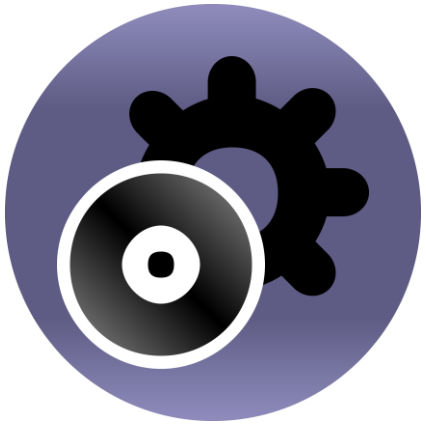
程序插桩及优化机会识别

黄 波
bhuang@dase.ecnu.edu.cn

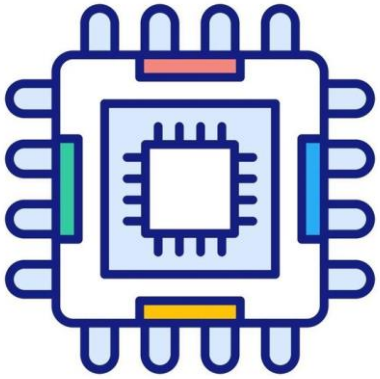
SOLE 系统优化实验室
华东师范大学

负载动态运行特征

软件特征数据



微体系架构特征数据

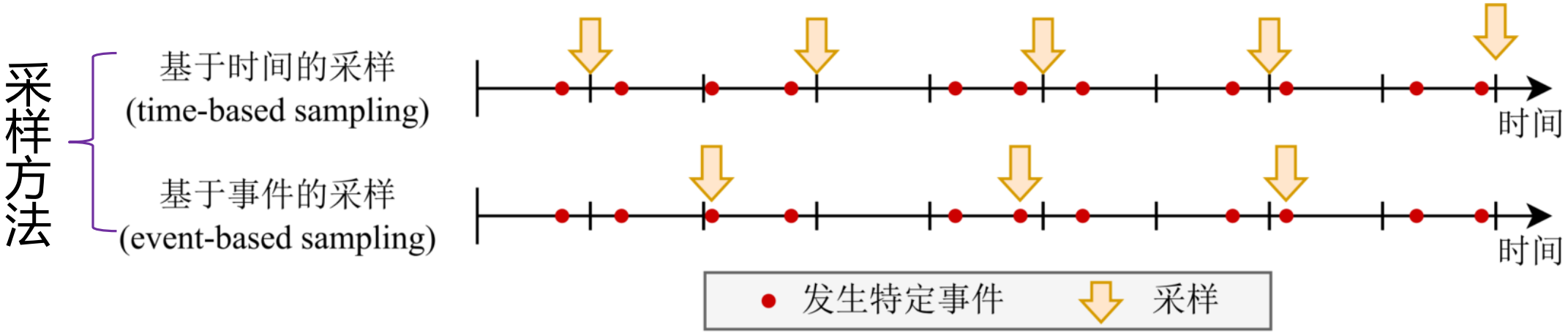


测量方法
外部测量
内部测量
仿真测量

收集策略
计数型
采样型
追踪型

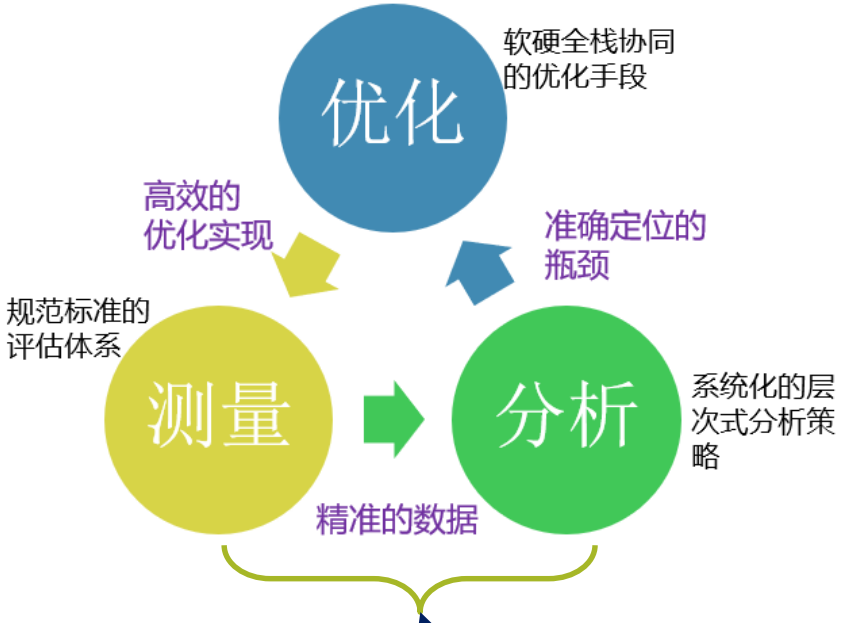
常见实用测量方法/收集策略的比较

测量方法	对负载的影响	准确性	开销
内部测量（传统插桩）	需要修改程序的源代码或二进制文件	比较准确	开销比较大，不大适合产品化部署环境
外部测量（*perf*）	无需修改程序	依赖于PMU是否有相应的支持及收集策略；一般比传统插桩的准确性低。	比传统插桩开销小



本次课的关注点

Scale up
全栈思维

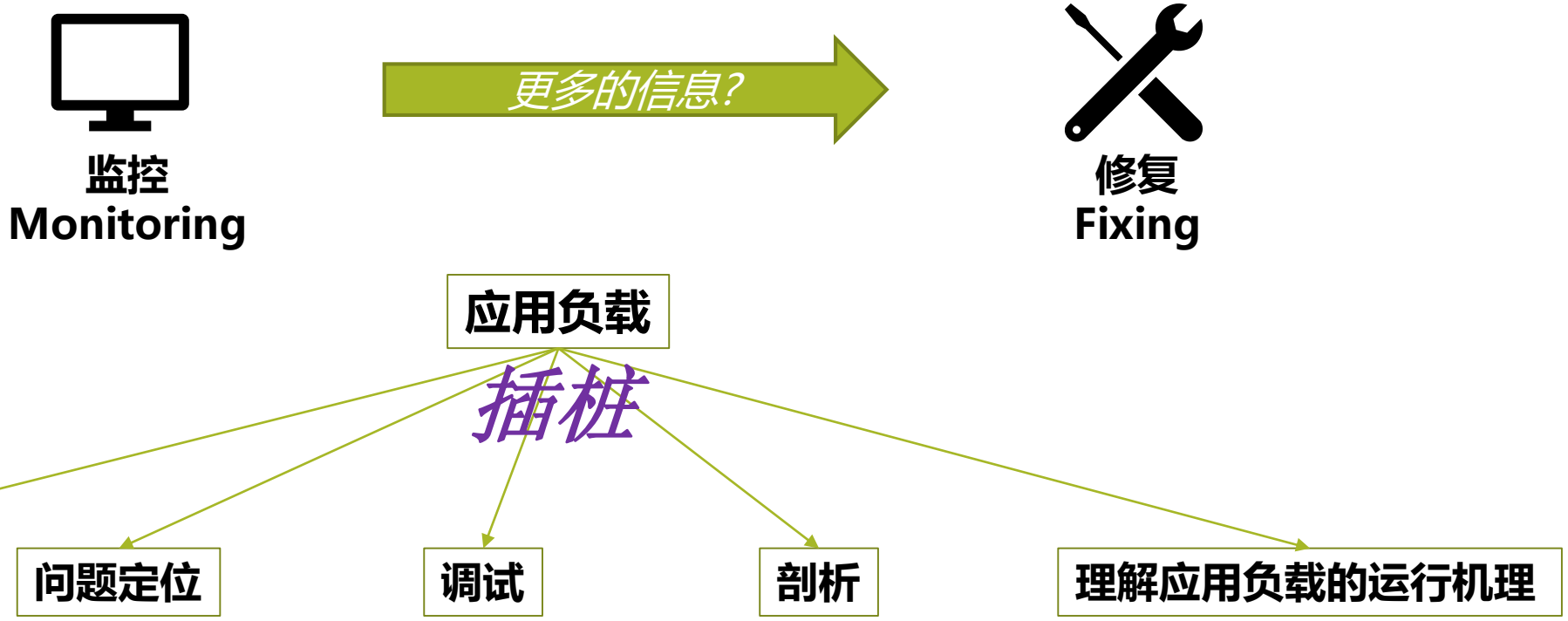


- 理解应用负载的动态特征及行为是程序优化的前提和基础
- 插桩（Instrumentation）是理解应用负载动态特征的一种常见手段
- 插桩工具的实现通常需要用到程序分析及编译技术
- 插桩信息可以用来帮忙寻找优化机会

本次课程内容

- 什么是程序插桩?
- 二进制翻译如何助力程序插桩?
- 如何利用插桩信息识别编译优化机会?

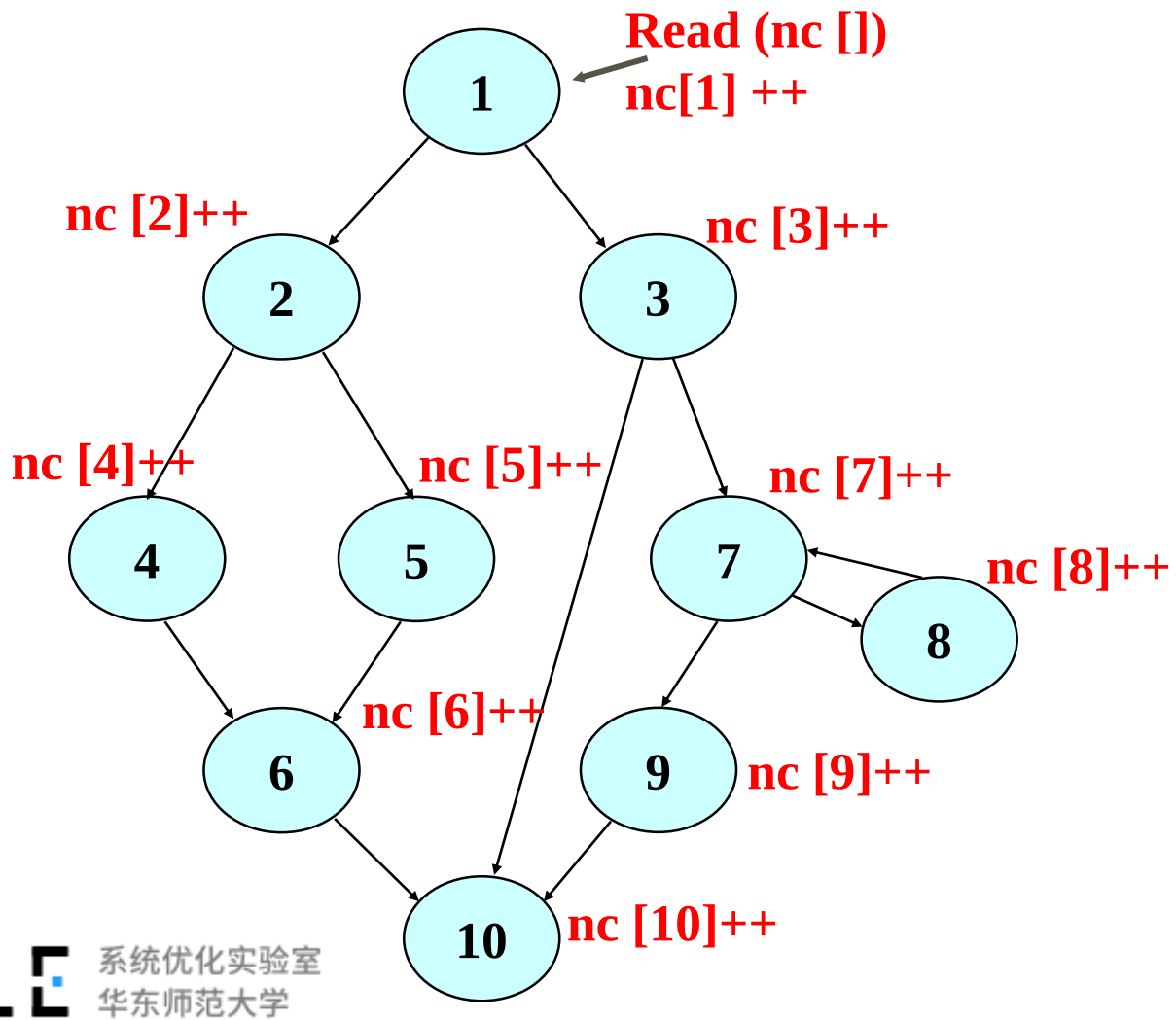
为什么需要进行程序插桩?



什么是插桩?

- 从根本上来说，插桩是从一个应用负载中获得一些特别关注的测量结果的过程；
- 在实践中，插桩也可以是非常简单的测量操作，譬如记录每个函数的执行时间以方便定位性能瓶颈的发生点；
- 严格来说，程序插桩是指在被插桩程序中插入一条或者多条语句/指令以获取程序运行时的某些信息或者特征，比如：
  - 完成诸如典型输入情况下程序运行时的行为特征收集、代码执行时间测量、日志记录、执行路径获取等
- 插入的语句不能影响被插桩程序的行为，也应该尽量减少对被插桩程序的性能影响
- 根据被插桩程序的文件格式，程序插桩包括源代码级别的程序插桩和二进制代码级别的程序插桩

程序中语句覆盖的插桩示例

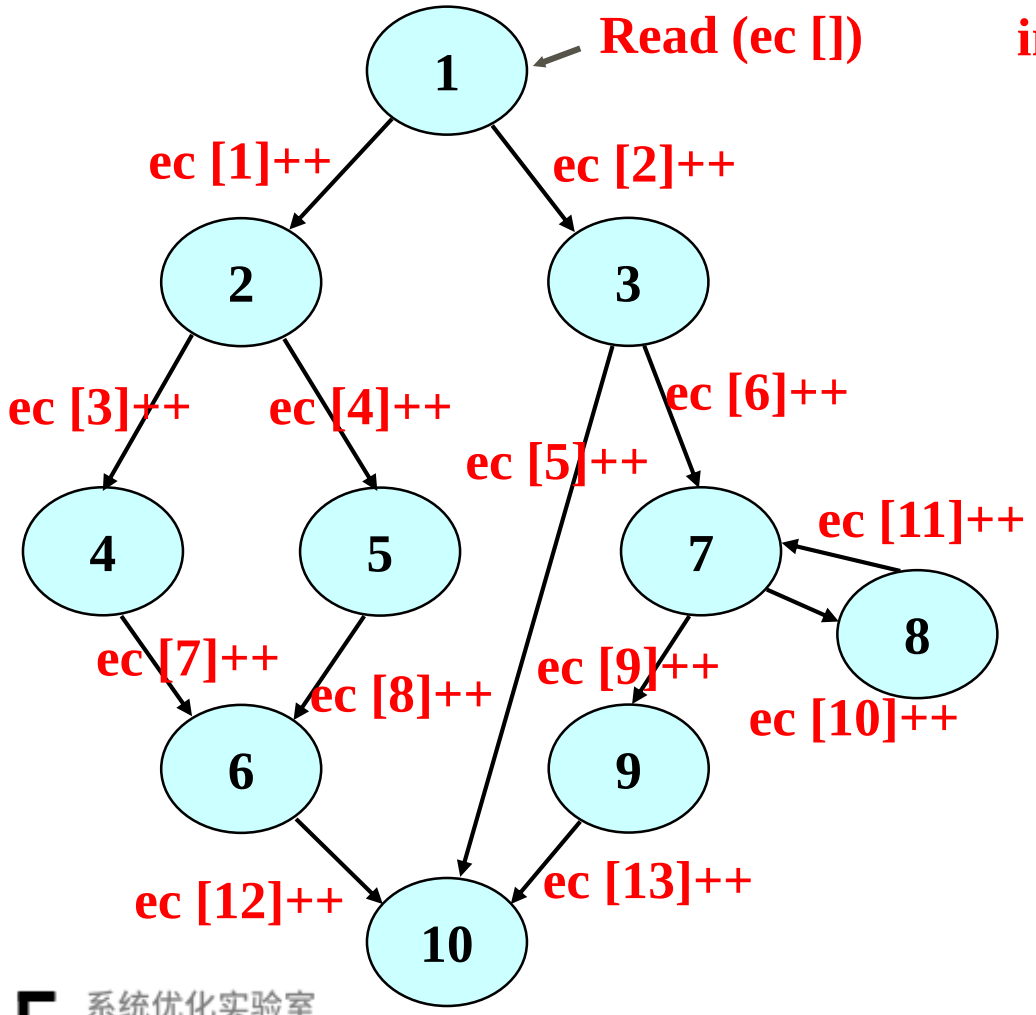


int nc[] = {0,0,0,0,0,0,0,0,0,0,0}

每一个节点代表一个基本块

如果运行完一系列的测试后，某个基本块（假设为node）的执行次数为零（即*nc[node]*的值为零），则可以得出此基本块没有被覆盖的结论。如果某些基本块的执行次数很大，说明这些基本块是潜在的优化热点。

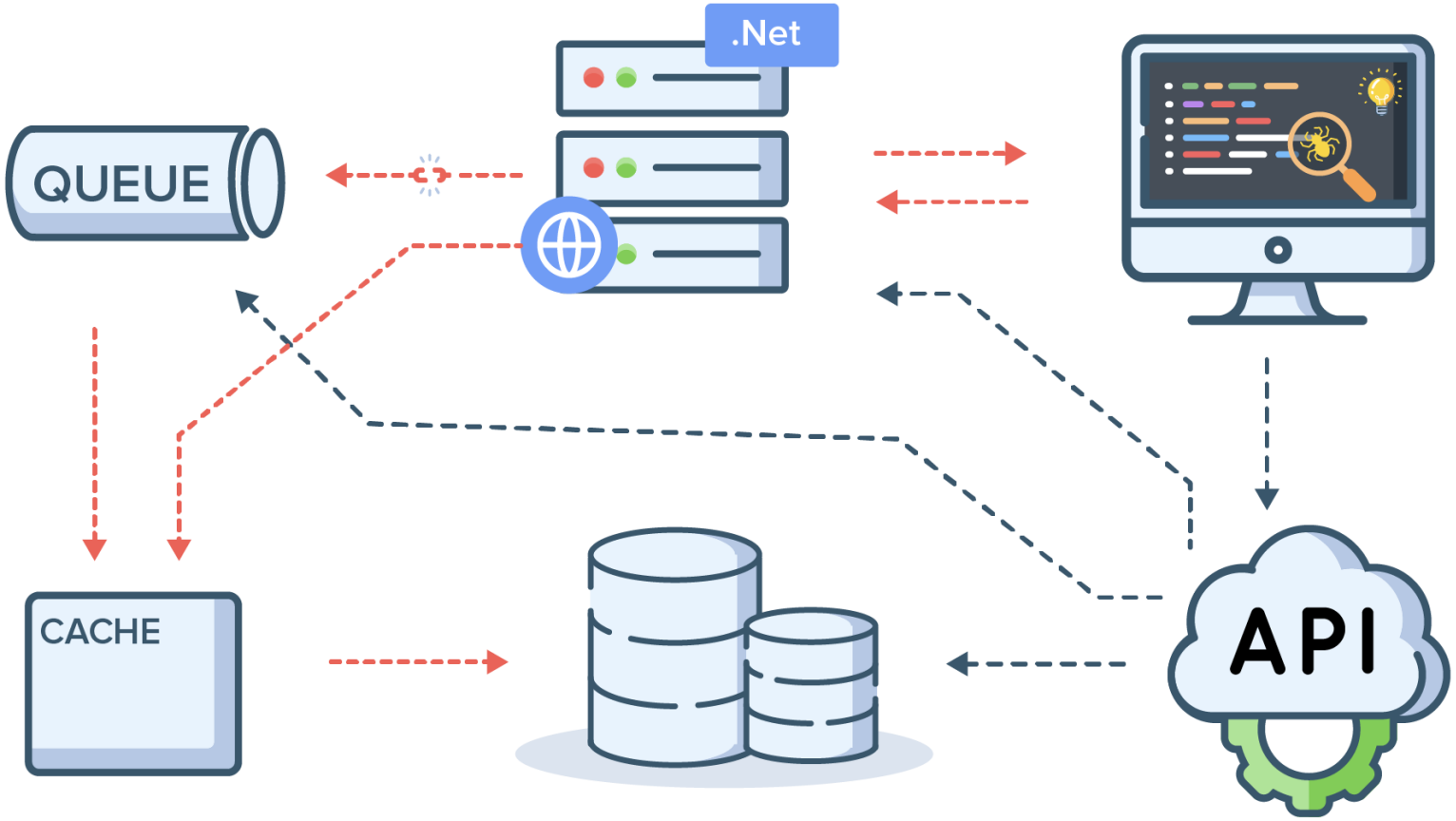
程序中边覆盖的插桩示例



int ec[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0}

- 如果某条边e对应的执行次数为零（即*ec[e]*的值为零），则说明e没有被覆盖；
- 如果某些边的执行次数很大，说明这些边对应的控制流跳转频繁发生，这些信息对形成程序执行时的热点路径（hot trace）非常重要；
- 如果ec变成一个布尔数组，则ec中每个数组元素记录的便是对应的边是否被覆盖的信息。

分布式执行路径跟踪（Distributed Tracing）



程序插桩的手段

- 人工插桩（Manual）
- 源代码级别的自动插桩（Automatic Source Level）
- 中间语言级别的插桩（Intermediate Language）
- 编译器助力插桩（Compiler Assisted）
- 静态二进制重写（Static Binary Rewriting）

静态插桩

- 运行时插桩（Runtime Instrumentation）
- 运行时注入（Runtime Injection）

动态插桩

https://en.wikipedia.org/wiki/Profiling_(computer_programming)

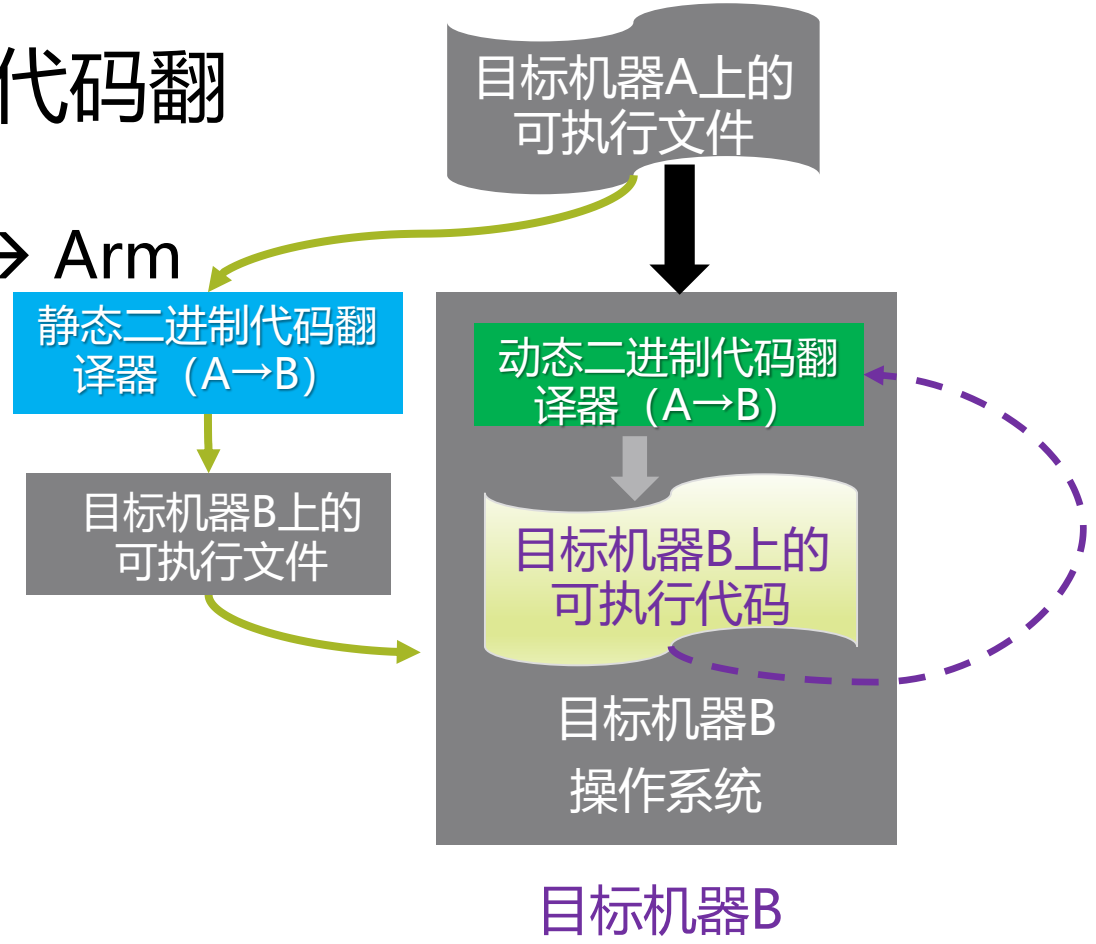
静态插桩 vs. 动态插桩

静态插桩	动态插桩
● 离线插桩，无需过多考虑被插桩程序的解析时间及插桩代码插入的时间； ● 可能产生更加有效的插桩后代码； ● 无法对运行时的事件进行立即响应；	● 在程序运行的特定时间内进行插桩，时间和开销敏感； ● 在程序运行过程中插入/移除插桩代码； ● 对运行时的事件可以第一时间及时响应；

什么是二进制翻译(Binary Translation)?

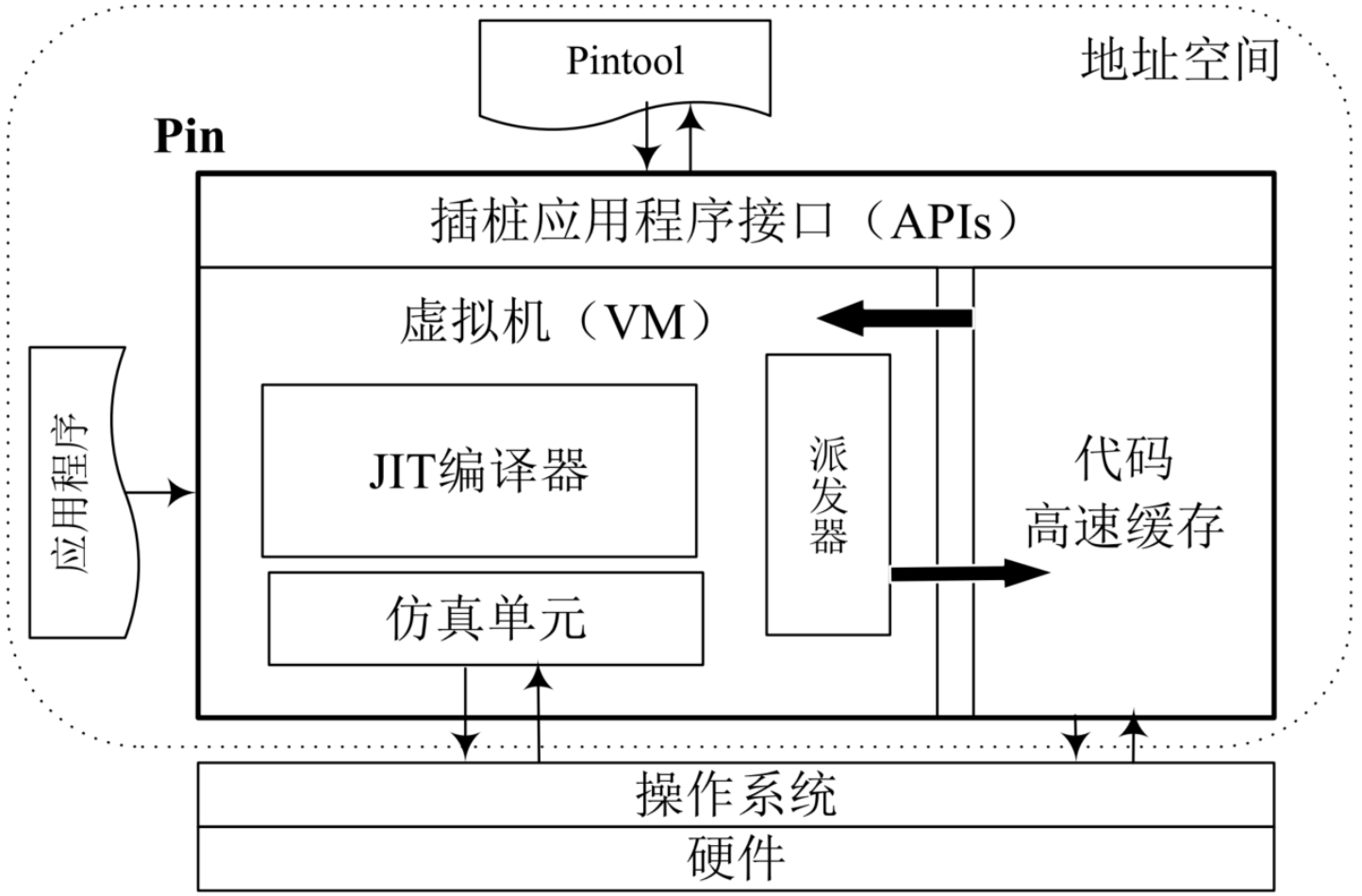
- 二进制翻译的功能是把一种二进制代码翻译成另外一种二进制代码:
  - 可以是跨指令集架构的翻译，如X86-64 → Arm
  - 也可以是同指令集架构之间的翻译

- 同指令集架构之间的二进制翻译是插桩工具开发时常用的技术:
  - 静态二进制重写
  - 运行时插桩



Pin的软件系统架构

https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

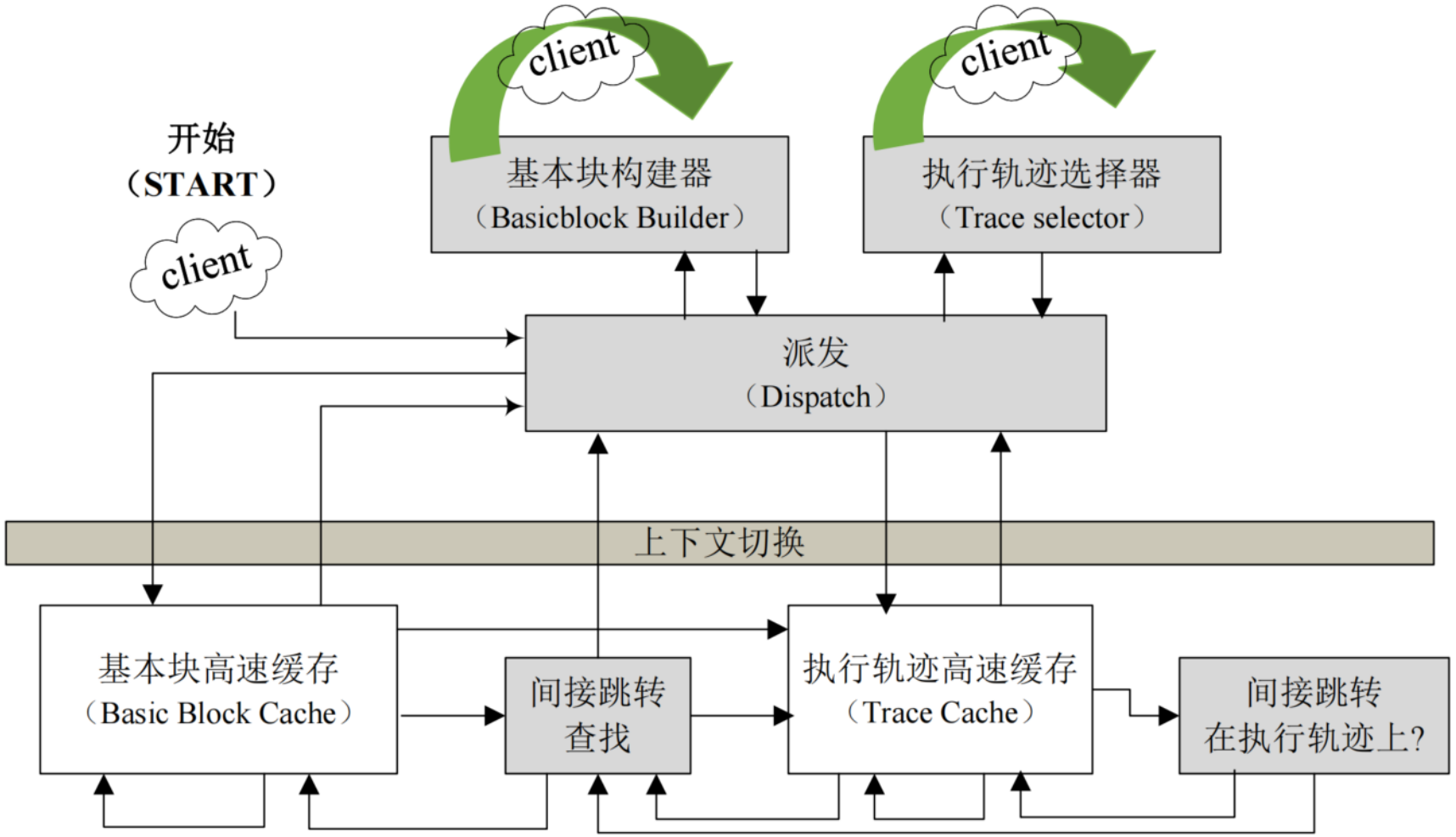


免费
但不开源

只支持在Intel平台上的插桩及动态优化！

Dynamo-RIO系统架构

https://dynamorio.org/



DynamoRIO是开源的，支持多平台

编译优化机会识别：源码分析

```
do i = 1, n
  a[i] = a[i] + c * i
end do
```

(a)

```
T = c
do i = 1, n
  a[i] = a[i] + T
  T = T + c
end do
```

(b)

强度削弱

```
do i = 2, n
  a[i] = a[i] + c
  if (x < 7) then
    b[i] = a[i] * c[i]
  else
    b[i] = a[i-1] * b[i-1]
  end if
end do
```

(c)

```
if (x < 7) then
  do all i = 2, n
    a[i] = a[i] + c
    b[i] = a[i] * c[i]
  end do
else
  do i = 2, n
    a[i] = a[i] + c
    b[i] = a[i-1] * c[i-1]
  end do
end if
```

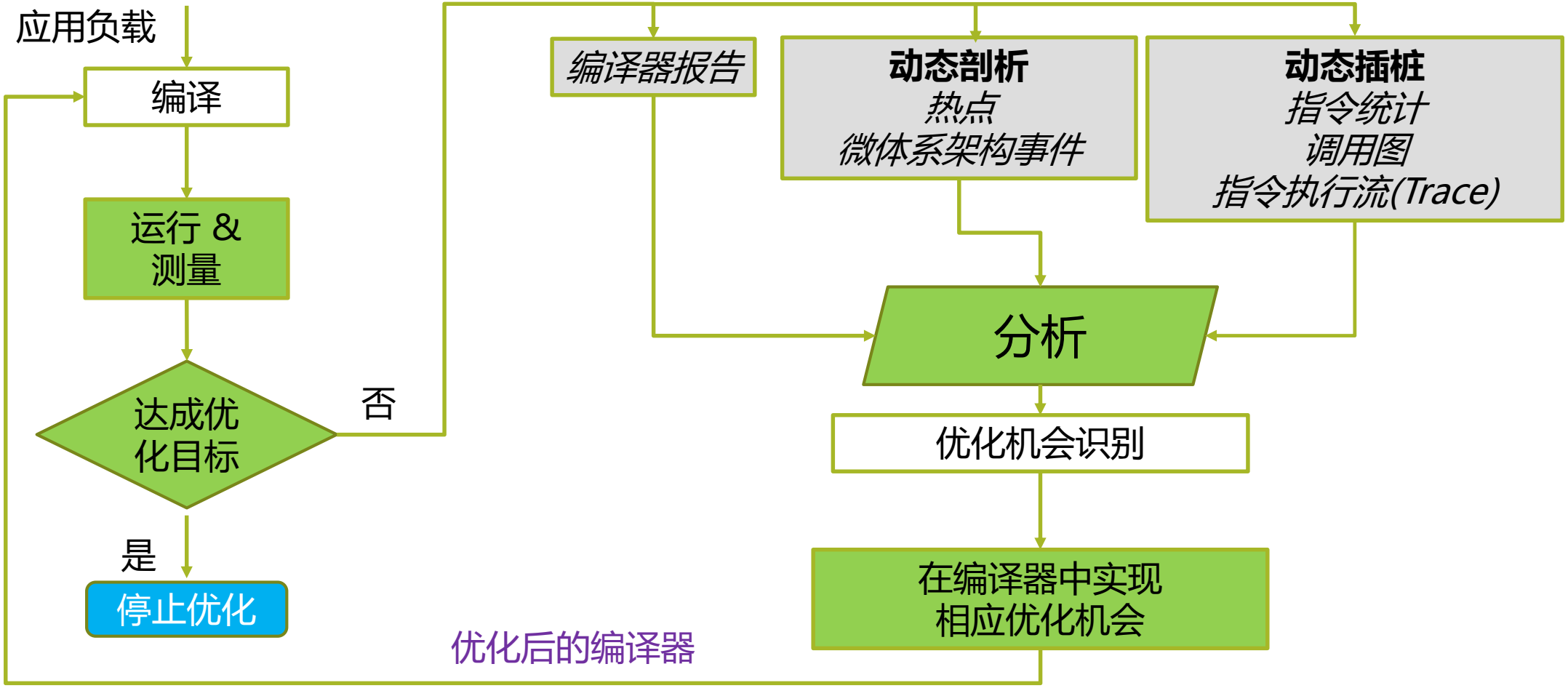
(d)

Unswitching

早期的优化机会识别通常针对循环来展开

Q: (a)与(b)哪个执行时间短?

编译优化机会识别：常用方法



高度依赖于从事编译优化分析工作人员的经验及专业程度

不同编译器的优化能力有差异

- 开源编译器

- 商业编译器产品

毕昇编译器

Intel® oneAPI DPC++/C++ Compiler
A Standards-Based, Cross-architecture Compiler

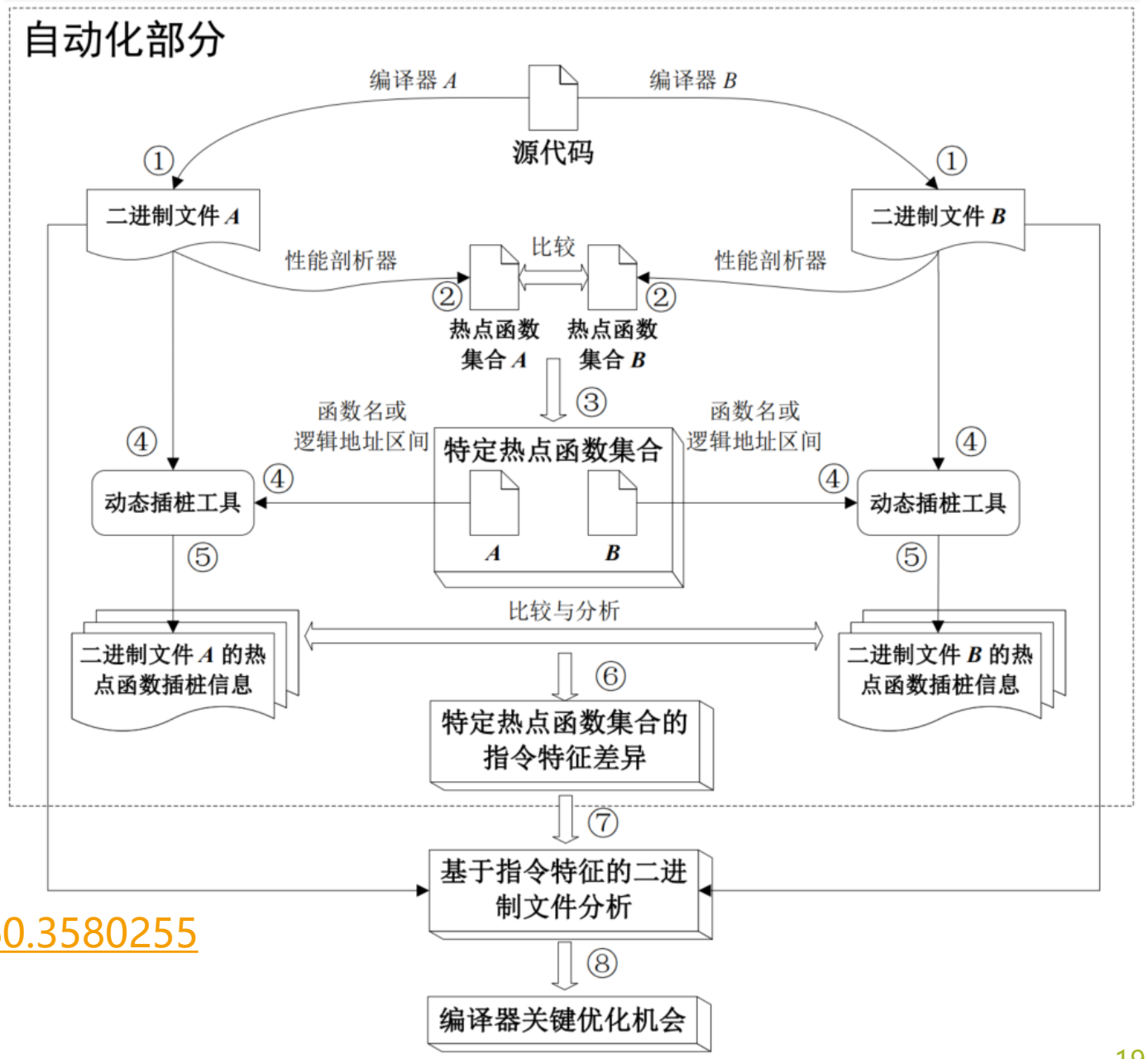
AMD Optimizing C/C++ and Fortran Compilers (AOCC)

Q: 是否可以"学习"别的编译器的优化方法?

一种新方法

一个热点驱动的半自动化编译器竞争分析框架

- ①-⑥是可以自动完成的，⑦和⑧是必须人工介入的
- ①-⑥的自动化大大节省了数据收集及分析的时间，也大大缩小了后期人工分析的程序范围



https://dl.acm.org/doi/10.1145/3578360.3580255

插桩工具对X64和AArch64的指令分类

指令分类			X86-64 平台	AArch64 平台
分支	间接分支	跳转	jmp<mem>, <reg>)	br
		函数调用	call(<mem>, <reg>)	blr
		返回	ret	ret
	条件跳转		jle / jne / je / ja /jna ...	b.<cond> / cbz / cbnz ...
	无条件直接跳转		jmp / call <imm>	b / bl
运算	算术		cmp / sub / mul ...	cmp / sub / mul / add ...
	逻辑		and / or / xor / test ...	and / or ...
	移位		shr / sha / sar / sal	asrv / rorv / lslv / bfm ...
数据传送			mov / movzx ...	movi / movz / fmov ...
栈操作			push / pop	–
内存读写			–	str / stp / ldr / ldp
向量化指令			SSE / AVX / AVX2 / AVX512	Neon / SVE
其它			lea / setb / nop ...	adrp / adr / nop / csel ...

实验配置

	X64	AArch64
处理器	Intel Xeon Gold 5218R	华为鲲鹏 920 - 5250
CPU 数量	80 (逻辑核, 2 插槽, 开启 Hyperthreading)	96 (物理核, 2 插槽)
开源编译器	GCC (版本 10.3.0)	GCC (版本 10.3.0)
专有编译器	Intel ICC 编译器 (版本 2021.6.0)	华为 BiSheng 编译器 (版本 2.1.0)
操作系统	Ubuntu 20.04.4 LTS Linux	Ubuntu 20.04.4 LTS Linux

- 编译SPEC CPU® 2017所用的配置是从SPEC CPU® 2017的官方网站上找到的跟上表实验平台最接近的配置
- *runcpu*命令行的主要选项用的是*tune=base, action=validate* 以及 *size=train*

案例分析负载选择

基准测试程序	平台	开源编译器 (GCC)	专有编译器 (BiSheng / ICC)	相对性能差距
Fortran				
648.exchange2_s	AArch64	38.9 s	25.0 s	55.6%
C++				
620.omnetpp_s	X64	45.8 s	31.2 s	46.8%
631.deepsjeng_s	X64	83.0 s	68.0 s	22.1%
641.leela_s	X64	89.2 s	76.0 s	17.4%
623.xalancbmk_s	X64	35.2 s	35.2 s	0.0%

注：编译器那两栏中的数据是对应行的基准测试程序用对应列的编译器编译出来可执行文件的运行时间。

648.exchange2_s(GCC)和648.exchange2_s(BiSheng)的热点比较

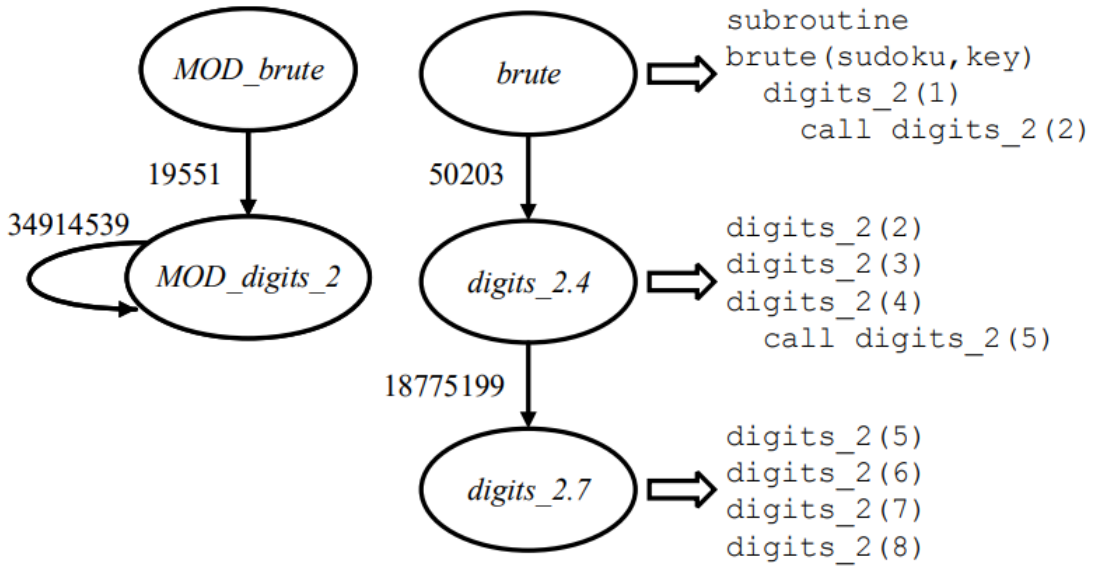
排名	BiSheng			GCC		
	相对时间 (%)	绝对时间 (秒)	函数名	相对时间 (%)	绝对时间 (秒)	函数名
1	**57.32**	14.33	digits_2.7	**82.38**	32.05	MOD_digits_2
2	**19.38**	4.85	digits_2.4	**7.30**	2.84	gfortran_mminloc
3	**18.30**	4.58	logic_new_solver	**4.15**	1.61	specific.4
4	1.46	0.37	free	**2.31**	0.90	logic_MOD_new_solver
5	0.80	0.20	malloc	1.08	0.42	hidden_triplets.0
6	0.43	0.11	covered	0.78	0.30	free
7	0.32	0.08	brute	0.55	0.21	naked_triplets.1
8	0.17	0.04	f90_dealloc	0.48	0.19	hidden_pairs.2
9	0.15	0.04	f90_alloc	0.25	0.10	MOD_brute
10	0.14	0.04	f90_set_intrin	0.25	0.10	malloc

嫌疑热点

编译优化机会识别（1）

指令分类	digit_2.4	digit_2.7	MOD_digit_2
分支	**3 593 142 992**	**11 609 217 450**	**29 321 533 296**
间接分支	**50 203**	**18 775 199**	**34 934 090**
跳转	0	0	0
函数调用	0	0	0
返回	**50 203**	**18 775 199**	**34 934 090**
条件跳转	**3 361 685 141**	**10 733 730 071**	**27 698 651 020**
无条件直接跳转	231 407 648	856 712 180	1 587 948 186
运算	**15 199 221 930**	**41 803 072 569**	**116 664 503 804**
算数	14 686 454 611	39 759 170 962	102 813 533 806
逻辑	498 710 685	1 733 736 947	12 559 811 188
移位	14 056 634	310 164 660	1 291 158 810
数据传送	32 692 266	214 161 627	1 636 029 315
内存读写	**11 911 031 090**	**37 507 748 969**	**81 757 176 763**
向量化	**5 426 852 903**	**10 429 222 578**	**268 662 027**
其他	534 241 804	1 514 087 575	2 581 871 188

(a) *648.exchange2_s(GCC)* **(b) *648.exchange2_s(BiSheng)***



函数内联 & 函数特化

编译优化机会识别（2）

指令分类	digit_2.4	digit_2.7	MOD_digit_2
分支	**3 593 142 992**	**11 609 217 450**	**29 321 533 296**
间接分支	**50 203**	**18 775 199**	**34 934 090**
跳转	0	0	0
函数调用	0	0	0
返回	**50 203**	**18 775 199**	**34 934 090**
条件跳转	**3 361 685 141**	**10 733 730 071**	**27 698 651 020**
无条件直接跳转	231 407 648	856 712 180	1 587 948 186
运算	**15 199 221 930**	**41 803 072 569**	**116 664 503 804**
算数	14 686 454 611	39 759 170 962	102 813 533 806
逻辑	498 710 685	1 733 736 947	12 559 811 188
移位	14 056 634	310 164 660	1 291 158 810
数据传送	32 692 266	214 161 627	1 636 029 315
内存读写	**11 911 031 090**	**37 507 748 969**	**81 757 176 763**
向量化	**5 426 852 903**	**10 429 222 578**	**268 662 027**
其他	534 241 804	1 514 087 575	2 581 871 188

(row=2,5,8)

(a) 函数特化前:

```
select case(mod(row, 3))
  case 1:
    block(row + 2, 4:6, i1) = block(row + 2, 4:6, i1) - 10
  case 2:
    block(row + 1, 1:3, i1) = block(row + 1, 1:3, i1) - 10
end select
```

(b) 函数特化后:

```
block(row + 1, 1:3, i1) = block(row + 1, 1:3, i1) - 10
```

函数特化
→ 常数传播
→ 死码删除

编译优化机会识别（3）

指令分类	digit_2.4	digit_2.7	MOD_digit_2
分支	**3 593 142 992**	**11 609 217 450**	**29 321 533 296**
间接分支	**50 203**	**18 775 199**	**34 934 090**
跳转	0	0	0
函数调用	0	0	0
返回	**50 203**	**18 775 199**	**34 934 090**
条件跳转	**3 361 685 141**	**10 733 730 071**	**27 698 651 020**
无条件直接跳转	231 407 648	856 712 180	1 587 948 186
运算	**15 199 221 930**	**41 803 072 569**	**116 664 503 804**
算数	14 686 454 611	39 759 170 962	102 813 533 806
逻辑	498 710 685	1 733 736 947	12 559 811 188
移位	14 056 634	310 164 660	1 291 158 810
数据传送	32 692 266	214 161 627	1 636 029 315
内存读写	**11 911 031 090**	**37 507 748 969**	**81 757 176 763**
向量化	**5 426 852 903**	**10 429 222 578**	**268 662 027**
其他	534 241 804	1 514 087 575	2 581 871 188

源代码

```
block(row + 1, 4:6, i4) = block(row + 1, 4:6, i4) + 10
```

AArch64 汇编代码

```
add  x8, x9, #0x32c
ldr  q0, [x8]
add  v0.4s, v0.4s, v3.4s
str  q0, [x8]
```

(a) *648.exchange2_s (BiSheng)*

AArch64 汇编代码

```
ldr  w20, [x14]
ldr  w19, [x14, #36]
ldr  w28, [x14, #72]
add  w27, w20, #0xa
add  w1, w19, #0xa
str  w27, [x14]
add  w8, w28, #0xa
str  w1, [x14, #36]
str  w8, [x14, #72]
```

(b) *648.exchange2_s (GCC)*

向量化优化

其它“嫌疑热点”

排名	BiSheng			GCC		
	相对时间 (%)	绝对时间 (秒)	函数名	相对时间 (%)	绝对时间 (秒)	函数名
1	**57.32**	**14.33**	digits_2.7	**82.38**	**32.05**	MOD_digits_2
2	**19.38**	**4.85**	digits_2.4	**7.30**	2.84	gfortran_mminloc
3	**18.30**	4.58	logic_new_solver	**4.15**	1.61	specific.4
4	1.46	0.37	free	**2.31**	0.90	logic_MOD_new_solver
5	0.80	0.20	malloc	1.08	0.42	hidden_triplets.0
6	0.43	0.11	covered	0.78	0.30	free
7	0.32	0.08	brute	0.55	0.21	naked_triplets.1
8	0.17	0.04	f90_dealloc	0.48	0.19	hidden_pairs.2
9	0.15	0.04	f90_alloc	0.25	0.10	MOD_brute
10	0.14	0.04	f90_set_intrin	0.25	0.10	malloc

编译优化机会识别（4）

函数名	编译器	向量化指令数	Ret 指令数
logic_new_solver	BiSheng	1 963 447 369	22 764
logic_MOD_new_solver	GCC	440 253 151	22 088
specific.4	GCC	127 785 213	32 759
gfortran_mminloc	GCC	0	24 517 022

函数内联 + 向量化优化

对GCC（AArch64）的优化建议

通过这个案例分析中以 *648.exchange2_s* 负载为例在 AArch64 平台上对 GCC 编译器和毕昇编译器的对比，我们可以对 GCC 在如下三个方面的优化能力进行增强，在 GCC 上实现这些优化后，*548.exchange2_r* 在鲲鹏平台上可以获得72.96% 的性能提升。

- 函数内联，包括对编译器内置函数（built-in functions）的内联
- 函数特化
- 自动向量化

本次课程总结

- 通过程序插桩来获取应用负载运行过程中的动态行为是对负载进行特征分析以及性能优化的一个重要手段；
- 静态程序插桩和动态程序插桩有各自的优缺点；
- 二进制翻译技术可以大大助力程序插桩工具的实现；
- 精准插桩信息的高效收集、性能瓶颈的分析和准确定位、优化机会的快速识别以及优化机会的高效实现是软件系统优化永恒的研究课题。

补充材料

- An infrastructure for adaptive dynamic optimization (https://ieeexplore.ieee.org/document/1191551)
- Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation (https://dl.acm.org/doi/10.1145/1064978.1065034)
- BOLT: A Practical Binary Optimizer for Data Centers and Beyond (https://ieeexplore.ieee.org/document/8661201)
- OCOLOS: Online COde Layout OptimizationS (https://ieeexplore.ieee.org/document/9923868)
- Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications (https://dl.acm.org/doi/pdf/10.1145/3575693.3575727)

课程预告

将通过两个典型应用场景（e.g. 数据中心、深度学习框架）的优化实践分享让大伙儿对软件系统优化获得更进一步的深刻认识

课程简介 + 矩阵乘法优化案例+方法论概述

性能测量
基准评测
配置优化
性能评价

处理器优化
存储器优化
微体系结构性能分析
异构计算与编程

源程序级别的常见优化方法
编译器概述
目标指令集架构及汇编语言
C程序的汇编代码生成
编译器的优化能力
程序插桩及优化机会识别

数据中心的性能优化
深度学习框架的优化


Q
&
A