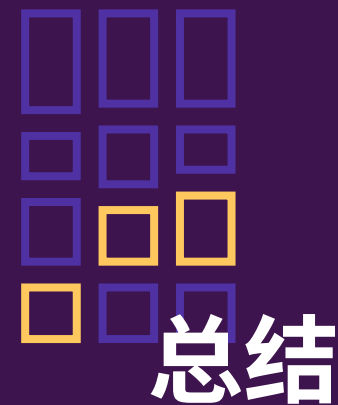
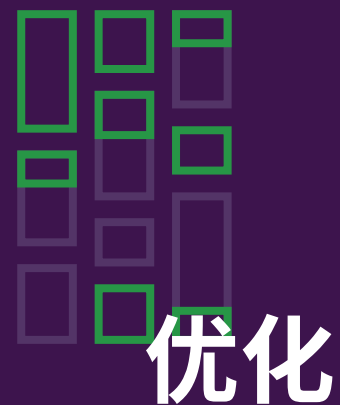
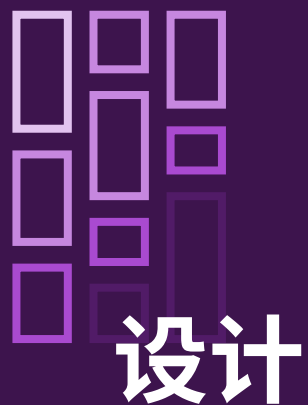
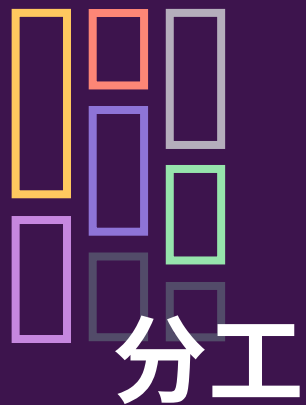




SEGVOL

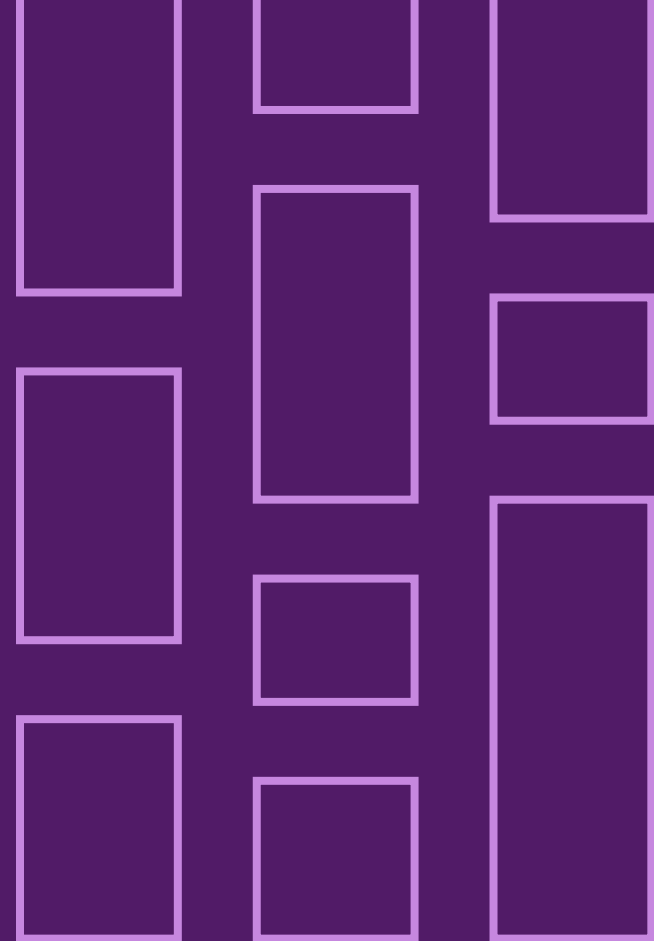
段地址不队

CONTENTS



DESIGN

编译器整体设计。

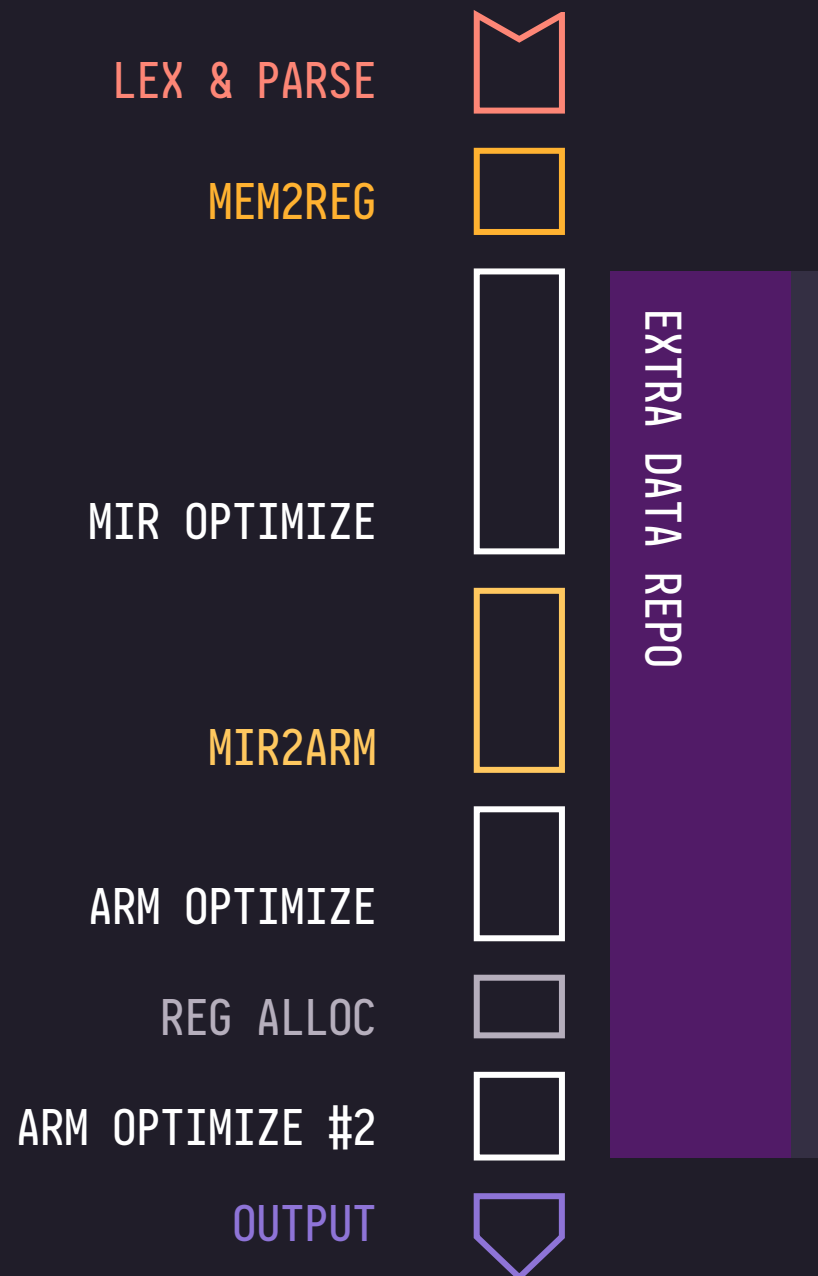


管线设计

在读入分析源程序后，首先将其转换成 SSA 形式的中间代码（MIR），进行中间代码的优化。

之后，编译器将中间代码翻译为使用虚拟寄存器的 ARM 汇编，进行汇编的优化。

最后，编译器为虚拟寄存器分配真实寄存器，做一些真实寄存器的优化，输出最终结果。



中间代码设计

中间代码，aka MIR，是输入程序在我们的编译器里大部分时间的表示形式。

MIR 通常是 SSA 的，但刚刚从文法分析处输出产生的不是。

大部分设计借鉴自 LLVM。

```
fn $$5_main() -> i32 {  
  $0: i32, priority: 0  
  %0: i32, priority: 0  
  %1: i32, priority: 0  
  %2: i32, priority: 0  
  %4: i32* temp, priority: 0  
  %6: i32 temp, priority: 0  
  %8: i32* temp, priority: 0  
  %10: i32 temp, priority: 0  
  %14: i32 temp, priority: 0  
  %18: i32 temp, priority: 0  
  %20: i32 temp, priority: 0
```

函数签名

变量表

```
bb1:    // preceding:  
  %4 = &@$$3_9  
  %6 = getint()  
  store %6 to [ %4 , 0 ]  
  %8 = &@$$3_10  
  %10 = getint()  
  store %10 to [ %8 , 0 ]  
  %20 = %6 == %10  
  br %20, 2, 3 if_branch
```

基本块

```
bb2:    // preceding: 1  
  %0 = 1  
  br 1048576  
bb3:    // preceding: 1  
  %1 = 0  
  br 1048576  
bb1048576: // preceding: 2, 3  
  %2 = phi [%0, %1]  
  $0 = %2  
  ret $0  
}
```

词/语法分析

MEM2REG

读入源代码，转换成非 SSA 格式的中间代码。

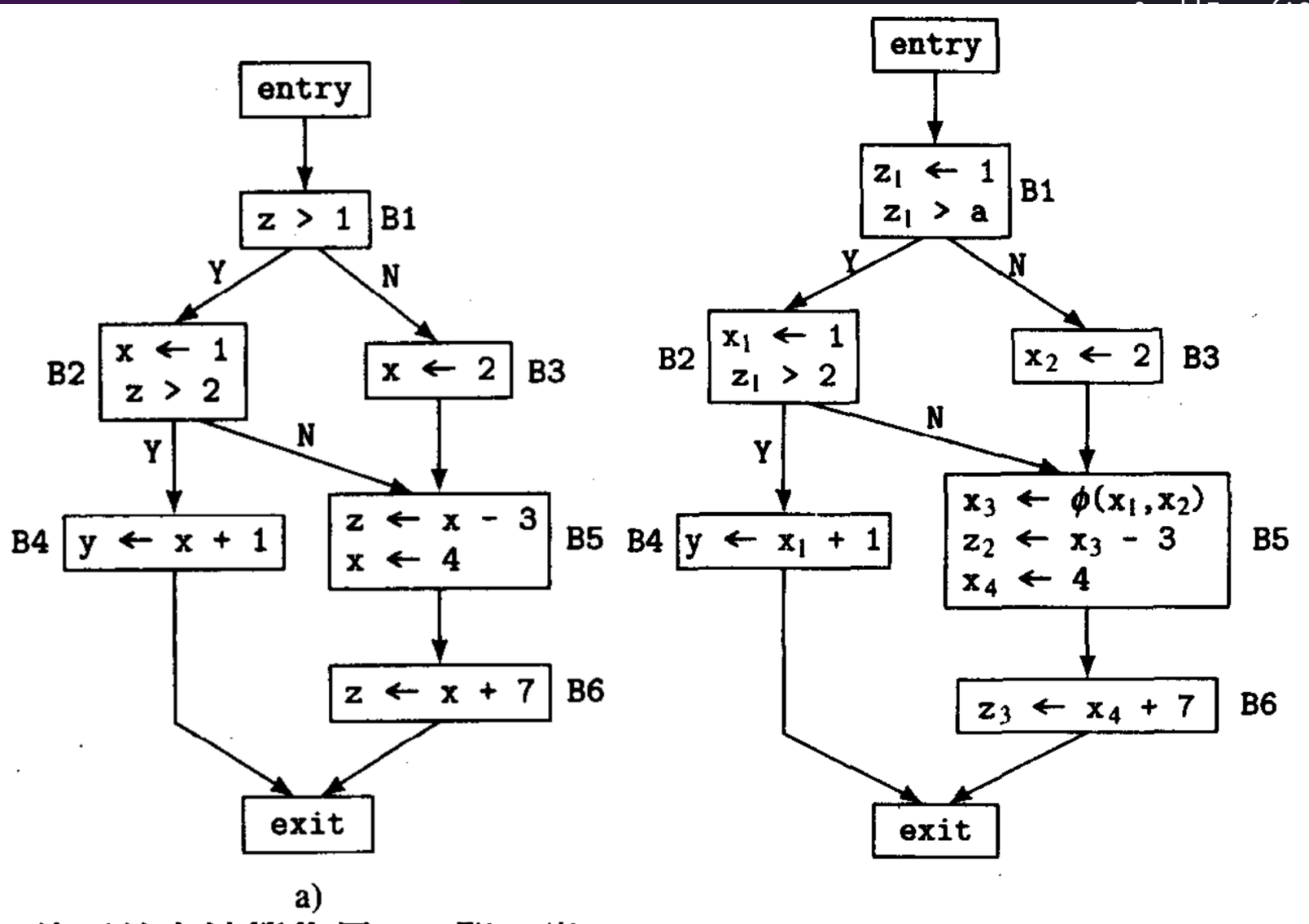
Mem2reg 算法将非 SSA 格式的中间代码转换成 SSA 格式。

```
label 1
$5 = $1
$6 = $2
$7 = $3
$8 = $4
$9 = 0
$10 = 0
$32768 = $9 < $5
br $32768, 2, 3 loop
label 2
$10 = 0
$32769 = $10 < $5
br $32769, 5, 6 loop
label 5
$32770 = $9 * 1024
$32770 = $32770 + $10
$32770 = $32770 * 4
$32771 = $8 + $32770
store 0 to $32771
$32772 = $10 + 1
$10 = $32772
...
```

```
fn $$5_mm(i32, i32*, i32*,
i32*) -> void {
bb1:    // preceding:
    %163 = $1
    %176 = $2
    %185 = $3
    %194 = $4
    %207 = 0
    %225 = 0
    %253 = %207 < %163
    br %253, 2, 3 loop
bb2:    // preceding: 1, 4
    %208 = phi [%207, %211]
    %227 = 0
    %255 = %227 < %163
    br %255, 5, 6 loop
bb3:    // preceding: 1, 4
    %213 = 0
    %232 = 0
    %243 = 0
    %271 = %243 < %163
    br %271, 8, 9 loop
...
bb1048576:
    $0 = 0
    ret $0
}
```

词/语法 MEM2RE

读入源代码
式的中间代
Mem2reg 算
中间代码转



, i32*, i32*,
{
ceding:

< %163
3 loop
ceding: 1, 4
[%207, %211]

< %163
6 loop
ceding: 1, 4

< %163
9 loop

OPTIMIZATIONS

我们做的优化。



优化总览

右侧展示了我们的编译管线中实际使用的优化。之后的页面将展示其中的一部分。

高亮的优化我们会重点讲解。

MIR OPTIMIZE

全局

常量传播
死代码删除
基本块合并
循环展开
高级死代码删除
循环不变量外提
基本块重排
图着色
常数组全局化

局部

函数内联
公共子表达式删除
内存变量传播
运算强度削弱
底层 IR 合并

LEX & PARSE

MEM2REG

MIR2ARM

ARM OPTIMIZE

条件执行
寄存器分配
窥孔优化
代码对齐

OUTPUT

中间代码优化

下面介绍的优化的对象都是 MIR

INLINE FUNCTION

函数内联。

将符合某些要求的函数体复制到被调用的地方，省去调用函数的开销。

```
fn $$5_func(i32*) -> i32 {  
  // -- snip --  
  bb1:    // preceding:  
    %2 = $1  
    %4 = &$$3_10  
    %5 = %4  
    %7 = load %5  
    %9 = 3 - %7  
    %11 = %9 * 4  
    %13 = %2 + %11  
    %0 = load %13  
    br 1048576  
  bb1048576: // preceding: 1  
    $0 = %0  
    ret $0  
}
```

```
fn $$5_main() -> i32 {  
  // -- snip --  
    %47 = %21  
    %49 = $$5_func(%47)  
    %51 = %49 + 2  
    %17 = %51 + %23  
  // -- snip --  
}
```

```
fn $$5_main() -> i32 {  
  // -- snip --  
    %47 = %21  
    %54 = %47  
    %55 = &$$3_10  
    %56 = %55  
    %57 = load %56  
    %58 = 3 - %57  
    %59 = %58 * 4  
    %60 = %54 + %59  
    %53 = load %60  
    br 4  
  bb4:    // preceding: 2  
    %52 = %53  
    %49 = %52  
    %51 = %49 + 2  
  // -- snip --  
}
```

CONSTANT PROPAGATION

常量传播。

将值为常数的变量替换成常数。

将参数都是常数的运算替换成运算的结果。

```
fn $$5_main() -> i32 {  
  // -- snip --  
bb0:  
  %1 = 1234  
  %2 = 3456  
  %3 = %1 + %2  
  %4 = 7890  
  %5 = %3 * %4  
  $0 = %5  
  ret $0  
}
```

>

```
fn $$5_main() -> i32 {  
  // -- snip --  
bb0:  
  $0 = 37004100  
  ret $0  
}
```

COMMON SUB-EXPR ELIMINATION

公共子表达式删除。

数组访问会在中间代码中产生重复计算偏移的冗余代码，源程序中也可能包含各类重复计算。

这个 pass 可以检测并删除上述冗余计算。

```
int arr[20][20]={};

// -- snip --

int my_func(...) {
    int i = 0, j;
    while (i < 15){
        j = 0;
        while (j < 15){
            arr[i][j] =
                arr[i][j] + i + j;
            j = j + 1;
        }
        i = i + 1;
    }
}
```

```
bb5:    // preceding: 2, 5
        %17 = phi [%16, %18]
        %26 = %9 * 20
        %27 = %26 + %17
        %28 = %27 * 4
        %30 = %2 + %28
        %32 = %9 * 20
        %33 = %32 + %17
        %34 = %33 * 4
        %36 = %2 + %34
        %38 = load %36
        %40 = %38 + %9
        %42 = %40 + %17
        store %42 to %30
        %18 = %17 + 1
        %46 = %18 < 15
        br %46, 5, 6 loop
```

MORE ON

COMMON SUB-EXPR ELIMINATION

通过节点表构造 DAG 图并重新导出，我们可以消除源程序及中间代码中的公共子表达式冗余。

```
bb5:    // preceding: 2, 5
        %17 = phi [%16, %18]
        %26 = %9 * 20
        %27 = %26 + %17
        %28 = %27 * 4
        %30 = %2 + %28
        %32 = %9 * 20
        %33 = %32 + %17
        %34 = %33 * 4
        %36 = %2 + %34
        %38 = load %36
        %40 = %38 + %9
        %42 = %40 + %17
        store %42 to %30
        %18 = %17 + 1
        %46 = %18 < 15
        br %46, 5, 6 loop
```



```
bb5:    // preceding: 2, 5
        %17 = phi [%16, %18]
        %26 = %9 * 20
        %27 = %26 + %17
        %28 = %27 * 4
        %30 = %2 + %28
        %38 = load %30
        %40 = %38 + %9
        %42 = %40 + %17
        store %42 to %30
        %18 = %17 + 1
        %46 = %18 < 15
        br %46, 5, 6 loop
```

MORE ON

COMMON SUB-EXPR ELIMINATION

在以上方法的基础上，我们还会将合并对纯函数的调用。

* 此处的纯函数，指的是返回值只与参数相关、没有副作用（不改变函数以外的东西）的函数。

```
bb26:    // preceding: 23, 26
        %232 = phi [%231, %233]
        %239 = %232
        %369 = $$5_getNumPos(%239, %141)
        %371 = %369 * 4
        %373 = %175 + %371
        %375 = load %373
        %377 = %375 * 4
        %379 = %154 + %377
        %233 = load %379
        %383 = $$5_getNumPos(%239, %141)
        %385 = %383 * 4
        %387 = %175 + %385
        %389 = load %387
        %391 = %389 * 4
        %393 = %154 + %391
        store %239 to %393
        %395 = $$5_getNumPos(%239, %141)
        %397 = %395 * 4
        %399 = %175 + %397
        %401 = $$5_getNumPos(%239, %141)
        %403 = %401 * 4
        %405 = %175 + %403
        %407 = load %405
        %409 = %407 + 1
        store %409 to %399
        %411 = $$5_getNumPos(%233, %141)
        %413 = %411 != %220
        br %413, 26, 27 loop
```

```
bb26:    // preceding: 23, 26
        %232 = phi [%231, %233]
        %239 = %232
        %369 = $$5_getNumPos(%239, %141)
        %371 = %369 * 4
        %373 = %175 + %371
        %375 = load %373
        %377 = %375 * 4
        %379 = %154 + %377
        %233 = load %379
        %389 = load %373
        %391 = %389 * 4
        %393 = %154 + %391
        store %239 to %393
        %407 = load %373
        %409 = %407 + 1
        store %409 to %373
        %411 = $$5_getNumPos(%233, %141)
        %413 = %411 != %220
        br %413, 26, 27 loop
```

MORE ON

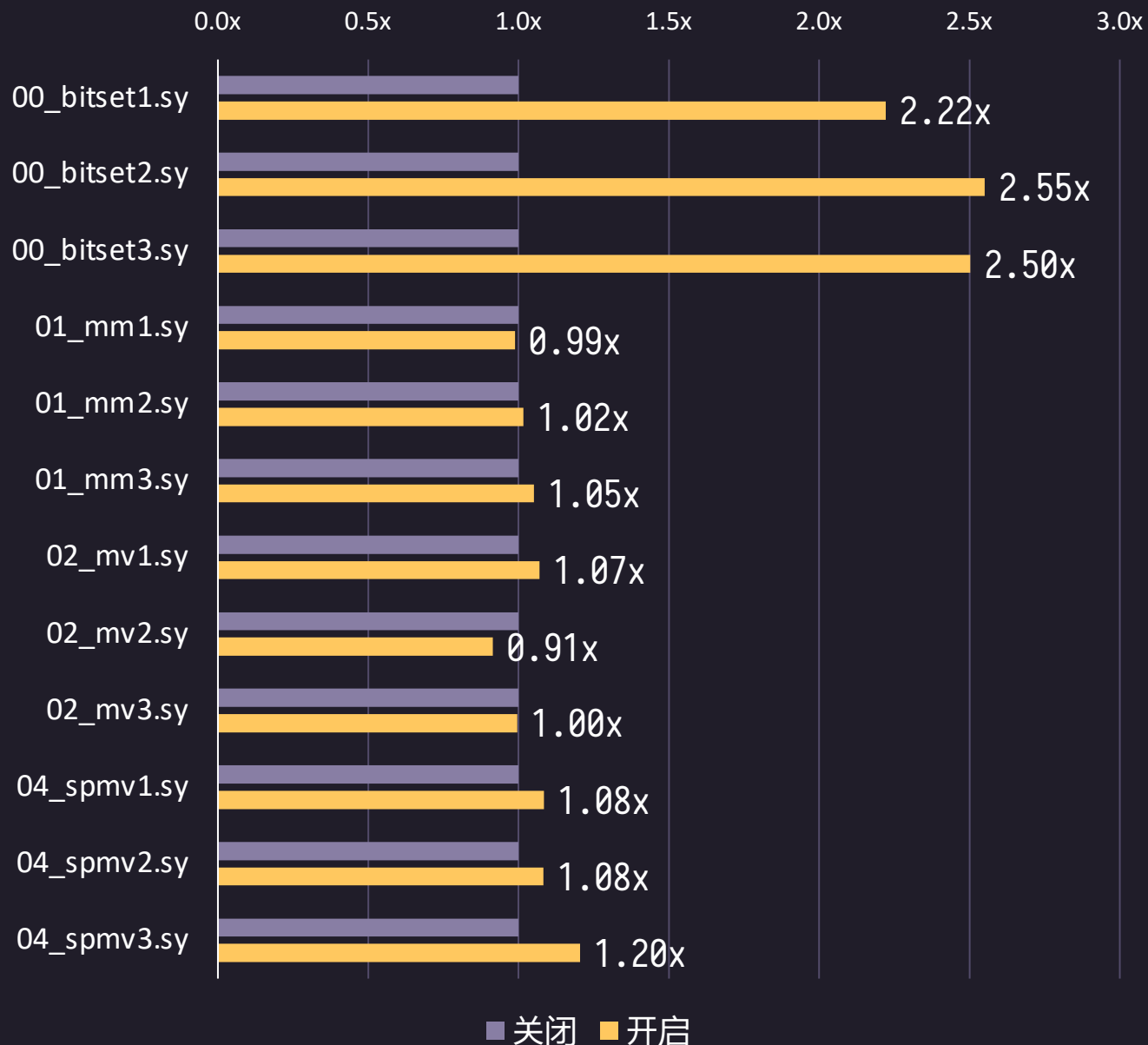
COMMON SUB-EXPR ELIMINATION

开关此优化的性能对比。

可以看到，经过公共子表达式删除，除了 mv 之外的测试程序均有不同程度的加速，部分用例速度甚至达到了之前的 2.5 倍。

样例中的 mv 和 mm 可能因为可合并的公共子表达式数量较少，没有出现明显提升。

(评测设备波动大约在 0.05-0.1x，下同)



DEAD CODE ELIMINATION

死代码删除。

输入的样例程序中会出现一些对返回值或输出没有贡献的代码。这个 pass 可以将这类代码删除掉。

这个 pass 分为基本版和高级版，对死代码的排查能力（以及消耗的时间）不同。

```
void mv(
    int n, int A[][N], int b[],
    int res[]
) {
    // -- snip --
    while (i < n) {
        j = 0;
        while (j < n) {
            if (A[i][j] == 0){
                x = x * b[i] + b[j];
                y = y - x;
            } else {
                // -- snip --
            }
            j = j + 1;
        }
        i = i + 1;
    }
}
```

```
fn $$5_mv(
    i32, i32*, i32*, i32*
) -> void {
    // -- snip --
bb11:    // preceding: 8
    %194 = load [%113, (%150, LSL 2)]
    %196 = %131 * %194
    %202 = load [%113, (%160, LSL 2)]
    %132 = %196 + %202
    %140 = %139 - %132
    br 13
bb12:    // preceding: 8
    // -- snip --
    br 13
bb13:    // preceding: 11, 12
    %141 = phi [%140, %139]
    %133 = phi [%132, %131]
    %162 = %160 + 1
    %238 = %162 < %98
    br %238, 8, 9 loop
    // -- snip --
}
```

MORE ON

COMPLEX DEAD CODE ELIMINATION

高级死代码删除。

类似于公共子表达式删除，在这个 pass 里，编译器扫描代码，产生变量之间的依赖图，同时确定哪些操作对返回值有直接影响或产生了副作用。

算法从副作用和返回值开始沿着依赖图搜索，将所有用到的变量标记。在搜索完之后，算法删除所有未标记变量。

```
fn $$5_mv(i32, i32*, i32*, i32*) -> void {  
    // -- snip --  
    bb5:    %150 = $2 ...  
    // -- snip --  
    bb8:    // preceding: 5, 13  
           %160 = $3 ...  
    // -- snip --  
           %186 = ...  
           %188 = %186 == 0  
           br %188, 11, 12 if_branch  
    X bb11: // preceding: 8  
           %194 = load [ %113 , (%150, LSL 2) ]  
           %196 = %131 * %194  
           %202 = load [ %113 , (%160, LSL 2) ]  
           %132 = %196 + %202  
           %140 = %139 - %132  
           br 13  
    bb12:   // preceding: 8  
           %216 = load [ %120 , (%150, LSL 2) ]  
           %230 = load [ %113 , (%160, LSL 2) ]  
           %232 = %186 * %230  
           %234 = %216 + %232  
           store %234 to [ %120 , (%150, LSL 2) ]  
           br 13  
    bb13:   // preceding: 11, 12  
           %141 = phi [ %140, %139 ]  
           %133 = phi [ %132, %131 ]  
           %162 = %160 + 1  
           %238 = %162 < %98  
           br %238, 8, 9 loop  
    bb1048576: // preceding: 6  
             ret void  
}
```

间接影响

直接影响

可能有副作用

```
fn $$5_mv(i32, i32*, i32*, i32*) -> void {  
    bb5:    // preceding: 3, 9  
           %150 = ...  
    // -- snip --  
    bb8:    // preceding: 5, 13  
           %160 = ...  
    // -- snip --  
           %186 = ...  
           %188 = %186 == 0  
           br %188, 13, 12 if_branch  
    // 注意这里 bb11 被完全删掉了  
    bb12:   // preceding: 8  
           %216 = load [ %120 , (%150, LSL 2) ]  
           %230 = load [ %113 , (%160, LSL 2) ]  
           %232 = %186 * %230  
           %234 = %216 + %232  
           store %234 to [ %120 , (%150, LSL 2) ]  
           br 13  
    bb13:   // preceding: 8, 12  
           %162 = %160 + 1  
           %238 = %162 < %98  
           br %238, 8, 9 loop  
    bb1048576: // preceding: 3, 9  
             ret void  
}
```

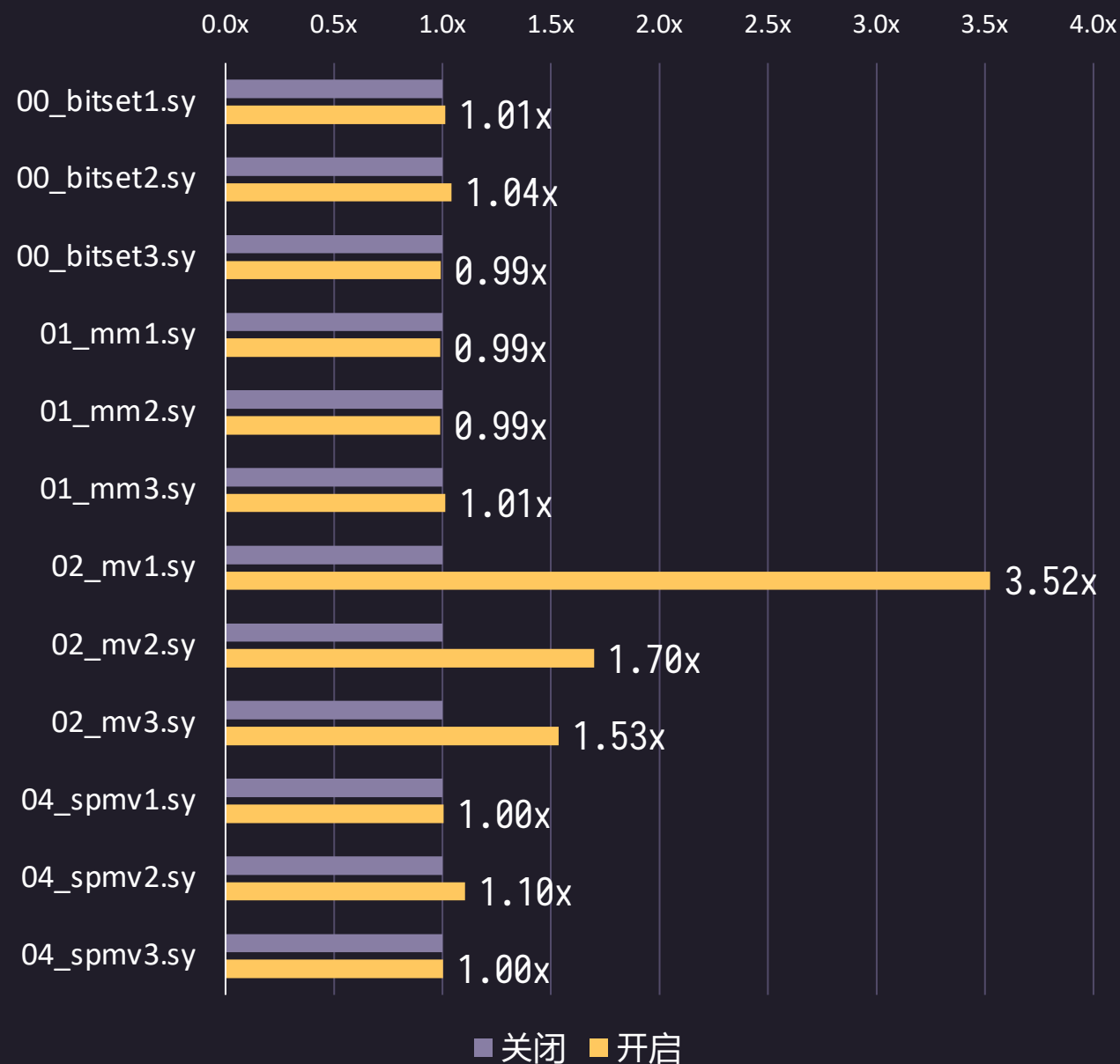
MORE ON

COMPLEX DEAD CODE ELIMINATION

开关本优化的性能对比。

高级死代码删除主要删除了 `mv` 用例中对结果没有影响的变量 `x` 和 `y`（上文例子），因此为这个用例带来了很大的提升。

其他用例中的死代码没有复杂到需要本优化处理，因此性能差别不大。



LOOP UNROLL

循环展开。

将简单的循环（包括常量和变量界限的）进行展开。被展开的循环体中通常含有冗余代码，之后的 pass 会将这些冗余消除掉，从而使得运行速度更快。

展开循环体非常小的循环也可以减少跳转次数，提高速度。

```
bb2:    // preceding: 1, 2
        %134 = phi [%133, %135]
        %135 = %134 + 1
        %234 = %135 * 4
        %236 = %123 + %234
        %238 = %135 - 1
        %240 = %238 * 4
        %242 = %123 + %240
        %244 = load %242
        %246 = %244 * 2
        store %246 to %236
        %248 = %135 < 30
        br %248, 2, 3 loop
```

```
bb2:
        %359 = %133 + 1
        %360 = %359 * 4
        %361 = %123 + %360
        %362 = %359 - 1
        %363 = %362 * 4
        %364 = %123 + %363
        %365 = load %364
        %366 = %365 * 2
        store %366 to %361
        %367 = %359 < 30
        %368 = %359 + 1
        %369 = %368 * 4
        %370 = %123 + %369
        %371 = %368 - 1
        %372 = %371 * 4
        %373 = %123 + %372
        %374 = load %373
        %375 = %374 * 2
        store %375 to %370
        %376 = %368 < 30
        %377 = %368 + 1
        %378 = %377 * 4
        %379 = %123 + %378
        %380 = %377 - 1
        %381 = %380 * 4
        %382 = %123 + %381
        %383 = load %382
        // .....
```

MEMORY VARIABLE PROPAGATION

内存变量传播。

对于刚写入全局变量或数组后又用到该全局变量或数组值的情况，可以不用重新 Load 而是直接使用 Store 前的变量。

```
int func(int i) {  
    a[1] = 2;  
    putint(a[1]);  
    a[i] = 4;  
    return (a[1]);  
}
```

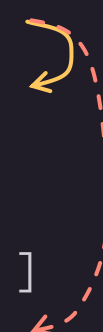
```
bb2:    // preceding:  
        %63 = 0  
        %66 = %63  
        %67 = &@$$3_10  
        %68 = %67 + 4  
        store 2 to [ %67 , 4 ]  
        %71 = load [ %67 , 4 ]  
        %77 = putint(%71)  
        %72 = %66 * 4  
        %74 = %67 + %72  
        store 4 to [ %67 , %72 ]  
        %65 = load [ %67 , 4 ]  
        %64 = %65  
        br 1048576
```

MORE ON


MEMORY VARIABLE PROPAGATION

如果 Load 和 Store 之间有对于同一地址的存操作或者调用了对内存有更改的函数，则不能传播，否则正确性将无法保障。

```
bb2:    // preceding:
        %63 = 0
        %66 = %63
        %67 = &@$$3_10
        %68 = %67 + 4
        store 2 to [ %67 , 4 ]
        %71 = load [ %67 , 4 ]
        %77 = putint(%71)
        %72 = %66 * 4
        %74 = %67 + %72
        store 4 to [ %67 , %72 ]
        %65 = load [ %67 , 4 ]
        %64 = %65
        br 1048576
```



```
bb2:    // preceding:
        %63 = 0
        %66 = %63
        %67 = &@$$3_10
        %68 = %67 + 4
        store 2 to [ %67 , 4 ]
        %77 = putint(2)
        %72 = %66 * 4
        %74 = %67 + %72
        store 4 to [ %67 , %72 ]
        %65 = load [ %67 , 4 ]
        %64 = %65
        br 1048576
```



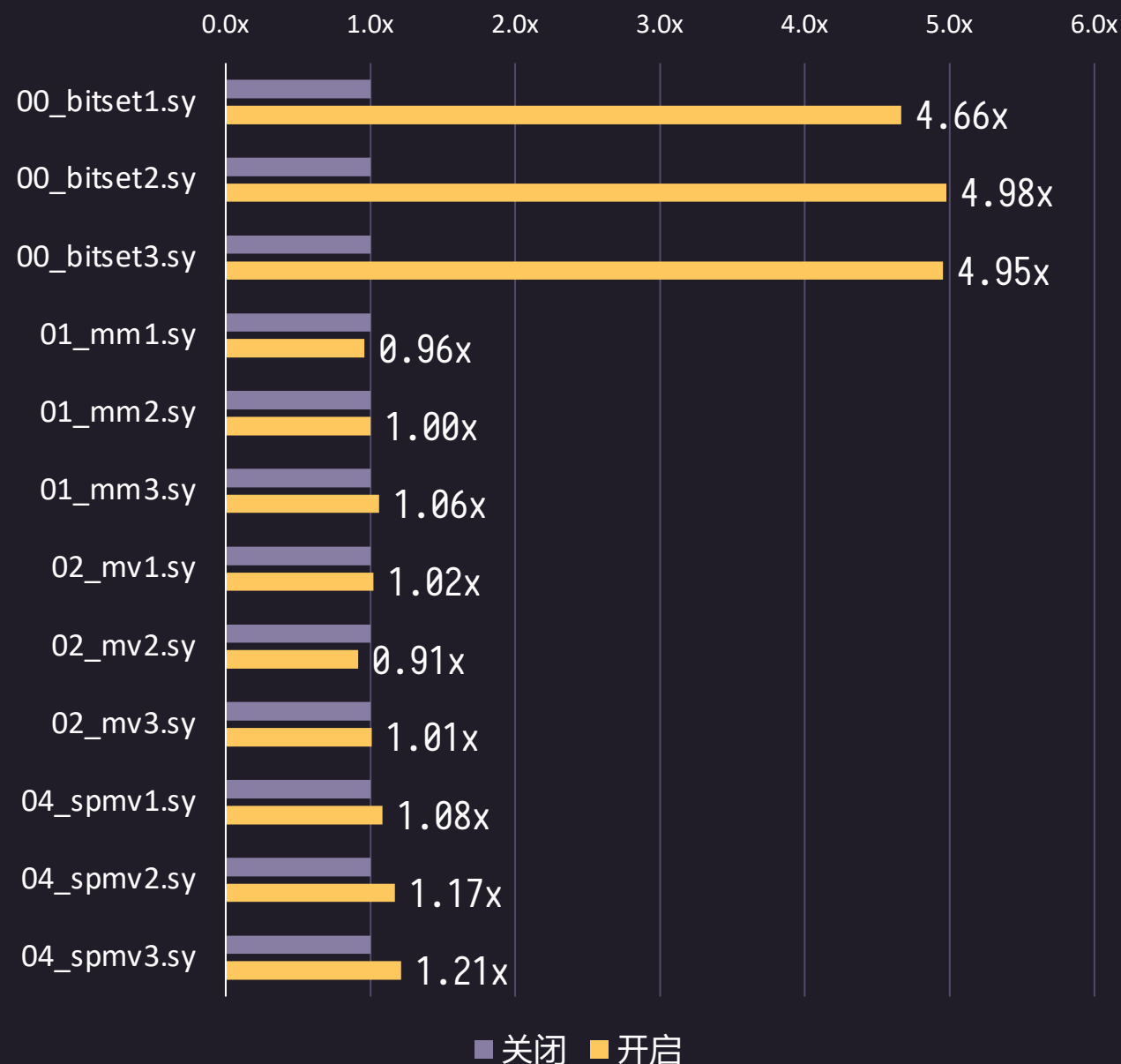
MORE ON

MEMORY VARIABLE PROPAGATION

开关本优化的性能差别。

性能提升最大的是 bitset，因为循环体内有一个非常大的数组初始化部分。内存变量传播可以将其直接优化成常量数组。

其他测试程序因为减少了内存存取，基本也有一定的提升。



ALGEBRAIC SIMPLIFICATION

运算强度削弱。

将常量乘法、除法、模除等大运算量操作替换为等价的移位、加法、乘法等操作的组合。

乘、除以、模 2 的幂的操作被优化为左移、右移和按位与。

非 2 的幂的数字的除法和模除按照论文中的算法转化为乘法、移位和加减法。

* 使用的算法来自 O. Traub - Division by invariant integers using multiplication.

```
%43 = load [ %44 , 0 ]
%42 = %43
%21 = %42
%7 = %21 % 300000
%66 = %43 * 19971231
%67 = %66 + 19981013
%71 = %67 % 1000000007
store %71 to [ %44 , 0 ]
```

```
%43 = load [ %44 , 0 ]
%42 = %43
%21 = %42
%86 = -541967606 MulSh %21
%87 = %86 + %21
%88 = %87 >> 18
%89 = %21 >> 31
%85 = %88 - %89
%90 = %85 * 300000
%7 = %21 - %90
%66 = %43 * 19971231
%67 = %66 + 19981013
%92 = -1989124302 MulSh %67
%93 = %92 + %67
%94 = %93 >> 29
%95 = %67 >> 31
%91 = %94 - %95
%96 = %91 * 1000000007
%71 = %67 - %96
store %71 to [ %44 , 0 ]
```


MORE ON

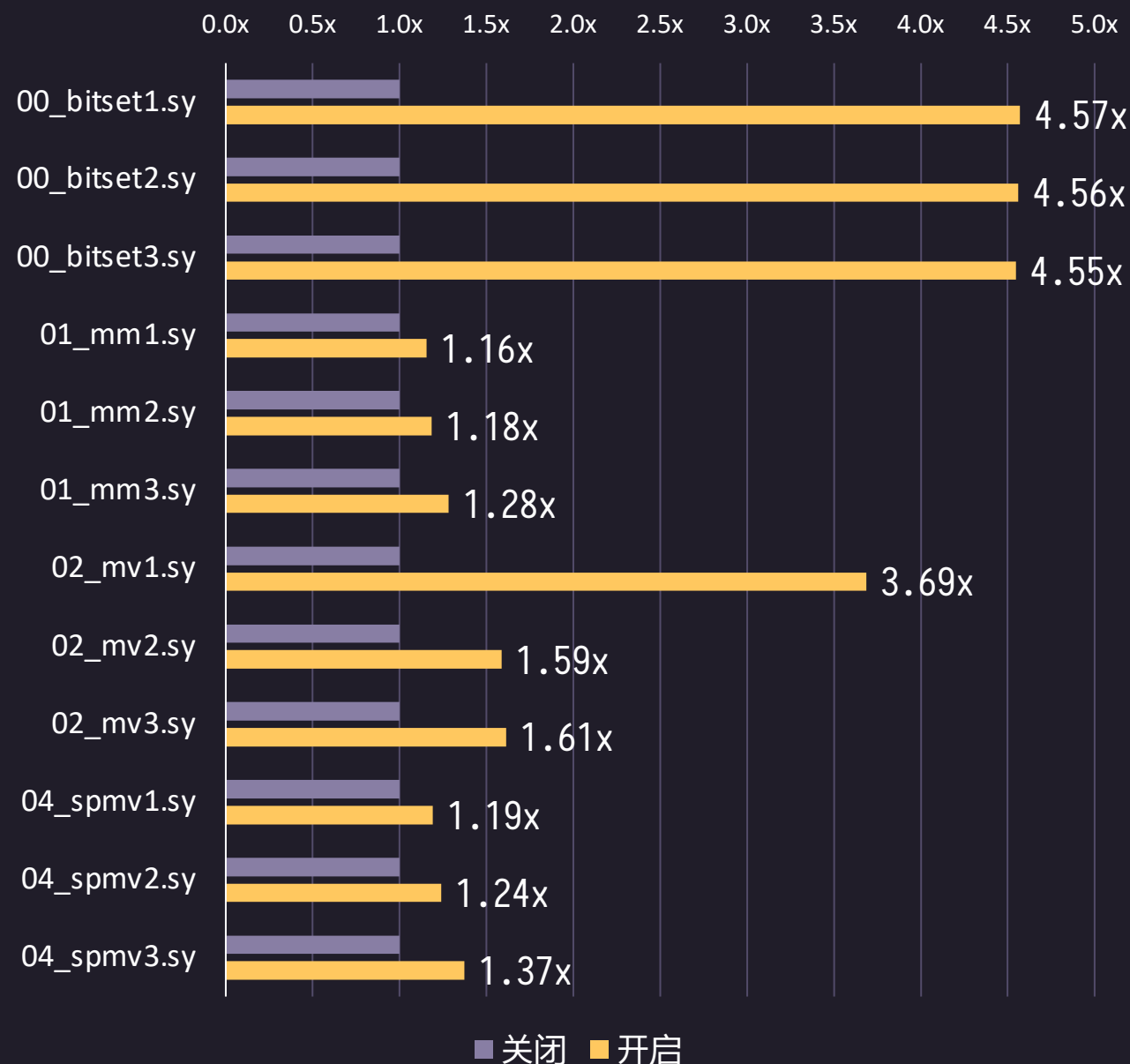
ALGEBRAIC SIMPLIFICATION

开关本优化的性能差别。

这个优化将乘除法优化成移位，使得内存偏移的移位可以嵌入 ARM 的操作数中，有效提高了大部分用例的性能。

在 bitset 中，循环内的常数模除从函数调用优化成了乘法。

在 mv 中，缩短的基本块使得条件执行成为可能。



LOOP INVARIANT EXTRACTION

循环不变量外提。

循环中有一些计算结果是不被循环改变的，这些计算可以提取到循环外进行。

因为提取不变量会将局部变量更改为全局变量进而增加寄存器分配的压力，所以仅当不变量有三行及以上时才进行提取。

```
bb9: // preceding: 5, 8
    %67 = load [ %9 , (%54, LSL 2) ]
    %49 = %54 + 1
    %55 = load [ %9 , (%49, LSL 2) ]
    %57 = %67 < %55
    br %57, 11, 12 loop
bb11: // preceding: 9, 11
    %68 = phi [%67, %69]
    %63 = load [ %17 , (%68, LSL 2) ]
    %79 = load [ %39 , (%63, LSL 2) ]
    %85 = load [ %25 , (%68, LSL 2) ]
    %91 = load [ %33 , (%54, LSL 2) ]
    %93 = %91 - 1
    %95 = %85 * %93
    %97 = %79 + %95
    store %97 to [ %39 , (%63, LSL 2) ]
    %69 = %68 + 1
    %1 = %54 + 1
    %7 = load [ %9 , (%1, LSL 2) ]
    %19 = %69 < %7
    br %19, 11, 12 loop
bb12: // preceding: 9, 11
    %59 = %54 + 1
    %13 = %59 < %3
    br %13, 5, 1048576 loop
```

```
bb9: // preceding: 5, 8
    %67 = load [ %9 , (%54, LSL 2) ]
    %49 = %54 + 1
    %55 = load [ %9 , (%49, LSL 2) ]
    %57 = %67 < %55
    br %57, 17, 12 loop
bb17: // preceding: 9
    %91 = load [ %33 , (%54, LSL 2) ]
    %93 = %91 - 1
    %1 = %54 + 1
    %7 = load [ %9 , (%1, LSL 2) ]
    br 11
bb11: // preceding: 11, 17
    %68 = phi [%67, %69]
    %63 = load [ %17 , (%68, LSL 2) ]
    %79 = load [ %39 , (%63, LSL 2) ]
    %85 = load [ %25 , (%68, LSL 2) ]
    %95 = %85 * %93
    %97 = %79 + %95
    store %97 to [ %39 , (%63, LSL 2) ]
    %69 = %68 + 1
    %19 = %69 < %7
    br %19, 11, 12 loop
bb12: // preceding: 9, 11
    %59 = %54 + 1
    %13 = %59 < %3
    br %13, 5, 1048576 loop
```

CONST ARRAY GLOBALIZE

常数组全局化。

将代码中出现的局部常量数组
(不管是直接初始化还是挨个
元素赋值) 转换为全局数组，
省去每次调用函数的初始化时
间。

本优化针对 bitset 用例。

```
%123 = &$7
store 1 to [ %123 , 0 ]
store 2 to [ %123 , 4 ]
store 4 to [ %123 , 8 ]
store 8 to [ %123 , 12 ]
store 16 to [ %123 , 16 ]
store 32 to [ %123 , 20 ]
store 64 to [ %123 , 24 ]
store 128 to [ %123 , 28 ]
store 256 to [ %123 , 32 ]
store 512 to [ %123 , 36 ]
store 1024 to [ %123 , 40 ]
store 2048 to [ %123 , 44 ]
store 4096 to [ %123 , 48 ]
store 8192 to [ %123 , 52 ]
// -- continues --
```

```
    $$6_$$5_set_7:
        .word 1, 2, 4, 8, 16, 32,
        64, 128, 256, 512, 1024, 2048,
        // -- snip --

%123 = &@$$6_$$5_set_7
```

ARM 代码优化

下面介绍的优化的对象都是 ARM 汇编



CONDITIONAL EXECUTION

条件执行。

利用 ARM 的体系结构特点，将小的基本块替换为条件执行语句，减少分支预测错误造成的性能损耗。

```
.bb_f__mv$8:
    mov r0, #2010
    mla r0, r9, r0, r8
    ldr r10, [r6, r0, LSL #2]
    cmp r10, #0
    beq .bb_f__mv$13
.bb_f__mv$12:
    ldr r0, [r4, r9, LSL #2]
    ldr r1, [r5, r8, LSL #2]
    mla r0, r10, r1, r0
    str r0, [r4, r9, LSL #2]
.bb_f__mv$13:
    add r8, r8, #1
    mov r8, r8
    cmp r8, r7
    bge .bb_f__mv$9
    b .bb_f__mv$8
```



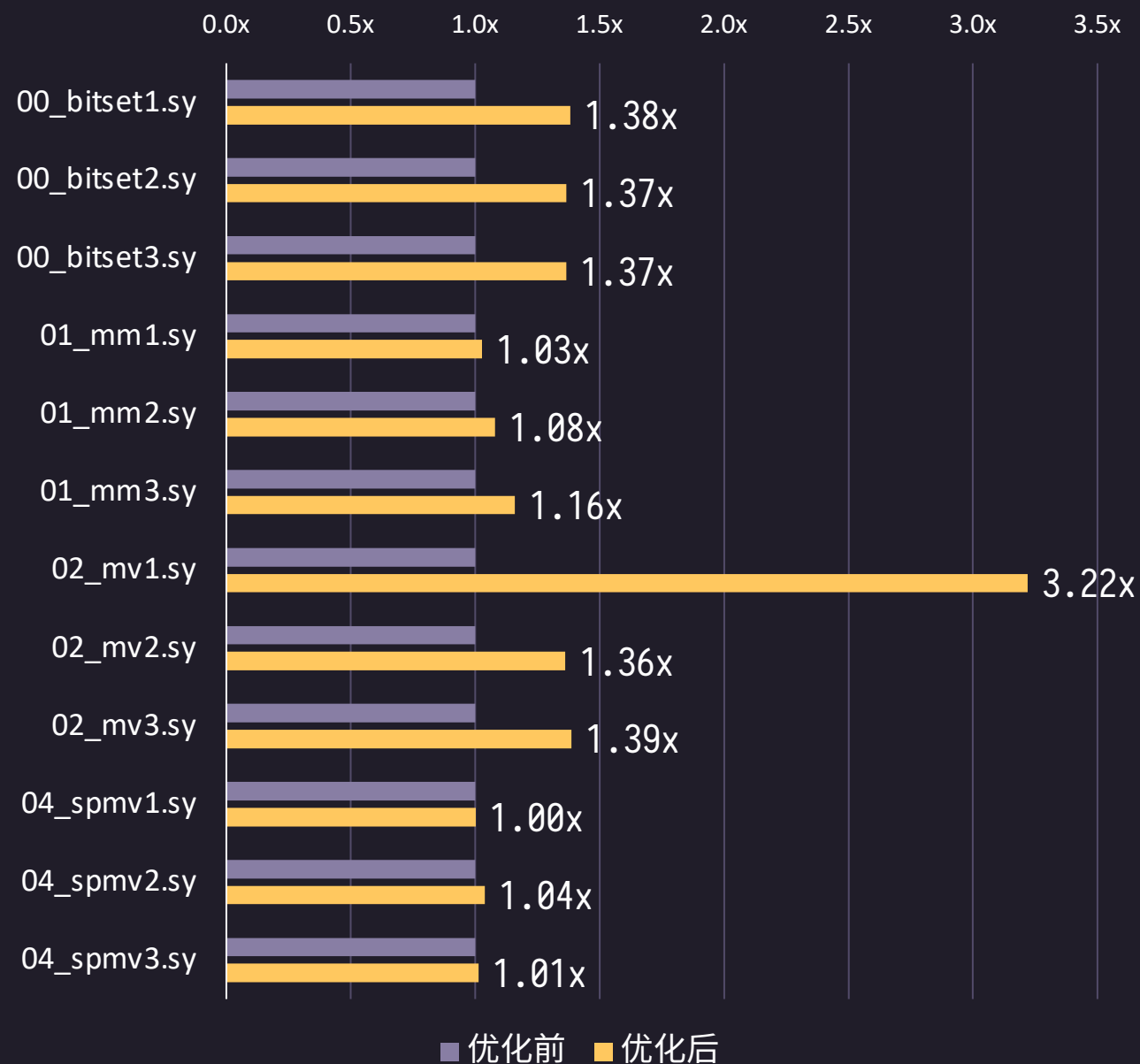
```
.bb_f__mv$8:
    mov r0, #2010
    mla r0, r9, r0, r8
    ldr r10, [r6, r0, LSL #2]
    cmp r10, #0
.bb_f__mv$12:
    ldrne r0, [r4, r9, LSL #2]
    ldrne r1, [r5, r8, LSL #2]
    mlane r0, r10, r1, r0
    strne r0, [r4, r9, LSL #2]
.bb_f__mv$13:
    add r8, r8, #1
    mov r8, r8
    cmp r8, r7
    bge .bb_f__mv$9
    b .bb_f__mv$8
```

CONDITIONAL EXECUTION

开关条件执行优化的速度对比。

条件执行对于条件分支两侧使用频率相近的样例有着很好的优化效果，比如 mv 和 bitset。

Cortex-A72 有 15 级流水线，一次分支预测错误惩罚约 10 时钟周期。对于指令少但时钟周期多的代码 yz'hx'xn'go



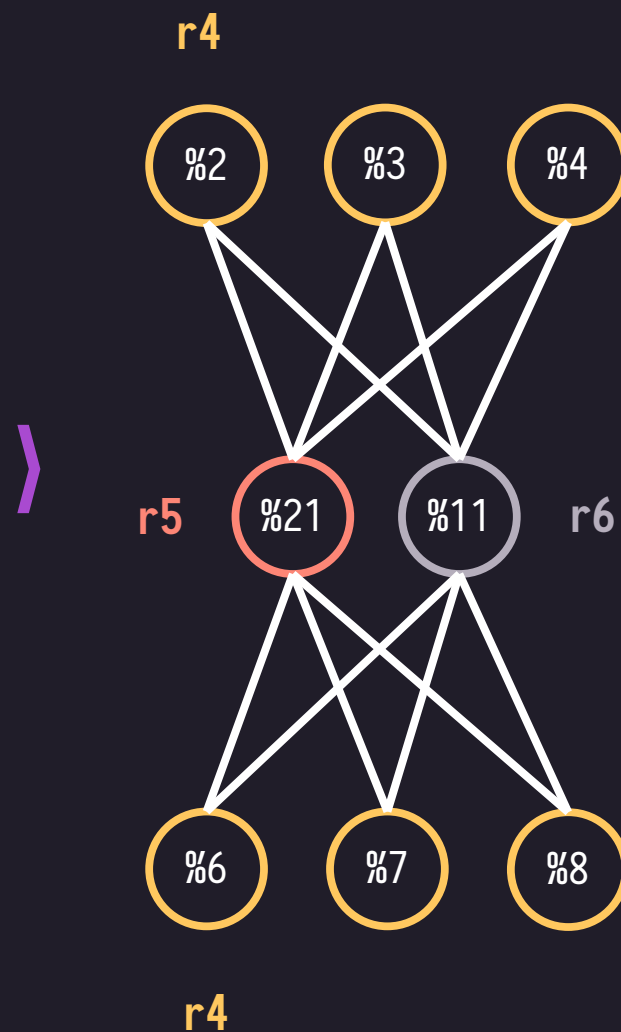
GRAPH COLORING

图着色。

寄存器分配的第一步，全局
(跨块) 寄存器分配。

根据引用计数的方法设置权重，
构建冲突图并着色。

```
bb1:
    %2 = 0
    %11 = getint()
    %21 = &@$$0_0
    %19 = %2 < %11
    br %19, 2, 3 loop
bb2:
    %3 = phi [%2,%4]
    %37 = putf(%21,%3)
    %38 = putch(10)
    %4 = %3 + 1
    %25 = %4 < %11
    br %25, 2, 3 loop
bb3:
    %6 = 0
    %27 = %6 < %11
    br %27, 5, 6 loop
bb5:
    %7 = phi [%6, %8]
    %39 = putf(%21, %7)
    %40 = putch(10)
    %8 = %7 + 1
    %33 = %8 < %11
    br %33, 5, 6 loop
bb6 :
    %0 = 0
    $0 = %0
    ret $0
```



REGISTER ALLOCATION

寄存器分配。

寄存器分配的第二步，线性扫描寄存器分配。同时将第一步中的着色结果应用到实际代码中。

```
.bb_$$5_get$72:  
    mov v91, #110  
    mul v90, v84, v91  
    add v92, v90, v88  
    lsl v93, v92, #2  
    ldr v94, [v5, v93]  
    str v94, [v8, v93]  
    add v95, v88, #1  
    sub v96, v3, #1  
    mov v88, v95  
    cmp v95, v96  
    bgt .bb_$$5_get$73  
    b .bb_$$5_get$72
```



```
.bb_$$5_get$72:  
    mov r0, #110  
    mul r0, r5, r0  
    add r0, r0, r4  
    lsl r0, r0, #2  
    ldr r1, [r8, r0]  
    ldr r2, [sp, #24]  
    str r1, [r2, r0]  
    add r4, r4, #1  
    sub r0, r9, #1  
    mov r4, r4  
    cmp r4, r0  
    bgt .bb_$$5_get$73  
    b .bb_$$5_get$72
```


PEEP HOLE OPTIMIZATION

窥孔优化。

消除掉一些因为其他原因（比如寄存器分配时给不同变量分配了同一寄存器）生成的显然无用的代码。

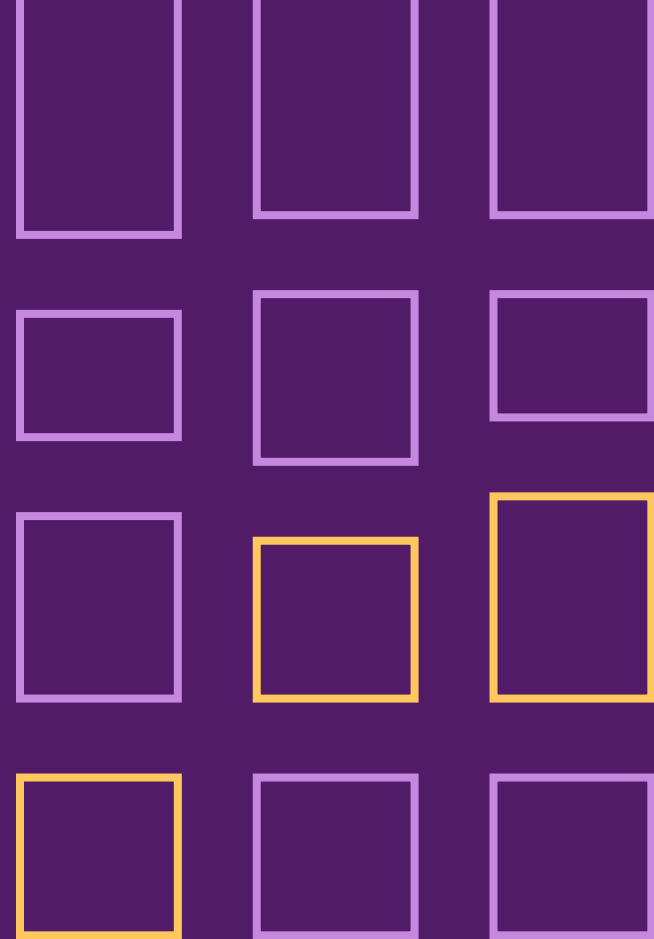
```
mov r4, #0
mov r0, #34953
movt r0, #34952
smmul r0, r8, r0
add r0, r0, r8
asr r0, r0, #4
asr r1, r8, #31
sub r5, r0, r1
mov r4, r4
mov r4, r4
mov r4, r4
mov r0, #10000
cmp r5, r0
blt .bb_$$5_set$6
b .bb_$$5_set$5
.bb_$$5_set$5:
```



```
mov r4, #0
mov r0, #34953
movt r0, #34952
smmul r0, r8, r0
add r0, r0, r8
asr r0, r0, #4
asr r1, r8, #31
sub r5, r0, r1
mov r0, #10000
cmp r5, r0
blt .bb_$$5_set$6
.bb_$$5_set$5:
```

CONCLUSION

总结，还有展望。



性能水平

我们的编译器和 GCC / Clang 比起来速度怎么样？

.....还行吧。

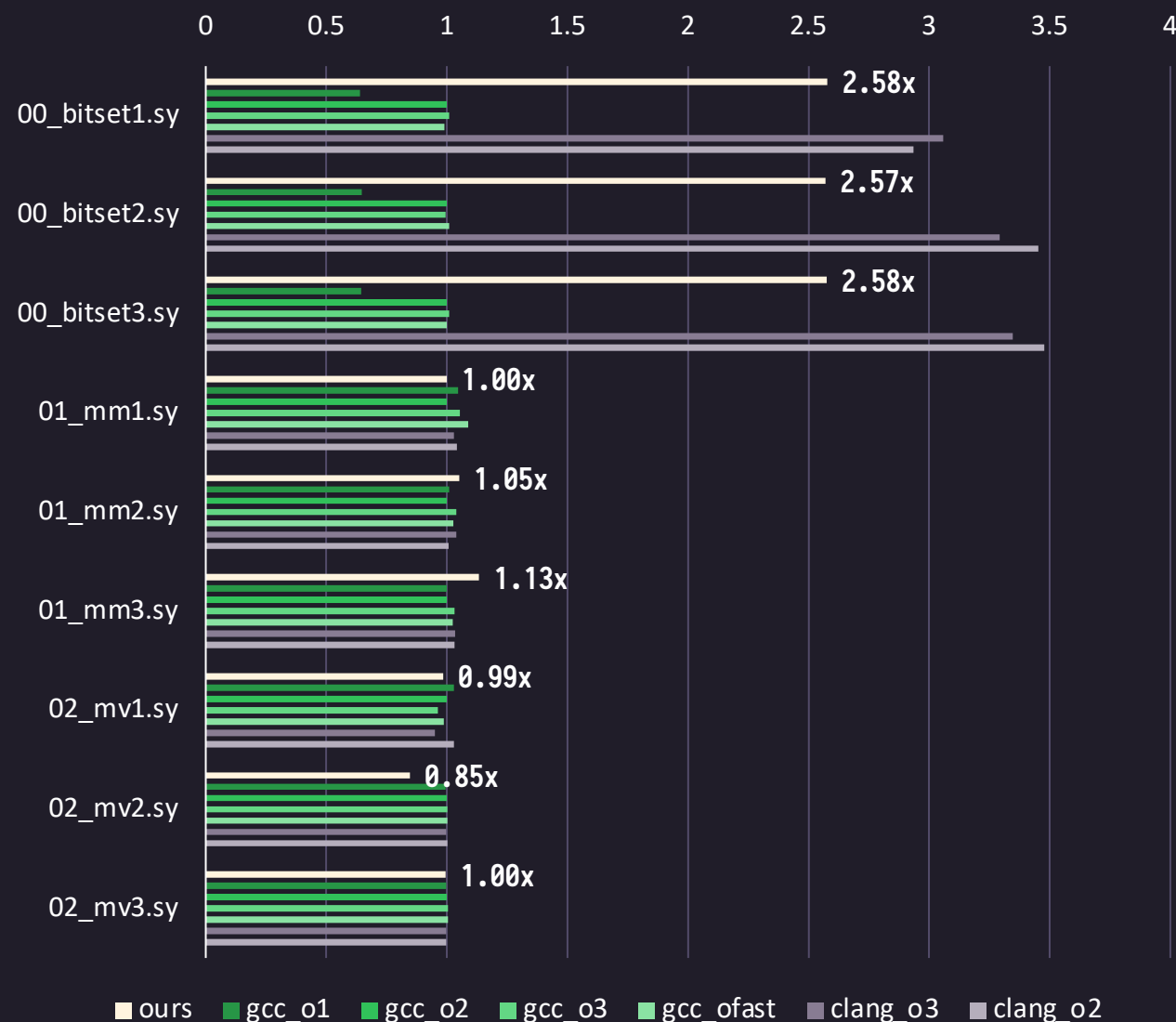
gcc_o1/2/3/fast:

```
$ gcc -march=armv7-a+neon-vfpv4 -mcpu=cortex-a7 -mfpu=neon -std=c11 -O1/2/3/fast
```

clang_o2/3:

```
$ clang -march=armv7-a+neon-vfpv4 -mcpu=cortex-a7 -mfpu=neon -std=c11 -O2/3
```

相对 GCC O2 的运行速度 (越高越好)



我们想做但是没来得及做的东西

- 自动向量化 (SIMD)

参考文献

在编写编译器的时候，我们参考了这些资料。

1. Traub, O., Holloway, G., & Smith, M. D. (1998). Quality and speed in linear-scan register allocation. ACM SIGPLAN Notices, 33(5), 142-151.
2. Granlund, T., & Montgomery, P. L. (1994, June). Division by invariant integers using multiplication. In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (pp. 61-72).
3. Mössenböck, H., & Pfeiffer, M. (2002, April). Linear scan register allocation in the context of SSA form and register constraints. In International Conference on Compiler Construction (pp. 229-246). Springer, Berlin, Heidelberg.

谢谢大家

2020. Team SEGVOL @ BUAA.