GoLLVM编译探索

马春辉

字节跳动程序语言团队
工程师

目 录

第一部分

团队与个人

团队与个人

- 字节跳动程序语言团队
  - go 编译器/Runtime/GC 优化
  - 基础库、性能分析工具、java、python
- 马春辉
  - 十多年的编译器领域相关工作经验
  - 先后就职于HP编译器组， IBM jvm组，华为虚拟机实验室，字节跳动程序语言团队

第二部分

GoLLVM背景

GoLLVM背景

- 字节内有大量的go微服务
  - 性能要求
- 在原生Go SDK上的一些传统编译优化收益超过几十万核

PSM	CPU	Latency	memory
微服务1 (60w+)	9%	10%	4%
微服务2	5%	5%	15%
微服务3	3%	10%	
微服务4	12%		18%

GoLLVM背景

- 传统编译优化在go compiler上的实现
  - Inline 策略调整
  - 栈大小调整
  - Fast path inline
  - Aggressive BCE

GoLLVM背景

- 两条路
  - 继续在原生Go SDK上开发
    - 优化pass 少
    - SSA 比较简陋，缺少一些优化的基础设施
  - 探索利用LLVM的优化能力：语言团队与STE-编译器组联合探索
    - LLVM作为一个基本的编译器框架，支持多种语言
      - C/C++/Fortran/Rust/Swift/Java(Falcon)
    - Tinygo GoLLVM

GoLLVM背景

- Tinygo
  - 嵌入式系统
    - 功能不完善或者不支持
      - maps/cgo/reflect/GC/recover…
  - GoLLVM
    - 基本上支持了所有的语言特性

第三部分

GoLLVM现状

GoLLVM现状

- 社区现状
  - 随着Go版本的发布和LLVM版本的更新一直在维持更新
    - 不包括重大的改动：比如泛型
  - 支持绝大部分常见的语言特性
    - channel
    - defer+recover
    - closure
    - interface

GoLLVM现状

- 问题：功能
  - asm 不支持
  - plugin 不支持
  - pprof 的支持不够完善
  - vaargs、go://embedded等不支持
- 问题： 性能
  - 和go compiler性能相差较远

GoLLVM现状

```
name                      old time/op    new time/op    delta
BinaryTree17-16              1.44s ± 2%     2.33s ± 1%   +62.22%  (p=0.029 n=4
Fannkuch11-16                1.61s ± 2%     1.35s ± 2%   -15.94%  (p=0.029 n=4
FmtFprintfEmpty-16           18.2ns ± 1%    24.6ns ± 1%  +34.99%  (p=0.029 n=4
FmtFprintfString-16          33.2ns ± 1%    43.2ns ± 1%  +29.98%  (p=0.029 n=4
FmtFprintfInt-16             37.4ns ± 1%    46.7ns ± 1%  +24.85%  (p=0.029 n=4
FmtFprintfIntInt-16          59.8ns ± 1%    69.2ns ± 0%  +15.67%  (p=0.029 n=4
FmtFprintfPrefixedInt-16     68.4ns ± 0%    75.3ns ± 2%  +10.02%  (p=0.029 n=4
FmtFprintfFloat-16           99.0ns ± 1%   133.3ns ± 1%  +34.64%  (p=0.029 n=4
FmtManyArgs-16                261ns ± 0%     277ns ± 1%   +6.03%  (p=0.029 n=4
GobDecode-16                 2.65ms ± 0%    4.59ms ± 0%  +73.14%  (p=0.029 n=4
GobEncode-16                 1.79ms ± 0%    3.51ms ± 0%  +95.88%  (p=0.029 n=4
Gzip-16                       120ms ± 2%     294ms ± 2%  +144.45%  (p=0.029 n=4
Gunzip-16                    16.7ms ± 0%    25.0ms ± 1%  +50.01%  (p=0.029 n=4
HTTPClientServer-16          33.1µs ± 1%    38.2µs ± 2%  +15.17%  (p=0.029 n=4
JSONEncode-16                4.43ms ± 2%    6.45ms ± 2%  +45.83%  (p=0.029 n=4
```

GoLLVM现状

- 问题分析
  - 不支持汇编
    - 基本上所有的微服务，都直接或者间接依赖plan9汇编
  - 前端来源于gccgo
    - 机制依赖于libdwarf/libunwind, 与runtime的兼容性不好
    - writebarrier/boundcheck的过早引入，影响了后续优化
  - 部分数据结构的兼容性
    - iface

GoLLVM现状

- GC: 性能差距
  - 空载: GoLLVM的时间是go的3倍左右
  - GC的性能差距
    - 保守式的栈扫描
    - runtime的生成代码
      - heapBits.next: 手动inline后，mallocgc提升10%

第四部分

解决办法

GoLLVM问题解决

- ASM的支持
  - 汇编指令转换
    - Plan9 -> gnu asm
  - ABI 转换：
    - Plan9: no callee save, stack-based
    - GoLLVM 基于C的calling convention
    - save all called-save regs
    - GC的支持

GoLLVM问题解决

- 兼容性问题
  - reflect.typelinks:
    - GoLLVM 没有moduledata相关的概念
  - 结构体类型
    - iface：第一个域指向itab vs 指向itab的方法
  - mangle机制
- 解决方法
  - 修改三方库，把常见的三方库自己维护一套gollvm的实现

GoLLVM问题解决

- 性能瓶颈
  - writebarrier/boundary check的过早引入，影响了优化
    - 在llvm侧增加优化pass，插入write barrier和boundary check
  - 支持GC精确式栈扫描
    - gollvm 内部有精确式栈扫描的代码，但是有bug
    - 编译时生成stackmap
      - spill register的开销很大

第五部分

阶段性成果

阶段性成果

Time	GoLLVM	gc
New goroutine	142 ms	150 ms
sliceRead	71 ns/op	116 ns/op
Reflect	288 ms	310 ms
GC	3.8s	2.89s
Defer	6s	2.53s

阶段性成果

- 微服务灰度测试
  - Latency： 10ms —> 30ms
  - CPU: 略有降低，效果不太明显

第六部分

未来与展望

未来与展望

- 任重而道远
  - gofrontend太旧
  - go SSA IR —> LLVM IR —> go obj

“

Q&A

谢谢！