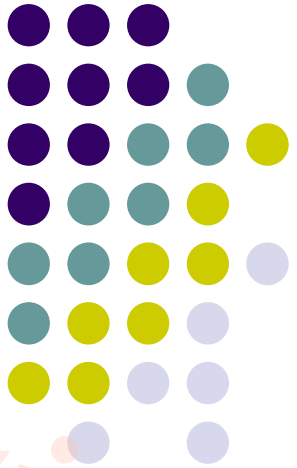
嵌入式系统设计与应用

第五章 程序设计与分析（3）

西安交通大学电信学院

孙宏滨



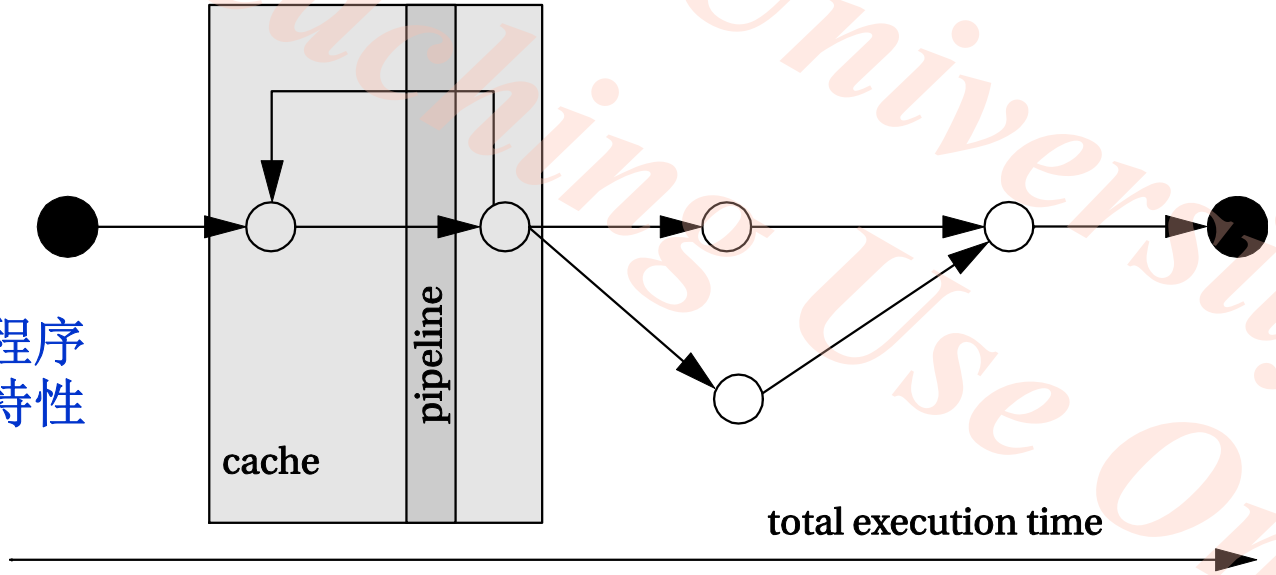
6 程序级性能分析

- 嵌入式软件为何需要性能分析？
  - 嵌入式系统必须满足一定的截止时限
    - 很快不一定代表一定足够快。
  - 需要能够分析程序执行时间
    - 最差情况，不是典型情况
  - 需要技术来可靠的改进程序执行的时间

程序级性能分析

- 我们需要详细的分析程序性能，优化依赖于分析：
  - 实时性能行为，不仅仅是典型性能
  - 面对复杂平台的分析
- 程序性能 ≠ CPU性能：
  - **Pipeline, cache**是程序中的窗口
  - 我们必须了解整个程序的执行时间，必须查看路径

执行时间是程序的一个全局特性



确定程序性能的困难度

- 程序执行时间随输入数值的改变而改变，因为:
  - 不同的数值可导致程序选择不同的路径：
    - 循环执行次数的改变
    - 不同的分支执行不同复杂度的语句块
  - 高速缓存的行为部分依赖于输入
  - 指令级执行的时间也有可能不同：
    - 浮点预算对数值最敏感
    - 引入数据相关（**data-dependent**），引起流水线互锁
    - 改变指令获取的次数

如何测量程序性能？

- 利用**CPU**仿真器模拟程序的执行
  - 有些仿真器可以度量程序执行时间，可以使**CPU**的内部状态可见
  - 速度慢，并非**100%**精确，且仿真**I/O**敏感代码很难
- 利用**CPU**总线上的定时器**timer**
  - 需修改程序控制，且被测程序长度受限于定时器精度
- 利用**CPU**外部总线上连接的逻辑分析仪
  - 依赖于能在总线上产生可识别事件的代码来区分代码的启动和停止
  - 被测量代码长度受限于逻辑分析仪缓冲区大小

程序性能的指标

- 平均执行时间(average-case execution time)
  - 执行典型输入数据的典型执行时间
- 最坏执行时间(worst-case execution time)
  - 最长执行时间对有时限要求的系统很重要
- 最佳执行时间(best-case execution time)
  - 任务级的交互可能导致最佳程序行为转变为最坏的系统行为
  - 对于多速率实时系统非常重要

程序性能的要素

- 程序执行时间可以看做:
  - 执行时间 = 程序路径 + 指令耗时
- 独立解决上述两个问题可以帮助简化分析
  - 高级语言规格说明可以追踪程序的执行路径
  - 但是，很难直接准确估计程序的总体执行时间
- 精确的程序性能分析要依赖:
  - 汇编或二进制目标代码
  - 执行平台

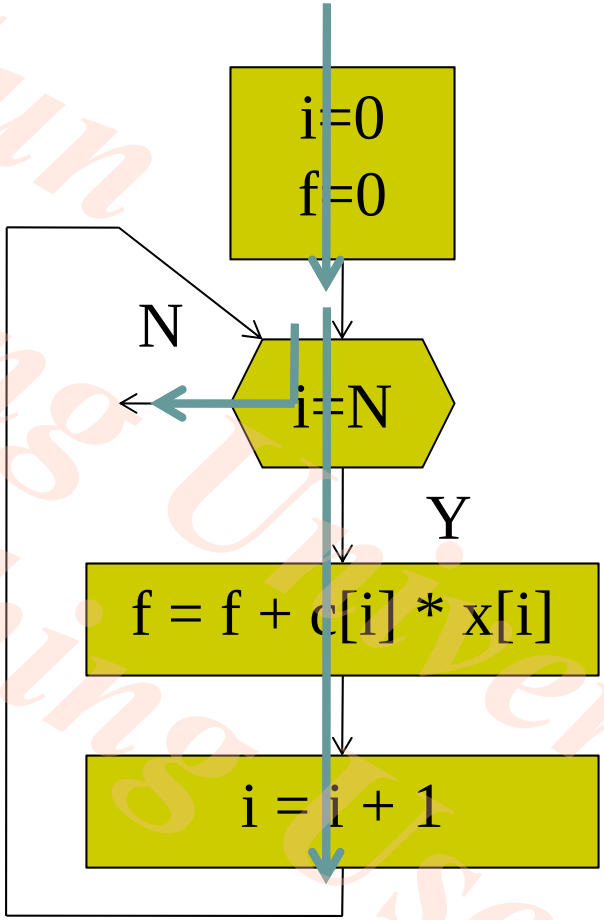
If语句中的数据相关路径

```
if (a || b) { /* T1 */
    if ( c ) /* T2 */
        x = r*s+t; /* A1 */
    else y=r+s; /* A2 */
    z = r+s+u; /* A3 */
    }
else {
    if ( c ) /* T3 */
        y = r-t; /* A4 */
}
```

a	b	c	path
0	0	0	T1=F, T3=F: no assignments
0	0	1	T1=F, T3=T: A4
0	1	0	T1=T, T2=F: A2, A3
0	1	1	T1=T, T2=T: A1, A3
1	0	0	T1=T, T2=F: A2, A3
1	0	1	T1=T, T2=T: A1, A3
1	1	0	T1=T, T2=F: A2, A3
1	1	1	T1=T, T2=T: A1, A3

循环中的路径

```
for (i=0, f=0; i<N; i++)
    f = f + c[i] * x[i];
```



循环初始化代码

循环测试

循环体

循环变量更新

指令耗时

- 并不是所有的指令都花费相同的时间
  - 多重装入-存储指令：**LDMIA**
  - 浮点指令
- 指令的执行时间不是独立的
  - 流水线互锁
  - 高速缓存的影响
- 指令的执行时间可能依赖于操作数的值
  - 浮点指令
  - 某些多周期整数操作指令

测量驱动的性能分析

- 确定程序执行时间最直接的方法就是测量，但听起来容易做起来难！
  - 必须可以访问CPU
  - 必须使内部状态可见：需要仿真器
  - 必须知道可以提供最坏/最佳性能的输入数据集！
- 尽管如此，测量仍然是确定嵌入式软件执行时间最常用的方法。

提供输入数据

- 问题一：确定期望的输入集
  - 使用基准数据集
  - 从运行系统中捕获数据
  - 用算法分析最差执行时间的输入

- 问题二：需要软件脚手架(software scaffolding)来将数据输入程序并且得到输出数据。
  - 对于大型系统，将软件的一部分提取出来进行独立测量很难，需要添加新的测试模块。

轨迹驱动的测量

- 程序执行路径的记录成为轨迹（**Trace**）
  - 通过设备或软件来监测程序
  - 保存关于路径的信息
- 需要修改程序
- 轨迹文件通常都很大
- 常用于高速缓存分析

物理测量

- 利用内部定时器，但是无法跟踪轨迹
- 在线调试器（**In-circuit emulator, ICE**）可以允许跟踪轨迹
- 逻辑分析仪：内存有限
  - 通过地址总线分析事件
  - 需要修改代码使事件可以观察

CPU仿真

- 我们需要周期精确的仿真器（**cycle-accurate simulator**），可以确定执行所需的确切时钟周期数目。
  - 仿真器具有完备的处理器模型，能够对**CPU**内部进行足够详细的仿真
  - 仿真器开发者必须知道**CPU**内部是如何工作的

- **Simplescalar, http://www.simplescalar.com**

SimpleScalar的FIR滤波器仿真

```
int x[N] = {8, 17, … };
int c[N] = {1, 2, … };
main() {
    int i, k, f;
    for (k=0; k<COUNT; k++)
        for (i=0; i<N; i++)
            f += c[i]*x[i];
}
```

COUNT	total sim cycles	sim cycles per filter execution
100	25854	259
1,000	155759	156
1,0000	1451840	145

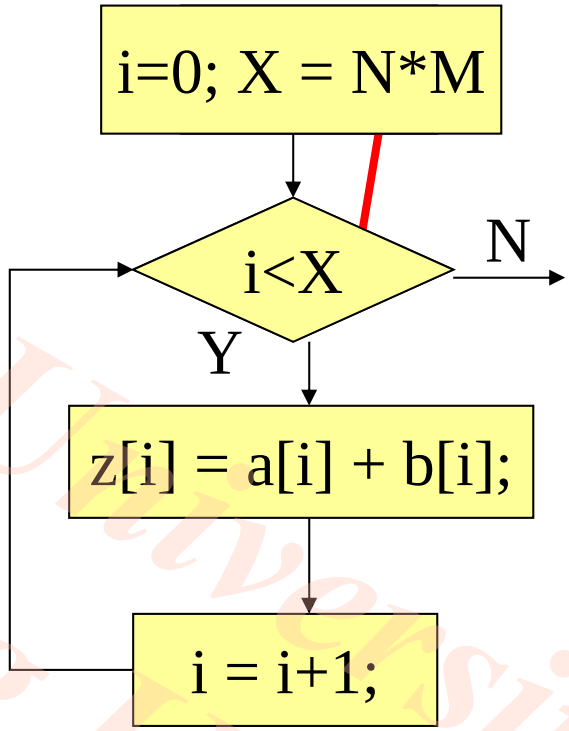
7 软件性能优化

- 循环是优化的重要对象，带有循环结构的程序需要花费较长的执行时间。

- 基本的循环优化技术:
  - 代码移出（**code motion**）
  - 归纳变量消除（**induction-variable elimination**）
  - 强度削减（**strength reduction (x*2 -> x<<1)**）

代码移出

```
for (i=0; i<N*M; i++)
    z[i] = a[i] + b[i];
```



归纳变量消除

- 归纳变量源于循环变量的值: **loop index.**

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        z[i,j] = b[i,j];
```

```
zbinduct = i*M + j;
*(zptr + zbinduct) = *(bptr + zbinduct);
```

- 改变每次都重新计算**i*M+j**，利用数组的步进顺序遍历.

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        *(zptr + zbinduct) = *(bptr + zbinduct);
        zbinduct ++;
```

强度削减

- y = x*2;
- 用左移替代乘2

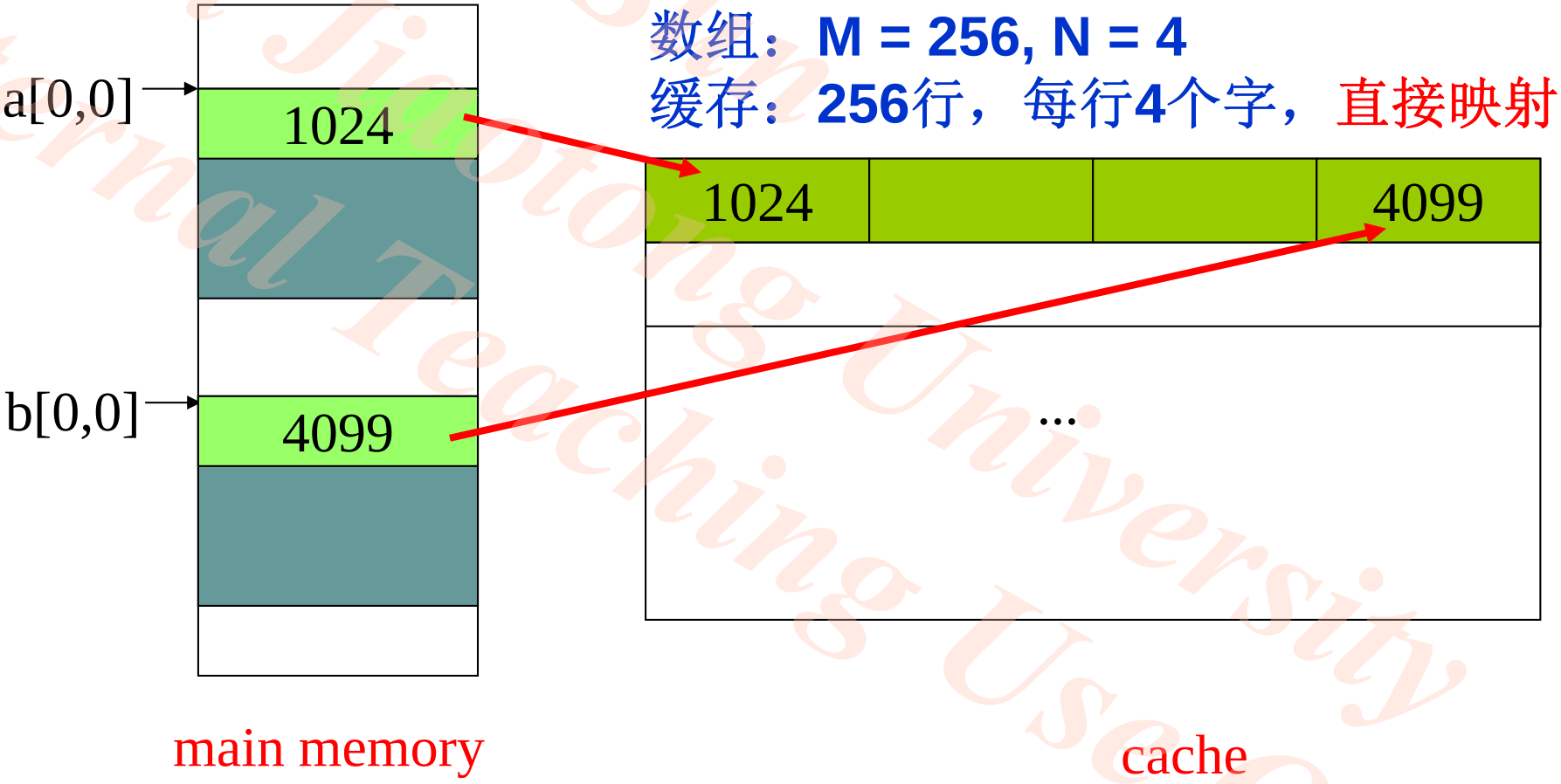
高速缓存优化

- 循环嵌套：一系列循环，一个包着一个。

- 由于循环通常使用大量数据，因此受高速缓存影响大。

```
for (j=0; j<M; j++)
        for (i=0; i<N; i++)
                b[j][i] = a[j][i] * c;
```

高速缓存中的数组冲突



高速缓存中的数组冲突

- **Array elements conflict because they are in the same line, even if not mapped to same location.**
- **Solutions:**
  - **move one array;**
  - **pad array.**

性能优化策略

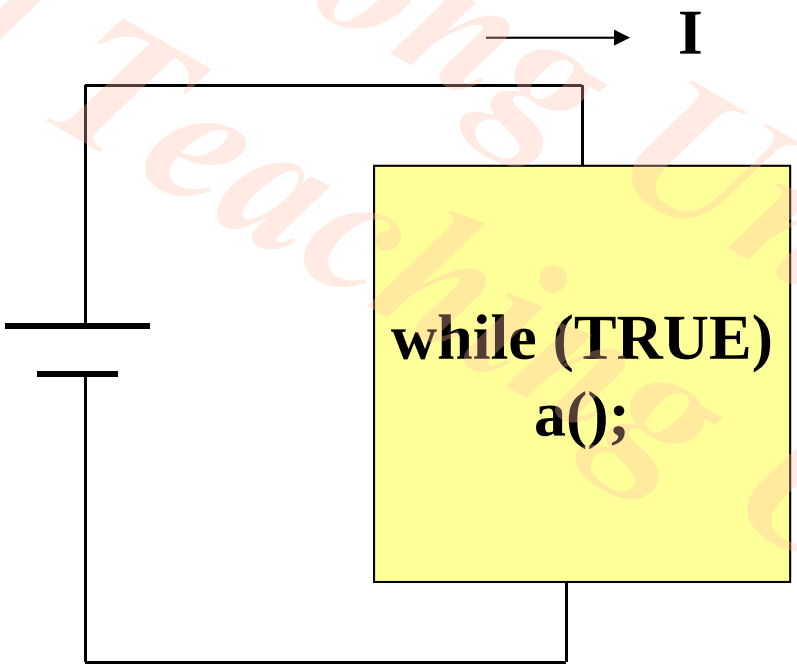
- 尽量有效的使用寄存器
- 尽可能在存储系统中利用页访问模式的优势
- 分析高速缓存行为查找主要的缓存冲突:
  - 指令冲突：小段代码，重写使其更小，尽量装进缓存中，可能必须要用汇编；大段代码，移出冲突指令或填充**NOP**。
  - 标量数据冲突：将数据搬移到不同的存储单元
  - 数组数据冲突：搬移数组或改变数组的访问模式

8 能量和功率的分析与优化

- 能量Energy: 做功的能力。
  - 对于电池供电的系统最为重要
- 功率Power: 单位时间能量。
  - 即使对于电网供电系统也很重要 --- 高速芯片发热量大，控制功耗是提高可靠性降低系统成本的重要因素。

测量能耗

- 在循环中反复执行测试代码，测量电流。
- 单独测量没有循环体的空循环，测量电流。

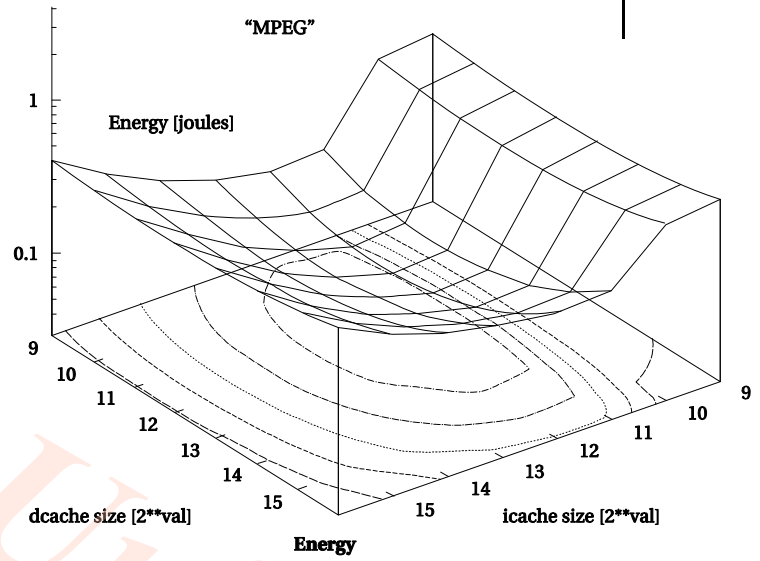


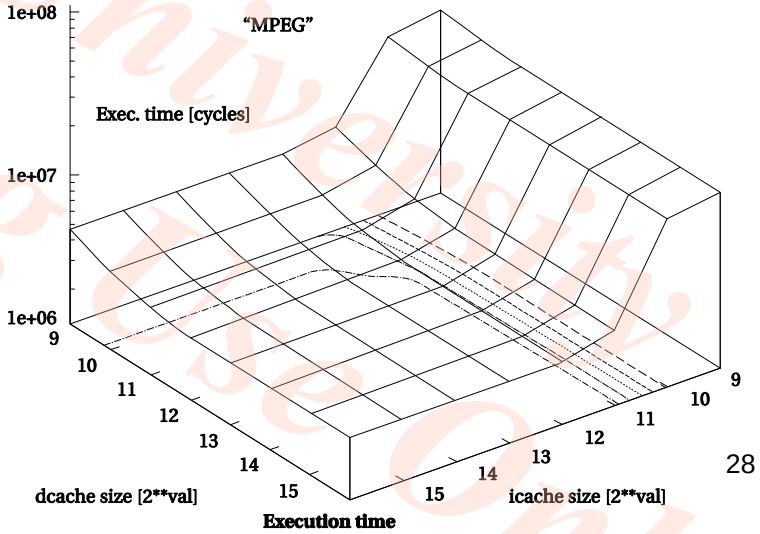
能耗来源比较

- 每次操作的相对能耗比较 **(Catthoor et al):**
  - **memory transfer: 33**
  - **external I/O: 10**
  - **SRAM write: 9**
  - **SRAM read: 4.4**
  - **multiply: 3.6**
  - **add: 1**

高速缓存是能耗的重要因素

- 随着高速缓存大小变化的能耗最佳点:
  - **cache too small**: **program thrashes, burning energy on external memory accesses;**
  - **cache too large**: **cache itself burns too much power.**





能耗优化（1）

- 最好的总体建议：高性能 = 低能耗
  - **high performance = low energy.**
- 性能优化的策略对降低能耗经常也有效
  - 尽量有效的使用寄存器
  - 分析、发现并避免主要的高速缓存冲突
  - 尽可能在存储系统中利用页模式访问

能耗优化（2）

- 适当的循环展开可以消除一些循环控制开销
  - 但是展开过多，会由于高速缓存命中率降低而使功耗增加
- 软件流水减少了流水线的停顿，因此减少每条指令的平均功耗.
- 尽可能的消除递归调用，节省函数调用的开销。尾递归（？）通常可以被消除；某些编译器会自动消除它。
- 内嵌调用：减少调用耗费，但可能导致缓存冲突

程序大小的优化

- 目标:
  - 减少存储器的硬件耗费
  - 减少存储部分的功耗

- 两个机会:
  - 数据**Data**
  - 指令**Instruction**

Data size最小化

- 识别并去除低效率程序保存的数据副本
- 小心确定缓冲区的大小，不要定义程序永远用不到的大数组
- 数据压缩：几个标志位存在在一个字中
- 重用**constants, variables, data buffers**
  - **Requires careful verification of correctness.**
- 运行时用代码创建数据而不是存储数据
  - 创建数据的代码占空间，但是当使用复杂数据结构时，用代码创建数据更省空间。

减小code size

- 代码大小优化需要同时考虑高级语言程序变换和指令选择。
  - 将函数封装在子例程中：最小函数体要求？传递参数
  - 用汇编编写可变长度指令的代码
  - 合理选择指令：乘加代替普通算术运算
  - 合理使用子例程，很重要！！！
  - 密集指令集（**dense instruction set**）：**ARM Thumb**指令集和**MIPS**的**MIPS-16**指令集。
    - 密集指令程序大致是标准指令程序的**70%-80%**。

10 程序验证与测试

- 这里我们主要关注的是功能验证
- 两种主要测试策略分别是：
  - 黑盒（**Black box**）测试：测试方法的产生无需了解程序的内容结构。
  - 白盒（**Clear box or white box**）：测试方法的产生要以程序结构为基础。
- 两者以不同的方式测试程序而互为补充。

白盒测试

- 控制/数据流图是开发程序白盒测试的重要工具。
- **Examine the source code to determine whether it works:**
  - **Can you actually exercise a path?**
  - **Do you get the value you expect along a path?**
- 测试过程包括:
  - 能控性**Controllability**: 给程序提供输入
  - 执行程序
  - 能观性**Observability**: 检查输出

控制和观察程序

```
firout = 0.0;
for (j=curr, k=0; j<N; j++, k++)
    firout += buff[j] * c[k];
for (j=0; j<curr; j++, k++)
    firout += buff[j] * c[k];
if (firout > 100.0) firout = 100.0;
if (firout < -100.0) firout = -100.0;
```

测试限制代码是否有效？

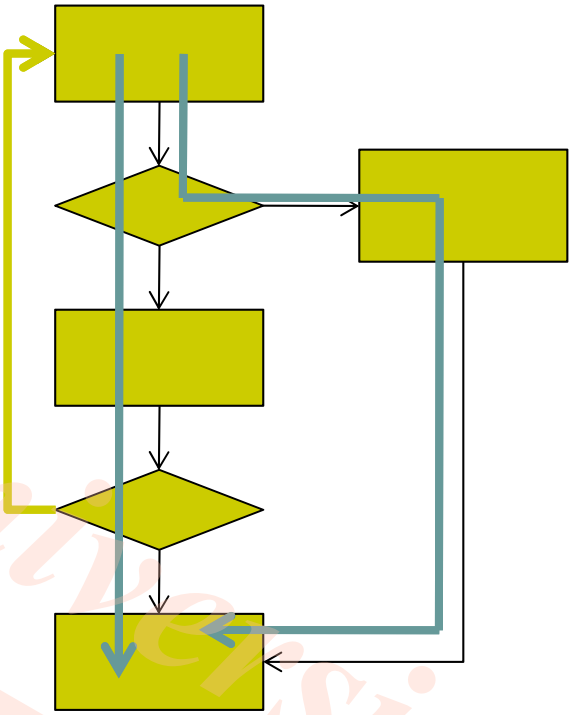
- 能控性:
  - 为firout建立两组越界值
  - 必须用N个期望值填充循环缓冲区
  - 其他代码保证我们能访问循环缓冲区
- 能观性:
  - 在限制代码执行前需要查看firout的值

执行路径与测试

- 与性能分析一样，路径在功能测试中也很重要。
- 我们是否有可能执行完每一条路径？
  - **while**循环
  - 流数据
  - 通常，整个程序中路径数是指数级的。
- 如何选择测试路径的子集？

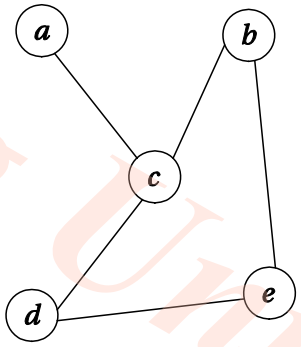
选择测试路径

- 合理的选择:
  - 每条语句至少执行一次
  - 每条分支至少执行一次
- 对于结构化编程语言是等效的。
- 但是，对非结构化语言（汇编）和**goto**语句不同.



基本路径

- Approximate CDFG with undirected graph.
- Undirected graphs have basis paths:
  - All paths are linear combinations of basis paths.



Graph

	a	b	c	d	e
a	0	0	1	0	0
b	0	0	1	0	1
c	1	1	0	1	0
d	0	0	1	0	1
e	0	1	0	1	0

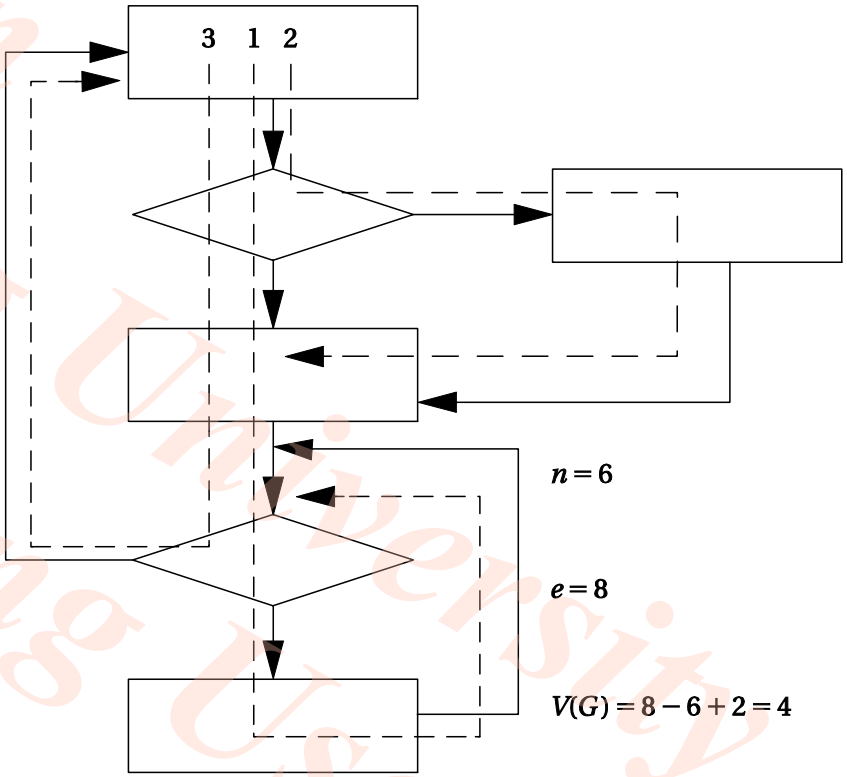
Incidence matrix

a	1	0	0	0	0
b	0	1	0	0	0
c	0	0	1	0	0
d	0	0	0	1	0
e	0	0	0	0	1

Basis set

环路复杂度

- Cyclomatic complexity is a bound on the size of basis sets:
  - e = # edges
  - n = # nodes
  - p = number of graph components
  - M = e – n + 2p.



分支测试的例子1

- Correct:
  - if (a || (b >= c)) { printf(“OK\n”); }
- Incorrect:
  - if (a && (b >= c)) { printf(“OK\n”); }
- Test:
  - a = F
  - (b >=c) = T
- Example:
  - Correct: [0 || (3 >= 2)] = T
  - Incorrect: [0 && (3 >= 2)] = F

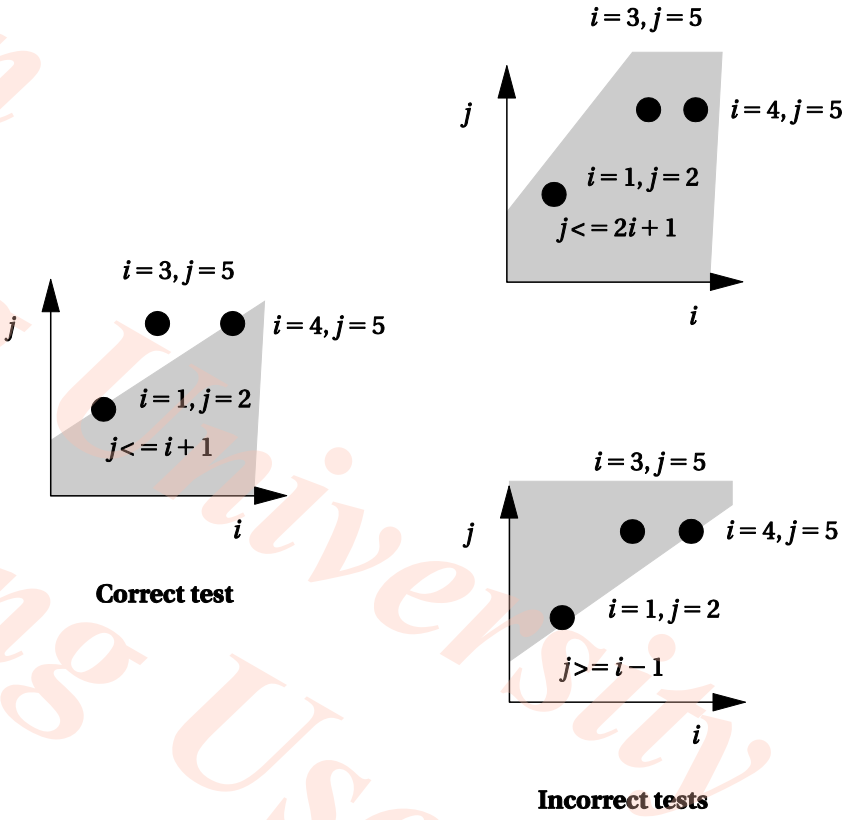
分支测试例子2

- Correct:
  - if ((x == good_pointer) && x->field1 == 3)) { printf("got the value\n"); }
- Incorrect:
  - if ((x = good_pointer) && x->field1 == 3)) { printf("got the value\n"); }

- Incorrect code changes pointer.
  - Assignment returns new LHS in C.
- Test that catches error:
  - (x != good_pointer) && x->field1 = 3)

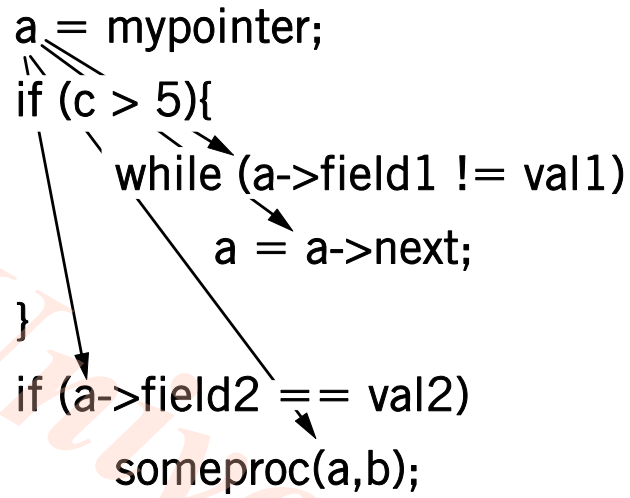
域测试

- 主要用于测试线性不等式
- 测试不等式的每一边 + 边界



定义-使用对

- Variable def-use:
  - Def when value is assigned (defined).
  - Use when used on right-hand side.
- Exercise each def-use pair.
  - Requires testing correct path.



循环测试

- **Loops need specialized tests to be tested efficiently.**
- **Heuristic testing strategy:**
  - **Skip loop entirely.**
  - **One loop iteration.**
  - **Two loop iterations.**
  - **# iterations much below max.**
  - **n-1, n, n+1 iterations where n is max.**

黑盒测试

- 在对代码不了解的情况下进行的测试就是黑盒测试。

- 如单独使用，黑盒测试发现程序全部错误的概率很低。但可作为对白盒测试的补充。
  - 更容易发现那些从代码结构抽象出的测试方法不易发现的程序错误。

黑盒测试向量

- 随机测试**Random tests.**
  - 随机值按照给定分布创建。期望值可以独立于系统计算出来，然后使用测试输入。
- 回归测试**Regression tests.**
  - 使用程序早期版本的测试来测试新版本程序。

多少次测试才算充分?

- 彻底详尽的测试是不切实际的。
- 对测试质量评估的一项重要指标：有多少bugs躲过测试到达应用现场。
- 错误注入分析测试覆盖率：
  - 向原有代码中加入已知程序错误，记住位置
  - 运行测试
  - 计算通过测试发现的注入错误的 占全部注入错误的%