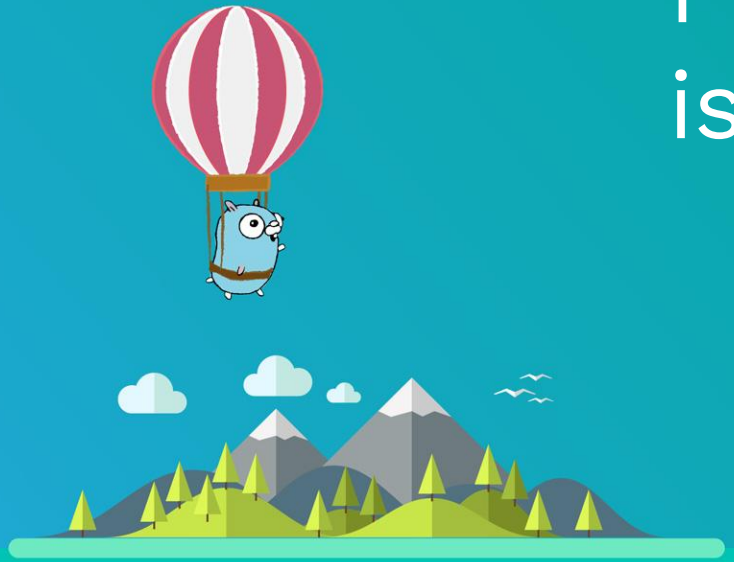
Shenzhen China, May 27 2018

Go汇编优化入门

蒙卓

hi@mzh.io



“Premature optimization is the root of all evil

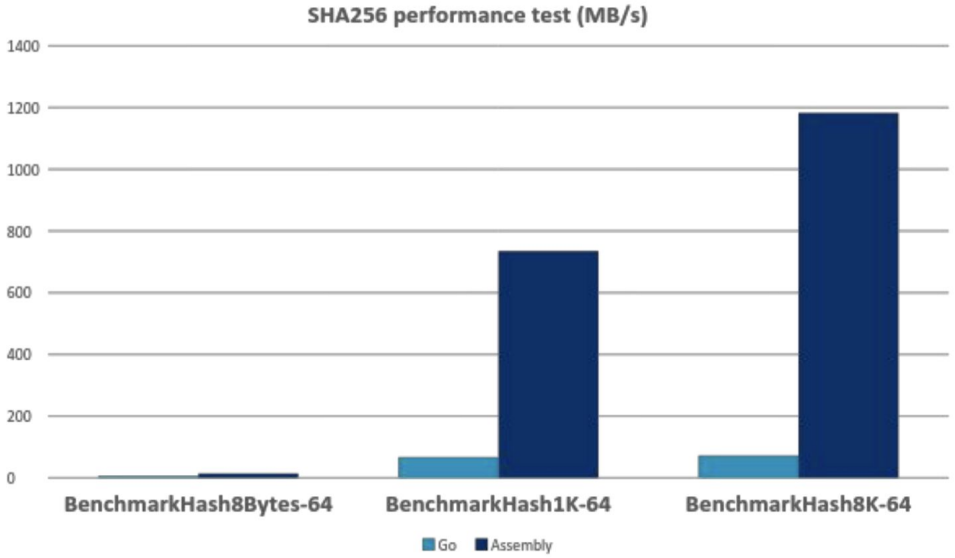
Donald Ervin Knuth ”

个人优化经历(arm64平台)

- AES hashmap ~ 6x
- Linux vdso syscall ~ 3x
- Md5 ~ 2x
- Chacha20 ~ 3x
- Duffcopy ~ 1x

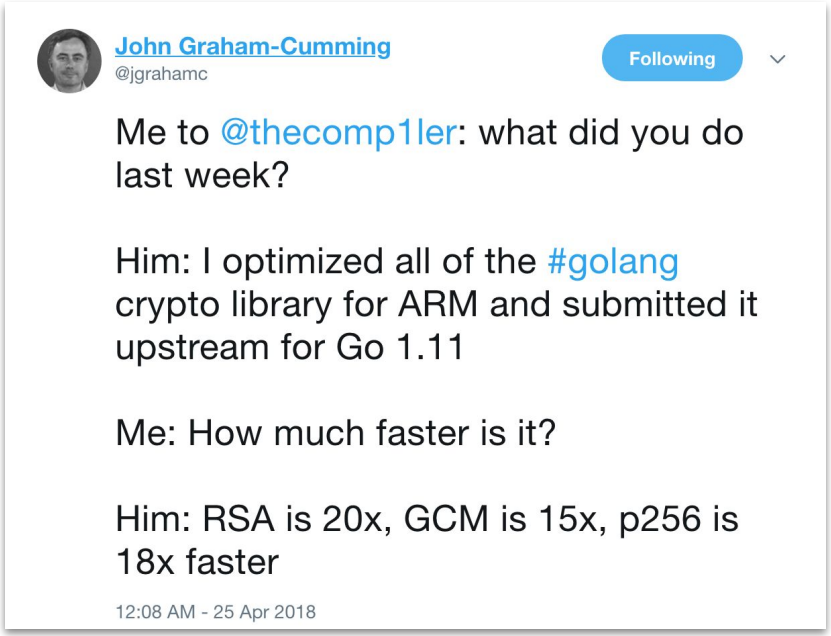
Go1.11

大牛们的优化(arm64平台)



肖玮 sha256优化

大牛们的优化（arm64平台）



Vlad Krasnov RSA优化

目录

- 基础知识
- 汇编语法
- Demo
  - 基本程序
  - debug

如何让程序跑得更快？

- 减少读写
- 并行操作
- 硬件加速

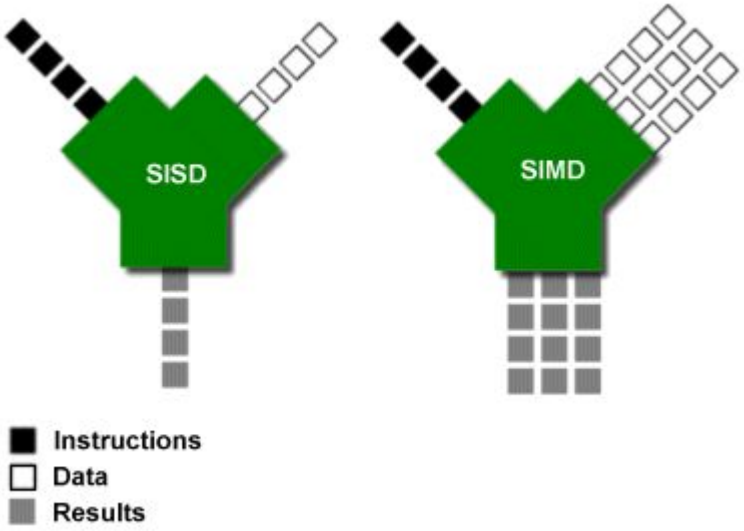
1.1 减少读写

- 层 = 10x性能下降
- 少用内存
- 对齐地址

```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us
Disk seek                           10,000,000   ns   10,000 us
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us
```

1.2 并行操作

- 同样的时间
- 更多的数据



1.3 硬件加速

- 算法再好 < 10x
- 硬件指令 > 16x
- 简单粗暴

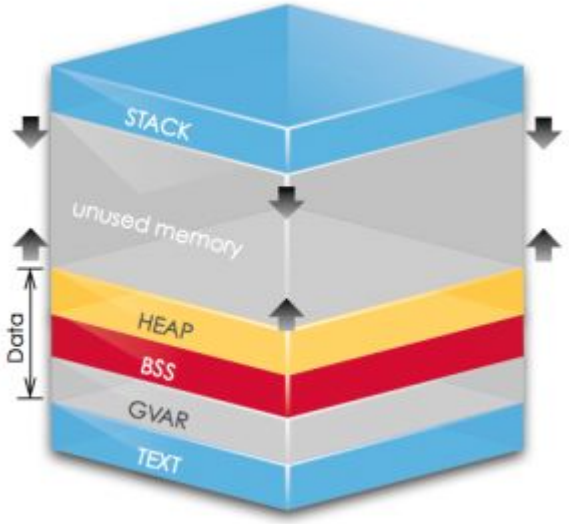
1.4 目标小结

- 减少读写
- 并行化
- 硬件加速

一起上，性能杠杠滴

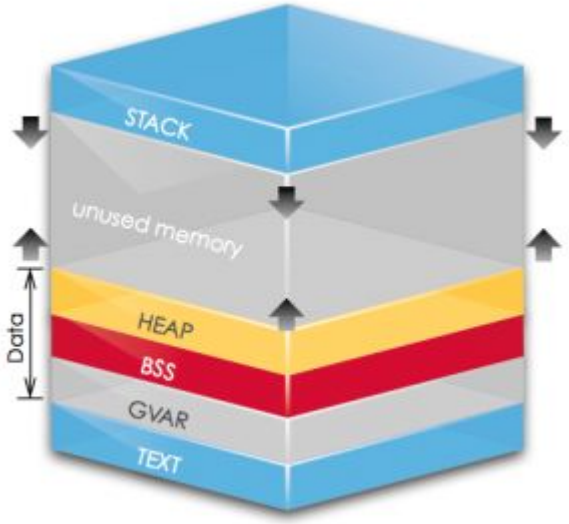
1.5 程序内存分布

- 构造与其他程序一致
- TEXT = 可执行代码
- DATA = 堆+全局变量
- frame = 函数参数+临时数据



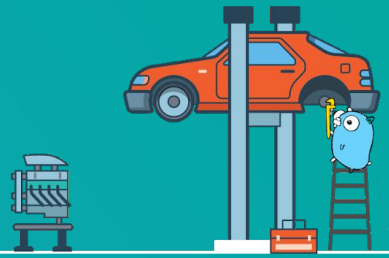
1.5 程序内存分布

- 构造与其他程序一致
- TEXT = 可执行代码
- DATA = 堆+全局变量
- frame = 函数参数+临时数据
- stack = Go 调度器/信号处理

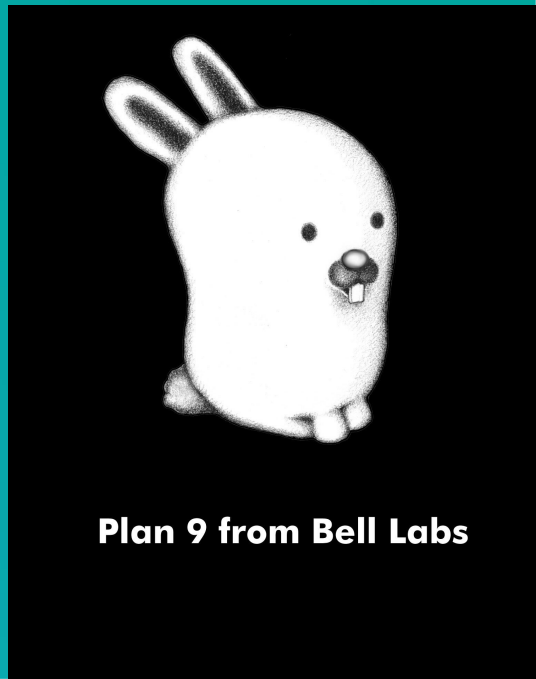


“

Don’t panic



2 汇编语法



2. Go汇编语法特点

- 准抽象汇编语言
- AT & T 风格(左到右)
- 指令 参数xN 目标 (N = 0 ... 3)

2.1 汇编语法例子

将 add函数 转化成 汇编写法

```
func add(a, b int64) (c int64) {
    c = a + b
    return c
}
```

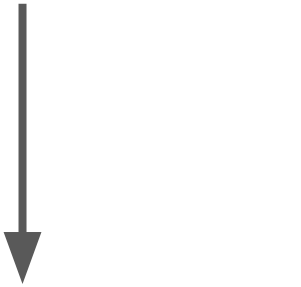
2.1 汇编语法例子

```
func add(a,b int64) (c int64)
```

```
TEXT ·add(SB)
```

2.1 汇编语法例子

```
func add(a,b int64) (c int64)
```



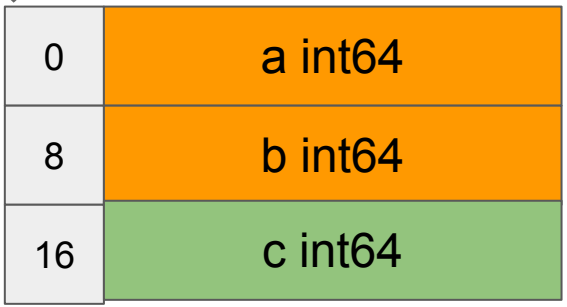
```
TEXT ·add(SB)
```

告诉汇编器
这是基于静态地址
的数据
（static base）

告诉汇编器
这数据该放TEXT区

2.1 汇编语法例子

```
func add(a,b int64) (c int64)
```



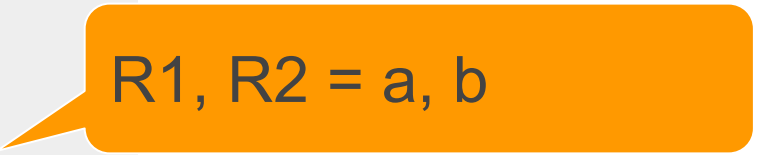

```
TEXT •add(SB), $24
```

3 x 8 = 24

栈帧长24字节

2.2 例子代码讲解

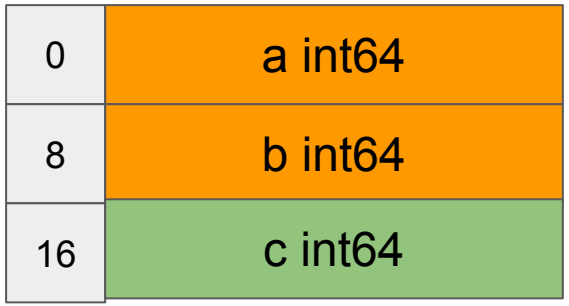
```
TEXT ·add(SB),$24
        MOVD    a+0(FP), R1
        MOVD    b+8(FP), R2
        ADD     R1, R2, R3
        MOVD    R3, c+16(FP)
        RET
```



2.2 例子代码讲解

```
TEXT ·add(SB),$24
        MOVD    a+0(FP), R1
        MOVD    b+8(FP), R2
        ADD     R1, R2, R3
        MOVD    R3, c+16(FP)
        RET
```

FP (Frame Pointer)
栈帧指针
指向栈帧最低位



2.2 例子代码讲解

```
TEXT ·add(SB),$24
        MOVD    a+0(FP), R1
        MOVD    b+8(FP), R2
        ADD     R1, R2, R3
        MOVD    R3, c+16(FP)
        RET
```

R3 = a + b

2.2 例子代码讲解

```
TEXT ·add(SB),$24
        MOVD    a+0(FP), R1
        MOVD    b+8(FP), R2
        ADD     R1, R2, R3
        MOVD    R3, c+16(FP)
        RET
```

c = tmp

2.2 例子代码讲解

```
TEXT ·add(SB),$24
        MOVD    a+0(FP), R1
        MOVD    b+8(FP), R2
        ADD     R1, R2, R3
        MOVD    R3, c+16(FP)
        RET
```

return 的简写

2.2 例子代码讲解

```
TEXT ·add(SB),$24
        MOVD    a+0(FP), R1
        MOVD    b+8(FP), R2
        ADD     R1, R2, R3
        MOVD    R3, c+16(FP)
        RET
```

很简单对吧？

2.2 例子代码讲解

```
TEXT ·add(SB),$24
        MOVD    a+0(FP), R1
        MOVD    b+8(FP), R2
        ADD     R1, R2, R3
        MOVD    R3, c+16(FP)
        RET
```

别用汇编写复杂语句

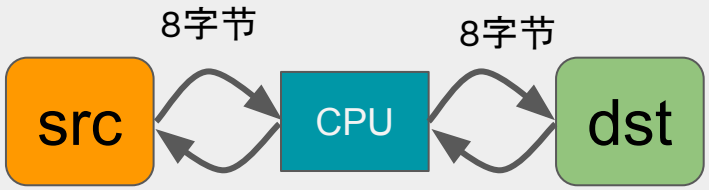
c = a + b

2.3 汇编优化目标

- 减少读写
- 并行操作
- 硬件加速

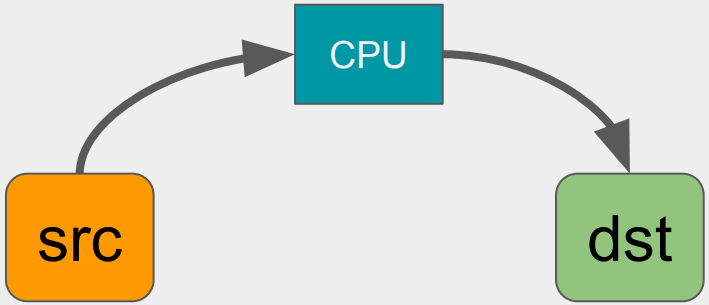
2.3 减少读写

```
func memmove(dst, src []byte)
```



2.3 减少读写

```
func memmove(dst, src []byte)
```



- 塞满寄存器
- 占满流水线
- 处理块数据

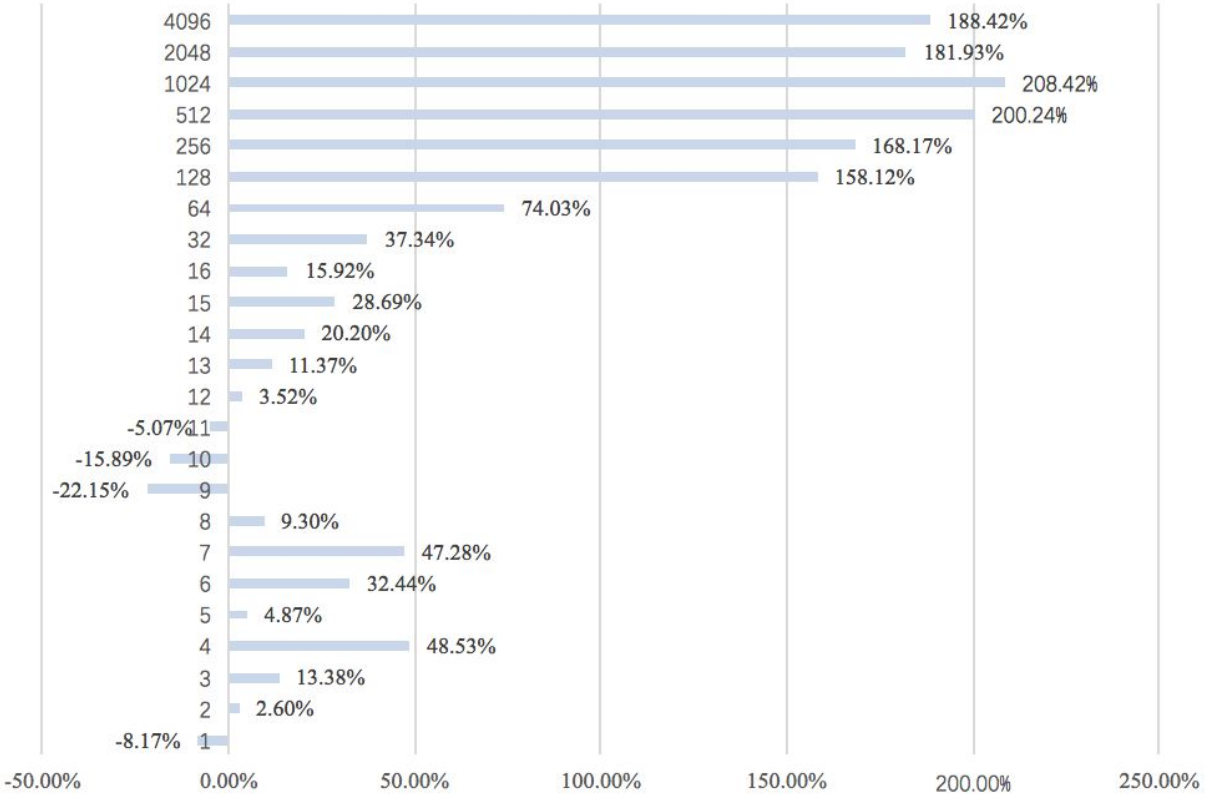
2.3 减少读写

```
TEXT ·memmove(SB),$48
    // 初始化 src, dst, length

    LDP (src), (R0, R1)
    LDP 16(src), (R2,R3)

    // R4……R28

    STP (R0,R1), (dst)
    STP (R2,R3), 16(dst)
```

- 塞满寄存器
- 占满流水线
- 处理块数据

LDP
(Load double word pair)
一次加载两个寄存器指令

2.3 减少读写效果


4096 188.42%
2048 181.93%
1024 208.42%
512 200.24%
256 168.17%
128 158.12%
64 74.03%
32 37.34%
16 15.92%
15 28.69%
14 20.20%
13 11.37%
12 3.52%
11 -5.07%
10 -15.89%
9 -22.15%
8 9.30%
7 47.28%
6 32.44%
5 4.87%
4 48.53%
3 13.38%
2 2.60%
1 -8.17%
-50.00%
0.00%
50.00%
100.00%
150.00%
200.00%
250.00%

2.4 并行操作

```
func vadd(dst, src []uint8) {
    for i, s := range src {
        dst[i] += s
    }
}
```

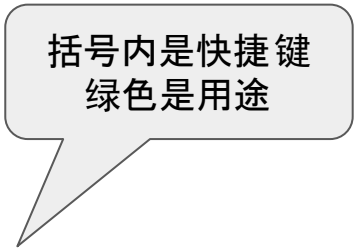
Demo

2.4 并行操作小结

- build tag区分OS/平台(x_linux_arm64.s)
- 测试/benchmark很重要

2.5 如何debug

- gdb
- Run(r) 运行程序
- Break(b) 文件名:行号 断点
- Next(n) 下一行
- Info Register (i r) 寄存器名 查看寄存器内容
- eXamine (x) 地址/寄存器... 查看内存数据



Q&A

参考资料

- Go ARM64高性能优化 肖玮
- 每个程序员都应该知道的延迟 Jeff Dean
- plan9 assembly 完全解析 Xargins
- Golang build Doc
- demo 在线录像
- demo 源代码