AMPERE™

AArch64 performance analysis and resulted enhancements on GCC

Feng Xue, Jiangning Liu

November 23, 2019

Agenda

- Loop split on semi-invariant conditional statement
- IPA constant propagation and recursive function versioning
- Some issues in current register allocator
- Trapless conditional selection instruction generation

Loop conditional statement elimination

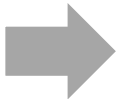
- Loop Split

```
for (i = 0; i < 100; i++) {
  if (i < 40)
    S1;
  else
    S2;
}
```

```
for (i = 0; i < 40; i++)
  S1;
for (i = 40; i < 100; i++)
  S2;
```

- Loop Unswitch

```
for (i = 0; i < 100; i++)
{
  if (a != b)
    S1;
  else
    S2;
}
```



```
if (a != b) {
  for (i = 0; i < 100; i++)
    S1;
}
else {
  for (i = 0; i < 100; i++)
    S2;
}
```

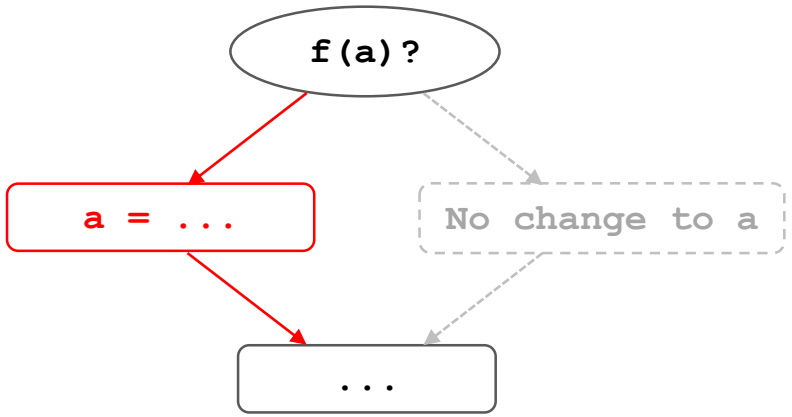
Loop semi-invariant conditional statement

- Loop invariant condition?

```
extern int flag;

for (i = 0; i < 100; i++) {
  if (flag)
    printf(…);
}
```

- Simple semi-invariant pattern




```
for (i = 0; i < 100; i++) {
  if (a < 10)
    a = new_value ();
}
```

How to eliminate semi-invariant condition?

- Loop Unswitch

```
if (flag) {
  for (i = 0; i < 100; i++) {
    if (flag)
      printf(…);
    S1;
  }
}
else {
  for (i = 0; i < 100; i++) {
    S1;
}
```

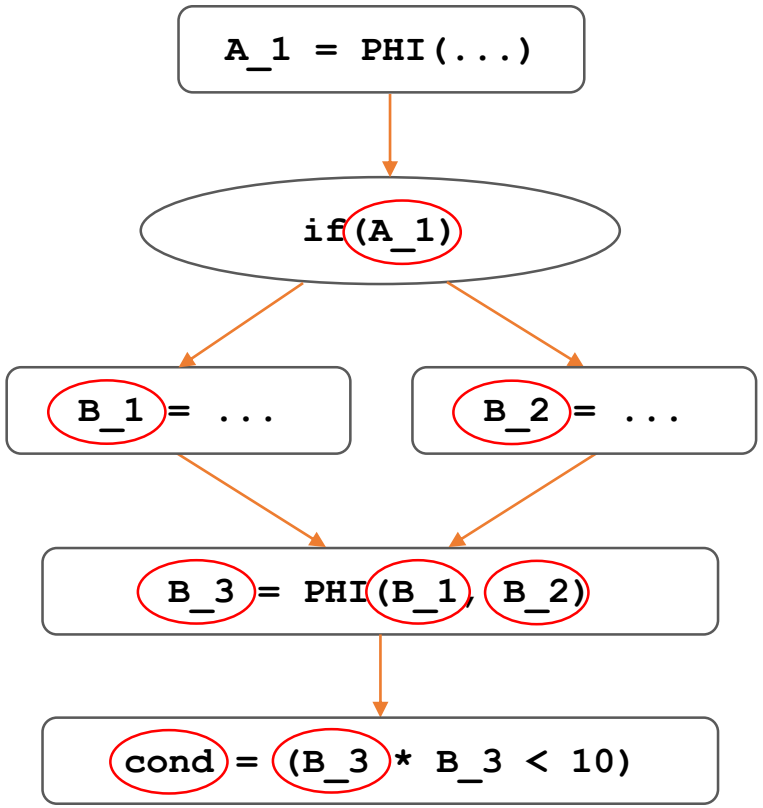
- Loop Split

```
for (i = 0; i < 100; i++) {
  if (flag)
    printf(…);
  else {
    S1;
    i++;
    break;
  }
}
for (; i < 100; i++)
  S1;
```

Identify semi-invariant condition

- Conditional expression tree evaluation
  - Normal value operation
  - SSA-PHI merge operation

```
foo(int p, int q, int r) {
  a = r;
  for (i = 0; i < 100; i++) {
    if (a)
      b = q;
    else
      b = p;
    if (b * b < 10)
      a = new_value();
  }
}
```

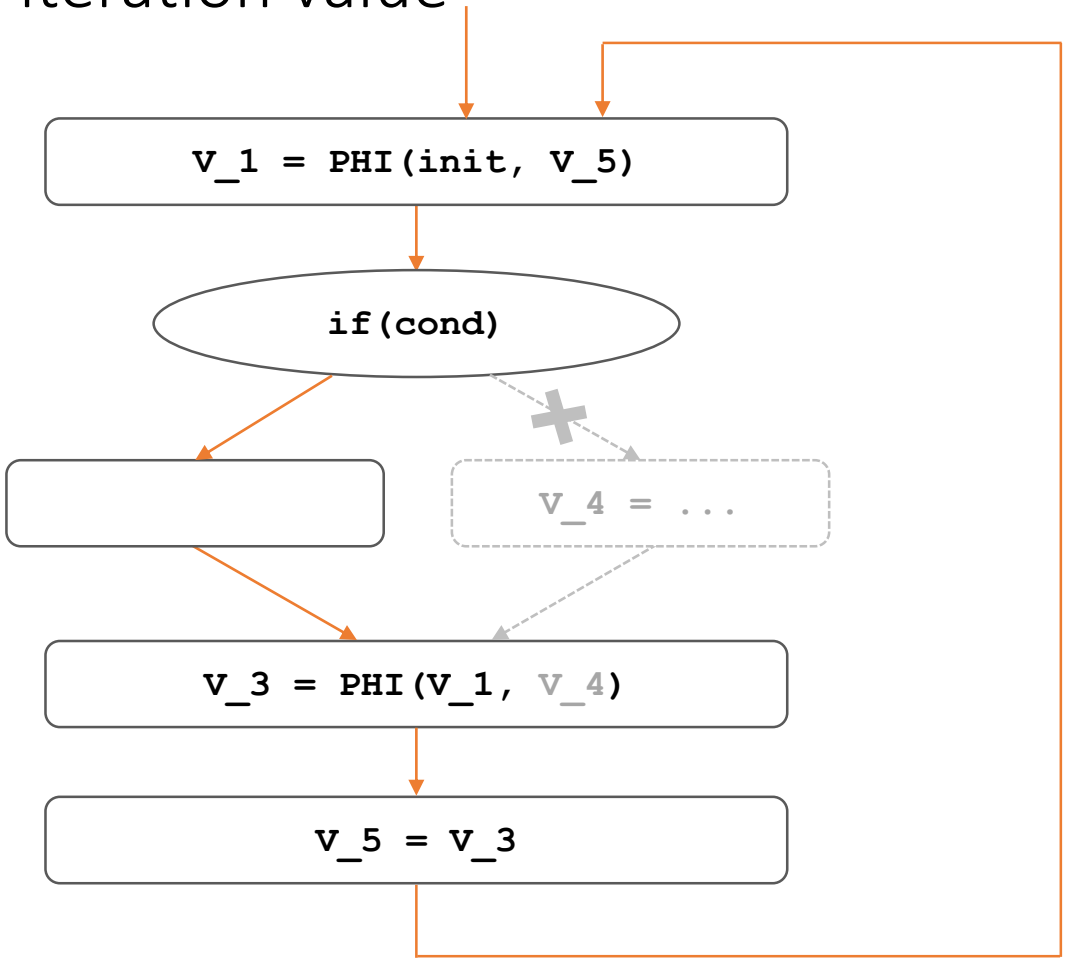


Both value expression and the condition that it control-depends on should be semi-invariant.

AMPERE.

Identify semi-invariant condition

- Semi-invariant loop iteration value



IPA constant propagation

- Jump function

```
f(int a, int b) {
  g(b, 3, -a, a + 1);
}
JF{f->g}[0] = param#1
JF{f->g}[1] = 3
JF{f->g}[2] = -param#0
JF{f->g}[3] = param#0 + 1
```

- In-memory constant

```
f() {
  int a = 1;
  struct {f0, f1} b = {2, 3};
  g(&a, b);
}
JF_agg{f->g}[0, @0] = 1
JF_agg{f->g}[1, @0] = 2
JF_agg{f->g}[1, @4] = 3
```

IPA constant propagation

- Parameter passing in FORTRAN

```
subroutine f(a)
integer, intent(in) a
  call g(a + 1)
end subroutine
```

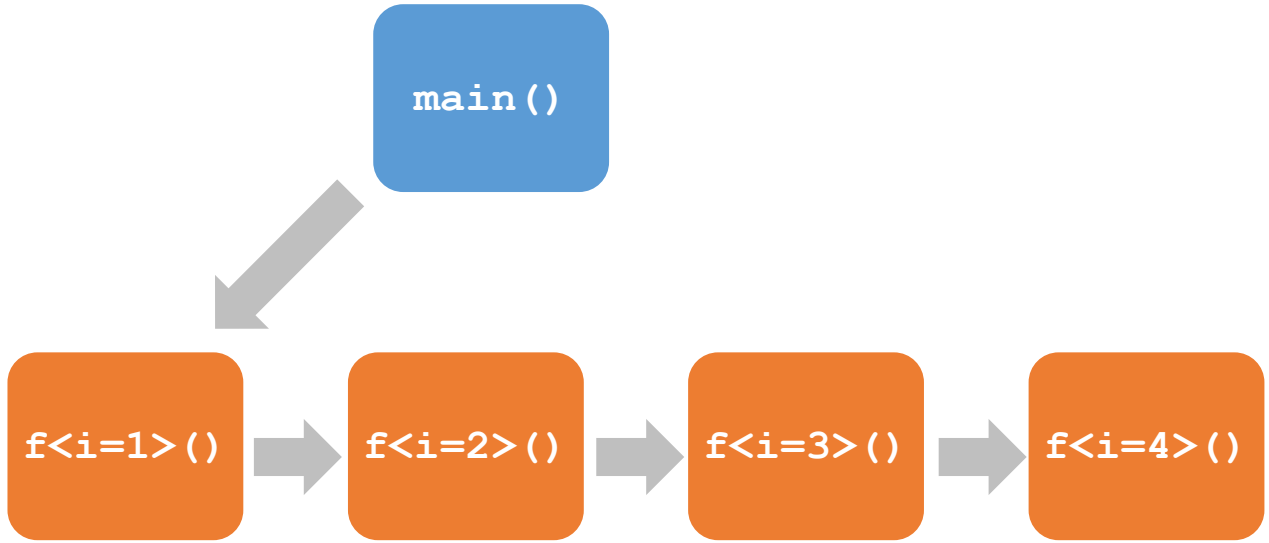
```
f(int *a) {
  int t = *a + 1;
  g(&t)
}
```

- Enhanced in-memory constant propagation
  - **JF_agg[i, @offset] = constant**
  - JF_agg[i, @offset] = param#j OP constant
  - JF_agg[i, @offset] = *(param#j + offset2) OP constant

Recursive function optimizations

```
f(int i) {
  if (i == 4) {
    do_work();
    return;
  }
  do_prepare();
  f(i + 1);
  do_post();
}
main() {
  f(1);
}
```

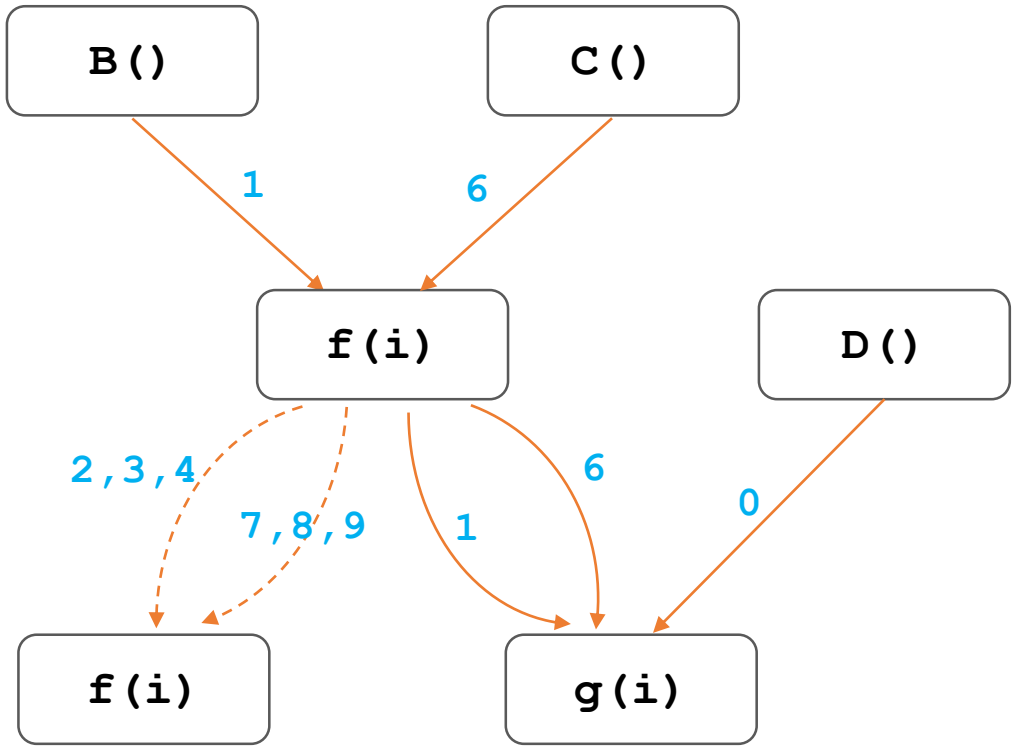
- Recursive tail call transformation
- Recursive inlining
- Recursive versioning



Recursive function versioning

- Only for self-recursive function
- New option for recursive versioning depth
- Recursive constant propagation strategy

```
f(int i) {
  g(i);
  f(i + 1);
}
B() { f(1); }
C() { f(6); }
D() { g(0); }
```



Versioning depth is supposed to be 4.

AMPERE.

IPA constant propagation TODOs

- Global variable value propagation

```
int CST;
init() { CST = 4; }
calc(int i) { return i / CST; }
main() {
  init();
  ... = calc(100);
}

calc(100) -> calc(100, CST)
```

- Extend jump function

```
f(int a, int b) {
  g(1 - a, b ? 1 : 2, a + b);
}

JF{f->g}[0] = 1 - param#0
JF{f->g}[1] = param#1 ? 1 : 2;
JF{f->g}[2] = param#0 + param#1
```

Issues in register allocator

- Context sensitive

```
f1() {
  S1
}
f2() {
  if (cond)
    S1
  else
    S2
}
```

Different allocation result

Irrelevant code

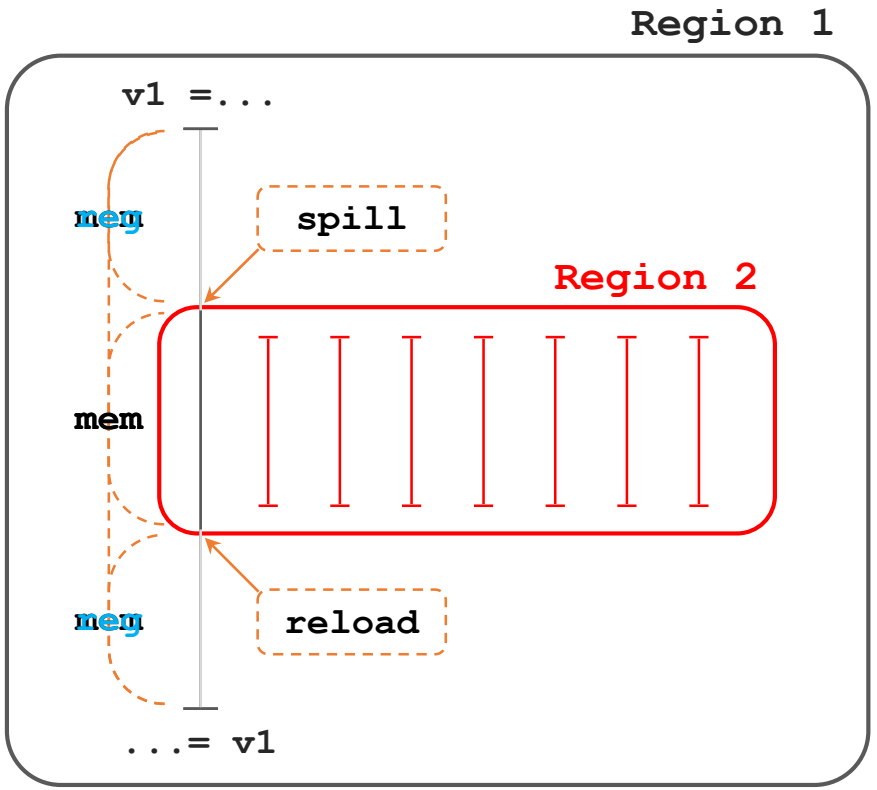
  - Code generation instability impacts inlining
  - Hard to do code and performance comparison

- Root cause
  - Execution profile normalization error

```
f1() {
  BB1 (30)   -> 30/10 = 3
  BB2 (1000) -> 1000/10 = 100
}
f2() {
  if (cond)
    BB1 (3)  -> 3/10 = 0.3 ≈ 1
    BB2 (100)-> 100/10 = 10
}
```

Issues in register allocator

- Top-down allocation order



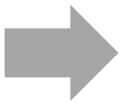
  - Local information impacts global allocation decision in too early stage

- Possible solutions
  - Use live range split to replace spilling
  - Do post refinement on outside region

Trapless conditional selection instruction generation

```
int f(int k, int b) {
    int a[2];
    if (b < a[k]) {
        a[k] = b;
    }
    return a[0]+a[2];
}
```

```
    sp, sp, #16
    uxtw    x0, w0
    add     x2, sp, 8
    ldr     w3, [x2, x0, lsl 2]
    cmp     w3, w1
    bls     .L2
    str     w1, [x2, x0, lsl 2]
.L2:
    ldr     w1, [sp, 8]
    ldr     w0, [sp, 16]
    add     sp, sp, 16
    add     w0, w1, w0
    ret
```



```
    uxtw    x2, w0
    add     x3, sp, 8
    ldr     w5, [sp, 16]
    ldr     w4, [x3, x2, lsl 2]
    cmp     w4, w1
    csel    w1, w1, w4, hi
    str     w1, [x3, x2, lsl 2]
    ldr     w0, [sp, 8]
    add     sp, sp, 16
    add     w0, w0, w5
    ret
```

- For “a” is local variable, always writable, introducing extra write on “a” will not cause trap.

Build something with us.

与我们一起创造未来!

http://developer.amperecomputing.com

AMPERE.