指令流分析制导的动态二进制翻译优化技术

胡起

2023 年 5 月 18 日

微处理器研究中心
中国科学院计算技术研究所

目录

研究背景

为什么需要二进制翻译？

一个新指令集架构的产生，需要建立其配套的软件生态：

软件迁移 + 重新编译

- ✓ 原生，性能优秀
- × 闭源软件无法迁移
- × 开源软件迁移需要投入大量时间 *Tab.*1

表 1: ArchLinux **中** LoongArch **架构软件支持清单**

仓库	二进制包数量	完成度
core	257	100%
extra	2651	88%
community	5691	41%

为什么需要二进制翻译？

一个新指令集架构的产生，需要建立其配套的软件生态：

软件迁移 + 重新编译

二进制翻译

× 翻译，性能较差

✓ 闭源软件直接翻译运行

✓ 开源软件直接翻译运行

二进制翻译打破指令集间“壁垒”，在新指令架构上运行已有程序：

- 快速提高新指令架构上软件丰富度
- 降低迁移的工作量
- 闭源软件运行的可能性

性能优化为什么是关键性问题？

二进制翻译

× **翻译，性能较差**

✓ 闭源软件直接翻译运行

✓ 开源软件直接翻译运行

翻译后程序性能直接影响使用体验：

- 看视频“卡成 PPT” *Fig.*1
- 存在超时机制的程序
  - V8 引擎
  - TCP 连接

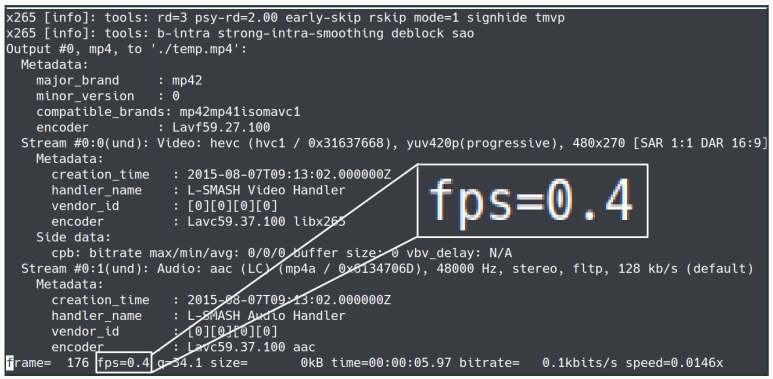


图 1: Qemu **运行** FFmpeg **解码视频运行帧率**

相关研究

Digital 公司的 FX!32 将 32 位 X86 程序迁移到 Alpha 架构:

- 首次运行产生概要文件
- 后续运行持续优化[a]
- Cache 优化
- 跳转优化
- 代码生成优化
- 地址对齐优化

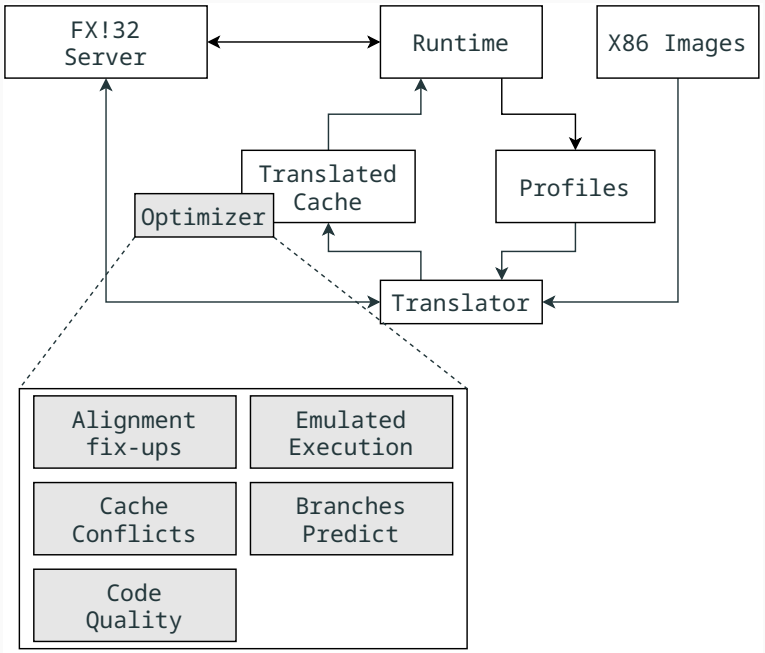


图 2: FX!32 **架构图**

[a] Drongowski et al., “Studying the Performance of the FX!32 Binary Translation System”.

Transmeta Crusoe

使用 VLIW 架构的全系统二进制翻译：

- **硬件辅助的二进制翻译**
- Shadow Register **机制**[a]
- **激进优化，错误回退**

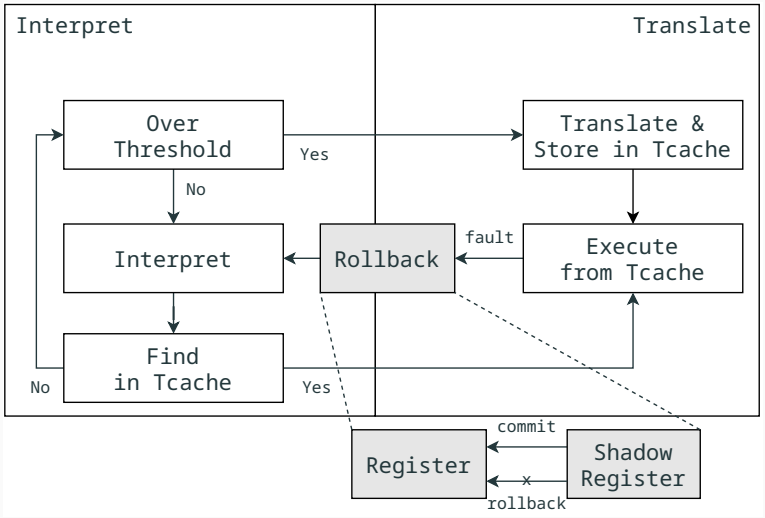


图 3: Transmeta Crusoe **架构图**

[a] Dehnert et al., “The Transmeta Code Morphing/spl trade/ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges”.

Dynamo

Dynamo **在运行过程中收集性能信息，用于对原程序优化**[a]：

- **轻量级性能收集算法** MRET
- trace **优化，热点块合并**
- **发展到** DynamoRIO

[a] Bala, Duesterwald, and Banerjia, “Dynamo: a transparent dynamic optimization system”.

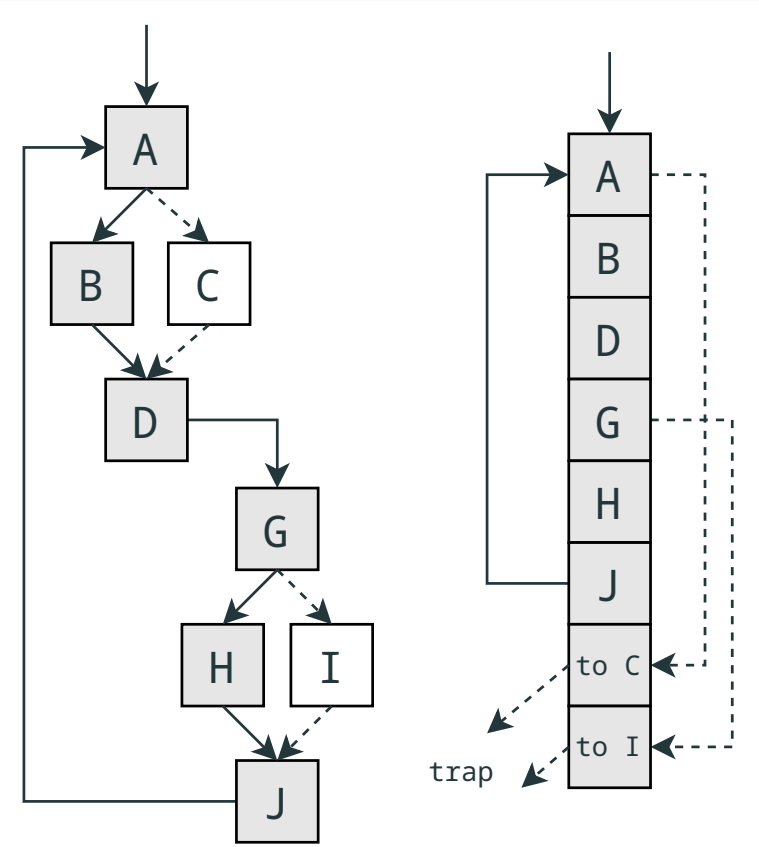


图 4: Dynamo **热点块合并优化**

MAMBO-X64 与 JTLT

MAMBO-X64 和 JTLT 优化都是通过硬件的一些修改，降低间接的跳转开销：

MAMBO-X64[a]

- 硬件加入跳转表
- 函数调用压栈时 SPC 和 TPC 同时入栈
- 返回时查找跳转表，跳转到 TPC

[a] D'Antras et al., "Optimizing Indirect Branches in Dynamic Binary Translators".

JTLT (Jump Target-address Lookup Table)[a]

- 加入 JTLT 和双地址返回值栈结构
- JTLT 与 BTB 联用，加速地址预测
- 双地址返回值栈和 MAMBO-X64 类似

[a] Kim and Smith, "Hardware support for control transfers in code caches".

HQEMU

HQEMU 在 QEMU 基础上加入 LLVM，用于指令优化：

- NET 的热点块收集
- LLVM 编译器优化
- 优化前后代码实时替换

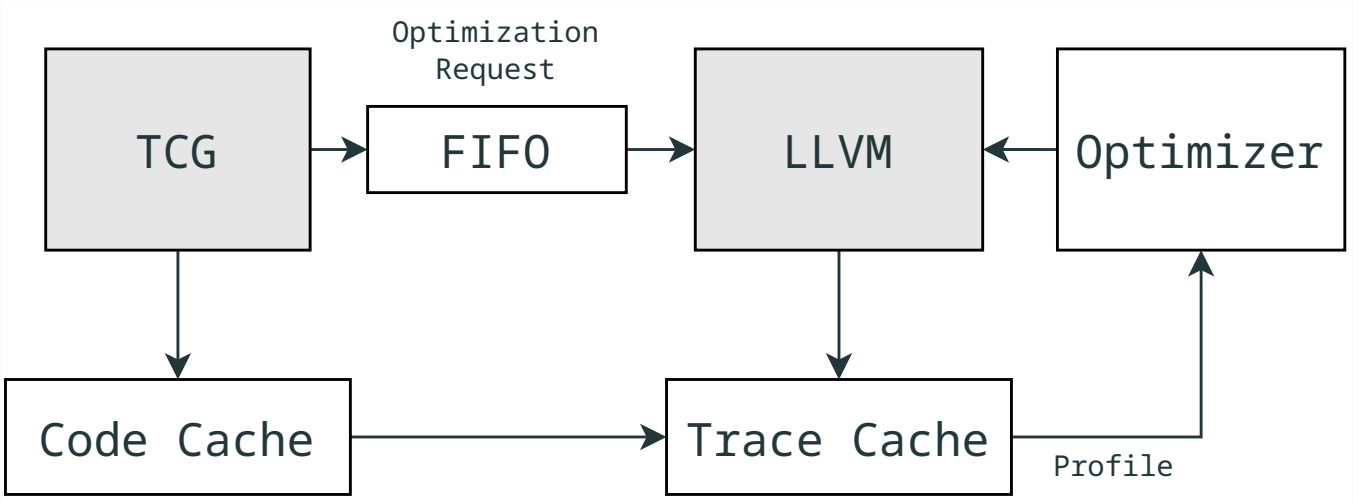


图 5: HQEMU **架构图**

Rosetta2

Rosetta2 **是近年来的商业二进制翻译软件**:

- X86 **到** ARM **二进制翻译器**
- **动静结合翻译**，AOT
- **较少的优化措施**
  - **活性分析等优化**

LATX (Loongson Architecture Translator from X86)

x86 到 LoongArch 用户态翻译器

- 正确性维护
  - 页大小不匹配
  - 指令翻译随机测试
  - EFLAGS/浮点栈硬件支持
- 翻译后性能
  - 原生性能 50-%
  - 单指令翻译优化

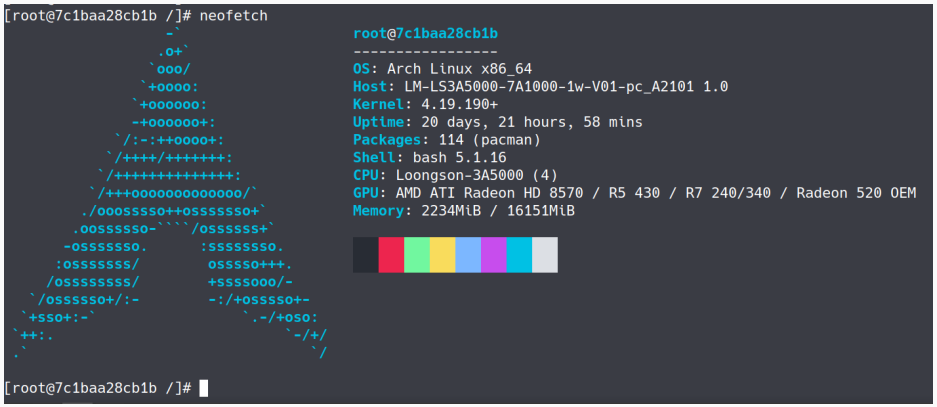


图 6: docker **内启动** X86_64 **的** Arch Linux

表 2: LATX **最初性能情况**

测试类型	优化前	优化前 $\frac{LATX}{Native}$	优化后	优化后 $\frac{LATX}{Native}$	性能提升
CINT	1220	45.3%	1295	48.1%	6.1%
CFP	1838	49.1%	1912	51.1%	4.0%

LATX 性能分析框架

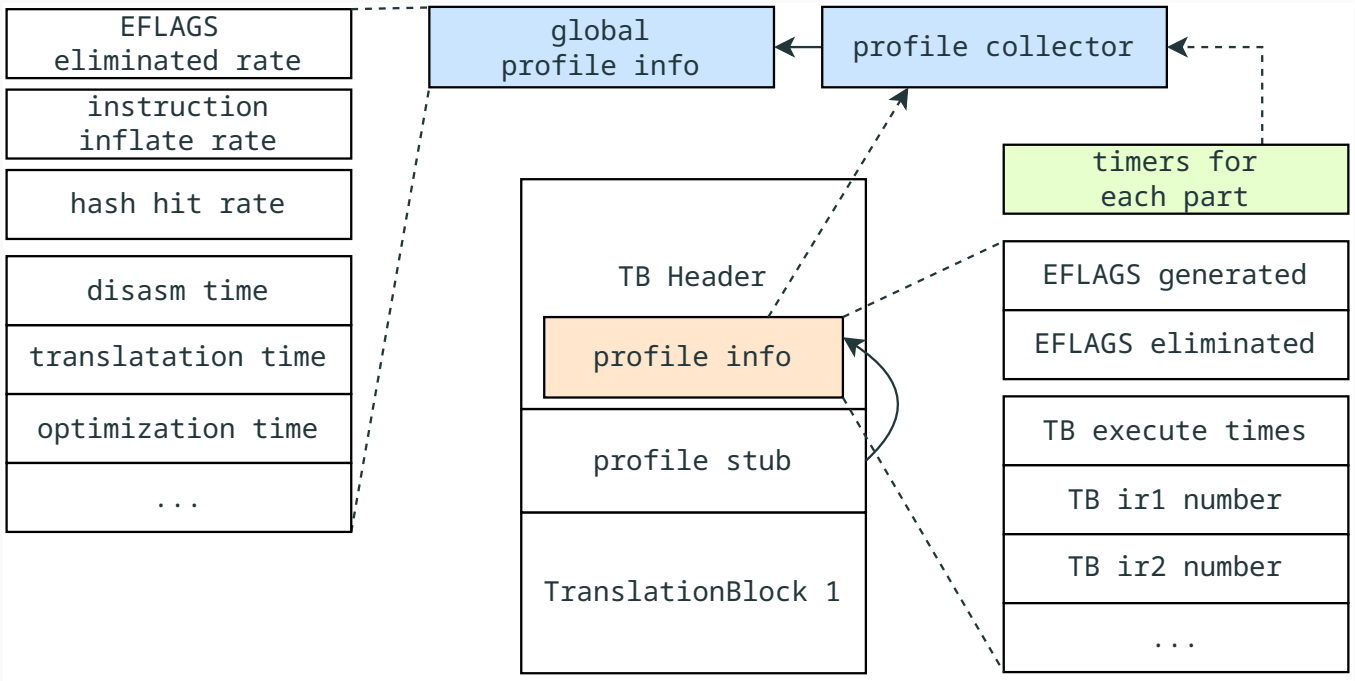


图 7: 性能分析系统架构

- 翻译开销统计
- EFLAGS 消除统计
- 指令翻译统计
- 直接/间接跳转统计

LATX 性能分析数据

表 3: gzip **中指令使用频率**

指令类型	动态运行数量	占比	exp.
mov	7,718,467,866	16.04%	1.78
cmp	6,987,729,783	14.52%	2.87
movzx	5,350,658,906	11.12%	2.97
jne	3,790,599,144	7.88%	6.74
lea	3,151,611,499	6.55%	2.66
sub	2,744,875,934	5.70%	3.77
and	2,513,651,036	5.22%	3.94
je	2,318,250,121	4.82%	6.93
movsxd	1,995,327,254	4.15%	1.06
jae	1,978,473,851	4.11%	6.95
add	1,850,150,565	3.84%	3.47
xor	1,563,243,497	3.25%	4.16
Other	6,165,295,262	12.81%	3.35
总计	48,128,334,718	100.00%	3.52

- 50% **以上运算指令需要计算/使用** EFLAGS
- LBT[a] **中** EFLAGS **计算类指令性能较低**

表 4: EFLAGS **运算指令性能**

指令类型	issue	cycle
搬运指令	1	2
运算指令	1	2
条件指令	1	1

[a] Loongson Binary Translation, **龙芯二进制翻译扩展指令**

LATX 性能分析数据

实验：SSE 标量运算指令优化

- 去除所有向量寄存器高位的“保存-恢复”操作[a]

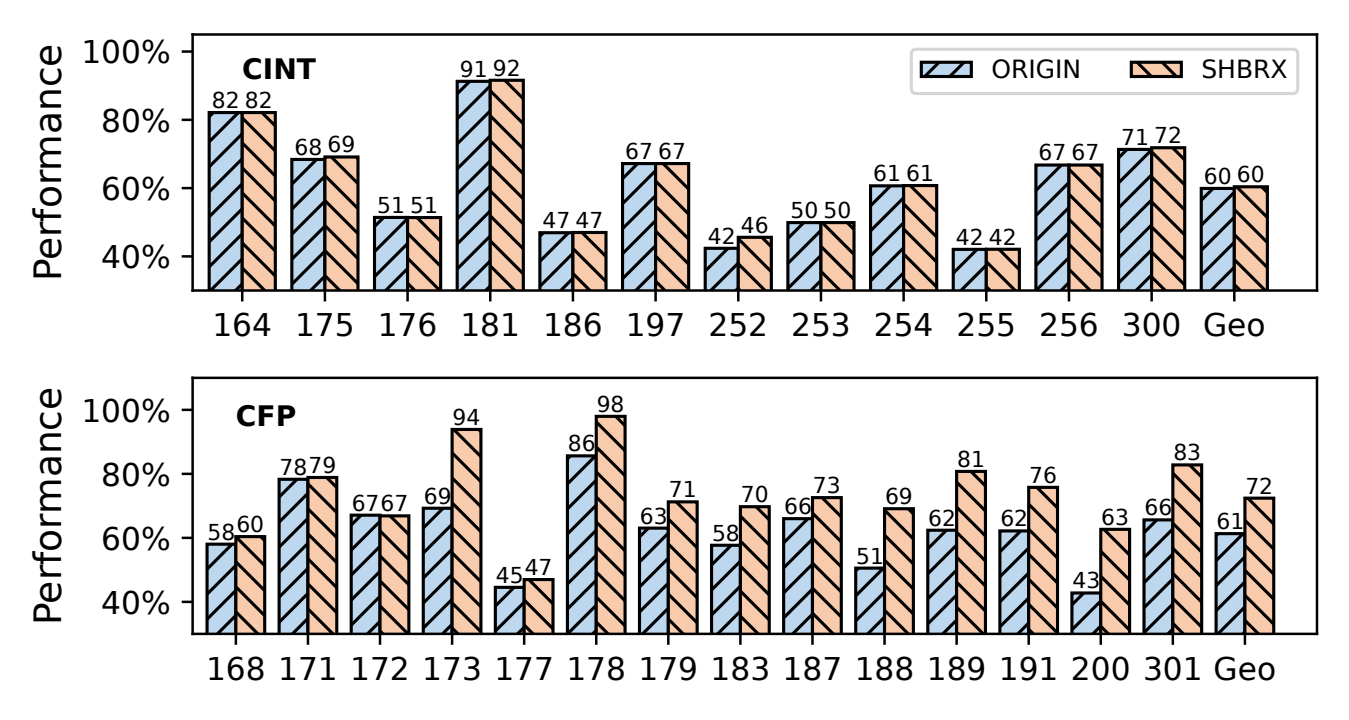


图 8: 高位操作完全消除性能数据

[a]X86 为了向前兼容，需要对高位**保留原值**

指令流分析制导的动态二进制翻译优化 (IFADO)

LATX 优化方向

- EFLAGS 延迟计算
- “反馈式”指令语义化翻译
- SSE 标量指令高位运算消除

指令流分析制导的动态二进制翻译优化

Instruction Flow Analysis Directing Optimization (IFADO)

指令流分析制导的动态二进制翻译优化（IFADO）

LATX 优化方向

- EFLAGS 延迟计算
- “反馈式”指令语义化翻译
- SSE 标量指令高位运算消除

指令流分析制导的动态二进制翻译优化

Instruction Flow Analysis Directing Optimization (IFADO)

	破坏性	优化系统	优化多样性	完备性
FX!32	–	需要 profile	较多	好
Transmeta	硬件	轻量级	激进的优化	好
Dynamo	–	需要 profile	热点块	较好
MAMBO-X64	硬件	轻量级	跳转	较好
JTLT				
HQEMU	–	LLVM	编译器支持	较差
Rosetta2	部分硬件	AOT	少	好
IFADO	已有指令	轻量级	支持浮点（SSE）	好

指令流分析制导的动态二进制翻译优化（IFADO）

总体架构

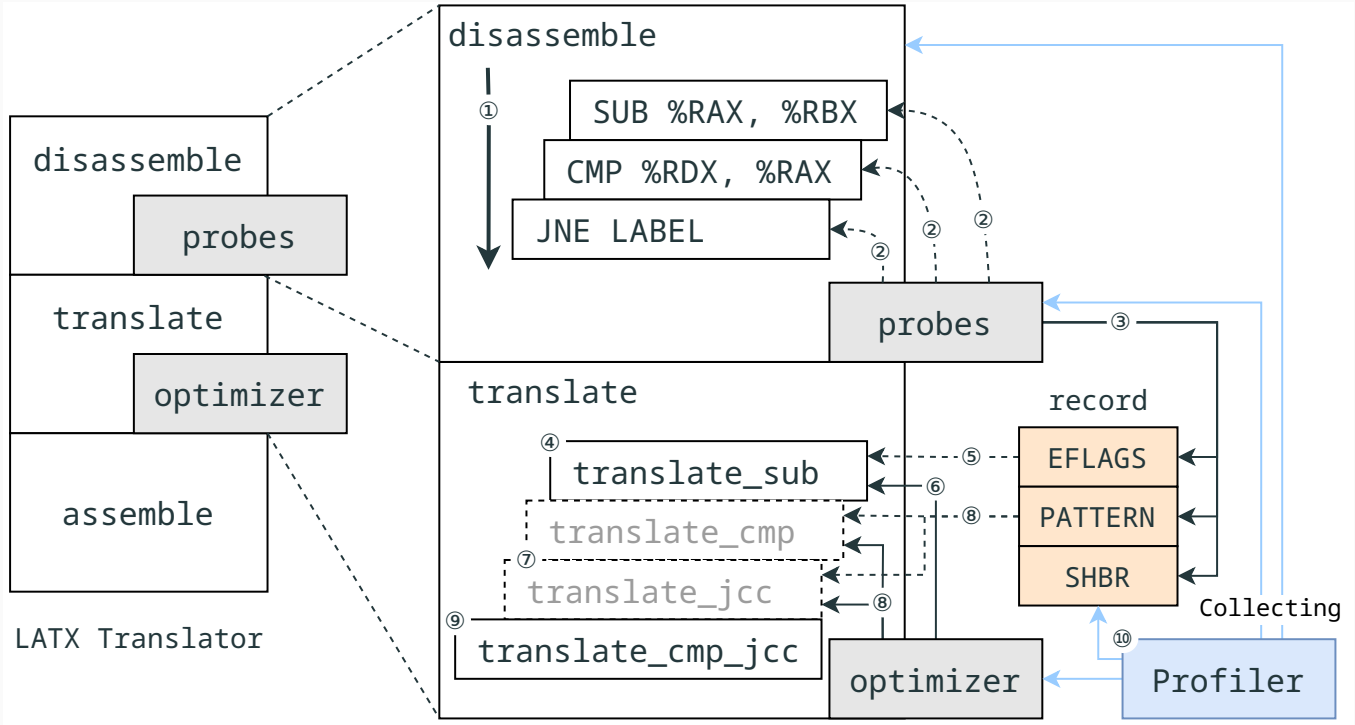


图 9: 指令流分析制导优化框架构图

- 反汇编阶段（①）
- 探针搜索优化（②）
- 记录已搜寻优化（③）
- 翻译过程执行优化（④～⑨）

EFLAGS 延迟计算

QEMU 等二进制翻译器对 EFLAGS 延迟计算的缺陷

- 额外开销
  - 需要从内存加载源数据
- 粗粒度
  - 任一 EFLAGS 被使用都会导致全部计算
- 可扩展性差
  - 基本块内优化
  - 跨基本块复杂处理（预翻译、超级块、热路径等）

EFLAGS 延迟计算 – 架构

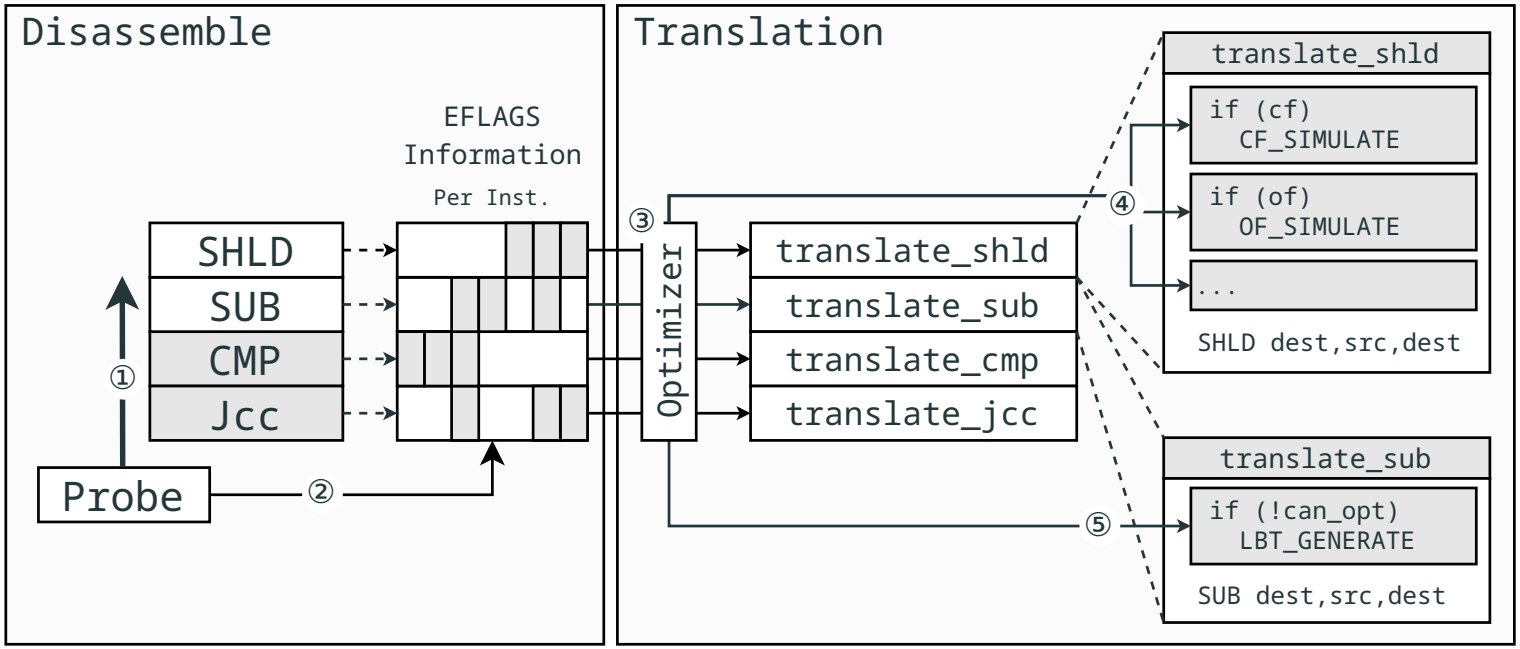


图 10: EFLAGS 延迟计算架构

- 反汇编阶段探针识别，翻译阶段执行优化

EFLAGS 延迟计算 – 探针识别

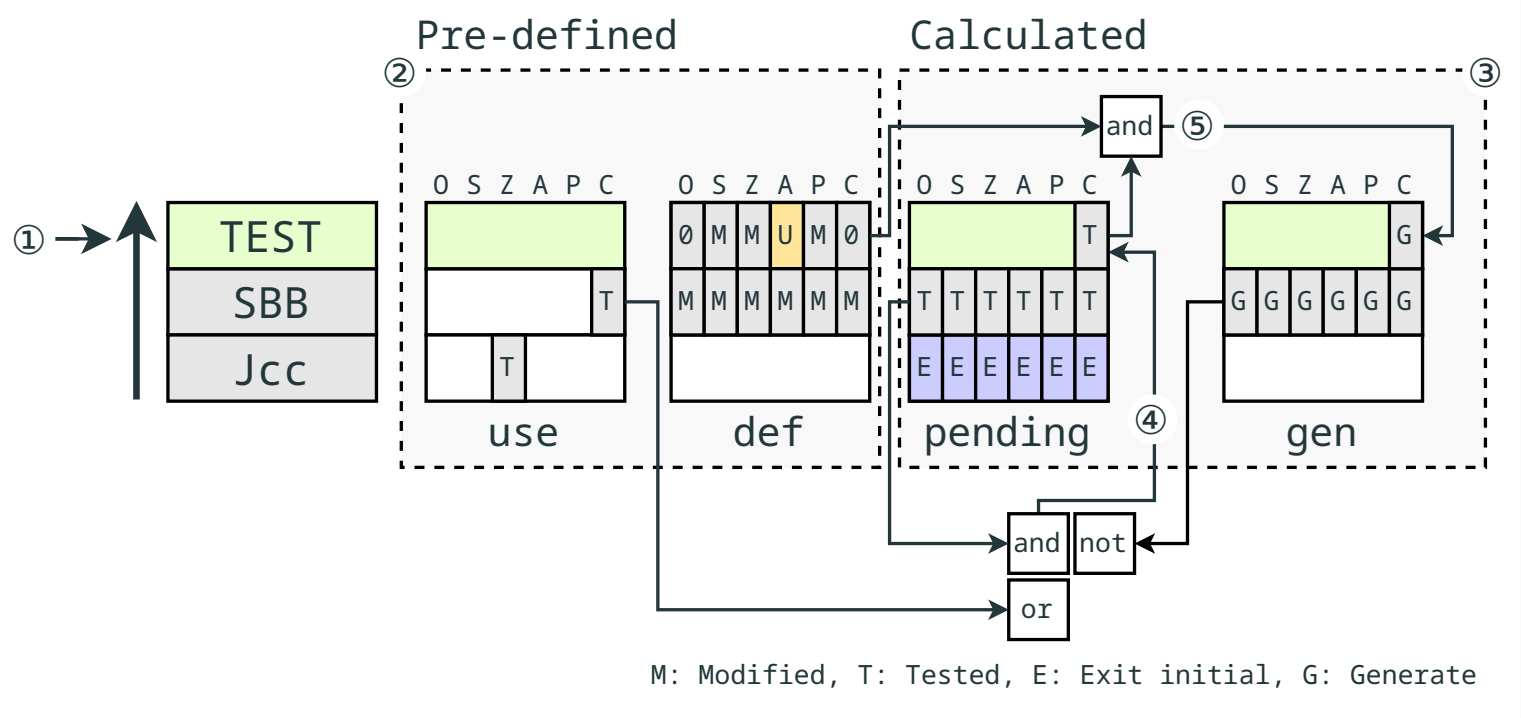


图 11: EFLAGS 延迟计算优化指令流分析过程

- 逆序扫描
- 根据后续使用的情况计算该指令的需要计算的状态位

EFLAGS 延迟计算 – 单指令翻译优化

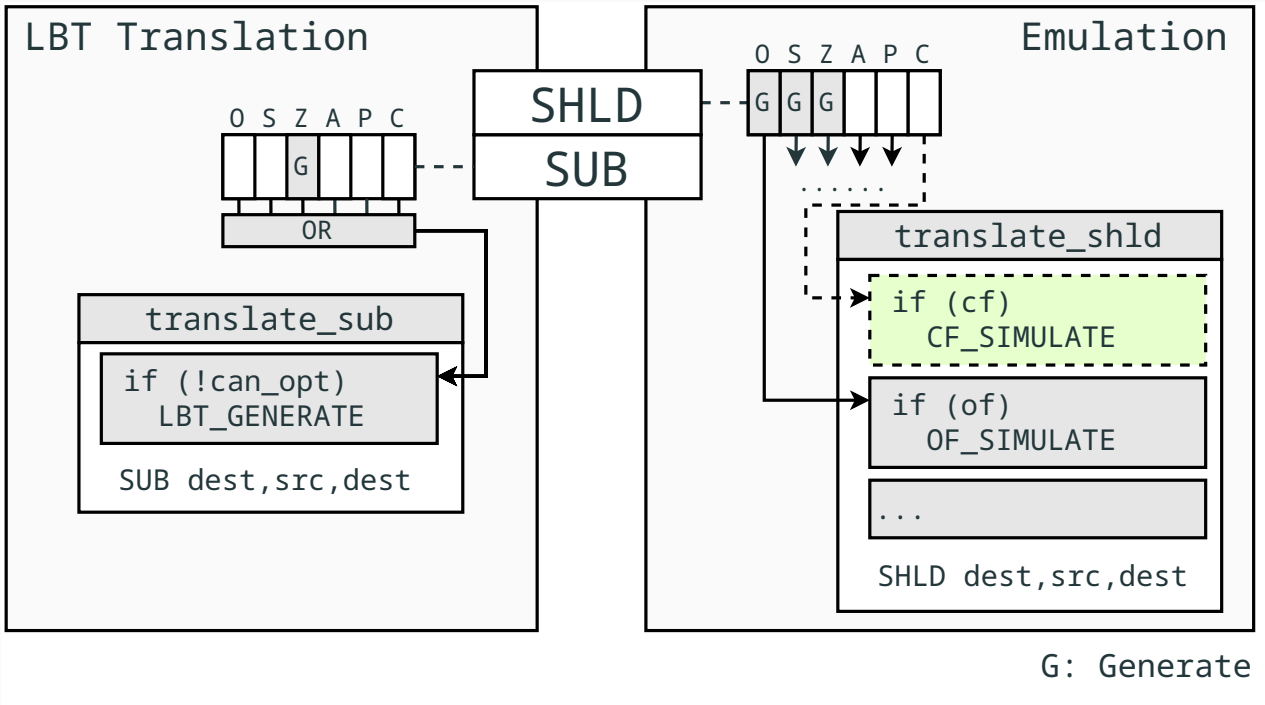
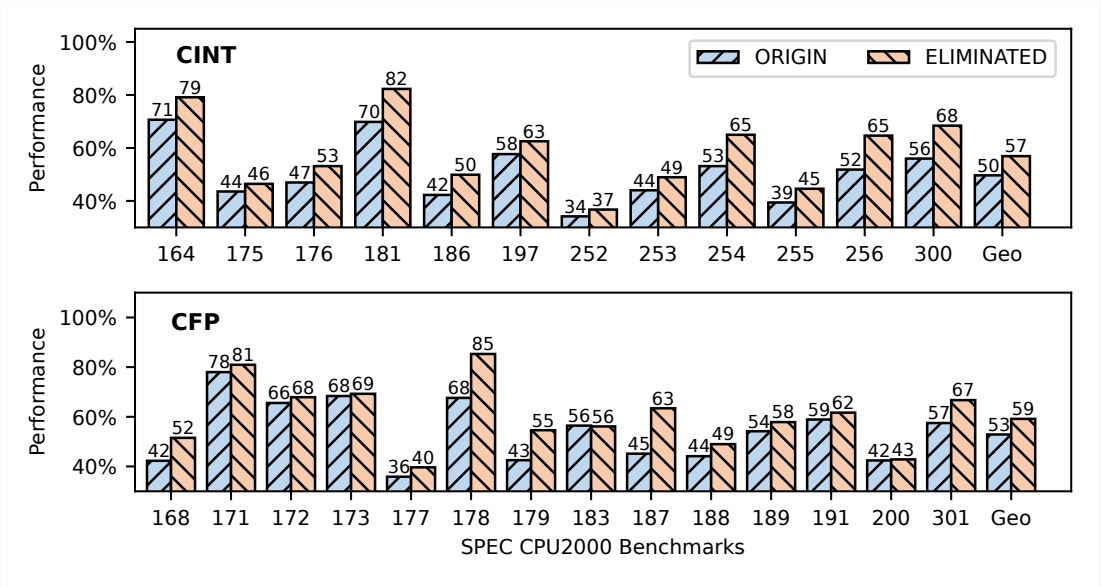
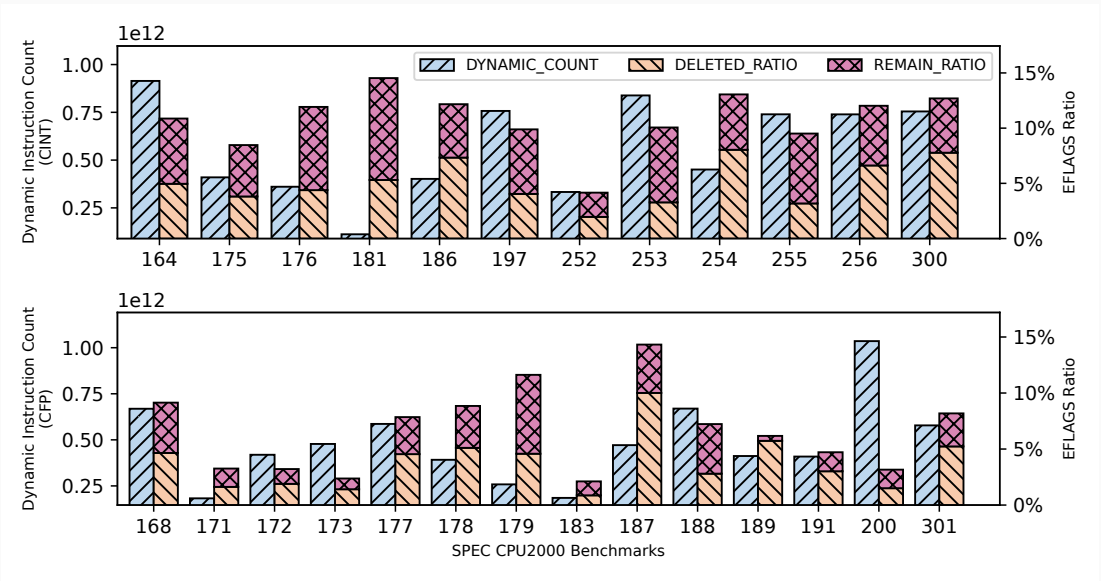


图 12: EFLAGS 指令消除

- LBT 指令：当任一 EFLAGS 无需计算时可以消除
- 模拟 EFLAGS：对每种类型 EFLAGS 标志单独运算

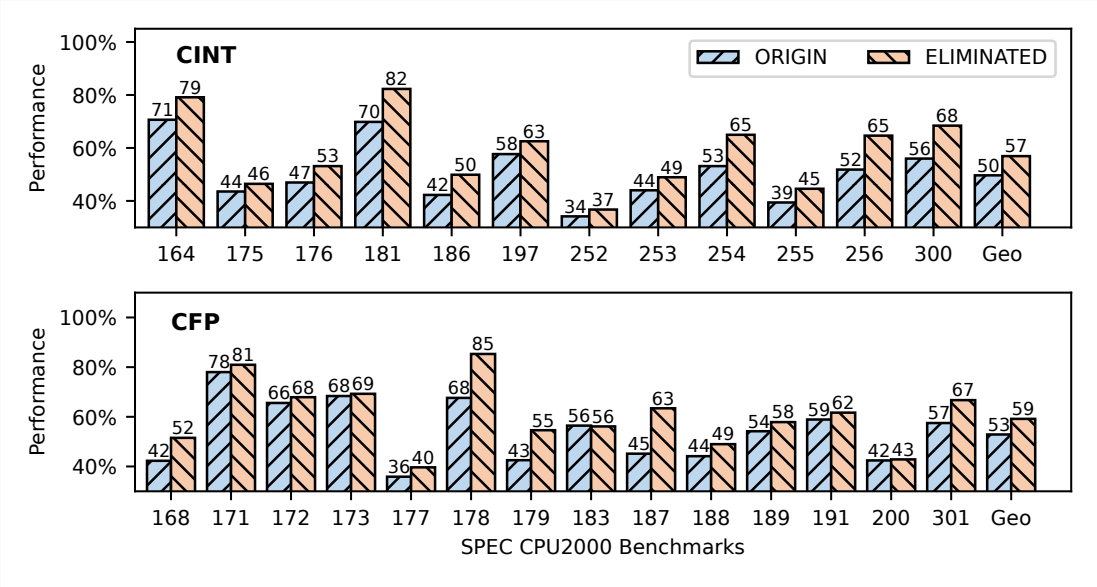
EFLAGS 延迟计算效果

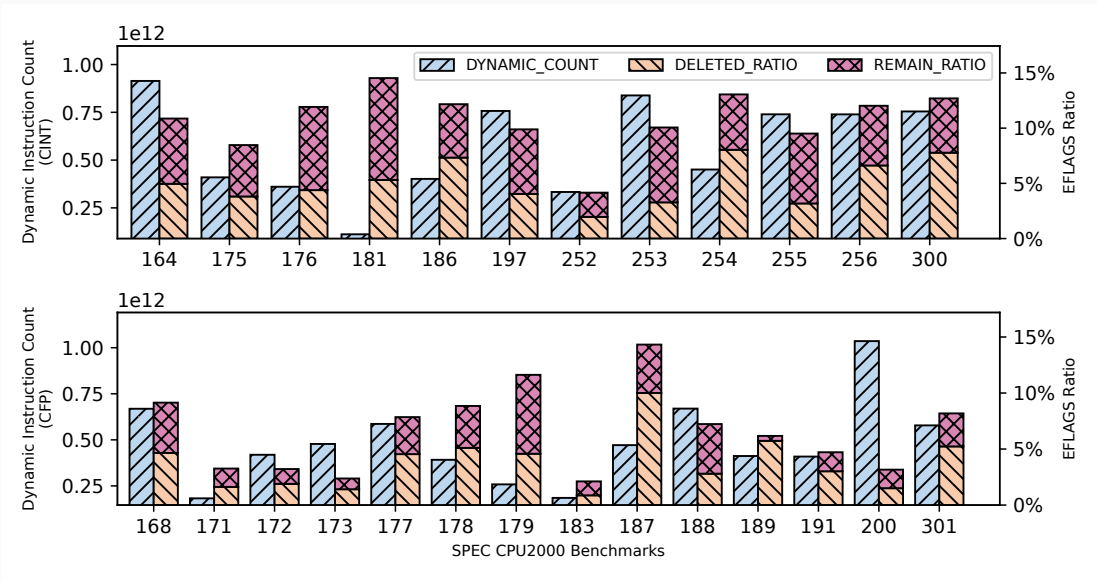




- 性能 6%+ 提升
- 部分程序消除不明显
  - 176, 181, ...

EFLAGS 延迟计算效果



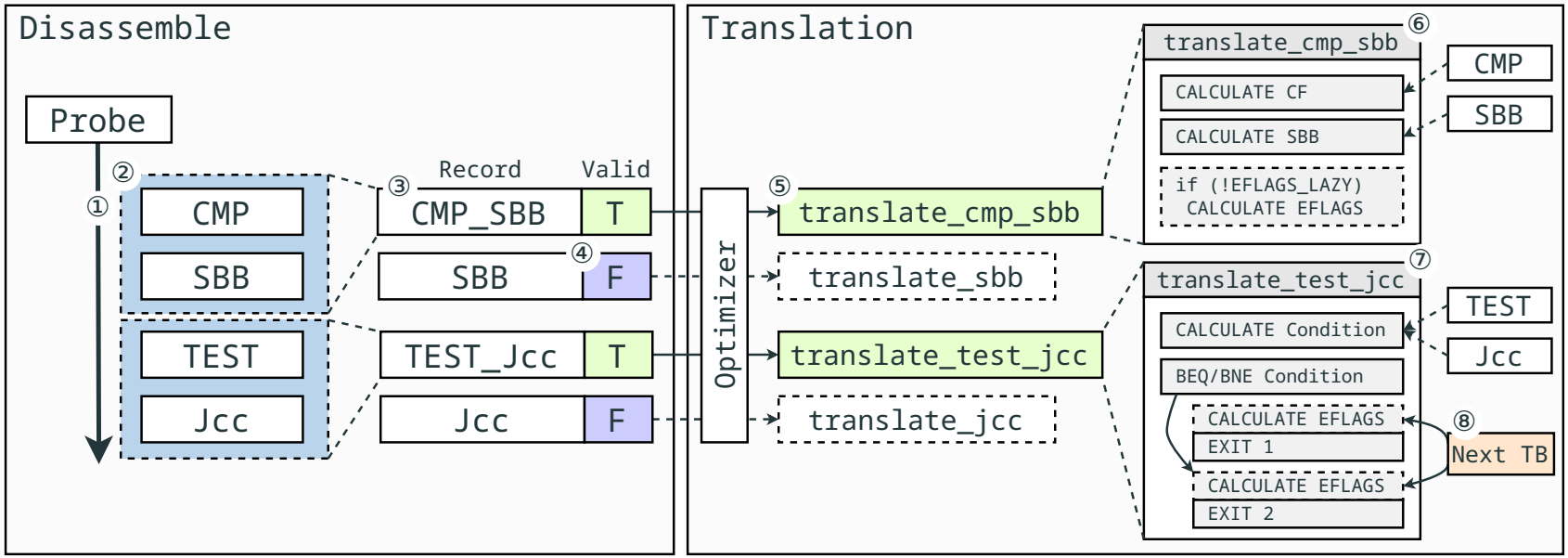


- 性能 6%+ 提升
- 部分程序消除不明显
  - 176, 181, ...

主要原因

- 后续指令使用
  - CMP/TEST+Jcc
- 基本块结尾产生

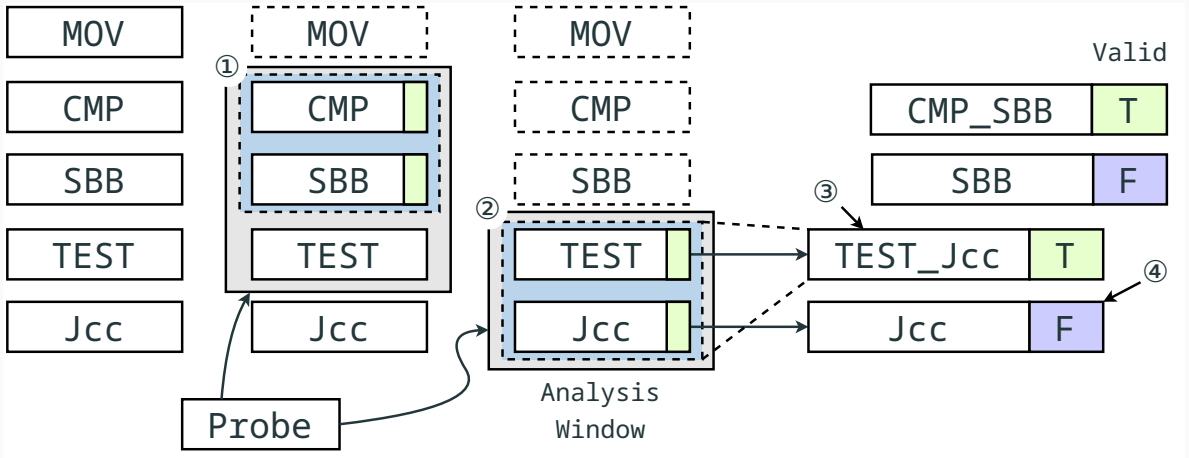
“反馈式”指令语义化翻译



解决问题

- EFLAGS 消除不完全
- 基本块内单指令翻译语义冗余

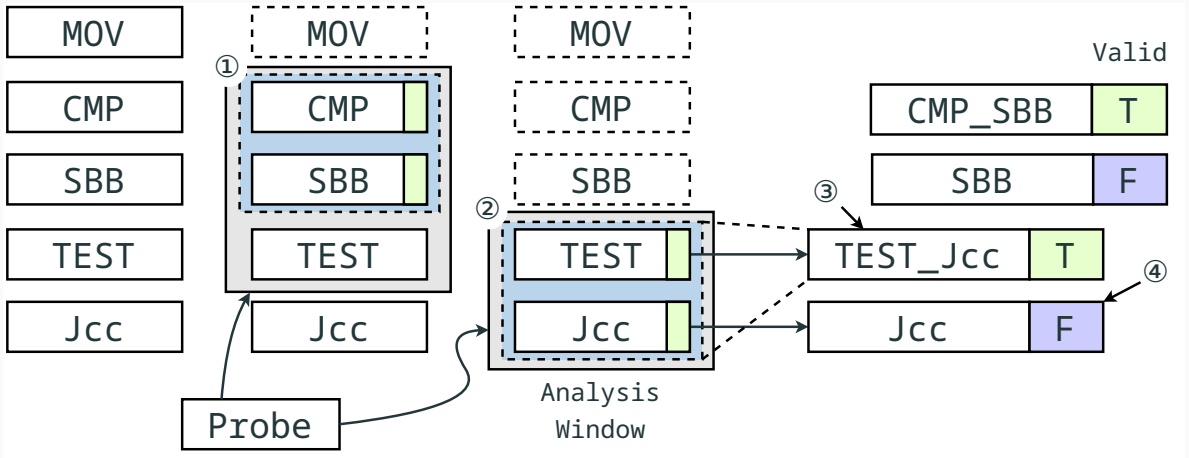
“反馈式”指令语义化翻译 – 探针识别



与窥孔优化类似

- 更加激进 – 只关心主要语义
- 其余语义 – 异常处理/基本块结尾处理

“反馈式”指令语义化翻译 – 探针识别



与窥孔优化类似

- 更加激进 – 只关心主要语义
- 其余语义 – 异常处理/基本块结尾处理

“激进”存在的问题

- 部分优化需要跨越基本块，优化不全？
- 优化后如何弥补与原有指令序列的语义差异？

“反馈式”指令语义化翻译 – 多指令翻译优化

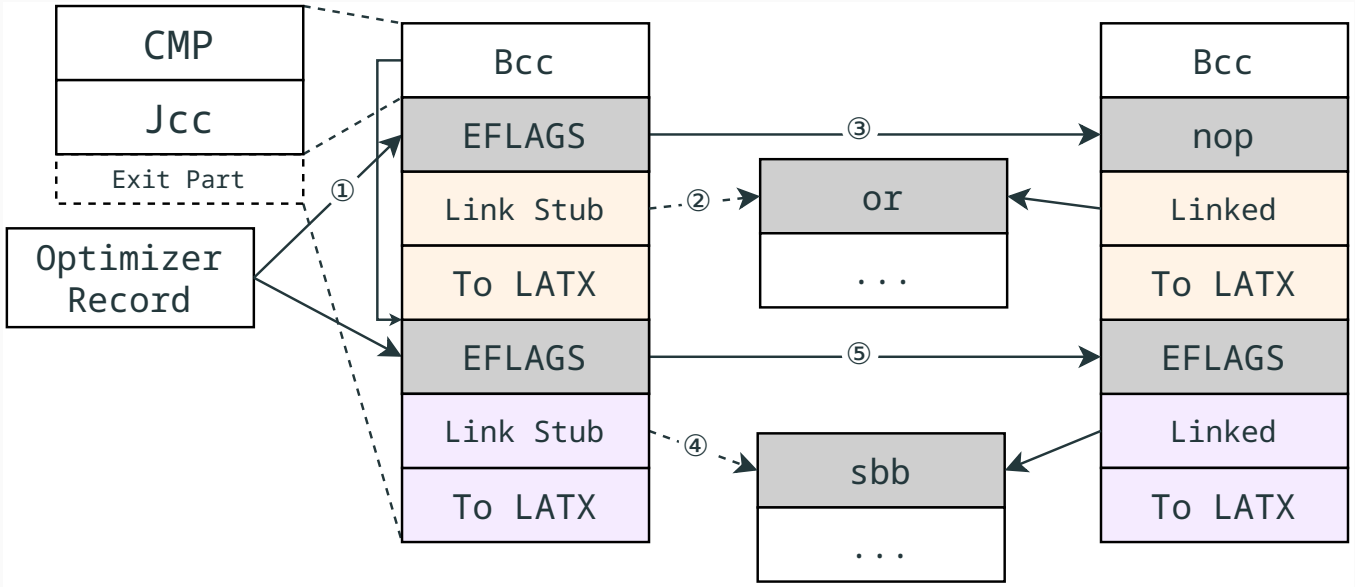
部分优化需要跨越基本块，优化不全？– “反馈式”

- 消除式跨基本块反馈优化 – LBT 指令
- 链接式跨基本块反馈优化 – EFLAGS 模拟

“反馈式”指令语义化翻译 – 多指令翻译优化

消除式跨基本块反馈优化

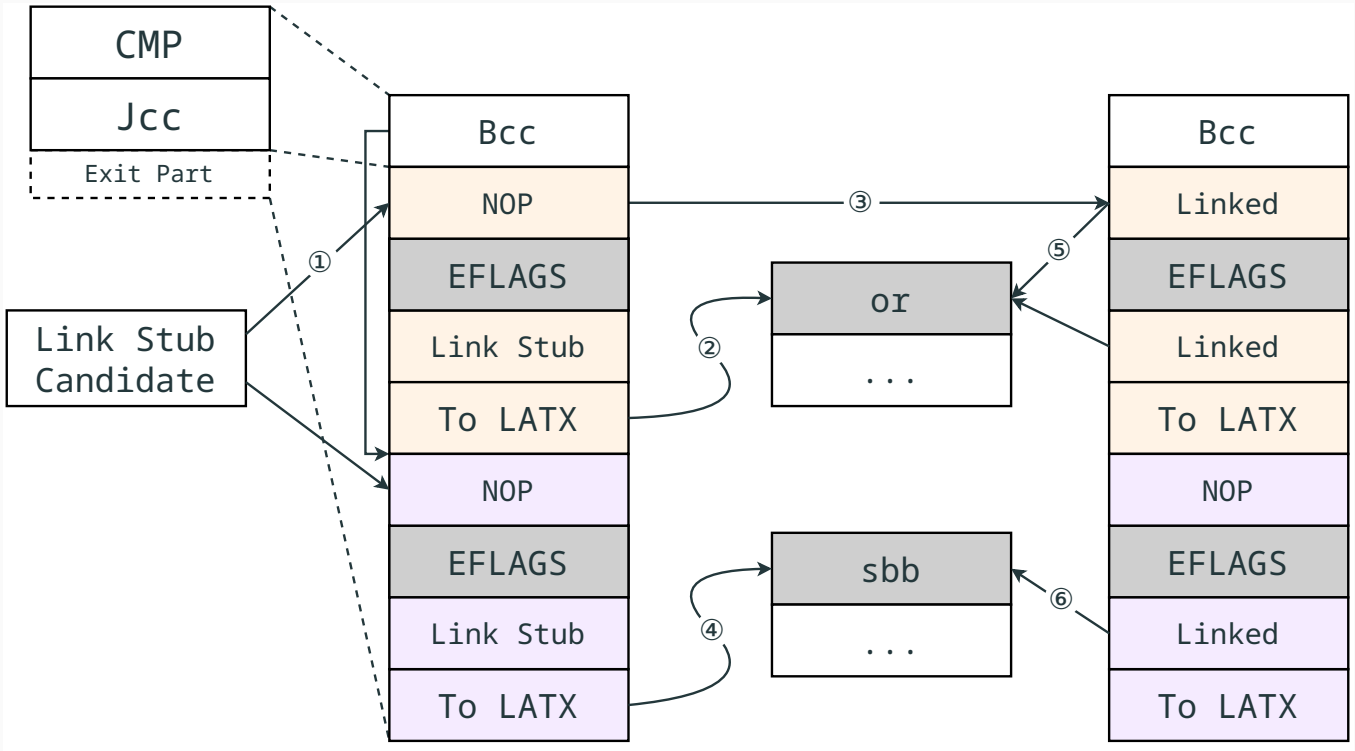
- 记录可能存在的优化（①）
- 后续“反馈”信息（②, ④）
- 可以优化情况（③）
- 不可优化情况（⑤）



“反馈式”指令语义化翻译 – 多指令翻译优化

链接式跨基本块反馈优化

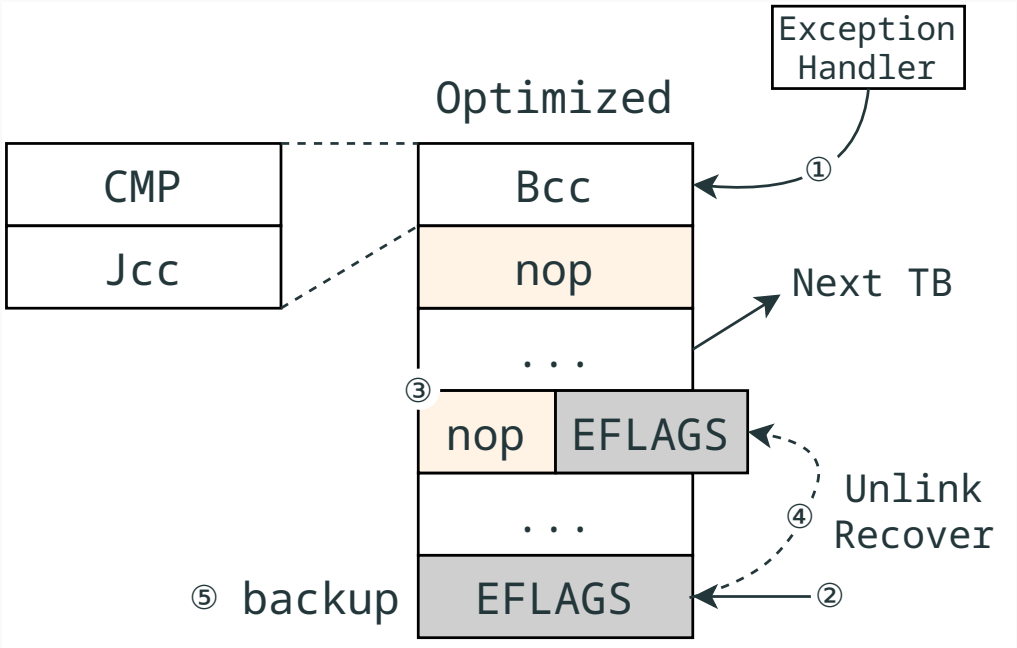
- 记录预留的链接点位（①）
- 后续“反馈”信息（②, ④）
- 可以优化情况（③, ⑤）
- 不可优化情况（⑥）



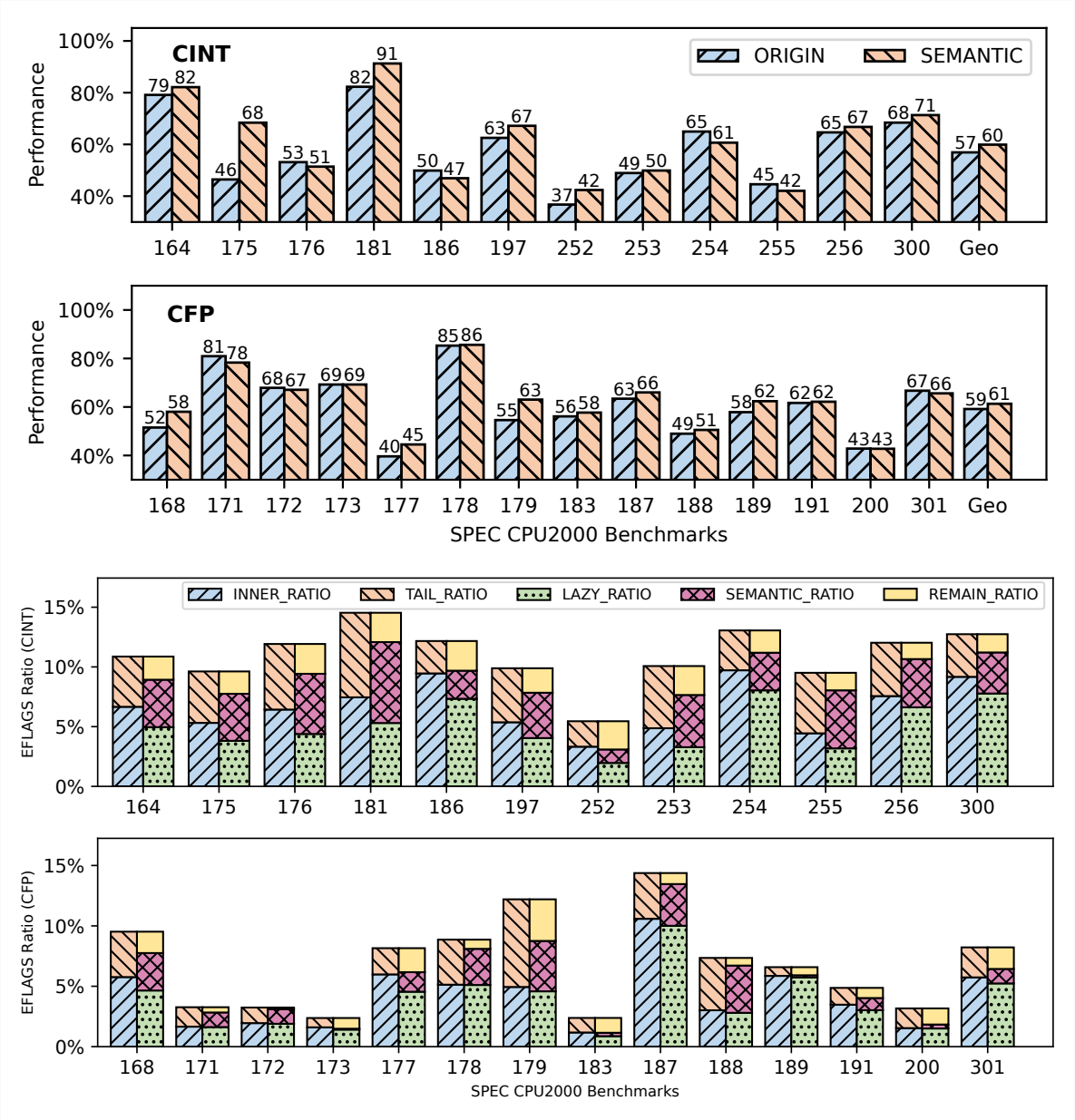
“反馈式”指令语义化翻译 – 从异常恢复

优化后如何弥补与原有指令序列的语义差异？

- 翻译时保留差异信息（②, ⑤）
- 恢复状态，修改基本块（③, ④）



“反馈式”指令语义化翻译效果



- **性能** 3%+ **提升**
- EFLAGS **整体消除比例达到** 80%

SSE 标量指令高位运算消除

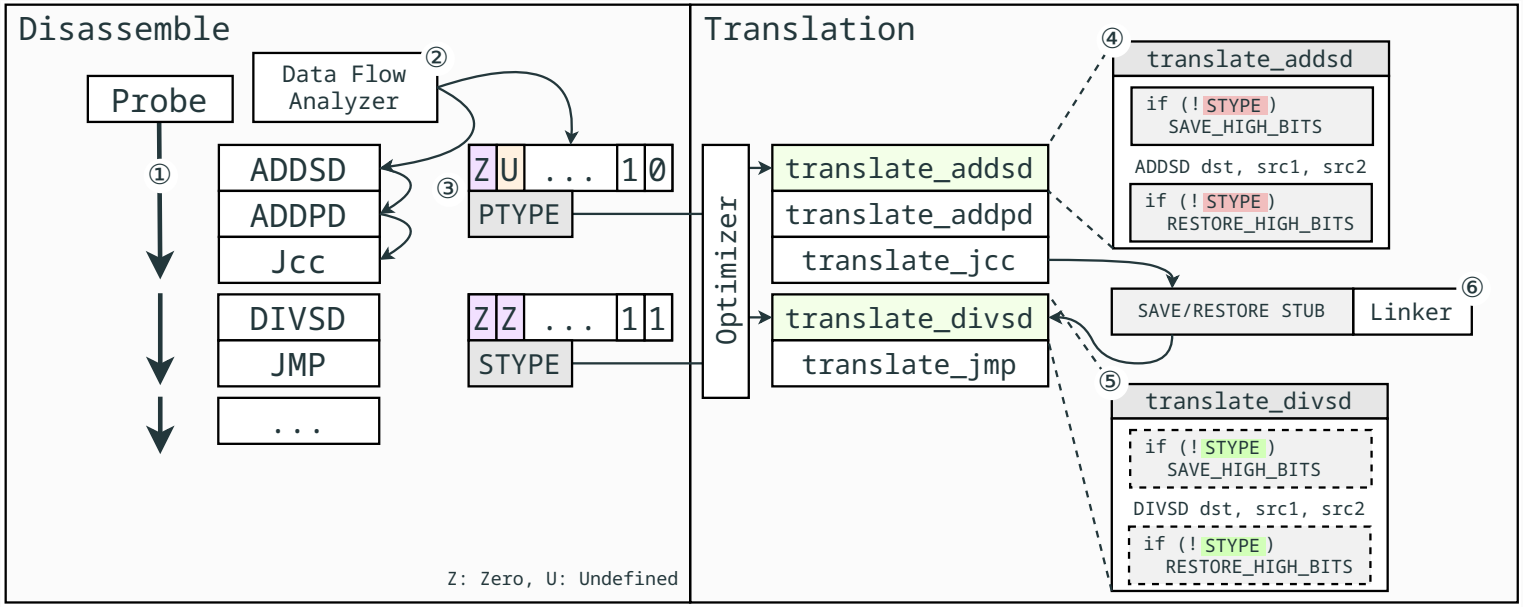
为什么要对 SSE 标量指令高位计算进行消除？

- 指令语义差距 - X86 计算高位保留
- 64 位程序默认使用 SSE 替代 X87 浮点计算指令
- 高位“保存-恢复”带来至少两条指令的冗余

```
000000000040c5b0 <_Z13ggOverlapBox2RK6ggBox2S1_>:
  40c5b0:       f2 0f 10 46 10          movsd  0x10(%rsi),%xmm0               <---- xmm0
  40c5b5:       66 0f 2f 07             comisd (%rdi),%xmm0                   ----> xmm0
  40c5b9:       76 35                   jbe    40c5f0 <_Z13ggOverlapBox2RK6ggBox2S1_+0x40>
  40c5bb:       f2 0f 10 47 10          movsd  0x10(%rdi),%xmm0               <---- xmm0
  40c5c0:       66 0f 2f 06             comisd (%rsi),%xmm0                   ----> xmm0
  40c5c4:       f2 0f 10 4f 08          movsd  0x8(%rdi),%xmm1                <---- xmm1
  40c5c9:       f2 0f 10 56 18          movsd  0x18(%rsi),%xmm2               <---- xmm2
  40c5ce:       f2 0f 10 67 18          movsd  0x18(%rdi),%xmm4               <---- xmm4
  40c5d3:       f2 0f 10 5e 08          movsd  0x8(%rsi),%xmm3                <---- xmm3
  40c5d8:       76 16                   jbe    40c5f0 <_Z13ggOverlapBox2RK6ggBox2S1_+0x40>
  40c5da:       66 0f 2f d1             comisd %xmm1,%xmm2                    ----> xmm1, xmm2
  40c5de:       76 10                   jbe    40c5f0 <_Z13ggOverlapBox2RK6ggBox2S1_+0x40>
  40c5e0:       31 c0                   xor    %eax,%eax
  40c5e2:       66 0f 2f e3             comisd %xmm3,%xmm4                    ----> xmm3, xmm4
  40c5e6:       0f 97 c0                seta   %al
  40c5e9:       c3                      ret
```

图 13: SSE 浮点运算片段

SSE 标量指令高位运算消除



SSE 标量指令高位运算消除（Scalar Higher Bits Remove, SHBR）

- 核心思想：“分类讨论”
  - 分类：将基本块按照一定规则分类
  - 讨论：根据不同分类执行相应优化

标量浮点高位计算消除 – “分类”（探针搜索）

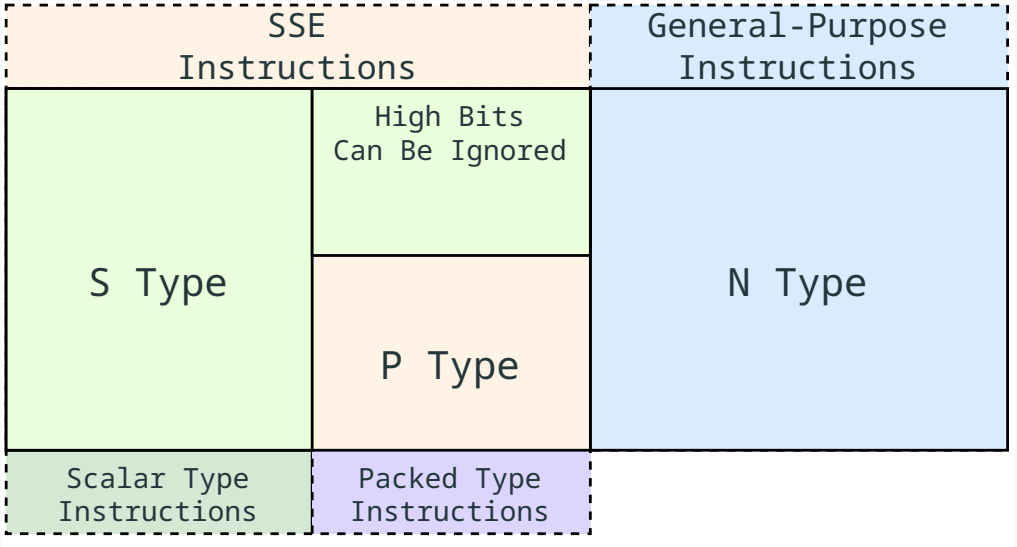
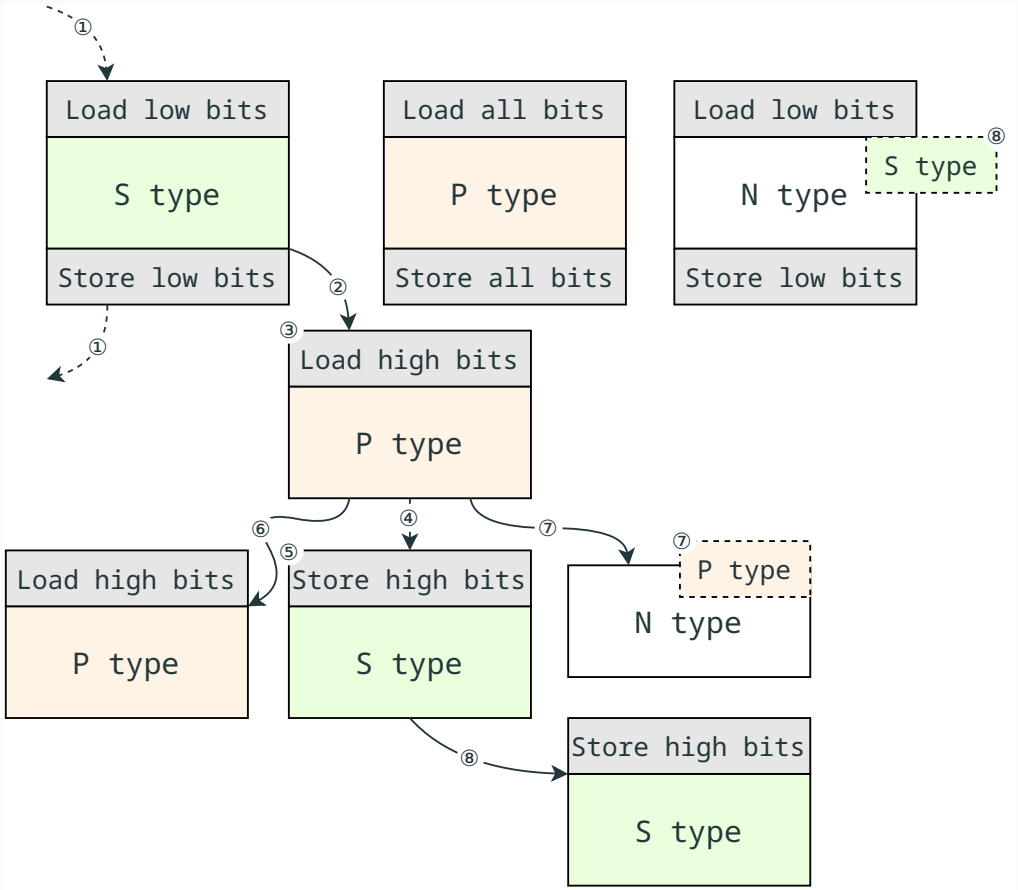


图 14: SHBR **算法对基本块的分类**

- S(Scalar)：**标量基本块**
- P(Packed)：**向量基本块**
- N(None)：**无** SSE **指令基本块**

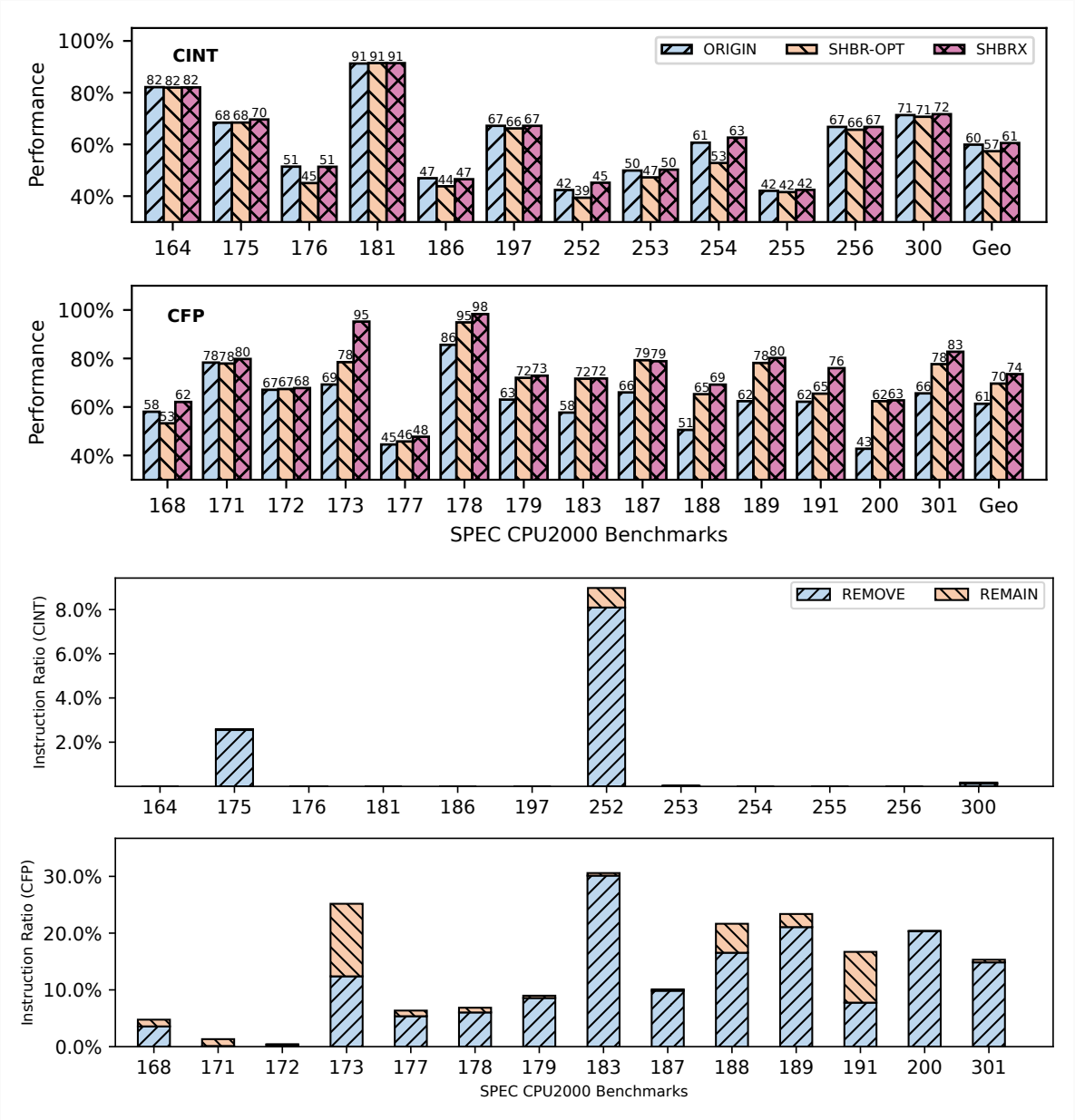
- **指令流分析后进行分类：**
- S **类型块内只有标量运算指令 或**
- **存在向量指令，但高位不影响结果**

标量浮点高位计算消除 - “讨论”（翻译优化）



- S 类型进行消除，其余不做处理
- 向量寄存器高位数据默认在内存
- 链接不同类型需要处理（②, ④）
- 相同类型基本块直接链接（⑥, ⑧）

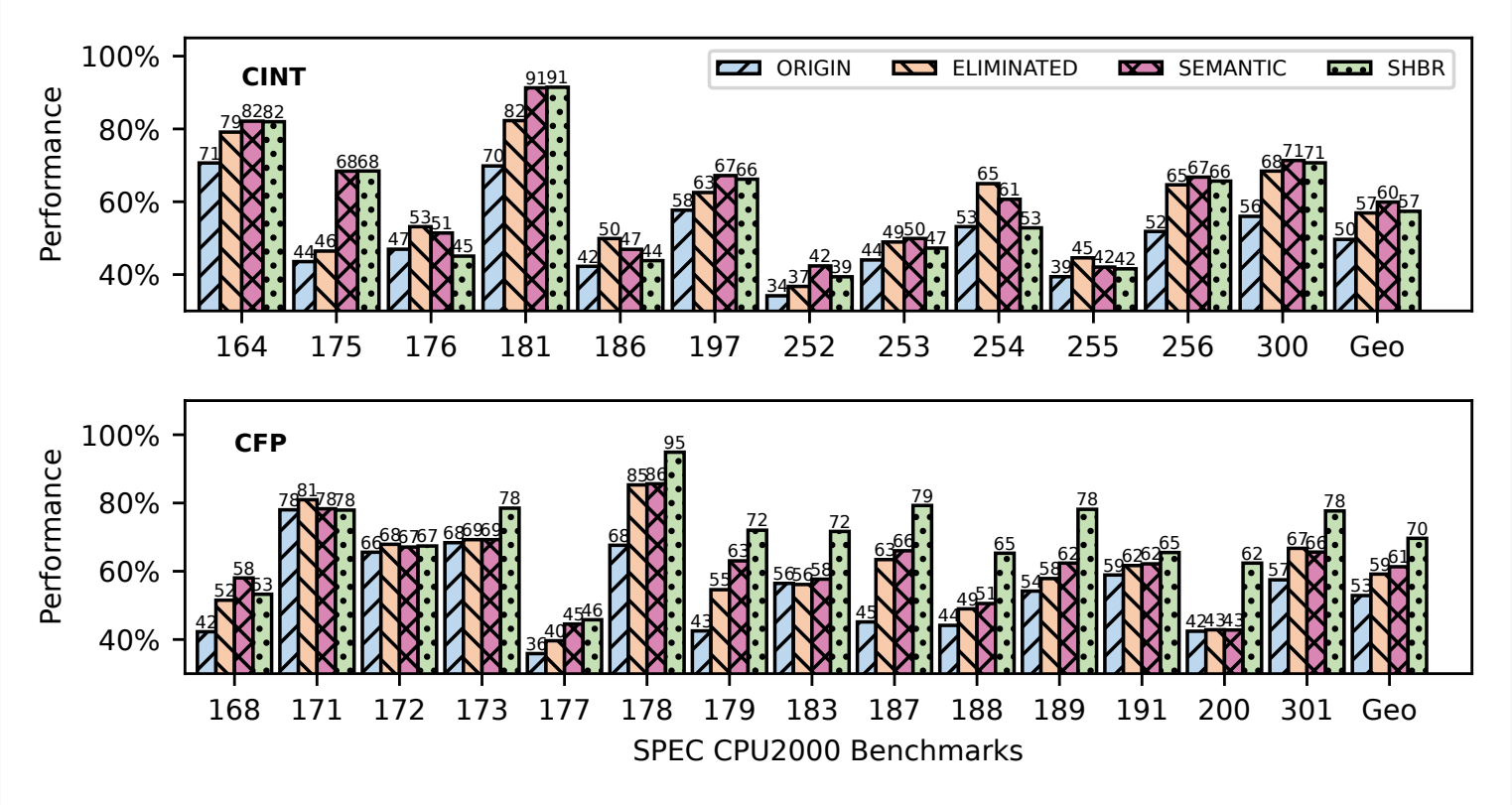
标量浮点高位计算消除性能数据



- 浮点 10% 提升
- 消除 90% 高位计算
- 定点性能略有下降
- 部分子项切换频繁

总结与展望

总结



- EFLAGS **消除**（6%）→ **语义化翻译**（3%）→ SSE **标量指令高位运算消除**（**浮点** 10%）
- EFLAGS **运算消除** 80%+
- **标量浮点高位计算消除** 90%+

展望

“反馈式”优化的探索深度不够

- 加大反馈深度
- 微小基本块合并，如只有一条无条件跳转的基本块

SSE 标量指令高位运算消除对定点程序不友好

- 热路径上 SHBR 优化
- 启发算法 – 缓解基本块类型“乒乓效应”

其他优化的探索

- 引入更加重量级别的优化
- 编译器的 PASS，AOT

请各位老师批评指正

References


Bala, Vasanth, Evelyn Duesterwald, and Sanjeev Banerjia. **“Dynamo: a transparent dynamic optimization system”.** In: *SIGP*. 2000.


D’Antras, Amanieu et al. **“Optimizing Indirect Branches in Dynamic Binary Translators”.** In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13 (2016), pp. 1 –25.


Dehnert, James C. et al. **“The Transmeta Code Morphing/spl trade/ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges”.** In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* (2003), pp. 15–24.

References ii


Drongowski, Paul J. et al. **“Studying the Performance of the FX!32 Binary Translation System”.** In: 2007.


Kim, Ho-Seop and James E. Smith. **“Hardware support for control transfers in code caches”.** In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* (2003), pp. 253–264.

LATX 性能分析数据

为什么要对 EFLAGS 延迟计算？

- EFLAGS 计算指令性能差距[a]
- 超过半数指令会产生 EFLAGS 计算指令

为什么要进行语义化翻译？

- 编译器特性 – 指令模板
- X86 条件跳转使用 EFLAGS – EFLAGS 指令的产生

为什么要进行 SSE 标量运算指令优化？

- 指令集差距 – X86 与 RISC 指令集的共性[b]
- 高位“保存-恢复”带来至少两条指令的冗余

---

[a] 1 条 EFLAGS 运算指令 ≈ 8 条不存在数据依赖关系的普通运算指令

[b] X86 为了向前兼容，需要对高位**保留原值**