代码尺寸优化

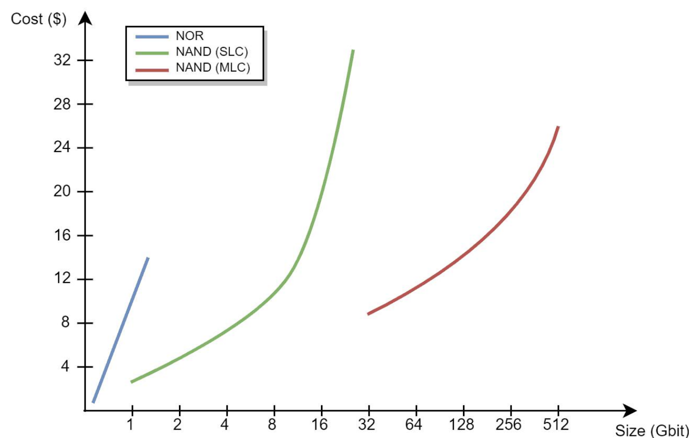
李清安

Outline

- 代码尺寸优化概述
- 相关工作介绍
  - 算法与编程（人工）
  - 删除无用代码
  - 精简重复代码
  - 压缩

代码尺寸优化的意义

- 存储资源成本敏感的场景
  - 存储资源很少，或者对存储设备的成本比较敏感
  - 嵌入式设备、移动设备等



- 存储资源无法扩容的场景
  - 比如，野外部署的嵌入式设备，存储使用率接近饱和
  - 但是，软件代码需要维护和更新
- 此外，也影响带宽成本、内存占用
  - 热门的移动App，通常都是100 MB级别的带宽消耗

代码尺寸的挑战

- 技术选型、算法设计等
  - 技术选型（依赖的第三方软件生态）
  - 算法和数据结构
- 无用代码
  - 软件中存在永远不需要的代码片段
    - 原因分析：第三方代码的通用性 vs 应用场景的特殊性
    - 优化思路：识别并删除不需要的代码
- 重复代码
  - 软件中存在重复或者相似的代码片段
    - 原因分析：多人开发、重复序列难以重构、编译器的模版化翻译等，信息熵
    - 优化思路：识别并精简重复代码

Outline

- 代码尺寸优化概述
- 相关工作介绍
  - 算法与编程（人工经验）
  - 删除无用代码
  - 精简重复代码
  - 压缩

算法与编程（人工经验）

- 有些算法需要更复杂的实现
  - 导致更大的code size

Commonly used	Cheaper alternative
Quick sort	Bubble sort
Binary search	Linear search

```
// clang++ -Oz test.cpp -o test.o, 13976 bytes
#include<map>
int main() {
  std::map<int, int> m;
  m[10] = 100;
  return m[0];
}
```

```
// clang++ -Oz test.cpp -o test.o, 15140 bytes
#include<unordered_map>
int main() {
  std::unordered_map<int, int> m;
  m[10] = 100;
  return m[0];
}
```

```
// clang++ -Oz test.cpp -o test.o, 12960 bytes
#include<list>
int main() {
  std::list<int> l;
  for (int i = 0; i < 1000; ++i)
    l.push_back(i);
  return *l.begin();
}
```

```
// clang++ -Oz test.cpp -o test.o, 14308 bytes
#include<vector>
int main() {
  std::vector<int> v;
  for (int i = 0; i < 1000; ++i)
    v.push_back(i);
  return v[0];
}
```

1. Code size compiler optimizations and techniques for embedded systems. https://llvm.org/devmtg/2020-09/slides/Kumar-LLVMDevCodeSizePdf.pdf

算法与编程（人工经验）

- 良好的编程风格，有助于编译优化效果
  - 比如，嵌入式编程风格
    - 对于局部函数，尽量声明static
    - 将频繁使用的宏定义转换成函数调用
    - 减少数据或字符串硬编码的使用
    - 使用够用的最低数据类型
    - 减少全局变量以及全局调用的使用
    - 控制函数的参数数目

3 Coding Techniques ....
3.1 Use Smallest Possible Types for Variables and Constants ....
3.2 Avoid Multiply and Divide ....
3.3 Use Lookup Tables Instead of Calculating ....
3.4 Use Word Accesses to Registers ....
3.5 Write to Registers Only Once (Where Possible) ....
3.6 Use the __even_in_range() Intrinsic ....
3.7 Use Functions Judiciously and Write for Reuse and Commonality ......

1. https://www.iar.com/knowledge/learn/programming/writing-optimizer-friendly-code/
2. Optimizing C Code for Size With MSP430™ MCUs: Tips and Tricks.
https://www.ti.com/lit/an/slaa801/slaa801.pdf?ts=1627212102615
https://aykevl.nl/2018/04/codesize

Outline

- 代码尺寸优化概述
- 相关工作介绍
  - 算法与编程（人工经验）
  - 删除无用代码
  - 精简重复代码
  - 压缩

删除无用代码

- 构建时删除
- 编译时删除
- link时删除
- 部署时删除

构建时删除

- 通过定制构建选项，删除不需要支持的功能代码
  - 冗余代码来自软件库的通用性和完整性
  - libc
    - ➢ IPv6 support can be removed with **USE_INET6=no:**
    - ➢ SSP buffer overflow protection from the GCC compiler can be disabled with **USE_SSP=no**
    - ➢ RPC support can be disabled with **MKRPC=no** variable and a patch
    - ➢ **glibc vs uclibc vs musl**
  - **Linux kernel**
    - ➢ kernel configuration file
    - ➢ tinyconfig
    - ➢ 各种补丁（2014年及以前）
  - 其他第三方库或者自有库

```
# CONFIG_CC_OPTIMIZE_FOR_PERFORMANCE is not set
CONFIG_CC_OPTIMIZE_FOR_SIZE=y
# CONFIG_KERNEL_GZIP is not set
# CONFIG_KERNEL_BZIP2 is not set
# CONFIG_KERNEL_LZMA is not set
CONFIG_KERNEL_XZ=y
# CONFIG_KERNEL_LZO is not set
# CONFIG_KERNEL_LZ4 is not set
CONFIG_OPTIMIZE_INLINING=y
# CONFIG_SLAB is not set
# CONFIG_SLUB is not set
```

设计者全局视角，子功能模块级别删除，粒度较粗糙…

https://wiki.netbsd.org/tutorials/how_to_reduce_libc_size
https://git.kernel.org/pub/scm/linux/kernel/git/josh/linux.git
Embedded Linux size reduction techniques. Embedded Linux Conference 2017
http://events17.linuxfoundation.org/sites/events/files/slides/opdenacker-embedded-linux-size-reduction-techniques_0.pdf

编译时删除

- 通过编译选项，可以改善code size
  - 很多常用的性能优化编译选项，能改善code size
    - constant fold，dead code elimination，不可达代码删除，
    - 需要关闭的选项： procedure inlining, loop unrolling, alignment options
    - 18%～20% reduction [pldi'99][toplos'00]
  - 专用于code size的优化选项
    - -Os in GCC， -Oz in clang

> `-Os`
>
> Optimize for size. `-Os` enables all `-O2` optimizations except those that often increase code size:
>
> ```
> -falign-functions  -falign-jumps
> -falign-labels  -falign-loops
> -fprefetch-loop-arrays  -freorder-blocks-algorithm=stc
> ```

编译过程以模块（文件）为视角单位，难以跨文件分析...

1. Cooper, K. D., & McIntosh, N. (1999). Enhanced code compression for embedded RISC processors. *ACM SIGPLAN Notices*, *34*(5), 139-149.
2. Debray, S. K., Evans, W., Muth, R., & De Sutter, B. (2000). Compiler techniques for code compaction. ACM Transactions on Programming languages and Systems (TOPLAS), 22(2), 378-415.

编译优化对code size的影响

- 机器无关优化
  - 消除重复计算
  - 编译时常量求值
  - 代数化简
  - loop unrolling
  - function inlining
- 机器相关优化
  - 指令选择
  - 指令调度（FIFO？）
  - 寄存器分配

常见的机器无关优化

- 消除重复计算
  - 公共子表达式

Example:

```
x := y + z                    x := y + z
...               ⇒           ...
w := y + z                    w := x
```

(the values of x, y, and z do not change in the … code)

  - 抽象数据类型访问
    - a[i][j][k] = x; y = a[i][j][k];

- 如何识别公共子表达式? no change in between.

常见的机器无关优化

- 编译时常量求值
  - 常量折叠

    Operations on constants can be computed at compile time

    - If there is a statement x := y op z
    - And y and z are constants
    - Then y op z can be computed at compile time

    Example: x := 2 + 2 ⇒ x := 4

    Example: if 2 < 0 jump L can be deleted

    When might constant folding be dangerous?
  - 如何评估常量表达式？常量是否可以传递？如何传递？

常见的机器无关优化

- 代数化简

Some statements can be deleted

x := x + 0

x := x * 1

Some statements can be simplified

x := x * 0 ⇒ x := 0

y := y ** 2 ⇒ y := y * y

x := x * 8 ⇒ x := x << 3

x := x * 15 ⇒ t := x << 4; x := t - x

(on some machines << is faster than *; but not on all!)

- 如何确保化简前后是恒等的?

常见的机器无关优化

- function inlining

```
inline void printSum(int num1,int
num2) {

    cout << num1 + num2 << "\n";

}
```

  - 函数内联通常增大代码尺寸
  - 但是，有时候
    - 函数调用需要的参数、frame、call指令+函数体，比不实现函数时的总函数体还大
    - 内联提供了编译器进行上下文相关优化的机会，这些优化可能会减小代码大小

```
inline void printSum(int num1, int num2){
    cout << num1 + num2 << "\n";
}
int main() {
    printSum(10, 20);
    printSum(2, 5);
    printSum(100, 400);
}
```

During Compilation

```
inline void printSum(int num1, int num2){
    cout << num1 + num2 << "\n";
}
int main() {
    cout << 10 + 20 << "\n";
    cout << 2 + 5 << "\n";
    cout << 100 + 400 << "\n";
}
```

Theodoros Theodoridis, Tobias Grosser, Zhendong Su:
Understanding and exploiting optimal function inlining. ASPLOS 2022: 977-989

常见的机器无关优化

- loop unrolling

Normal loop	After loop unrolling

`int x;`
`for (x = 0; x < 100; x++)`
`{`
`    delete(x);`
`}`
	`int x;`
	`for (x = 0; x < 100; x += 5 )`
	`{`
	`    delete(x);`
	`    delete(x + 1);`
	`    delete(x + 2);`
	`    delete(x + 3);`
	`    delete(x + 4);`
	`}`

  - 通常, loop unrolling被认为增加代码尺寸
  - 但是,
    - 有没有可能去掉loop的overhead， 从而减少代码?
    - 有没有可能提供更多优化机会，从而减少代码?

link时删除（静态链接）

- link时优化，删除**可执行文件**中无用的section
  - 静态链接，将多个.o文件与lib文件静态链接，可以同时分析同一个程序的多个文件
  - -lto
    - link time optimization，以性能优化为目标（如更好地支持inlining），附带code size优化
    - lto+global constant propagation
  - -gc-sections
    - -ffunction-sections：确保每个函数为一个section
    - -gc-sections：在link阶段进行符号解析，符号解析后发现的无用函数对应的section可以删除

link时删除（动态链接）

- 动态链接
  - 因为多个程序会共享相同的.so库，所以动态链接有潜力实现多个程序间的分析和优化
  - 假设不会有新增的.so库的调用者

Bartell, Sean, Will Dietz, and Vikram S. Adve. "Guided linking: dynamic linking without the costs." Proceedings of the ACM on Programming Languages 4.OOPSLA (2020): 1-29.

部署时删除——假设与思路

- 部署阶段，可以分析计算设备上的所有软件代码
- 假设：
  - 已有软件系统的完整二进制代码，包括
    - 应用程序：app1.out app2.out …
    - 共享库：lib.so
    - Linux内核：vmlinux
  - 应用程序集固定，所以使用到的库函数符号确定
    - 无用的库函数符号可以分析确定
- 思路：
  - 识别二进制代码中的所有unused代码块
  - 从二进制软件系统中直接移除这些unused代码块
  - 精简后的软件系统功能不变

部署时删除

- 实现路线1:
  - 删除无用的库函数
  - 需要重新link共享库
- 实现路线2:
  - 删除无用的代码块
  - 二进制重写，不需要重新link共享库
- 后续详述

部署时删除——路线1

1. 函数引用图分析
   - 利用重定位信息，构建函数间的引用图
   - 比如，f引用了g，如果:
     - f函数中，call g
     - f函数中，读取了g的地址（函数指针）
2. 函数调用图遍历
   - 从应用程序中的入口函数（比如main）出发，遍历全局函数引用图
   - 凡是不可达的函数符号，都是无用的函数符号
3. 删除无用的函数符号
   - 借用--ffunction-sections, -gc-sections选项，修改link脚本

部署时删除——路线1：实验评估

- 删除无用的.os文件
  - 在re-link过程中，去掉不需要的.os文件
- 实验评估（X86-64）：
  - 22个app

```
kp3@kp:~/qali/cy/app2Dir$ ls
addr2line  ar  as  bzip2  c++filt  elfedit  file  gprof  grep  iptables  kmod  ld  less  nm  objcopy  objdump  outDir  ranlib  readelf  sed  size  strings  strip  test.sh
```

  - 优化效果

	初始大小	优化大小
libc.so	16.6M	11.2M
libc.so(−debug)	2.1M	1.6M
libc.so(−unused)	1.8M	1.4M

部署时删除——路线1：实验评估

	初始大小	优化大小
libm.so	5.1M	2.6M
libm.so(-debug)	1.4M	828.4K
libm.so(-unused)	1.3M	784.6K
libpthread.so	2.4M	1.2M
libpthread.so(-debug)	145.7K	94.2K
libpthread.so(-unused)	114.1K	72.4K

- 删除无用的.section文件
- 评估结果
  - app集合（X86-64）

```
kp3@kp:~/qali/cy/app2Dir$ ls
addr2line  ar  as  bzip2  c++filt  elfedit  file  gprof  grep  iptables  kmod  ld  less  nm  objcopy  objdump  outDir  ranlib  readelf  sed  size  strings  strip  test.sh
```

  - 正确性
    - 动态loader在将控制权给app之前，会查询app中所有的外部符号是否在内存
    - 执行每个程序(--help)，确保不会输出错误

```
kp3@kp:~/qali/binary-size/testcase$ ./a.out
./a.out: symbol lookup error: ./a.out: undefined symbol: print
```

  - 优化效果

	初始大小	file-level	Func-level
libc.so	16.6M	11.2M	10.9M
libc.so(strip debug)	2.1M	1.6M	1.5M
libc.so(strip unneed)	1.8M	1.4M	1.3M

部署时删除——路线2

1. 反汇编：
   - 将.app+.so反汇编成.s
2. 汇编级控制流分析
   - intra-procedural：simple CFG building
   - inter-procedural：
3. 引用分析
   - 从应用程序中的入口函数（比如main）出发，根据ldd得到依赖的.so库；在.so中解析undef全局符号，得到全局引用关系图
4. 二进制重写优化
   - 删除.so中的无用符号，调整指令偏移
   - 删除app的动态符号表中的无用符号表项
     - 避免无谓的look up symbol error

部署时删除——路线2：实验评估

- benchmark（ARM64, 22个应用程序）

```
kp3@kp:~/qali/cy/app2Dir$ ls
addr2line  ar  as  bzip2  c++filt  elfedit  file  gprof  grep  iptables  kmod  ld  less  nm  objcopy  objdump  outDir  ranlib  readelf  sed  size  strings  strip  test.sh
```

- 效果

	优化前	优化后	优化比
libc	16391304	16123968	0.983690376
libc(-debug)	1738768	1471416	0.846240557
libc(-unneeded)	1417448	1150096	0.811384968

Outline

- 代码尺寸优化概述
- 相关工作介绍
  - 算法与编程（人工）
  - 删除无用代码
  - 精简重复代码
  - 压缩

精简重复代码

- 思路:
  - 识别重复或相似代码序列，合并优化
- 分类
  - 公共子表达式
  - local code factoring（near相同代码合并）
  - 过程抽象（far相同代码合并）
  - function merging（相似函数合并成新函数）
  - loop rerolling

- 消除重复计算
  - 公共子表达式

Example:

```
x := y + z                 x := y + z
...             ⇒          ...
w := y + z                 w := x
```

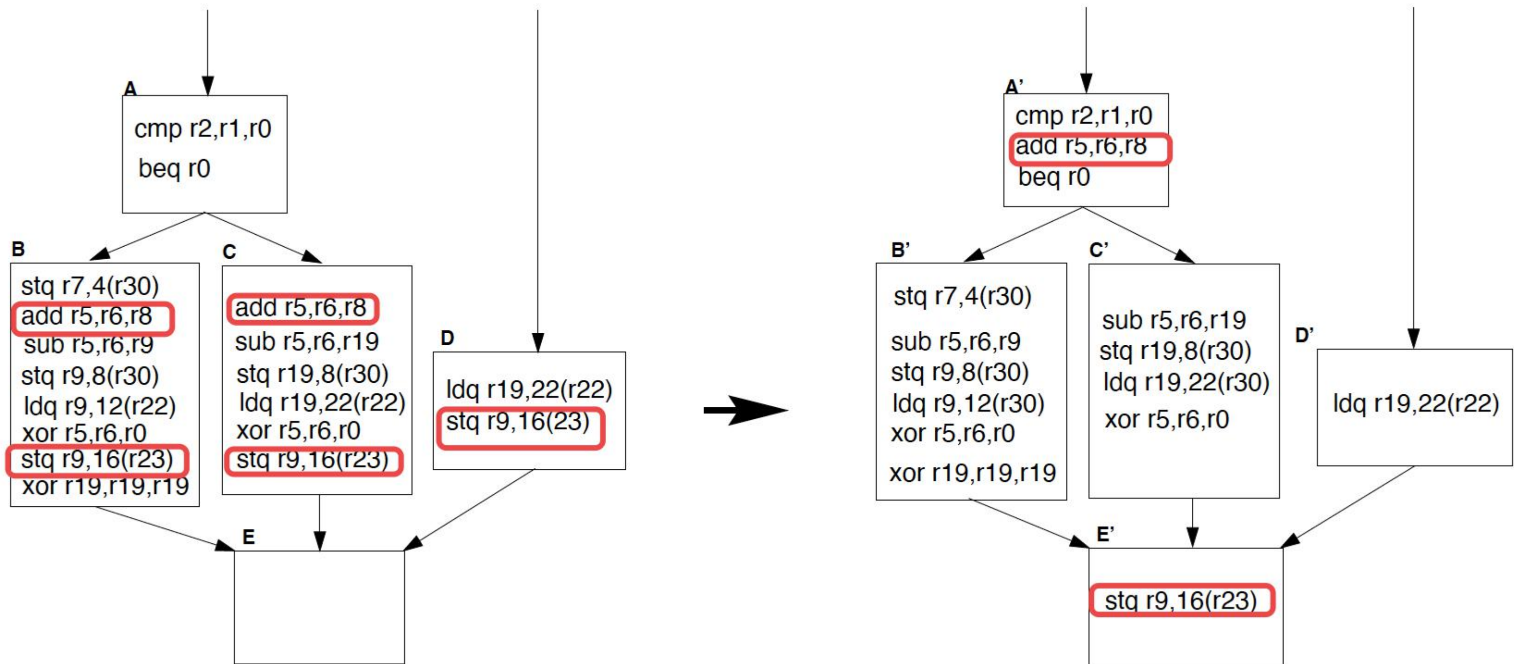
(the values of x, y, and z do not change in the … code)

  - 抽象数据类型访问
    - a[i][j][k] = x; y = a[i][j][k];

- 如何识别公共子表达式？ no change in between.

local code factoring

- 思路
  - 识别出分支（或循环）中的公共表达式，将其往前移动（hoist）到公共前驱路径，或往后移动（sink）到公共后继路径上



过程抽象

- 思路
  - 识别重复序列，替换函数调用

```
// Original source
{

  ...

  I0;
  I1;
  I2;
  I3;

  ...

  I0;
  I1;
  I2;
  I3;

  ...

  I0;
  I1;
  I2;
  I3;

  ...

  I0;
  I1;
  I2;
  I3;

  ...
}
```

```
// After sequence abstraction
{
  void *jump_label;
  ...
  jump_label = &&exit_0;
entry_0:
  I0;
  I1;
  I2;
  I3;
  goto *jump_label;
exit_0:
  ...
  jump_label = &&exit_1;
goto entry_0;

exit_1:
  ...
  jump_label = &&exit_2;
  goto entry_0;

exit_2:
  ...
  jump_label = &&exit_3;
  goto entry_0;

exit_3:
  ...
}
```

优化效果—machine outliner for UberRider

- 基于过程抽象的改进
  - machine outliner：LLVM中实现的过程抽象
  - 全程序优化
  - repeated machine ou

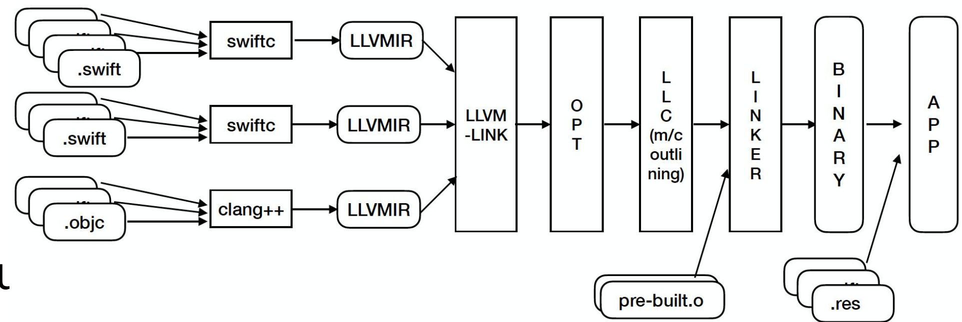


TABLE II: OUTLINING STATISTICS AT DIFFERENT LEVELS OF REPEATS.

Metric	rounds of outlining				
	1	2	3	4	5
# sequences outlined ($\times 10^6$)	3.08	4.30	4.62	4.70	4.71
# functions created ($\times 10^5$)	1.15	2.03	2.44	2.57	2.59
Bytes consumed by outlined functions ($\times 10^6$)	1.69	2.80	3.33	3.50	3.53

1. Chabbi, M., Lin, J., & Barik, R. (2021, February). An Experience with Code-Size Optimization for Production iOS Mobile Applications. In 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (pp. 363-377). IEEE.

优化效果—machine outliner for UberRider

- 特点：
  - UberRider on iOS
    - 145.7MB binary that has 114.5MB code
  - malloc allocation和reference counting导致大量重复
  - 高级语言特征
- 优化效果
  - -23%

TABLE I: SUMMARY OF DIFFERENT OPTIMIZATION CHOICES.

Level	Optimization considered	Note
AST	Source function replicas [40]	<1% replication
SIL	SIL outlining [41]	0.41% size saving
LLVM-IR	MergeFunction [42]	0.9% size saving
	FMSA [43]	2% size savings
	MergeSimilarFuncs [44]	high build times
	IROutliner [45]	not target aware
ISA	Repeated machine outlining	23% size reduction

```
1  $x0 = ORRXrs $xzr, $x20
2  BL swift_release
```

Listing 1: swift_release (46.3K repeats)

```
1  $x0 = ORRXrs $xzr, $x19
2  BL swift_release
```

Listing 2: swift_release (21.8K repeats)

```
1  $w2 = ORRWri $wzr, 2
2  BL swift_allocObject
```

Listing 3: swift_allocObject (64.3K repeats)

```
1  $x0 = ORRXrs $xzr, $x20
2  BL objc_release
```

Listing 4: objc_release (35.2K repeats)

```
1  $x0 = ORRXrs $xzr, $x20
2  BL objc_retain
```

Listing 5: objc_retain (29.5K repeats)

```
1  $x0 = ORRXrs $xzr, $x20
2  BL swift_retain
```

Listing 6: swift_retain (24.4K repeats)

```
1  $sp = STPXpre $x26, $x25,
       $sp(tied-def 0), -10
2  STPXi $x24, $x23, $sp, 2
3  STPXi $x22, $x21, $sp, 4
4  STPXi $x20, $x19, $sp, 6
```

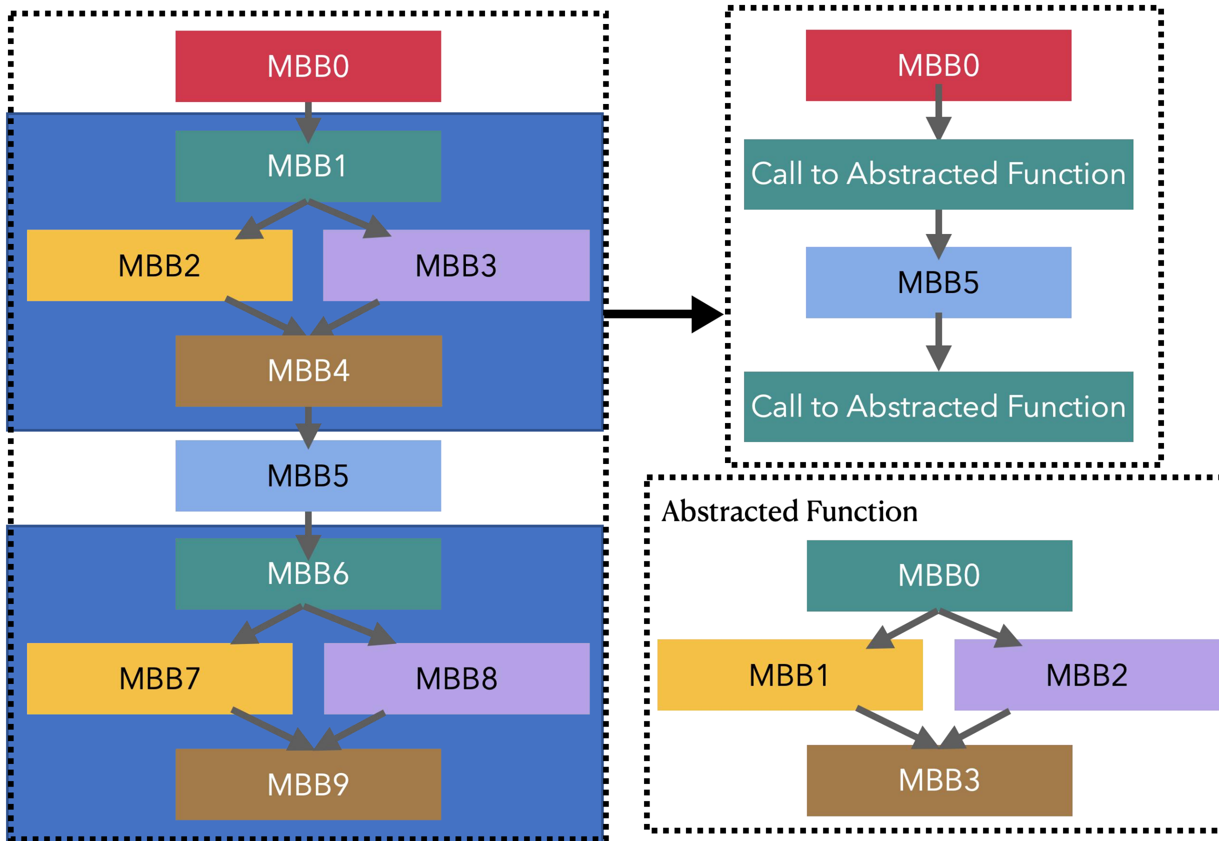
Listing 7: Frame setup (7K repeats)

```
1  $x20, $x19 = LDPXi $sp, 6
2  $x22, $x21 = LDPXi $sp, 4
3  $x24, $x23 = LDPXi $sp, 2
4  $sp, $x26, $x25 = LDPXpost
       $sp(tied-def 0), 10
```

Listing 8: Frame destroy (7K repeats)

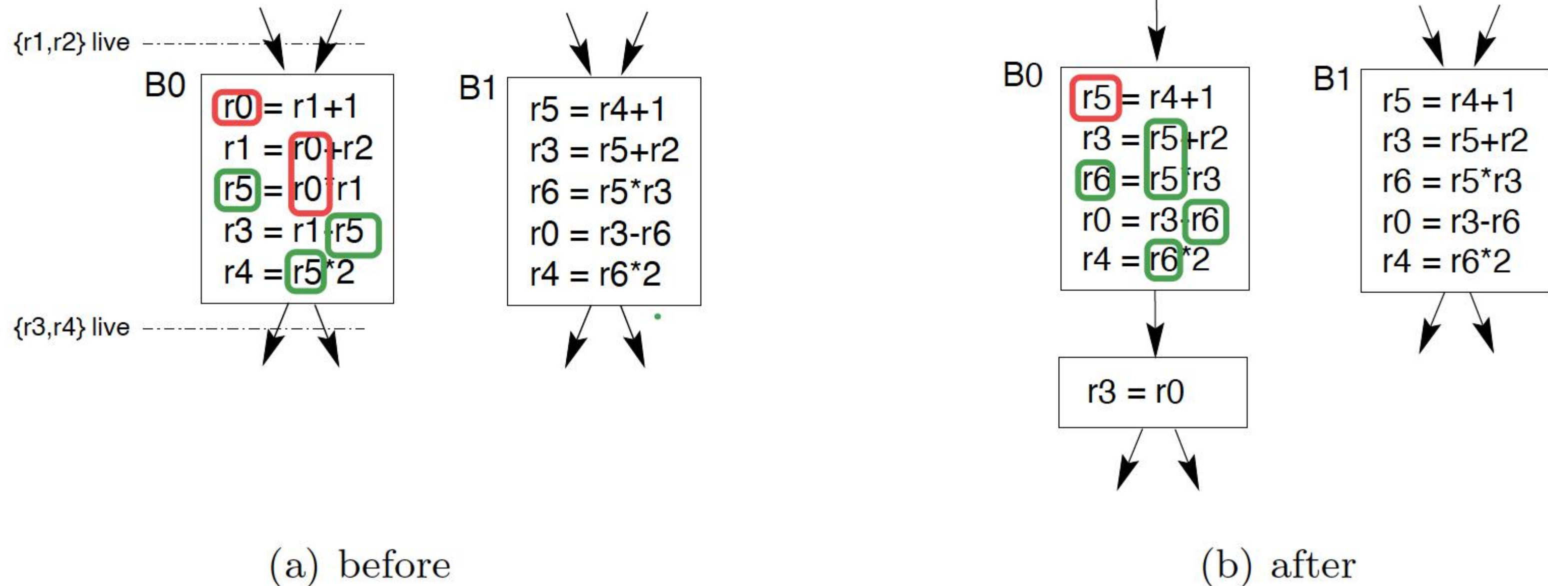
改进：跨基本块的过程抽象（跨branch指令）

- 单入口单出口



改进：register renaming

- 情形：代码不一样，且只是register不一样



(a) before (b) after

改进：more

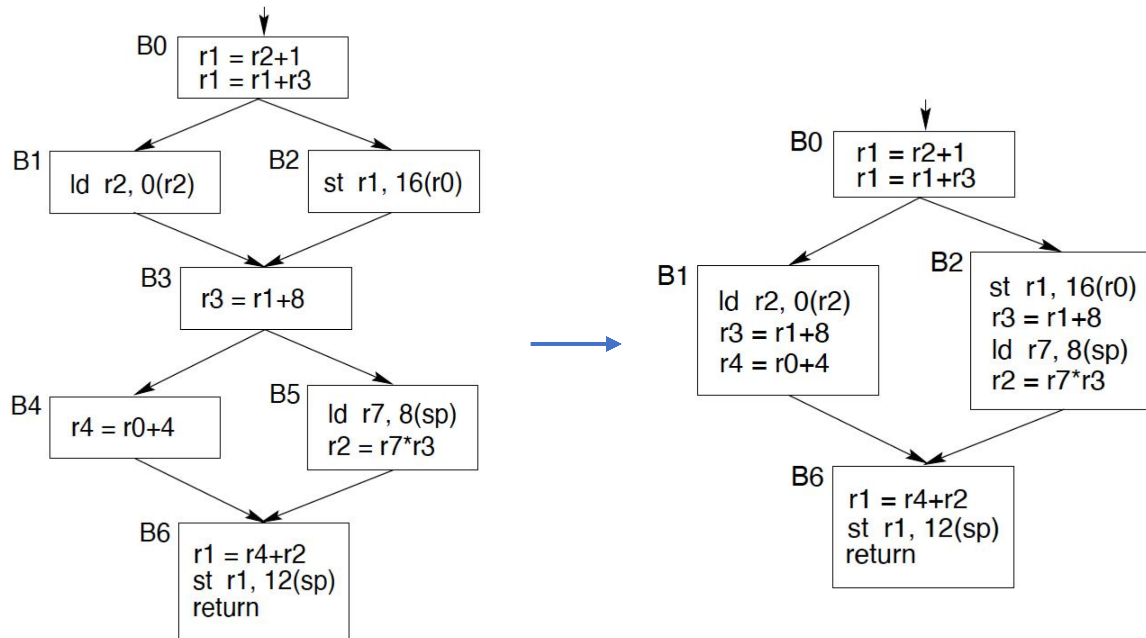
- 基于语义等价的normalizatoin
  - 算术、逻辑、关系表达式的结合律和交换律
  - if-else/for循环的normalized翻译
  - 代数恒等式

相似代码合并（block级别）

```
r1 = r2+1
r1 = r1+r3
ld r2, 0(r2)
r3 = r1+8
r4 = r0+4
r1 = r4+r2
st r1, 12(sp)
```

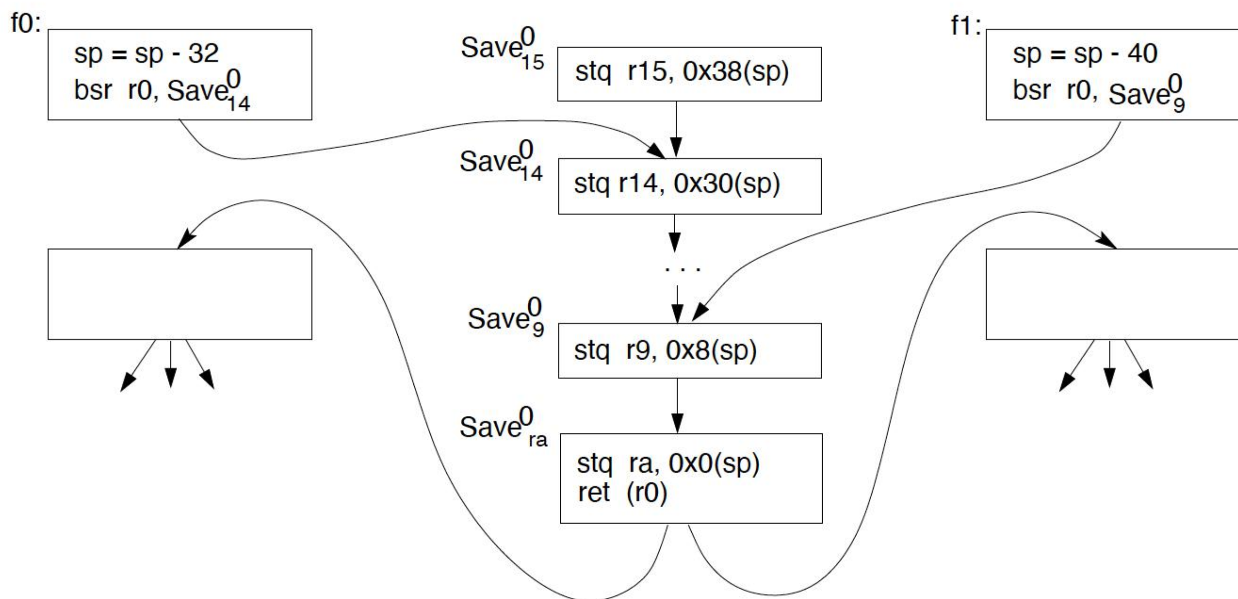
```
r1 = r2+1
r1 = r1+r3
st r1, 16(r0)
r3 = r1+8
ld r7, 8(sp)
r2 = r7*r3
r1 = r4+r2
st r1, 12(sp)
```

- 思路
  - 识别相似函数，合并成一个函数



例子

- 情形：利用calling convention，识别相似序列
- -msave-store (Risc-v), 寄存器保护与恢复
  - 例子：f0需要保护r14~r9，f1只需要保护r9



优化效果

- Effect in a bigger graph (reduction 28%~30%)
  - redundant computation elimination -> global pointer opti
  - eliminate redundant bank selection instructions?

Transformation	*Savings* (%)
redundant computation elimination	34.14
Basic block and region abstraction	27.42
Useless code elimination	22.43
Register save/restore abstraction	9.95
Other inter-procedural optimizations	6.06

- Close-up on code factoring:
  - local factor factoring + 过程抽象 + similar block merging ?
  - code size reduction: 5%~6%
  - performance degration: 4%~10%

1. Cooper, K. D., & McIntosh, N. (1999). Enhanced code compression for embedded RISC processors. *ACM SIGPLAN Notices*, *34*(5), 139-149.
2. **Debray, S. K., Evans, W., Muth, R., & De Sutter, B. (2000). Compiler techniques for code compaction. ACM Transactions on Programming languages and Systems (TOPLAS), 22(2), 378-415.**

改进：相似代码合并（函数级别）

Example from 400.perlbench

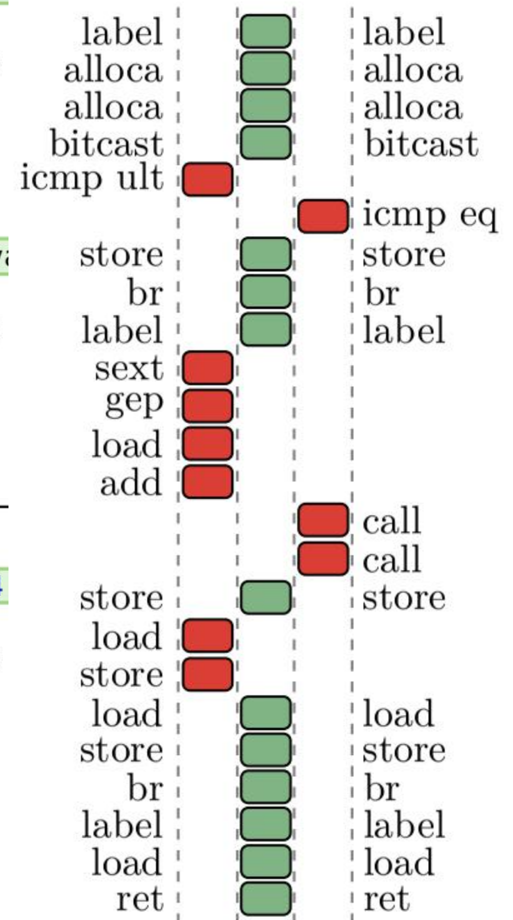
```
1 | OP *Perl_scalarkids(pTHX_ OP *o) {
2 |   OP *kid;
3 |   if (o && o->op_flags & OPf_KIDS) {
4 |     for (kid = cLISTOPo->op_first; kid; kid = kid->op_sibling)
5 |       scalar(kid);
6 |   }
7 |   return o;
8 | }
```

```
1 | OP *Perl_listkids(pTHX_ OP *o) {
2 |   OP *kid;
3 |   if (o && o->op_flags & OPf_KIDS) {
4 |     for (kid = cLISTOPo->op_first; kid; kid = kid->op_sibling)
5 |       list(kid);
6 |   }
7 |   return o;
8 | }
```

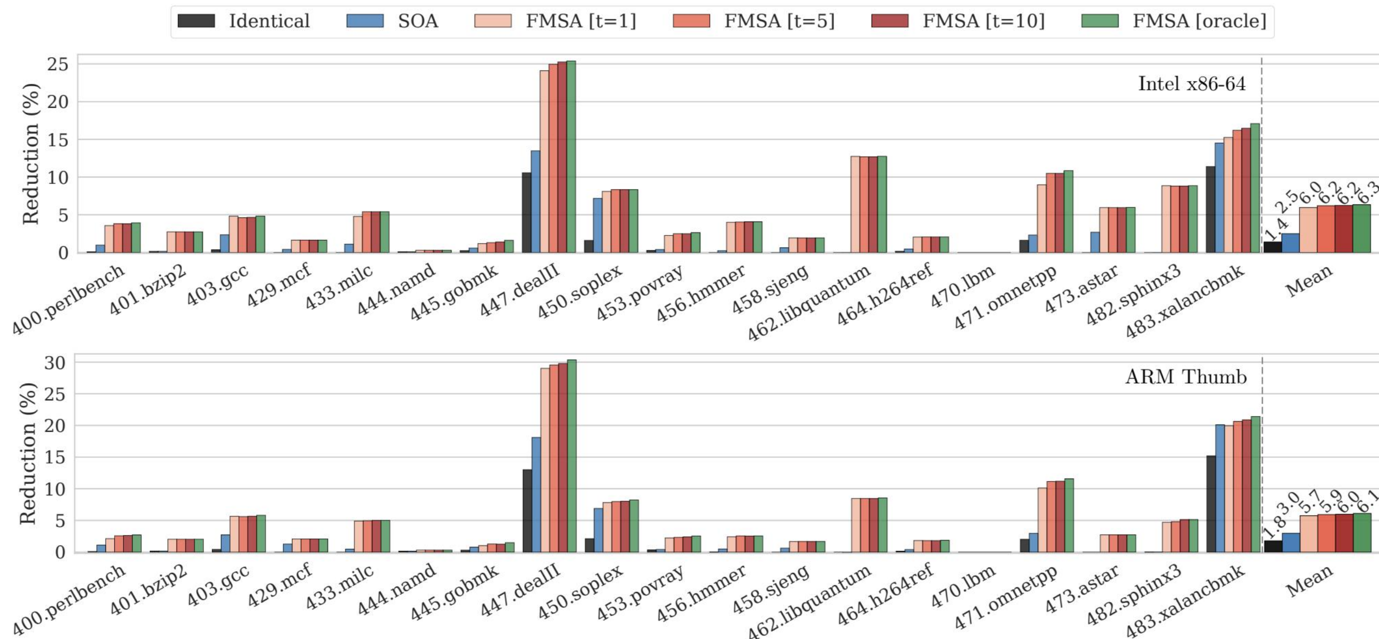
```
glist_t glist_add_float32(glist_t g, float32 va
  gnode_t *gn;
  gn = (gnode_t *) mymalloc (sizeof(gnode_t));
  gn->data.float32 = val;
  gn->next = g;
  return ((glist_t) gn);
}

glist_t glist_add_float64(glist_t g, float64 va
  gnode_t *gn;
  gn = (gnode_t *) mymalloc (sizeof(gnode_t));
  gn->data.float64 = val;
  gn->next = g;
  return ((glist_t) gn);
}
```

Merged Function

```
glist_t merged(bool func_id,
               glist_t g, float32 v32, float64
  gnode_t *gn;
  gn = (gnode_t *) mymalloc (sizeof(gnode_t));
  if (func_id)
    gn->data.float32 = v32;
  else
    gn->data.float64 = v64;
  gn->next = g;
  return ((glist_t) gn);
}
```



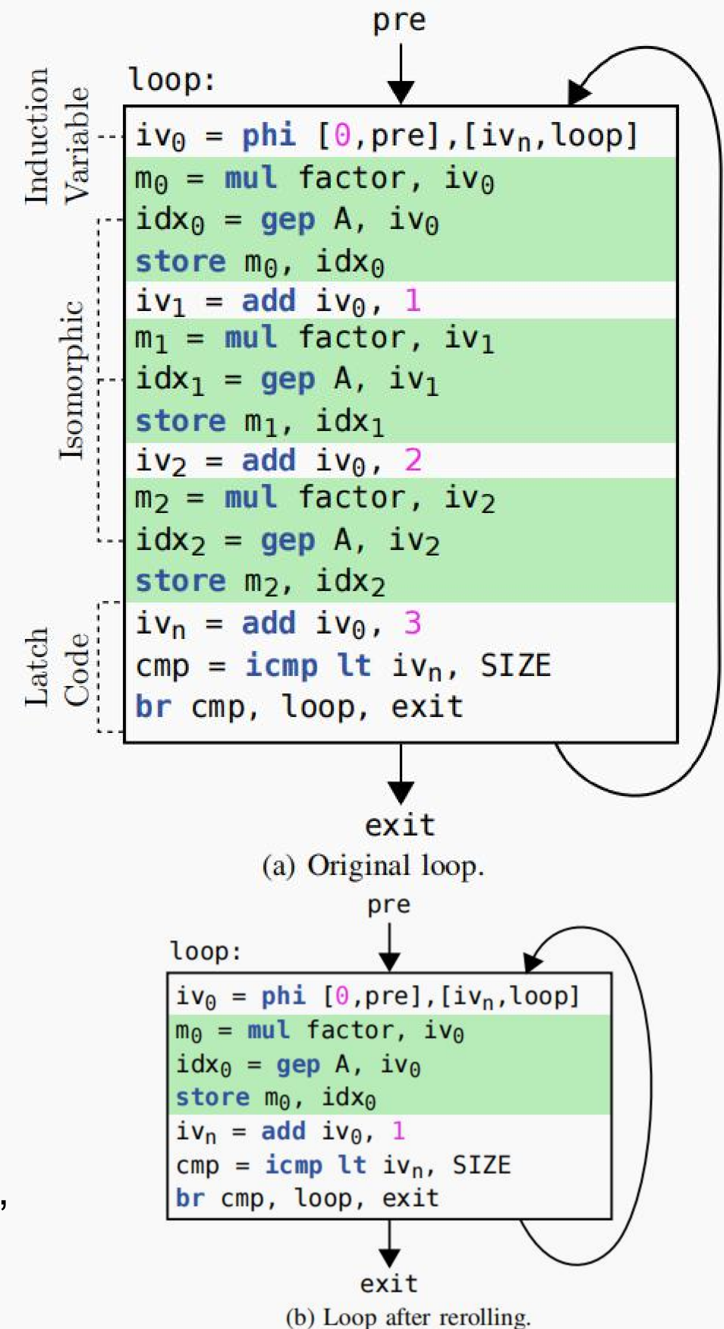
• 优化效果：～6.0%



1. **Rocha, R. C., Petoumenos, P., Wang, Z., Cole, M., & Leather, H. (2019, February). Function merging by sequence alignment. In 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (pp. 149-163). IEEE.**
2. Edler von Koch, T. J., Franke, B., Bhandarkar, P., & Dasgupta, A. (2014, June). Exploiting function similarity for code size reduction. In Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (pp. 85-94).
3. Rocha, R. C., Petoumenos, P., Wang, Z., Cole, M., Hazelwood, K., & Leather, H. (2021, April). HyFM: function merging for free. In *22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems: Co-located with PLDI 2021*.

loop rerolling

- 思路
  - 将unrolling的代码改成循环
  - 提取可roll的模式



(a) Original loop.

(b) Loop after rerolling.

Fig. 1. Example demonstrating LLVM's loop rerolling technique on a partially unrolled loop.

Rodrigo C. O. Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, Michael F. P. O'Boyle:
Loop Rolling for Code Size Reduction. CGO 2022: 217-229

Outline

- 代码尺寸优化概述
- 相关工作介绍
  - 算法与编程（人工）
  - 删除无用代码
  - 精简重复代码
  - 压缩

压缩

Method	ECS
4.1 Cooper & McIntosh	0.95
4.2 Squeeze	0.70
4.3 Narrow-Word	——
4.4 Wolfe (LAT-based)	0.73
4.5 Breternitz	0.56
4.6 Lekatsas & Wolf	0.50
4.7 CodePack*	0.60
4.8 Brazil*	0.43
4.9 ThumbScrews	0.70
4.10 Model inference*	0.19
4.11 Arizona-Microsoft*	0.20
4.12 Slim Binaries	0.35

- 思路：先压缩编码，执行前再解码
- 分类：
  - 编码方法
    - Huffman Coding, Arithmetic Coding, Dictionary-Based Methods
  - 编码粒度：bit，字符，指令、block等
  - 解码方法：运行时软件、硬件解码
- 优点：压缩比高
- 缺点
  - 解码的软硬件支持，性能及space overhead

Beszédes, Á., Ferenc, R., Gyimóthy, T., Dolenc, A., & Karsisto, K. (2003). Survey of code-size reduction methods. ACM Computing Surveys (CSUR), 35(3), 223-267.

压缩的工业应用

- linux内核镜像文件压缩
  - vmlinux -> zImage
- Android中的zram

压缩的工业应用：ISA

- 变长指令集
  - Intel x86：1～15 bytes
  - ARM：Thumb和Thumb2
  - MIPS：MIPS-16
  - RISC-V：RISC-V Compressed, and Huawei's extension
- Stack架构
  - 更紧凑的指令编码：简化了操作数寻址
    - ➢25% shorter than register-based byte code (see Taco 2008)
  - 更多的重复代码序列：过程抽象优化
    - ➢even shorter?

Shi, Y., Casey, K., Ertl, M. A., & Gregg, D. (2008). Virtual machine showdown: Stack versus registers. ACM Transactions on Architecture and Code Optimization (TACO), 4(4), 1-36.
Lozano, H., & Ito, M. (2016, May). Increasing the Code Density of Embedded RISC Applications. In 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC) (pp. 182-189). IEEE.

压缩的工业应用：ISA

- Huawei's RISC-V extension
  - -mpush-pop
    - push/pop a fixed set of regs
  - -enable-c-lbu-sb
    - load a byte (compressed)
  - -fimm-compare
    - Branch on Immediate op Reg

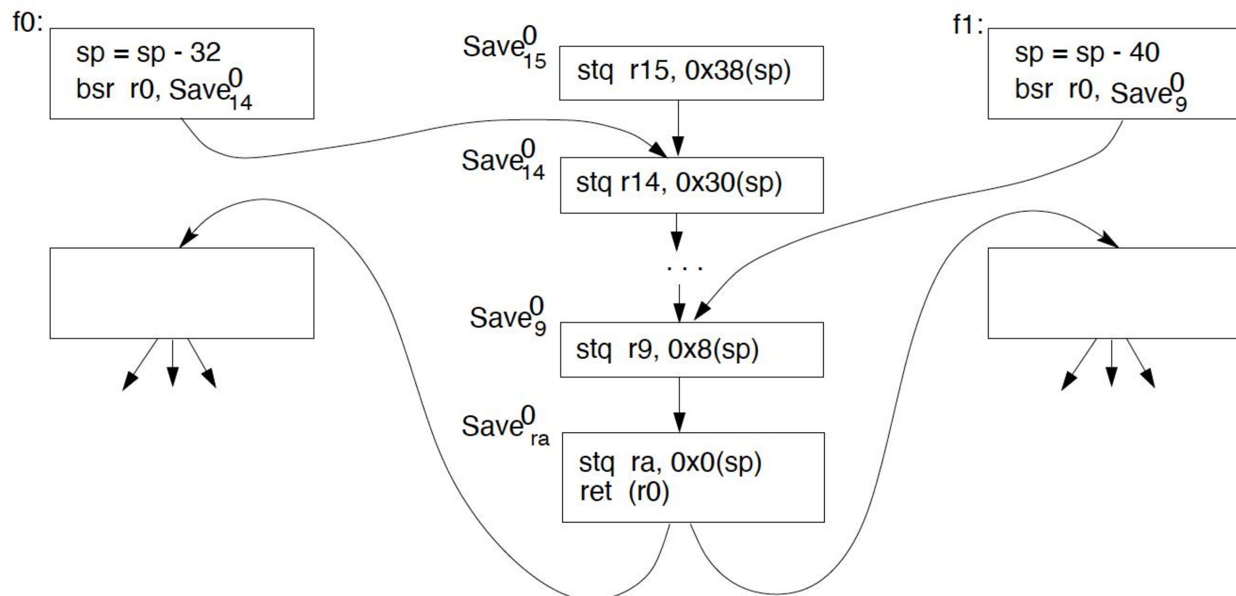
ISA condition	IoT [B]	Embench [B]
HCC reference	233804	1975
-mpush-pop	221860 (-5.11%)	1902 (-3.70%)
-Wa,-enable-c-lbu-sb	230328 (-1.49%)	1960 (-0.76%)
-Wa,-enable-c-lhu-sh	232084 (-0.74%)	1971 (-0.20%)
-femit-lli	232556 (-0.53%)	1988 (0.66%)
-fimm-compare	230636 (-1.35%)	1967 (-0.41%)
-Wl,-enjal16	233804 (0.00%)	1975 (0.00%)
-femit-muliadd	233396 (-0.17%)	1975 (0.00%)
-fmerge-immshf	232992 (-0.35%)	1915 (-3.04%)
-femit-uxtb-uxth	232864 (-0.40%)	1970 (-0.25%)
-fldm-stm-optimize	233796 (0.00%)	1964 (-0.56%)
HCC extension	206020 (-11.88%)	1805 (-8.61%)

ISA condition	IoT [B]	Embench [B]
ARM-Thumb2	209696	1766
RV32IMC HCC	206020 (-1.75%)	1805 (2.21%)
RV32IMC	233628 (11.41%)	1966 (11.33%)
RV32IMC push/pop	221860 (5.80%)	1902 (7.70%)
RV32IMC Xpulp	230264 (9.81%)	1896 (7.36%)
RV32IMC Xpulp push/pop	220956 (5.37%)	1860 (5.32%)

Perotti, M., Schiavone, P. D., Tagliavini, G., Rossi, D., Kurd, T., Hill, M., ... & Benini, L. (2020). Hw/sw approaches for risc-v code size reduction. In Workshop on Computer Architecture Research with RISC-V (CARRV 2020).

-mpush-pop

- 情形： 利用calling convention，识别相似序列
- -msave-store (Risc-v), 寄存器保护与恢复
  - 例子：f0需要保护r14~r9，f1只需要保护r9



Superpack: Compiler meet data compression

- For repeated sequences of bytes
  - compression
    - using pointer + distance to compress recurring sequence
  - compiler: collect context info for data, by differentiating immediates
    - recurring length: 2 -> 4 insts;
    - distance: 2 -> 0

Original	Group 1	Group 2
ADD x16, x16	ADD x16, x16	#0x2144
BR x17	BR x17	#0x0860
ADRP x16, 671000	ADRP x16, 671000	#0x2152
LDR x17, x16, #0x2144	LDR x17, x16	#0x0868
ADD x16, x16, #0x860	ADD x16, x16	#0x2160
BR x17	BR x17	
ADRP x16, 671000	ADRP x16, 671000	
LDR x17, x16, #0x2152	LDR x17, x16	
ADD x16, x16, #0x868	ADD x16, x16	
BR x17	BR x17	
ADRP x16, 671000	ADRP x16, 671000	
LDR x17, x16, #0x2160	LDR x17, x16	

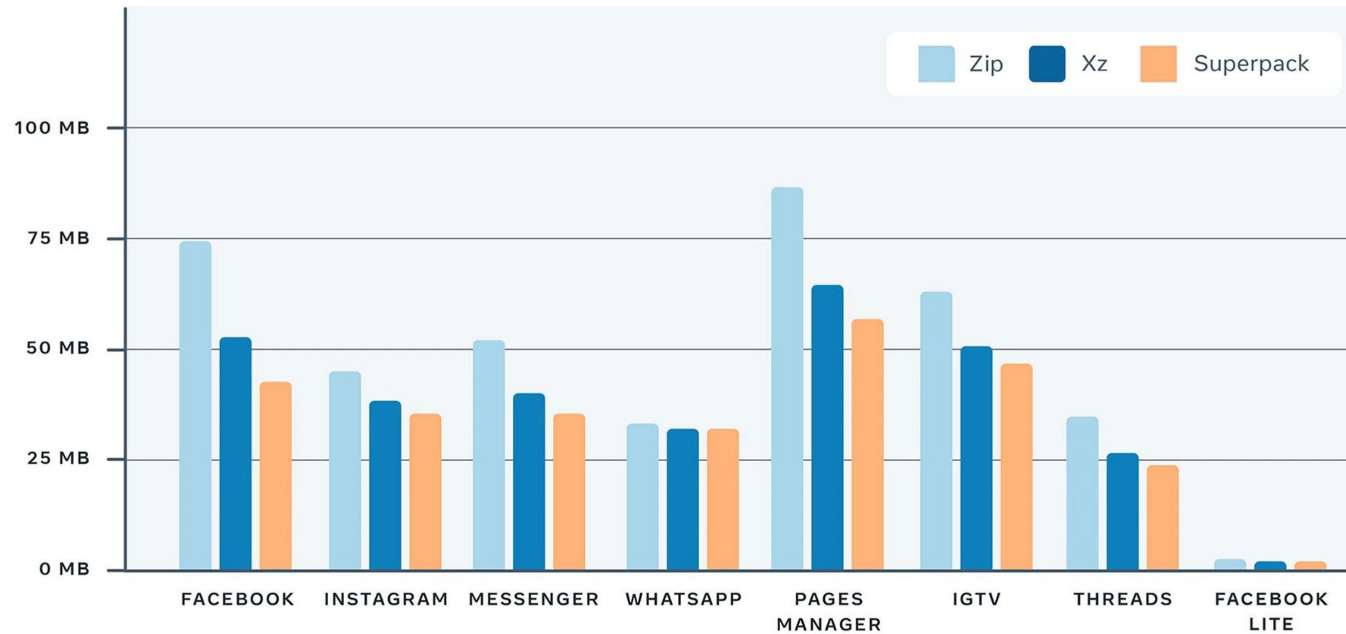
- https://engineering.fb.com/2021/09/13/core-data/superpack/

- For non-repeated sequences of bytes or short sequence cheaper than a pointer
  - entropy coding
  - compiler provides context info by differentiating Opcs
    - ➢ 3 -> 2bits for coding

Code		Regular coding (~3 bits)		Coding with opcode context (~2 bits)	
B	0x6b198				
B	0x6b228	0x6b198	000	0x6b198	00
BL	0x8a198	0x6b228	001	0x6b228	01
BL	0x8a248	0x8a198	101	0x8a198	01
B	0x6b448	0x8a248	111	0x8a248	11
B	0x6b1a8	0x6b448	011	0x6b448	11
BL	0x6b198	0x6b1a8	110	0x6b1a8	10
BL	0x89c70	0x89c70	100	0x89c70	00

Superpact

- 20%压缩比优化 vs Zip



总结

- 代码尺寸优化概述
- 相关工作介绍
  - 算法与编程（人工）
  - 删除无用代码
  - 精简重复代码
  - 压缩

Thanks for your time!