



# 编译系统针对RISC-V架构的 CodeSize优化

报告人

王锋研究员

2023年9月8日

# 动机

- 随着物联网的日益普及，物联网设备的功能越来越多，对嵌入式设备能储存的代码量要求越来越高。
- 提高代码密度可以装下更多功能，且间接优化了功耗，性能。
- 在嵌入式领域，ARM Thumb2 比 RISC-V 指令集的代码密度更优。
- 很大原因是因为编译器不能很好地利用 RISC-V 压缩指令集。

# CONTENTS

## 目录



**01.** RISC-V指令集架构介绍

---

**02.** 寄存器分配优化

---

**03.** 过程抽象

---

**04.** 编译、汇编、链接器协同优化

---

**05.** 总结

---



# 01

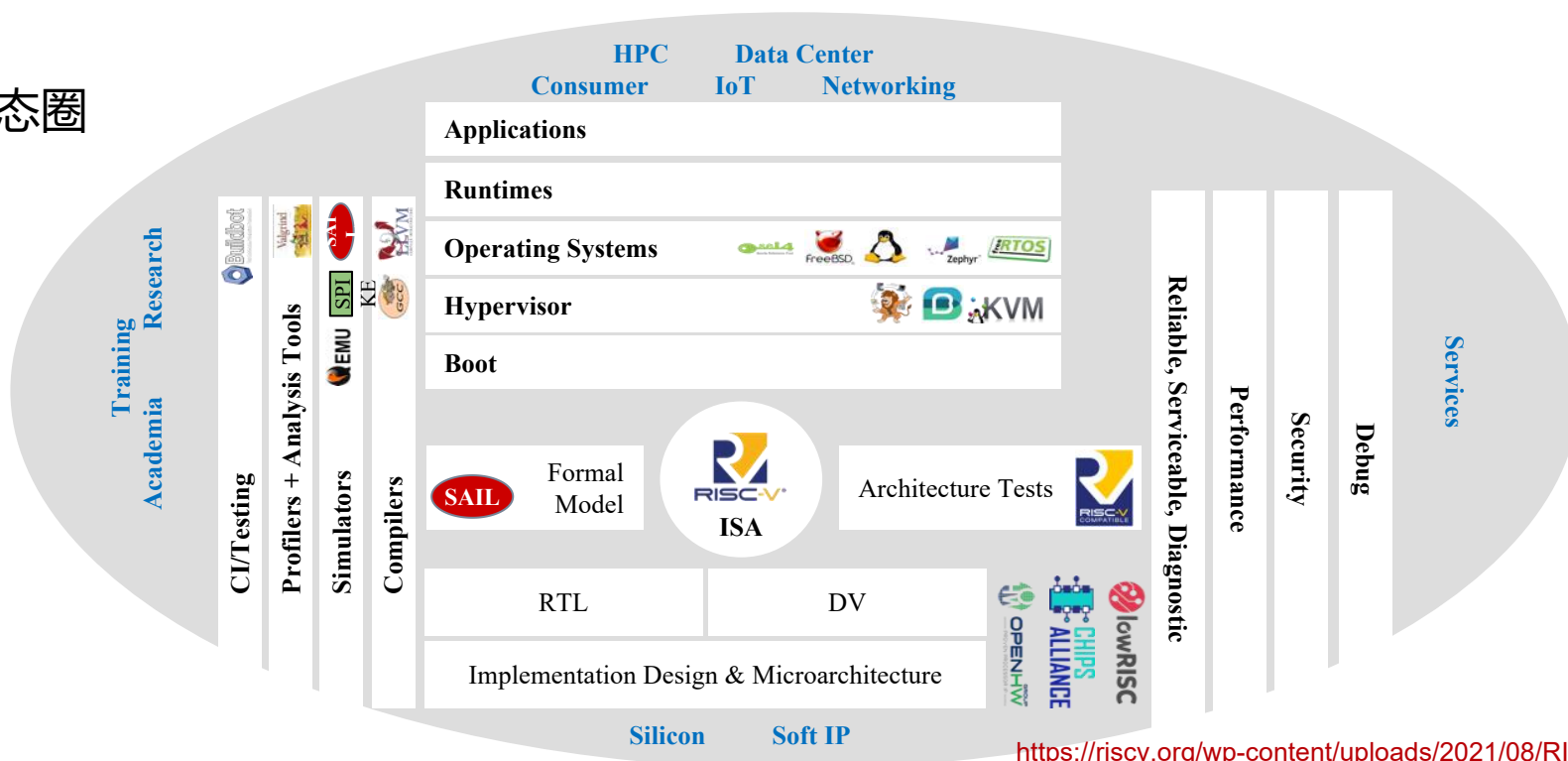
## RISC-V 指令集架构介绍

---

# RISC-V指令集架构介绍

- RISC-V: (RISC five) , 是一种开源的指令集架构
- RISC-V, 是一种精简指令集 ( RISC, Reduced Instruction Set Computer) 架构
- RISC-V, 是一种模块化的指令集架构

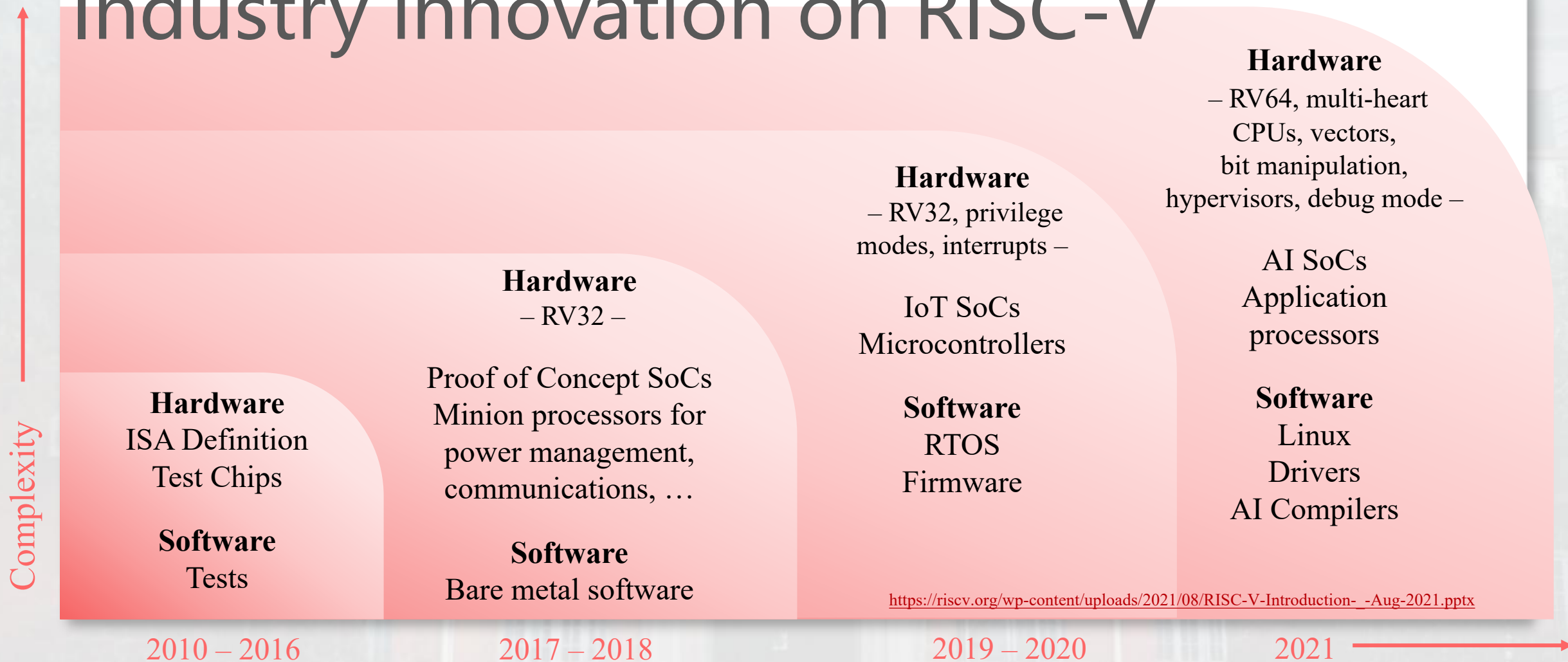
## RISCV生态圈



[https://riscv.org/wp-content/uploads/2021/08/RISC-V-Introduction\\_-\\_Aug-2021.pptx](https://riscv.org/wp-content/uploads/2021/08/RISC-V-Introduction_-_Aug-2021.pptx)

# RISC-V指令集架构介绍

## Industry innovation on RISC-V



# RISC-V指令集架构介绍

- RISC-V指令集架构的核心是RV32I的基础ISA，运行一个完整的软件栈
- RISC-V指令集架构的模块化来源于可选的标准扩展，根据应用程序的需要，硬件可以设计是否支持这些可选的扩展，如RV32M乘法扩展、RV32F单精度扩展、RV32D双精度扩展等。

| 基础指令集  | 描述   |
|--------|--|
| RV32I  | 32位地址空间与整数指令，支持32个通用整数寄存器                          |
| RV32E  | RV32I的子集，仅支持16个通用整数寄存器                             |
| RV64I  | 64位地址空间与整数指令及一部分32位的整数指令                           |
| RV128I | 128位地址空间与整数指令及一部分64位和32位的指令                        |
| 扩展指令集  | 描述   |
| M      | 整数乘法与除法指令  |
| A      | 存储原子（Atomic）操作指令和Load-Reserved/Store-Conditional指令 |
| F      | 单精度（32-bit）浮点指令                                    |
| D      | 双精度（64-bit）浮点指令，必须支持F扩展指令                          |

# RISC-V指令集架构介绍

| 扩展指令集 | 描述            |
|-------|---------------|
| C     | 压缩指令，指令长度为16位 |

RV32C 压缩指令集：

- 每条短指令必须和一条标准的 32 位 RISC-V 指令一一对应；
- 压缩指令的操作数是访问频率最高的十个常用寄存器(a0-a5, s0-s1, sp以及ra);
- 压缩指令的目的操作数和源操作数是同一个寄存器；
- 压缩指令的立即数的位数很小，大部分指令只有6位立即数。
- RV32C压缩指令集，让RISC-V程序显著缩短了其二进制代码长度。



# RISC-V指令集-Zc\*扩展指令级v1.0.0-RC5.7

| 扩展指令集 | 描述  |
|-------|---|
| Zca   | C扩展去掉浮点相关指令   |
| Zcf   | 单精度浮点load/store压缩指令: c.flw, c.flwsp, c.fsw, c.fswsp; 仅RV32适用  |
| Zcd   | 双精度浮点load/store压缩指令: c.fld, c.fldsp, c.fsd, c.fsdsp   |
| Zcb   | 扩展16位编码简单指令: c.lbu, c.lh, c.lhu, c.sb, c.sh, c.zext.b, c.sext.b, c.zext.h, c.sext.h, c.zext.w, c.mul, c.not |
| Zcmp  | 扩展的指令完成一系列32位RV指令: cm.push, cm.pop, cm.popret, cm.popretz, cm.mva01s, cm.mvsa01                             |
| Zcmt  | 扩展了跳转表指令: cm.jt, cm.jalt  |

- ❑ Zcf、Zcd、Zcb、Zcmp、Zcmt依赖Zca
- ❑ Zcb依赖其他扩展Zbb: basic bit manipulation.
- ❑ Zcmp与Zcd冲突
- ❑ Zcmt与Zcd冲突, 还依赖Zicsr

<https://github.com/riscv/riscv-code-size-reduction/tree/main/Zc-specification>



# 01

## 寄存器分配优化

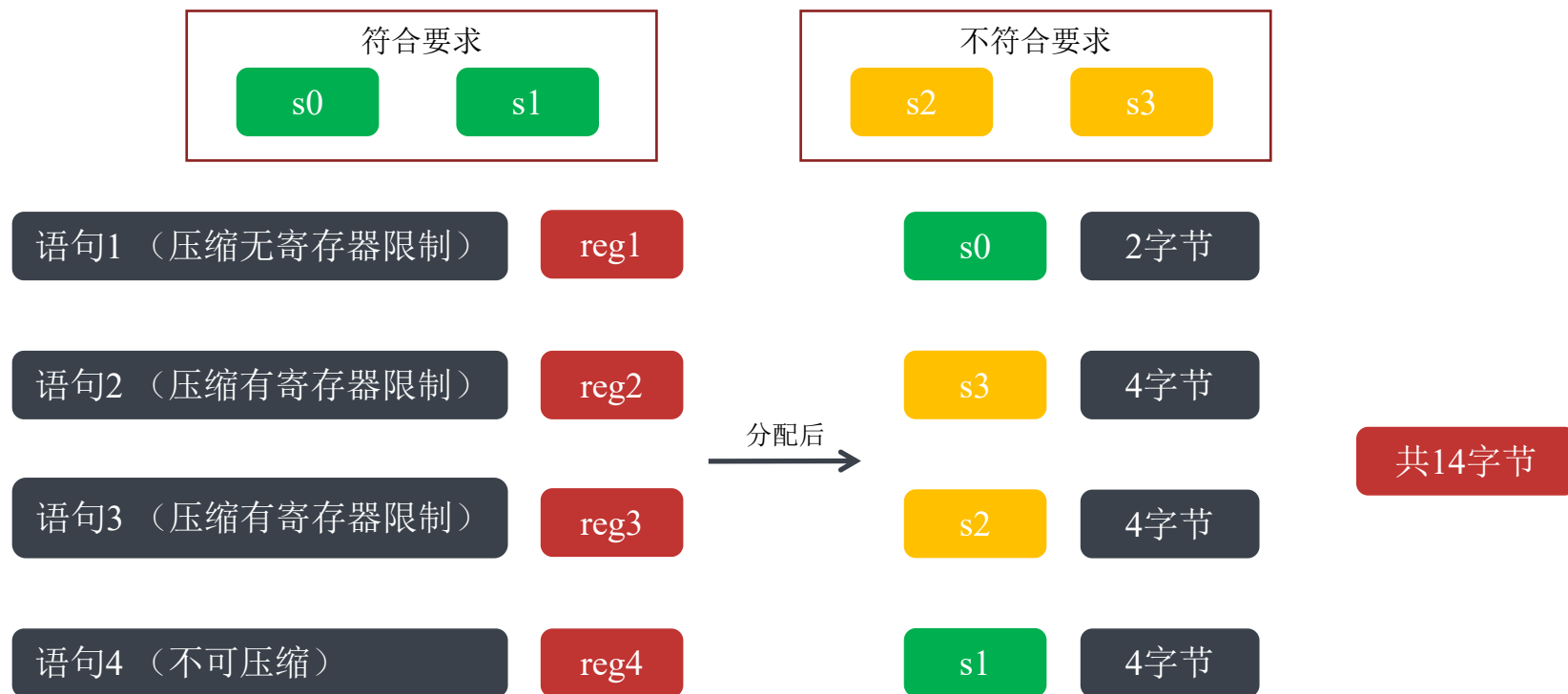
# 寄存器分配优化

RISC-V的汇编指令  
有部分指令的压缩  
指令对分配的硬件  
寄存器有要求。

| 指令类型 (典型例子)         | 压缩要求                              |
|---------------------|-----------------------------------|
| c.addi4spn rd, imm  | rd 需为常用寄存器                        |
| c.lw rd, rs1, imm   | rd 与 rs1 均需为常用寄存器                 |
| c.andi rd, rs1, imm | rs1 需与 rd 相同, 且为常用寄存器             |
| c.xor rd, rs1, rs2  | rs1 需与 rd 相同, 且 rs1 与 rs2 均为常用寄存器 |

# 寄存器分配优化：压缩指令优先策略

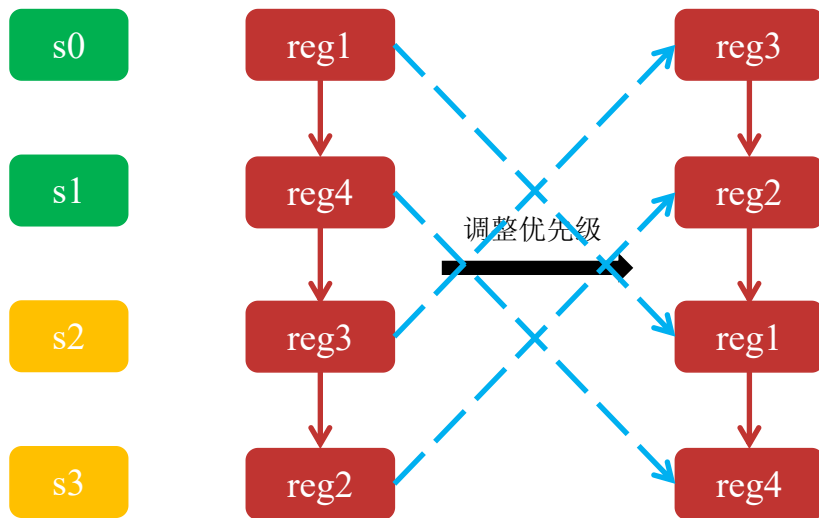
RISC-V的汇编指令有部分指令的压缩指令对分配的硬件寄存器有要求。



# 寄存器分配优化：压缩指令优先策略

优化算法

分配优先级



分配结果

|               |      |    |     |
|---------------|------|----|-----|
| 语句1（压缩无寄存器限制） | reg1 | s2 | 2字节 |
| 语句2（压缩有寄存器限制） | reg2 | s0 | 2字节 |
| 语句3（压缩有寄存器限制） | reg3 | s1 | 2字节 |
| 语句4（不可压缩）     | reg4 | s3 | 4字节 |
| 共10字节         |      |    |     |

# 寄存器分配优化：压缩指令优先策略

## 实例

```
00000000 <func>:
0: 8058      push {ra,s0-s3},-32
2: 4500      lw s0,8(a0)
4: 10060493  addi s1,a2,256
8: 892a      mv s2,a0
a: 89b2      mv s3,a2
c: 4581      li a1,0
e: 8626      mv a2,s1
10: 8522     mv a0,s0
12: 00000097 auipc ra,0x0
16: 000080e7 jalr ra # 12 <func+0x12>
1a: c119     beqz a0,20 <.L2>
1c: 408504b3  sub s1,a0,s0

00000020 <.L2>:
20: 8526     mv a0,s1
22: 0099f363 bgeu s3,s1,28 <.L3>
26: 854e     mv a0,s3

00000028 <.L3>:
28: 00a407b3 add a5,s0,a0
2c: 9426     add s0,s0,s1
2e: 00f92023 sw a5,0(s2)
32: 00892223 sw s0,4(s2)
36: 00892423 sw s0,8(s2)
3a: 8054     popret {ra,s0-s3},32
```

恒可压缩

不可压缩

优化后可压缩

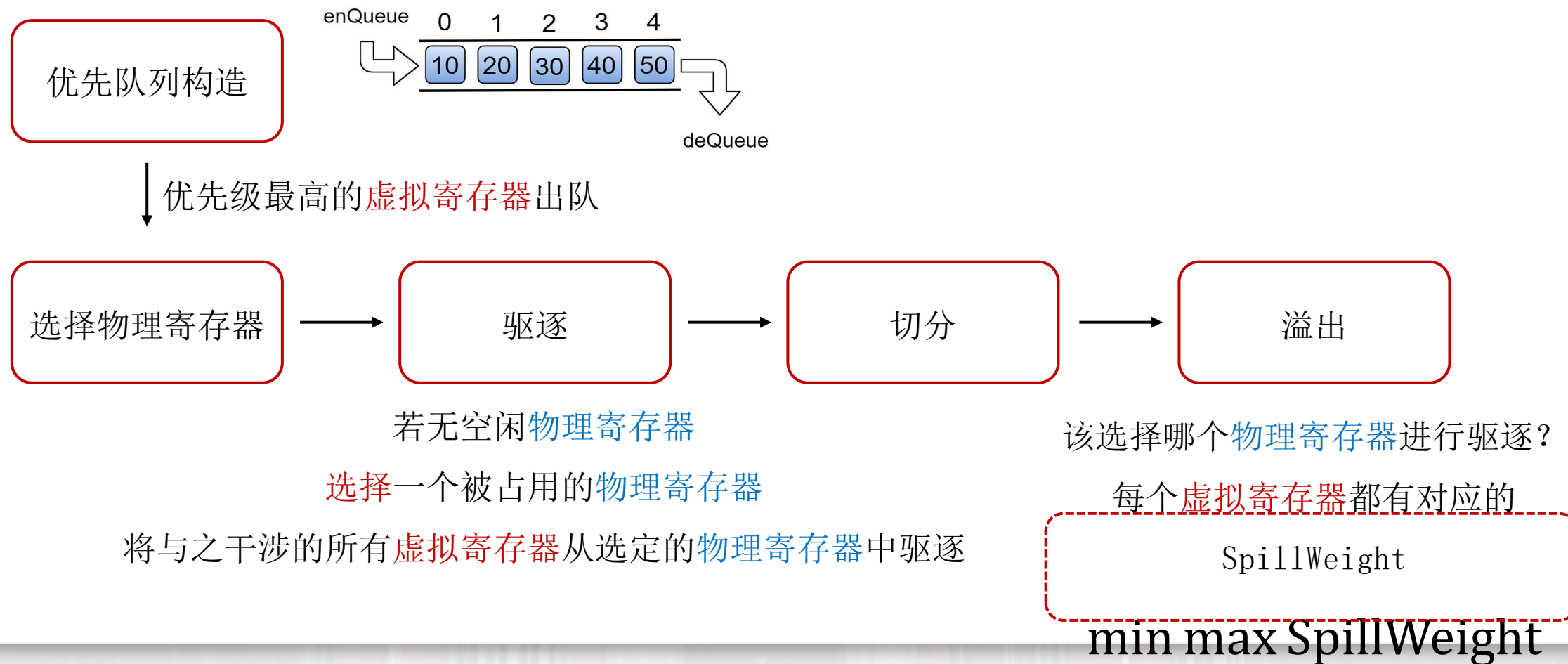
```
00000000 <func>:
0: 8058      push {ra,s0-s3},-32
2: 4500      lw s0,8(a0)
4: 10060913  addi s2,a2,256
8: 84aa      mv s1,a0
a: 89b2      mv s3,a2
c: 4581      li a1,0
e: 864a      mv a2,s2
10: 8522     mv a0,s0
12: 00000097 auipc ra,0x0
16: 000080e7 jalr ra # 12 <func+0x12>
1a: c119     beqz a0,20 <.L2>
1c: 40850933  sub s2,a0,s0

00000020 <.L2>:
20: 854a      mv a0,s2
22: 0129f363 bgeu s3,s2,28 <.L3>
26: 854e     mv a0,s3

00000028 <.L3>:
28: 01240633 add a2,s0,s2
2c: 00a407b3 add a5,s0,a0
30: c09c      sw a5,0(s1)
32: c0d0      sw a2,4(s1)
34: c490      sw a2,8(s1)
36: 8054     popret {ra,s0-s3},32
```

优化6字节

# 寄存器分配优化：压缩指令优先策略





# 寄存器分配优化：压缩指令优先策略

- 驱逐优化目的：不要让压缩优先级低的虚拟寄存器驱逐压缩优先级高的
- 正在为虚拟寄存器  $v$  分配物理寄存器
  - 驱逐：选择一个物理寄存器  $p$ ，把占用它的虚拟寄存器  $v_0 \sim v_n$  都驱逐出该寄存器。如此一来，现在可以把  $v$  分配给  $p$
  - 选择哪个物理寄存器？
  - 优化前：min max SpillWeight
  - 优化后：若  $\text{compression-priority}[v] < \sum_{i=0}^n \text{compression-priority}[v_i]$ ，则不驱逐此物理寄存器。若大于等于，则继续考虑 min max SpillWeight





# 03

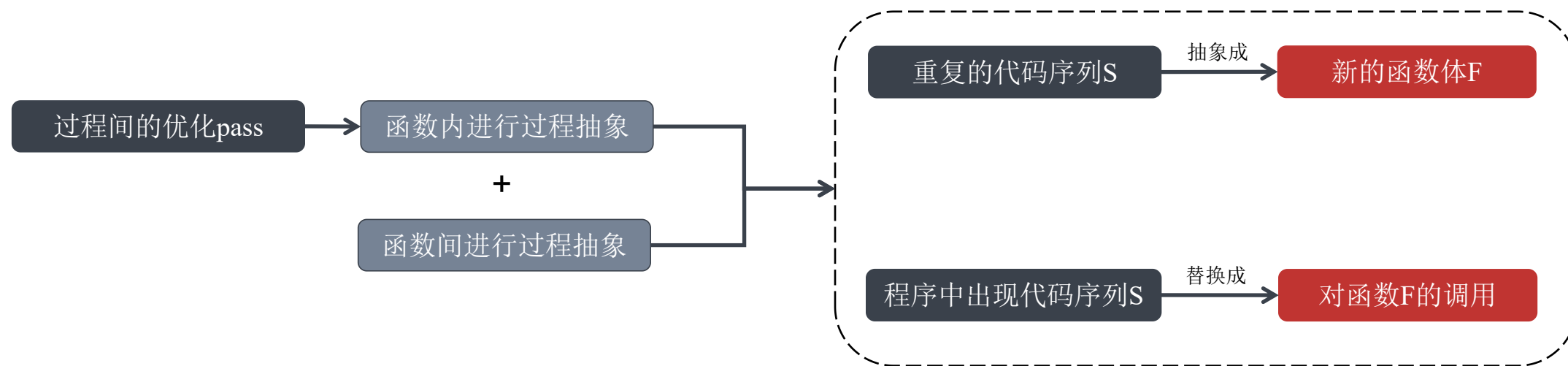
## 过程抽象

---

# 过程抽象优化

## LLVM过程抽象优化

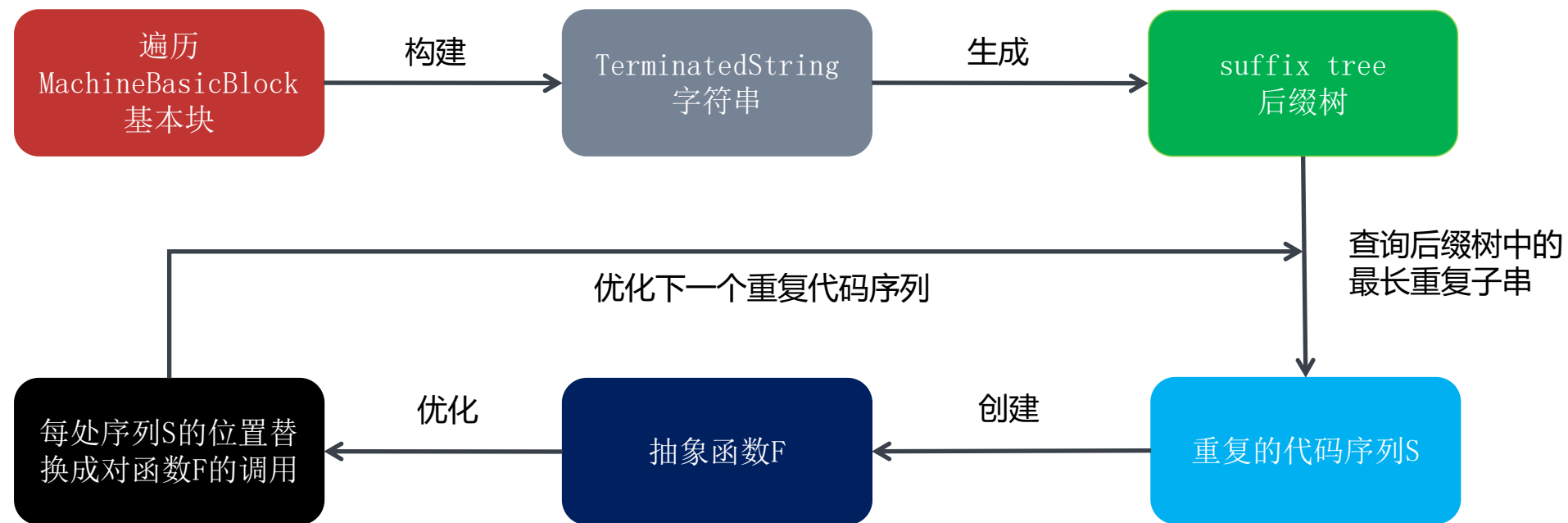
由 google公司Jessica Paquette 在2016 LLVM Developers' Meeting介绍并实现



Jessica Paquette, Apple, <https://llvm.org/devmtg/2016-11/Slides/Paquette-Outliner.pdf>

# 过程抽象优化

## 优化算法



# 过程抽象优化

优化示例-from Jessica Paquette's report(Paquette-Outliner)

## MachineBasicBlock

```
FOO()  
A R1 = 0xDEADBEEF  
B R3 = R2 + R1  
C R1 = *R5  
D R7 = 0xFEEDFACE  
E R1 = R1 - 1  
  BAR()  
F R7 = R3 + R2  
A R1 = 0xDEADBEEF  
B R3 = R2 + R1  
C R1 = *R5  
G R7 = 0xFACEFEED  
A R1 = 0xDEADBEEF  
B R3 = R2 + R1  
C R1 = *R5  
E R1 = R1 - 1
```

A B C D E \$0

F A B C G A B C E \$1

TerminatedString

A B C D E \$0 F A B C G A B C E \$1

生成后缀树

# 过程抽象优化

优化示例-from Jessica Paquette's report(Paquette-Outliner)

## MachineBasicBlock

X D E \$0 F X G X E \$1

```
FOO()
A R1 = 0xDEADBEEF
B R3 = R2 + R1
C R1 = *R5
D R7 = 0xFEEDFACE
E R1 = R1 - 1
  BAR()
F R7 = R3 + R2
A R1 = 0xDEADBEEF
B R3 = R2 + R1
C R1 = *R5
G R7 = 0xFACEFEED
A R1 = 0xDEADBEEF
B R3 = R2 + R1
C R1 = *R5
E R1 = R1 - 1
```

## OUTLINED()

```
A R1 = 0xDEADBEEF
B R3 = R2 + R1
C R1 = *R5
```

```
FOO()
X call OUTLINED()
D R7 = 0xFEEDFACE
E R1 = R1 - 1
  BAR()
F R7 = R3 + R2
X call OUTLINED()
G R7 = 0xFACEFEED
X call OUTLINED()
E R1 = R1 - 1
```



# 04

## 编译、汇编、链接 的协同优化

# 编译、汇编、链接的协同优化

CAL: a Joint Method of Compiler, Assembler, and Linker on Code-size Optimiztion

```
#include<stdio.h>
int main()
{
    printf ("Codesize test: %s, %s, %s\n",
"test1", "test2", "test3");
}
```

|      |                 |
|------|-----------------|
| lui  | a3,%hi(.LC0)    |
| addi | a3,a3,%lo(.LC0) |
| lui  | a2,%hi(.LC1)    |
| addi | a2,a2,%lo(.LC1) |
| lui  | a1,%hi(.LC2)    |
| addi | a1,a1,%lo(.LC2) |
| lui  | a0,%hi(.LC3)    |
| addi | a0,a0,%lo(.LC3) |

|      |                   |
|------|-------------------|
| lui  | a3,%hi(.LC0)      |
| addi | a3,a3,%lo(.LC0)   |
| lui  | a2,%hi(.LC1)      |
| addi | a2,a3,(.LC1-.LC0) |
| lui  | a1,%hi(.LC2)      |
| addi | a1,a2,(.LC2-.LC1) |
| lui  | a0,%hi(.LC3)      |
| addi | a0,a1,(.LC3-.LC2) |

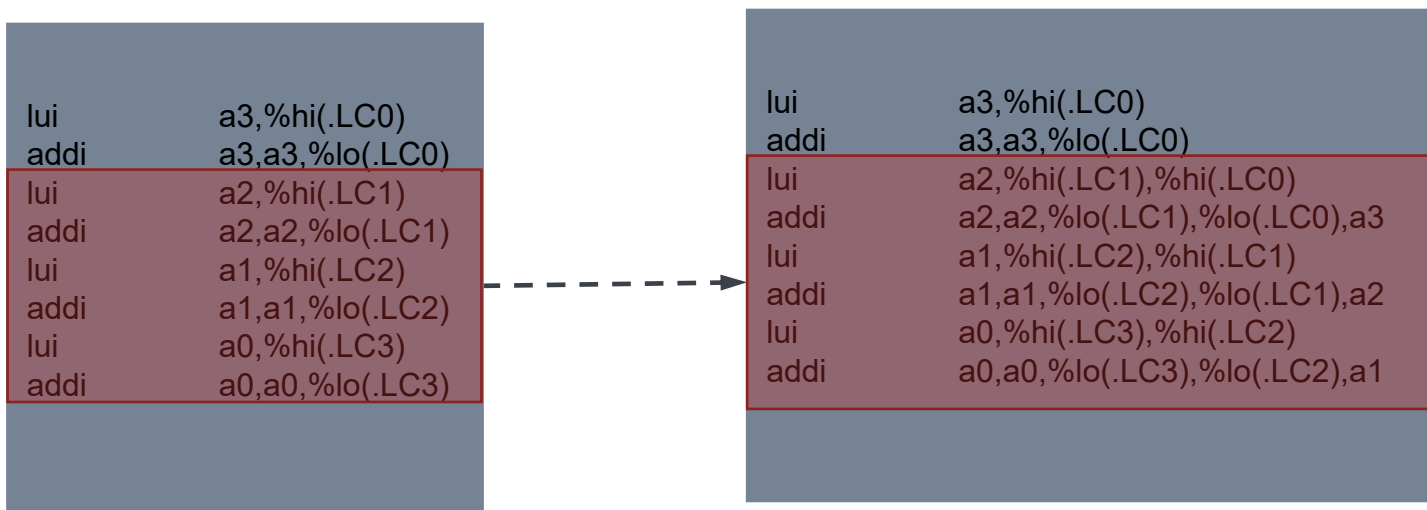
→ .LC1/.LC2/.LC3的地址通过基址+偏移量的方式计算

可将lui删除，每处可优化2/4字节

# 编译、汇编、链接的协同优化

## 编译器

标记可优化的符号加载指令，并生成特定的伪指令



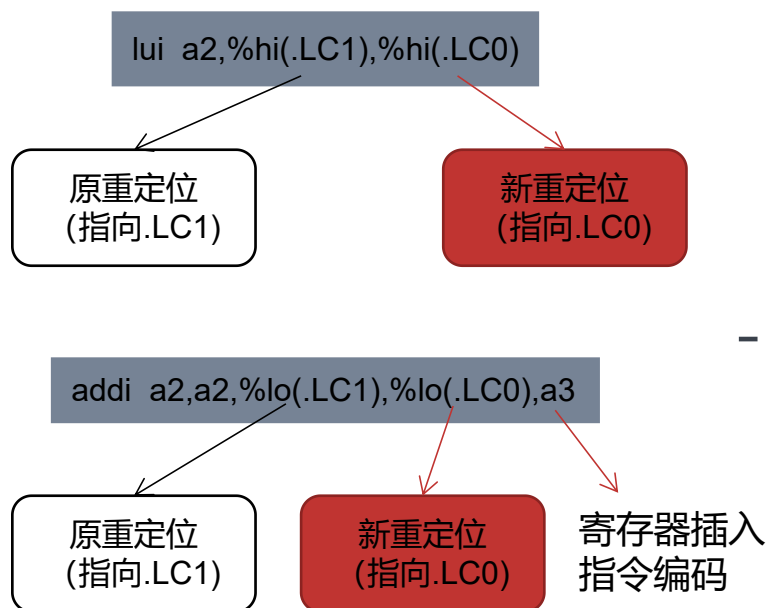
编译器需要通过数据流分析，确定两个符号的加载可以被优化才能打上标记。基址寄存器不能被修改。



# 编译、汇编、链接的协同优化

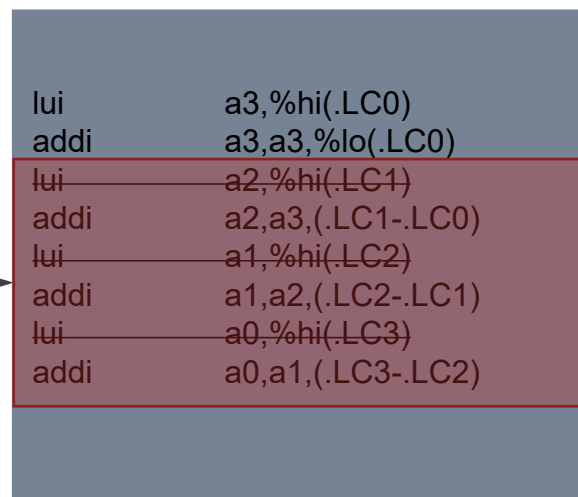
## 汇编器

识别特殊汇编，生成特定重定位



## 链接器

根据特定重定位，判断符号地址偏移，满足优化条件时将lui删除并修改addi



```
lui a3,%hi(.LC0)
addi a3,a3,%lo(.LC0)
lui a2,%hi(.LC1)
addi a2,a3,(.LC1-.LC0)
lui a1,%hi(.LC2)
addi a1,a2,(.LC2-.LC1)
lui a0,%hi(.LC3)
addi a0,a1,(.LC3-.LC2)
```

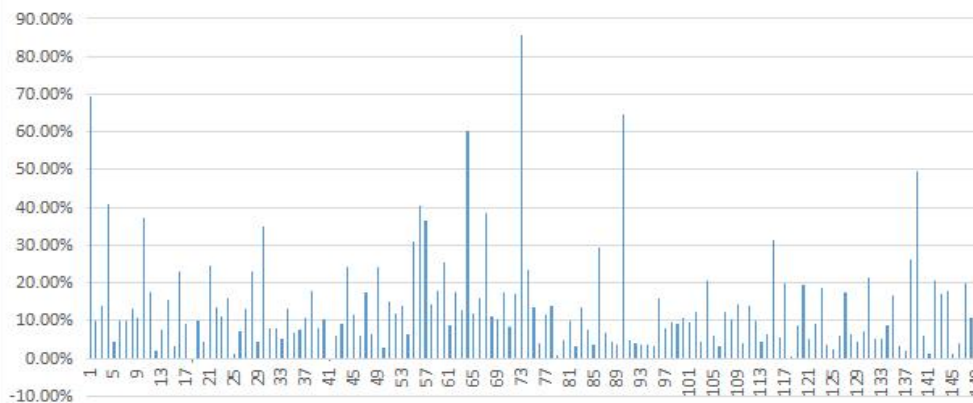
# 优化效果-CSiBE

- 编译、汇编、链接器协同优化
- 过程抽象
- 代码移动
- 浮点常量加载优化
- 内联代价模型优化
- 跳转指令优化
- 寄存器分配优化
- ...



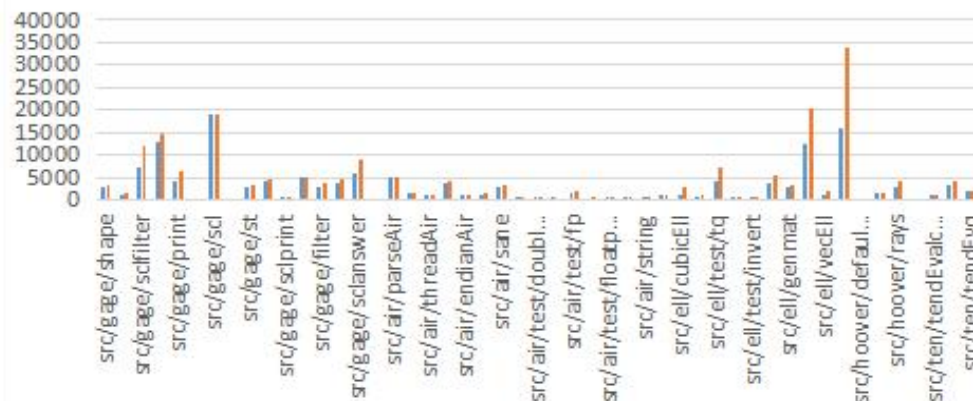
CSiBE

codesize优化百分比

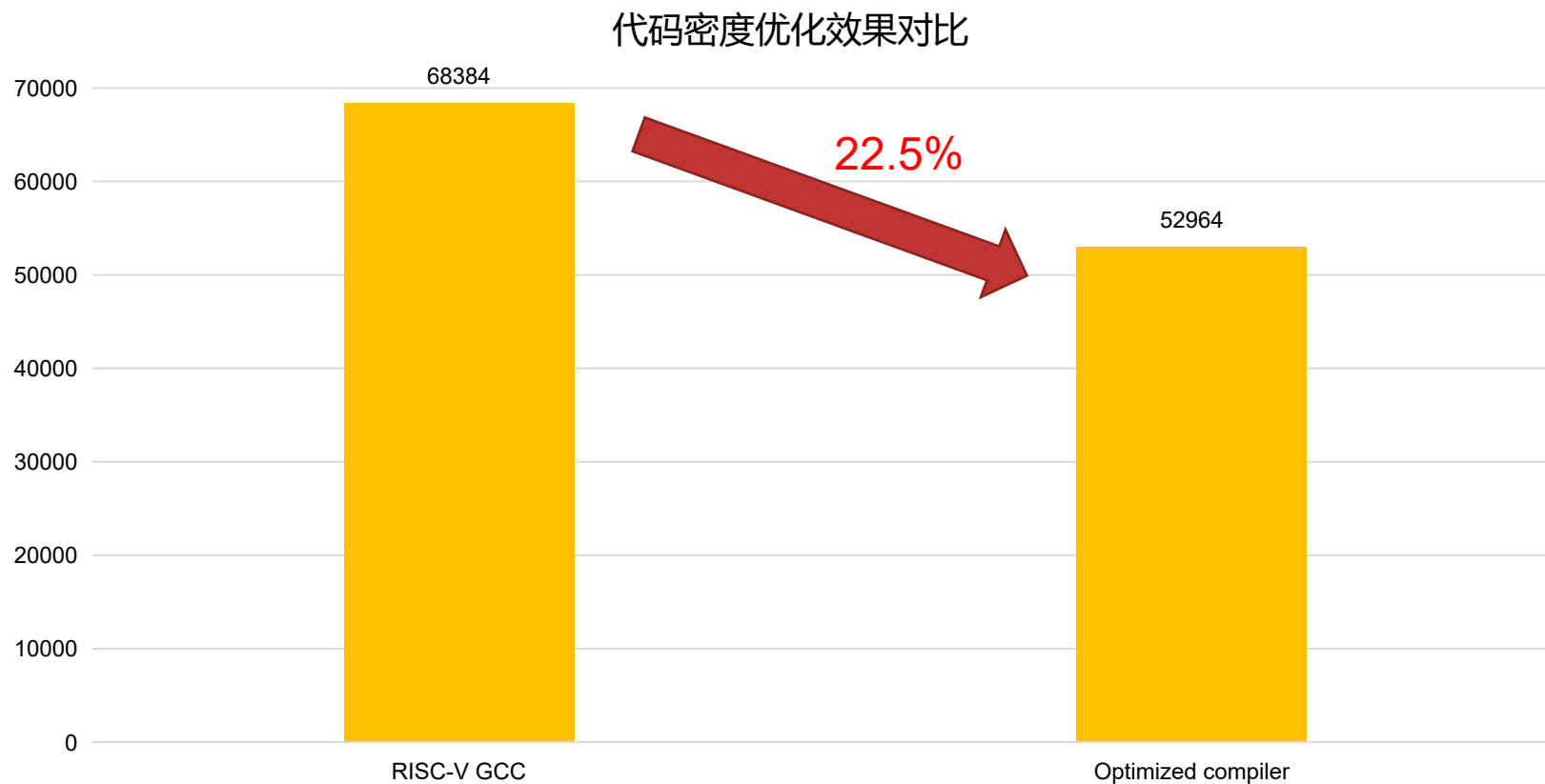


CSiBE

codesize优化大小对比



# 优化效果-鸿蒙LiteOS 5.0.0





# 05

## 总结

---

# 总结

- RISC-V架构发展迅速，对整个底层基础软件需求迫切
- 边缘计算、嵌入式领域最先替代ARM、X86
- 编译器针对代码密度有较大优化空间
- 进一步提高能效需要软硬件协同设计
- 机器学习方法成为编译器优化的重要趋势
  - 编译器内部有大量启发式算法和代价模型

# THANKS FOR ALL

王锋 wangfeng@hnu.edu.cn

