CLS = 64 bytes and data size = 8 bytes

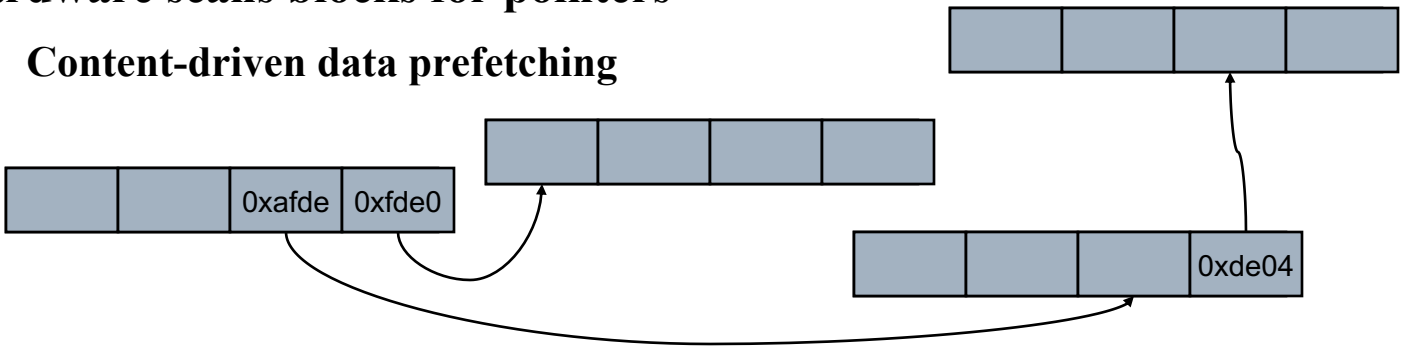
k and t are prefetch distance values

对复杂数据的预取

- 预取内容
  - Next level of tree: n+1, n+2, n+?
    - Entire tree? Or just one path
  - Next node in linked list: n+1, n+2, n+?
  - Jump-pointer prefetching



- 预取的时机
  - Software places jump pointers in data structure
  - Hardware scans blocks for pointers
    - Content-driven data prefetching



预取的困难点

- **Prefetching is effective only if all of these are true:**
  - **There is spare memory bandwidth to begin with**
    - **Otherwise prefetches could make things worse**
  - **Prefetches are *accurate***
    - **Only useful if you prefetch data you will soon use**
  - **Prefetches are *timely***
    - **Ie., prefetch the right data, but not early enough**
  - **Prefetched data doesn't displace other in-use data**
    - **Eg: bad if PF replaces a cache block about to be used**
  - **Latency hidden by prefetches outweighs their cost**
    - **Cost of many useless prefetches could be significant**
- **Ineffective prefetching can hurt performance!**

软件数据预取的代价

- Requires memory instruction resources
  - A prefetch instruction for each access stream
- Issues every iteration, but needed less often
  - If branched around, inefficient execution results
  - If conditionally executed, more instruction overhead results
  - If loop is unrolled, code bloat results
- Redundant prefetches get in the way
  - Resources consumed until prefetches discarded!
- Non redundant need careful scheduling
  - Resources overwhelmed when many issued & miss
- Redundant prefetches increases power/energy consumption

获取存储参数：lstopo

Run lstopo on UG machine, gives:

```
Machine (3829MB) + Socket #0
  L2 #0 (6144KB)

     L1 #0 (32KB)+Core #0+PU #0 (phys=0)
     L1 #1 (32KB)+Core #1+PU #1 (phys=1)

  L2 #1 (6144KB)

     L1 #2 (32KB) + Core #2 + PU #2 (phys=2)
     L1 #3 (32KB) + Core #3 + PU #3 (phys=3)
```

4GB RAM

2X 6MB L2 cache

2 cores per L2

32KB L1 cache per core

获取存储参数： 一级数据高速缓存

- ls /sys/devices/system/cpu/cpu0/cache/index0
  - coherency_line_size: 64 // 64B cache lines
  - level: 1 // L1 cache
  - number_of_sets
  - physical_line_partition
  - shared_cpu_list
  - shared_cpu_map
  - size:
  - type: data // data cache
  - ways_of_associativity: 8 // 8-way set associative

获取存储参数：二级高速缓存

- ls /sys/devices/system/cpu/cpu0/cache/index2
  - coherency_line_size: 64 // 64B cache lines
  - level: 2 // L2 cache
  - number_of_sets
  - physical_line_partition
  - shared_cpu_list
  - shared_cpu_map
  - size: 6144K
  - type: Unified // unified cache, means instructions and data
  - ways_of_associativity: 24 // 24-way set associative

使用perf工具访问硬件计数器

- Perf工具可以帮助访问性能计数器
- 例如要测量foo程序的L1 cache失效

```
 perf stat -e L1-dcache-load-misses foo
7803 L1-dcache-load-misses     # 0.000 M/sec
```

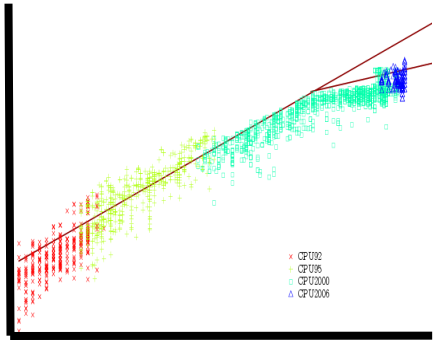
- 查看能够测量的所有事件：

```
perf list
```

- 可以同时测量多个事件

多核编译支持

Benchmark	Threads at Peak	Speedup	LOCs Changed
164.gzip	32+	29.91	26
175.vpr	15	3.59	1
176.gcc	16	5.06	17
181.mcf	32+	2.84	0
186.crafty	32+	25.18	9
197.parser	32+	24.50	2
253.perlbmk	5	1.21	0
254.gap	10	1.94	1
255.vortex	32+	4.92	0
256.bzip2	12	6.72	0
300.twolf	8	2.06	1
GEOMEAN	17	5.54	
ARITHMEAN	20	9.81	

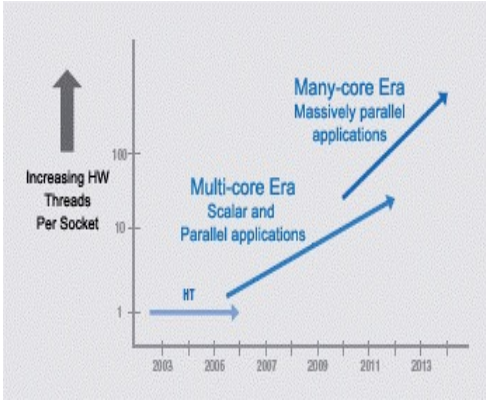


M.L.O.P.:

5 Generations

32 Cores

5.3x Speedup



“Hello Word” Example/1

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {



          printf("Hello World\n");



   return(0);
}
```

“Hello Word” - An Example/2

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    #pragma omp parallel
    {
            printf("Hello World\n");

    } // End of parallel region

    return(0);
}
```

“Hello Word” - An Example/3

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) {

#pragma omp parallel
  {
    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();

    printf("Hello World from thread %d of %d\n",
         thread_id, num_threads);
  }

   return(0);
}
```

Directives

Runtime Environment

“Hello Word” - An Example/4

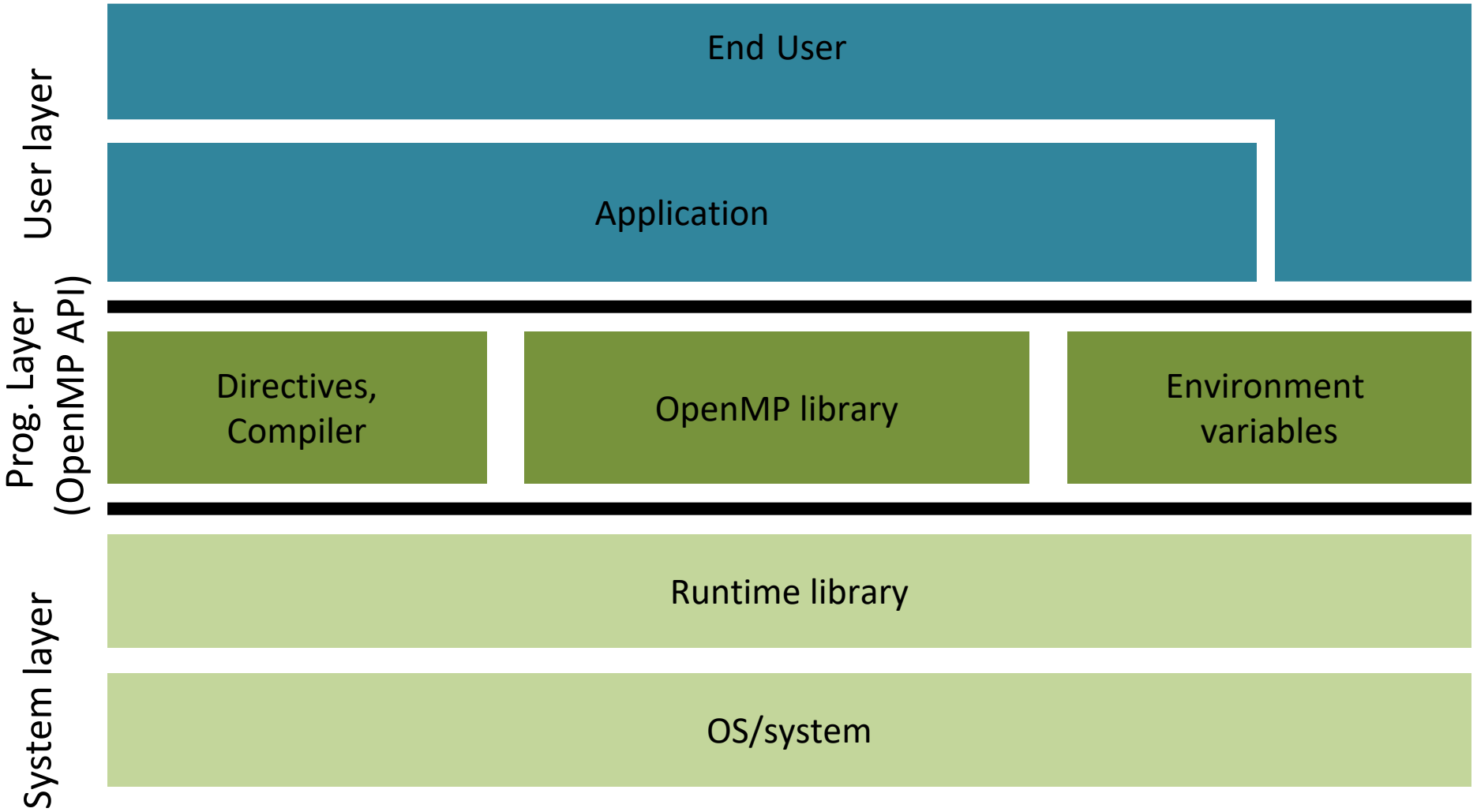
```
$
$ gcc -fopenmp helloomp.c -o helloomp
$ ls helloomp
helloomp
$
$ export OMP_NUM_THREADS=2
$  ./helloomp
Hello World from thread 1 of 2
Hello World from thread 0 of 2
$
$ export OMP_NUM_THREADS=4
$ ./helloomp
Hello World from thread 0 of 4
Hello World from thread 1 of 4
Hello World from thread 3 of 4
Hello World from thread 2 of 4
$
$ export OMP_NUM_THREADS=4
$ ./helloomp
Hello World from thread 1 of 4
Hello World from thread 2 of 4
Hello World from thread 3 of 4
Hello World from thread 0 of 4
```

Environment Variable

Environment Variable: it is similar to program arguments used to change the configuration of the execution without recompile the program.

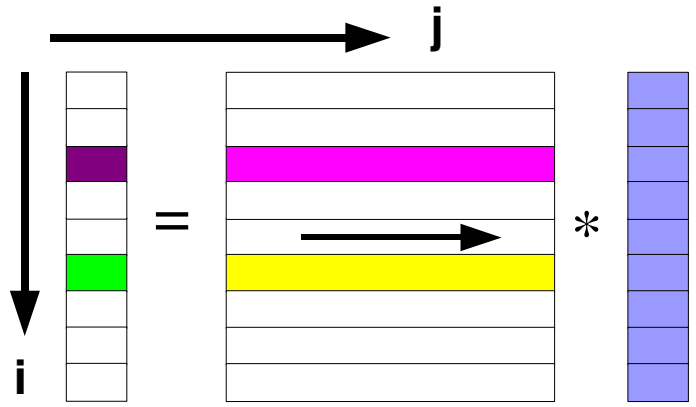
NOTE: the order of print

OpenMP并行计算栈框架

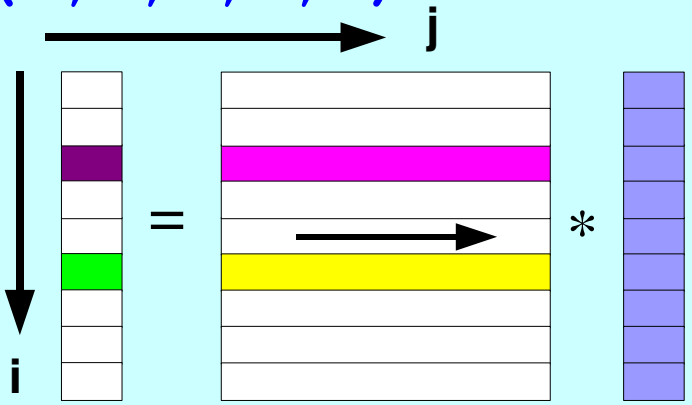


仍然是矩阵乘法

```
for (i=0; i<m; i++)
{
  a[i] = 0.0;
  for (j=0; j<n; j++)
     a[i] += b[i][j]*c[j];
}
```

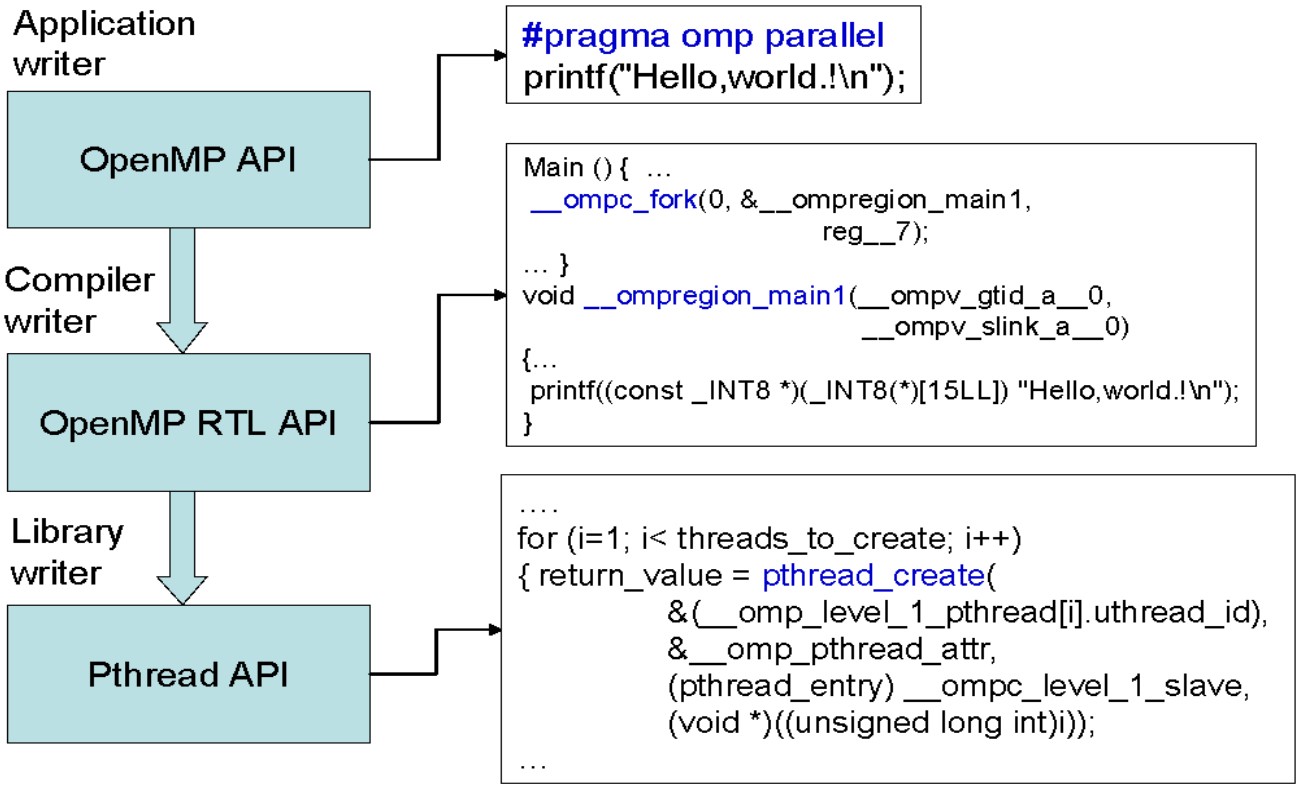



```
#pragma omp parallel for default(none) \
        private(i,j) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
  a[i] = 0.0;
  for (j=0; j<n; j++)
     a[i] += b[i][j]*c[j];
 }
```



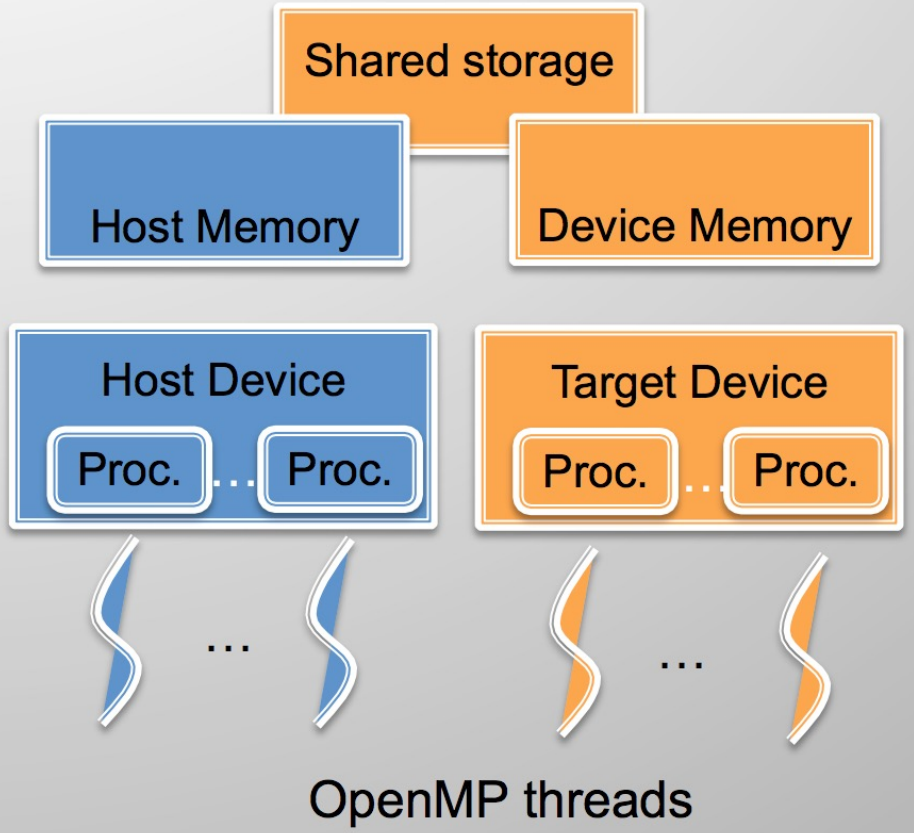
OpenMP实现

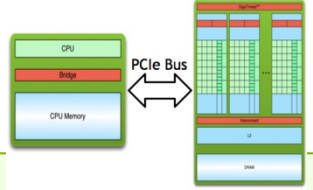
The role of the OpenMP runtime library



OpenMP 4.0 for Accelerators

- Device: a logical execution engine
  - Host device: where OpenMP program begins, one only
  - Target devices: **1 or more** accelerators
- Memory model
  - Host data environment: one
  - Device data environment: one or more
  - Allow shared host and device memory
- Execution model: Host-centric
  - Host device : “offloads” code regions and data to accelerators/target devices
  - Target Devices: still fork-join model
  - Host waits until devices finish
  - Host executes device regions if no accelerators are available /supported





CUDA Offloading Computation

```
// DAXPY in CUDA
__global__
void daxpy(int n, double a, double *x, double *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void) {
     int n = 1024;
    double a;
    double *x, *y; /* host copy of x and y */
    double *x_d; *y_d; /* device copy of x and y */
    int size = n * sizeof(double)
    // Alloc space for host copies and setup values
    x = (double *)malloc(size); fill_doubles(x, n);
    y = (double *)malloc(size); fill_doubles(y, n);

    // Alloc space for device copies
    cudaMalloc((void **)&d_x,  size);
    cudaMalloc((void **)&d_y, size);

    // Copy to device
    cudaMemcpy(d_x,  x,  size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);

    // Invoke DAXPY with 256 threads per Block
    int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x_d, y_d);

    // Copy result back to host
    cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(x); free(y);
    cudaFree(d_x); cudaFree(d_y);
    return 0;
}
```

Memory allocation and data copy-in

Offloading computation

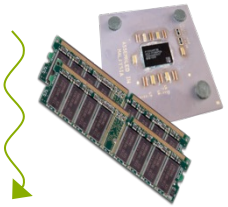
Data copy-out and deallocation

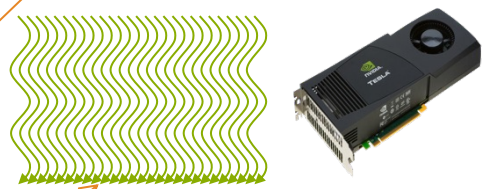
CUDA kernel

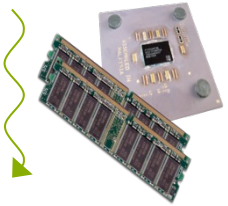
serial code

parallel exe on GPU

serial code







链接时刻优化

Technical Showcase

Linux Foundation CE Workgroup / Embedded Linux Conference 2013

gcc Link-Time-Optimization of ARM Linux kernel

What is demonstrated

- Possibly the most ~~boring~~ thrilling demo ever
- Gcc has compile-time option to do link-time optimization
- Andi Kleen created patches to support this compiler option
  - He demonstrated on an Intel CPU
- This is first demonstration of LTO kernel running on ARM
  - World's first, that I know of !!!
- LTO supports whole-program optimization, at final link time
  - Slow link step, but good code optimizations

Hardware Information

TI panda board
mem=24M

pandaboard.org

What was improved

System size and performance

6% reduction in image size (384K)

Kernel	non-lto	lto
Compile time	1m58s	3m22s
Image size	5.85M	5.46M
Meminfo Total	17804K	18188K
Meminfo Free	10908K	11260K
LTP time		

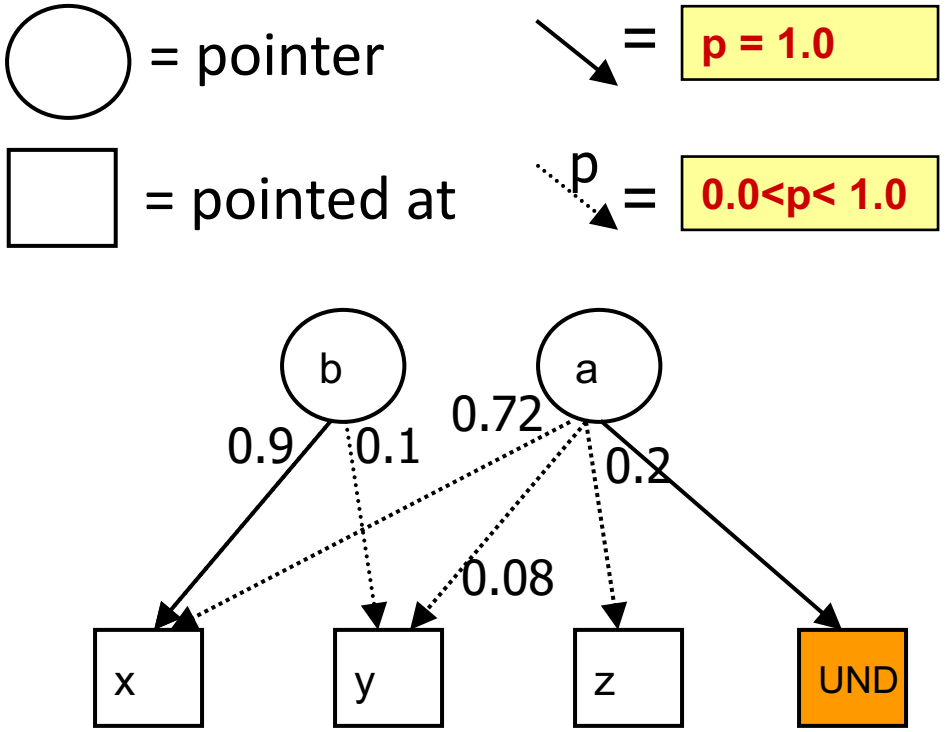
Source code or detail technical information availability

http://lwn.net/Articles/512548/

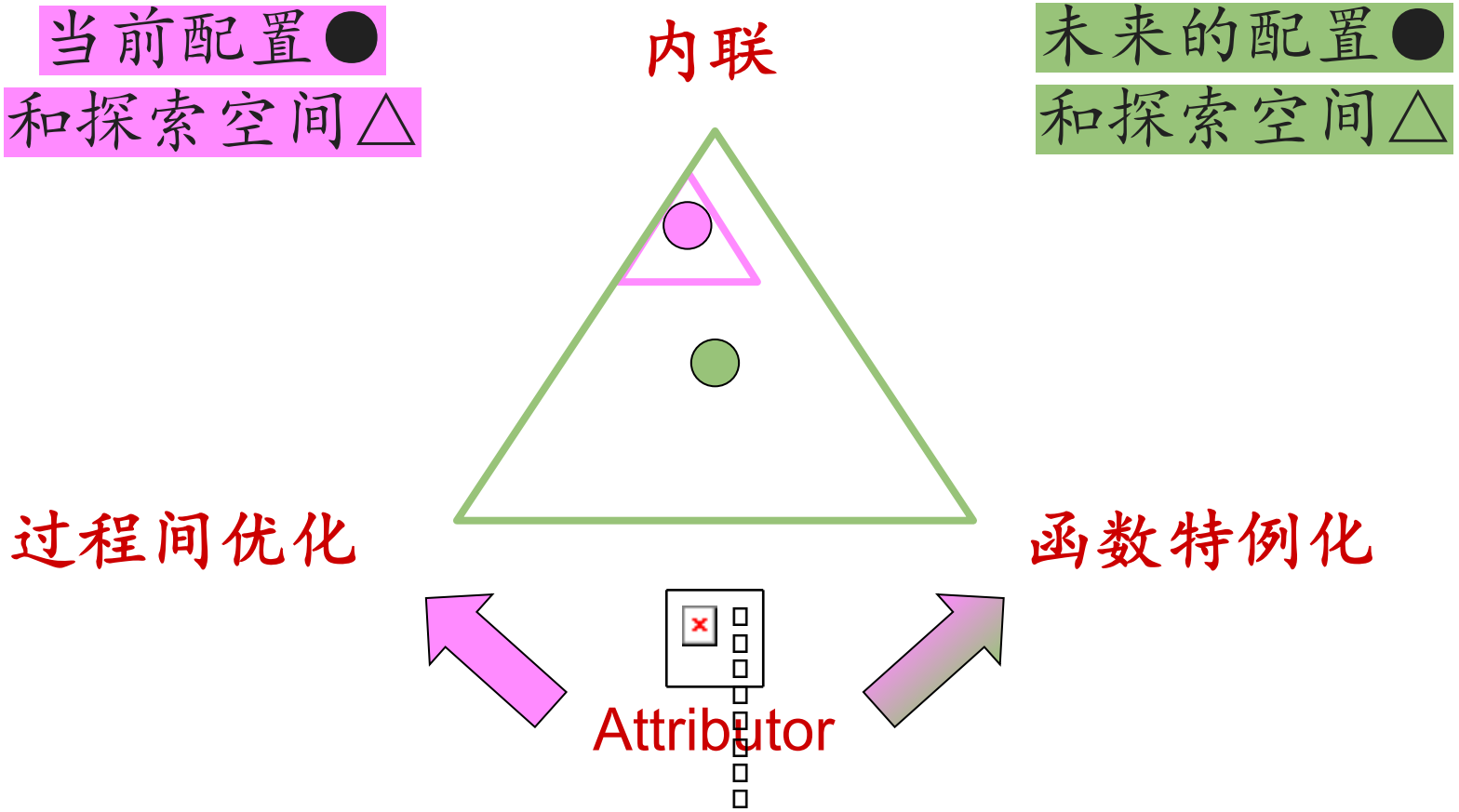
git://github.com/andikleen/linux-misc

基于剖视的优化

```
int x, y, z, *b = &x;
void foo(int *a) {

   if(…)⇨0.1 taken(edge profile)
      b = &y;

   if(…)⇨0.2 taken(edge profile)
      a = &z;
   else(…)
      a = b;

   while(…) {
      x = *a;
      …
   }
}
```

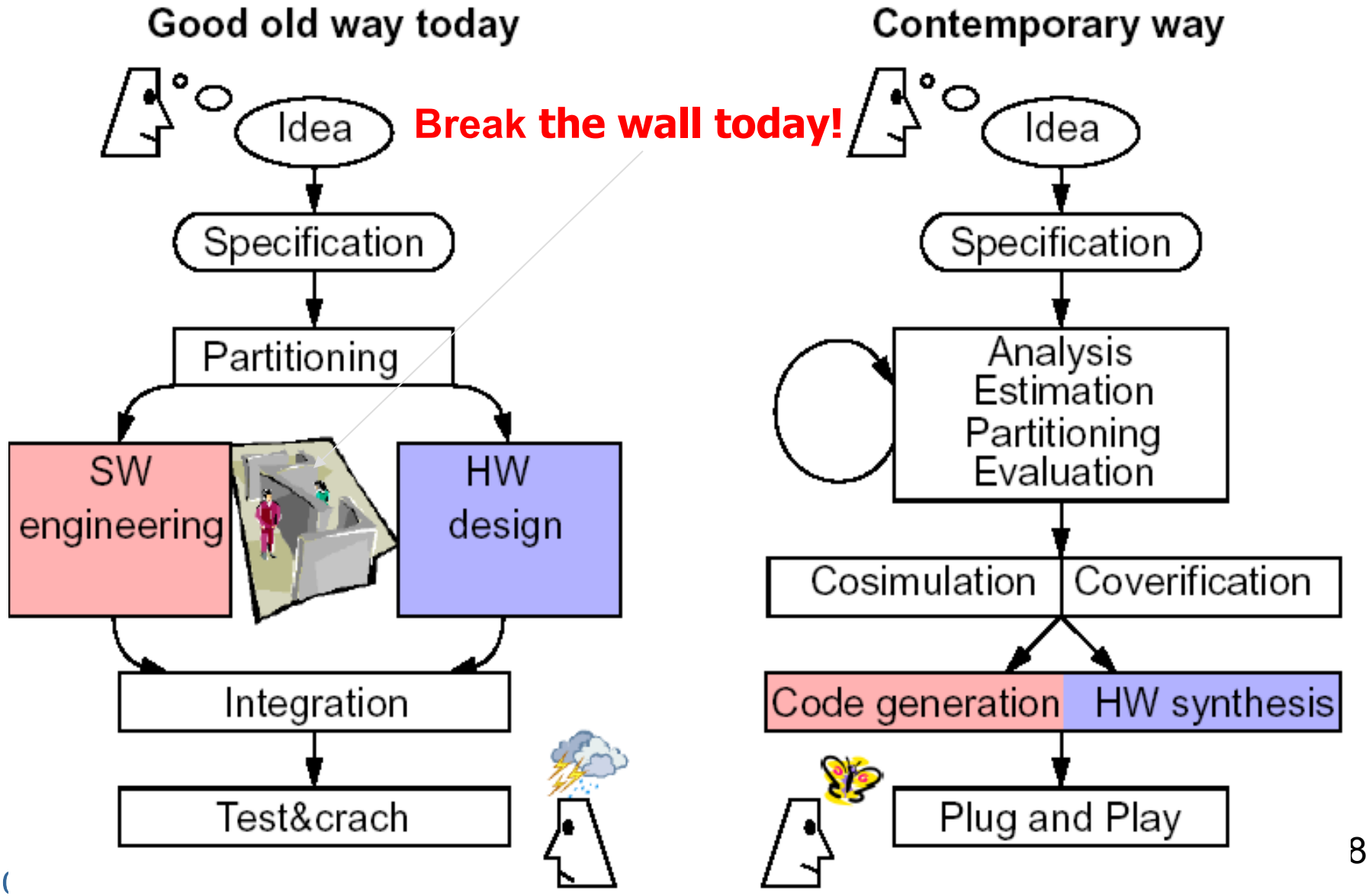


内联、过程间优化和特例化



The Attributoris an *interprocedural fixpoint iteration framework,* with lots of built-in features.

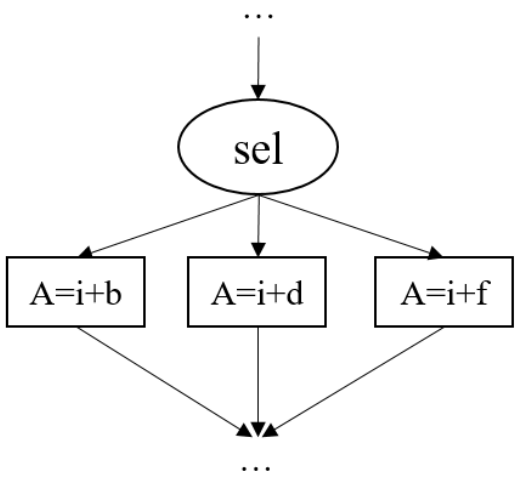
设计流程的变化


Good old way today
Idea
Specification
Partitioning
SW engineering
HW design
Integration
Test&crach
Break the wall today!
Contemporary way
Idea
Specification
Analysis
Estimation
Partitioning
Evaluation
Cosimulation
Coverification
Code generation
HW synthesis
Plug and Play

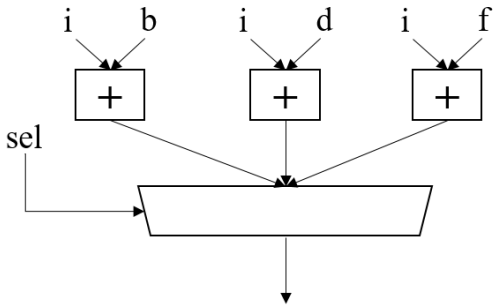
仍然还是公共子表达式消除

Common Operation - Chisel

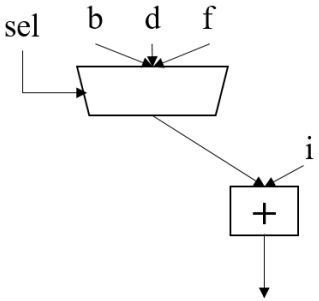
```
switch(sel){
    is(2.U){result := in+b};
    is(1.U){result := in+d};
    is(0.U){result := in+f};
}

io.out := result
```



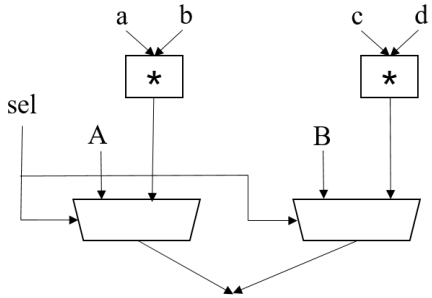
主要区别	编译	综合
信息传递	临时变量	电路连线
优化	操作	面积/速度
是否并行	串行	并行



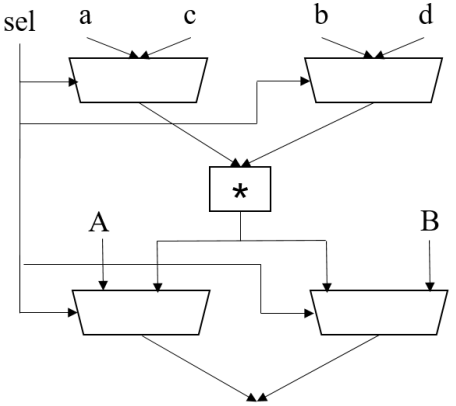
优化前



优化后



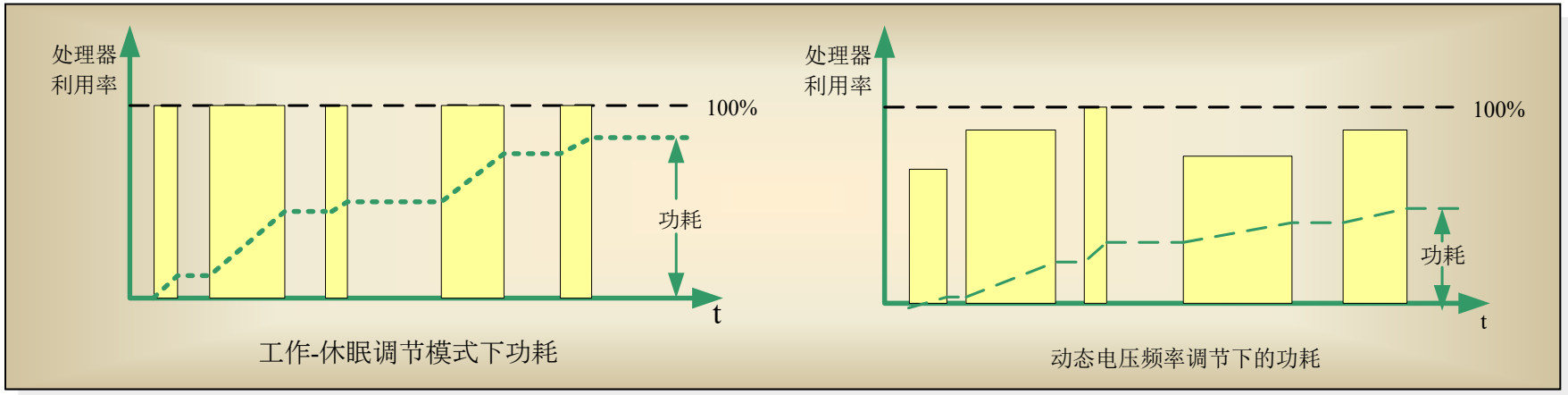
优化前



优化后

低功耗编译(1)

- 编译制导的基于DVS功耗优化
  - 通过剖视信息或者人工干预，加入制导信息。编译器识别制导信息加入电压切换片段，从而有效实现细粒度和程序内DVS。
  - 根据相关文献和初步评测数据，该方法带来的平均功耗下降可在20%左右。

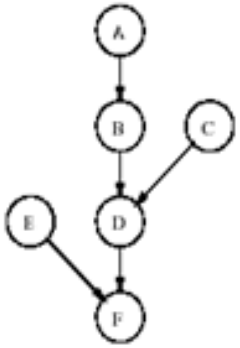


低功耗编译(2)

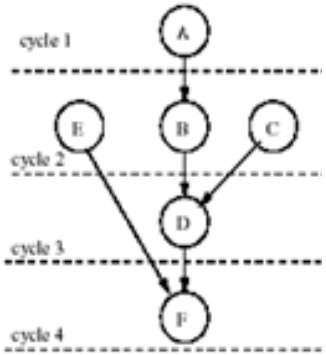
- 功耗约束的代码生成研究
  - 为每条指令添加功耗权重信息，可选择弱强度指令替换高强度指令以降低功耗，在编译器中根据选择选择不同的代码生成。
  - 重新排布代码以降低逻辑翻转次数，可以减少20-30%的翻转切换，但可能带来1-2%的性能损失。
  - 同样的，通过调整程序代码排布，可降低所需电压。

(a) 10010011
(b) 11101000
(c) 10111011
(d) 01110100
total switches: 24

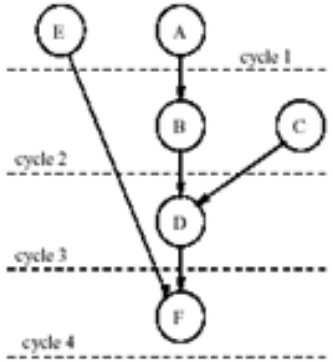
(a) 10010011
(c) 10111011
(b) 11101000
(d) 01110100
total switches: 18



(a) Example DDG



(b) Peak Power = 3
Energy = 6



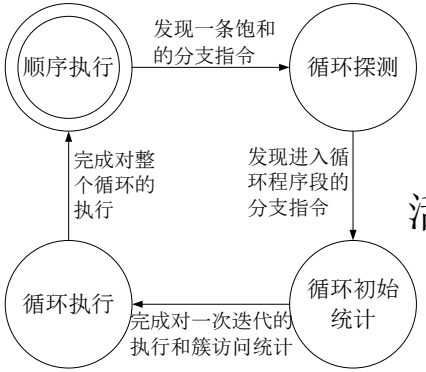
(c) Peak Power = 2
Energy = 6

低功耗编译(3)

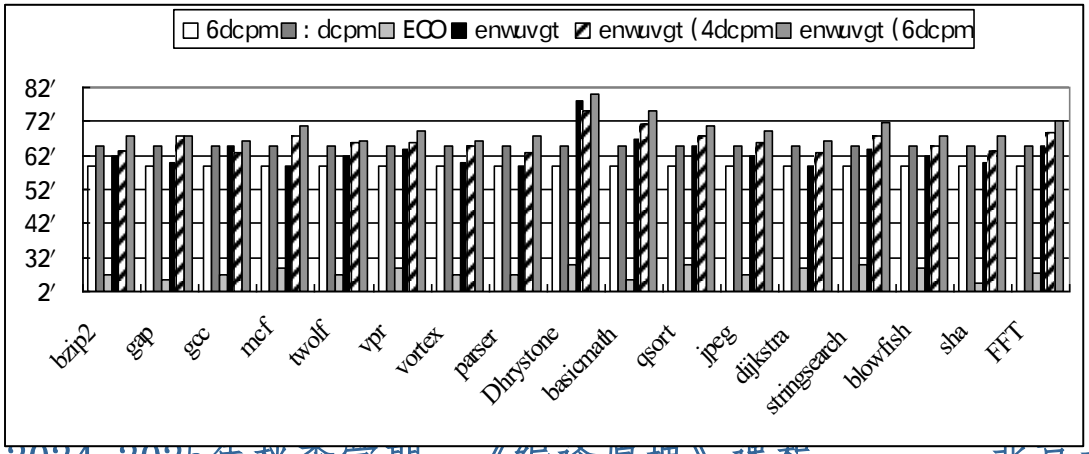
□ 基于编译指导和寄存器活跃度信息的低功耗寄存器堆

V	TAddr	LAAddr	Cause	Counter	SSR	Timer	Loop

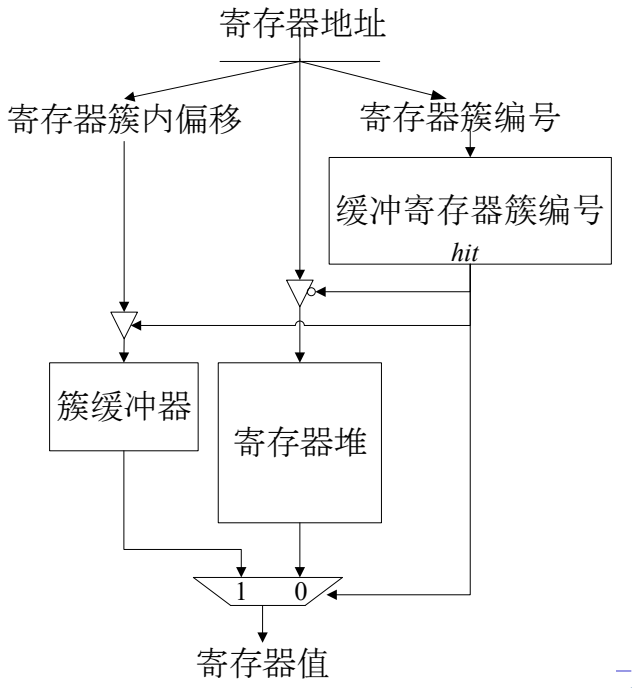
扩展RBB动态探测循环程序段



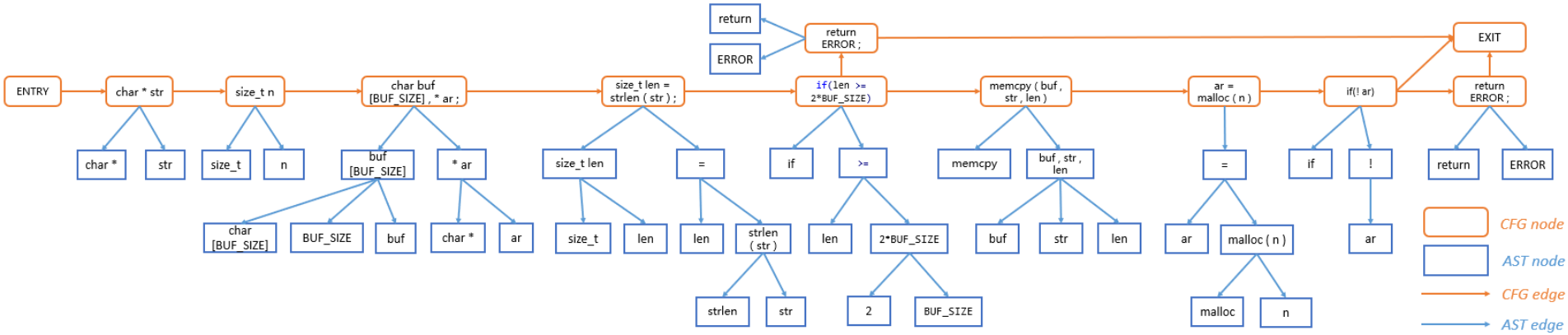
活动控制器状态转换

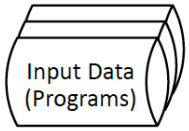


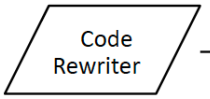
- 簇缓冲器中保存的寄存器簇
- 非循环执行状态：调用/返回时使用的寄存器簇
- 循环执行状态：迭代中少数频繁访问的簇
- 寄存器堆动态功耗和静态功耗节省40%

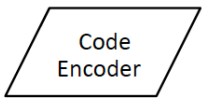


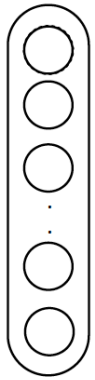
安全性增强（1）





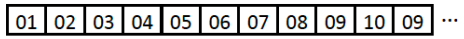






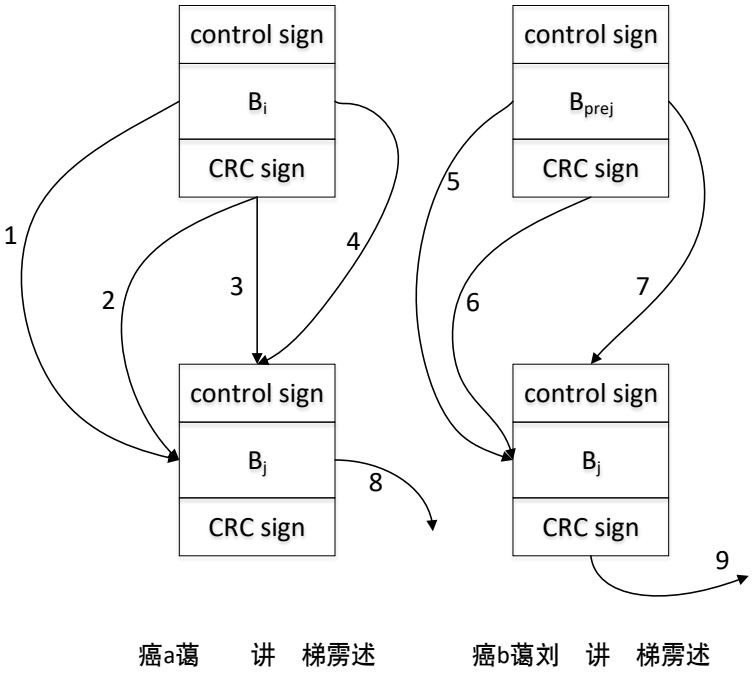

```c
/* goodG2B(): use goodsource and
badsink by changing the sources*/
static void goodG2B(){
   int64_t * data;
   data = NULL;
   while(1){
/* FIX: Use memory allocated  */
      int64_t dataGoodBuffer;
      data = &dataGoodBuffer;
/* Initialize and make use of data */
      *data = 5LL;
      printLongLongLine(*data);
      break;
   }
   while(1){
      break;
   }
}
```

```c
static void FUN1(){
   int6_t * VAR1;
   VAR1 = NULL;
   while(1){
      int64_t VAR2;
      VAR2 = &VAR3;
      *VAR2 = 5LL;
      FUN2(*VAR1);
      break;
        }
   while(1){
      break;
   }
}
```

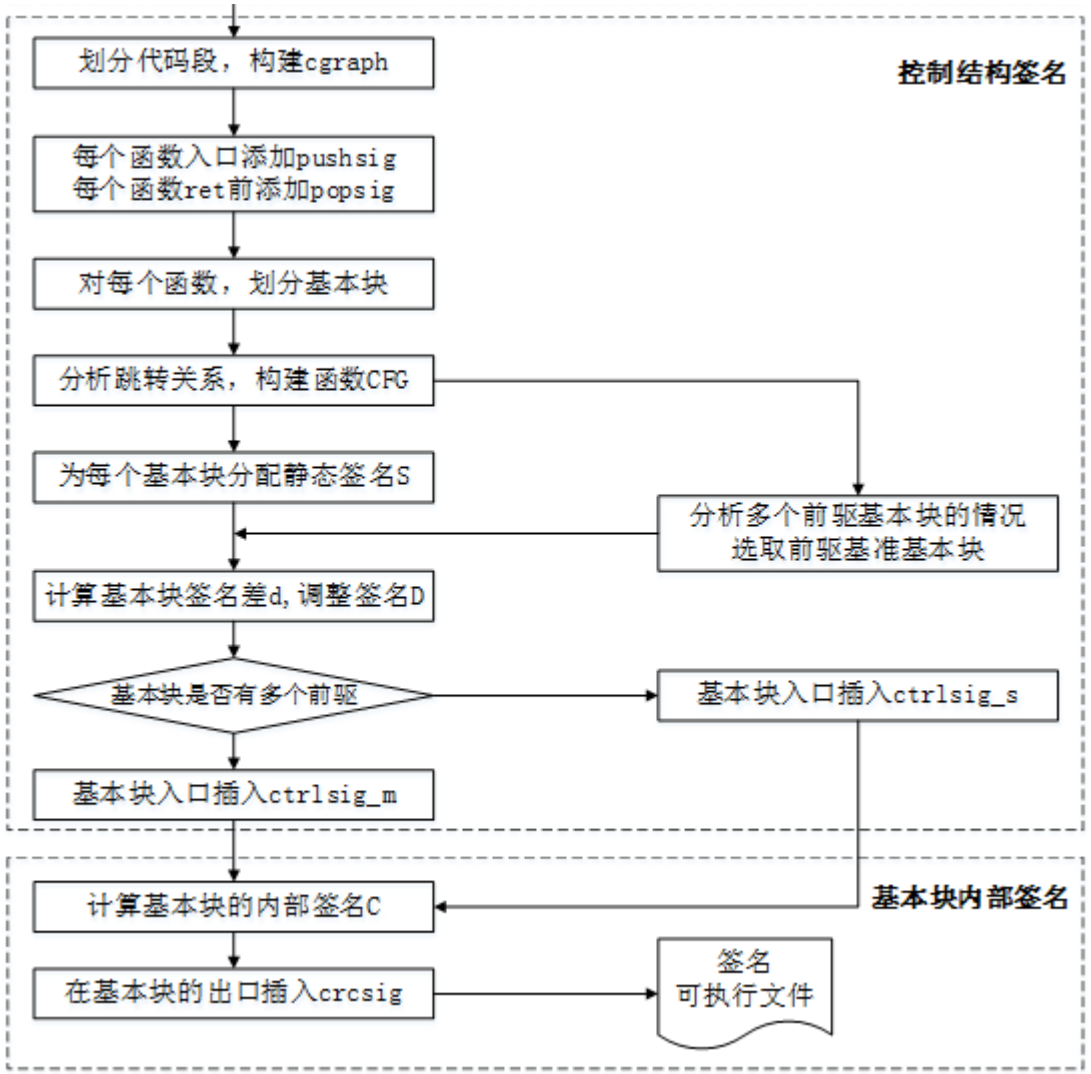


ID	01	02	03	04	05	06	07	08	09	10	...
Token	static	void	FUN1	(	)	{	int64_t	*	VAR1	;	

安全性增强（2）



癌a谶　　讲　梯霁述　　癌b谶刘　讲　梯霁述



GCC的编译优化选项

- -O1完成简单优化，例如CP, CF, CSE, DCE, LICM, 小函数内联等。-O2增加复杂、激进的优化。-O3在时间空间方面做出更多权衡，进行函数内联和循环展开。
- -floop-unroll-and-jam执行外部循环展开和内部循环融合，-floop-interchange执行嵌套循环中的循环互换，增强数据局部性。一些已有的优化器也经过了改进，包括-floop-nest-optimize和-ftree-loop-distribution。所有优化器都是使用-O3标识默认启用的。