Register Allocation and Instruction Scheduling in Unison

Roberto Castañeda Lozano – SICS, KTH

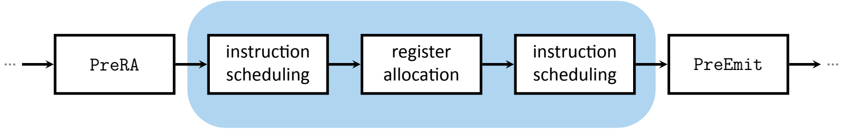
joint work with:

G. Hjort Blindell – KTH, SICS M. Carlsson – SICS

F. Drejhammar – SICS C. Schulte – KTH, SICS

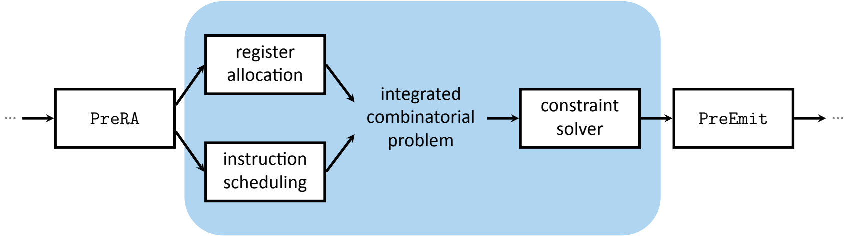
This research has been partially funded by Ericsson AB and the Swedish Research Council (VR 621-2011-6229)

Code Generation in LLVM



- Stages, heuristics
- Pros: compilation speed
- Cons: suboptimal, complex

Introducing Unison



- Integration, combinatorial optimization
- Pros: simple, optimal
- Cons: compilation slowdown

Unison Is Practical and Effective

- For LLVM Users
  - traditional LLVM for compile/debug cycle
  - LLVM + Unison for release builds
- For LLVM developers
  - evaluation of heuristics
  - identification of improvement opportunities

Earlier Optimal Approaches

- Global register allocation
  Local instruction scheduling
  - practical and effective
- Global instruction scheduling
  - scales up to medium-size problems
- Integrated optimal approaches
  - ignore essential register allocation subproblems
  - do not scale beyond small problems

Integrated Optimal Approaches

approach	GL	register allocation								instr. sched.			max.
		SP	RA	CO	SO	RP	LS	RM	MB	BD	MU	2D	size
Wilson 1994	✓	✓	✓	✓	-	-	✓	-	-	✓	✓	-	30
Chang 1997	-	✓	-	-	✓	-	-	-	-	✓	✓	-	~ 10
Gebotys 1997	-	✓	✓	-	✓	-	✓	-	✓	-	✓	-	108
ICG 1999	-	✓	✓	✓	✓	-	✓	-	✓	✓	-	-	23
PROPAN 2000	✓	-	✓	-	-	-	-	-	✓	✓	✓	✓	42
Nagar. 2007	-	✓	✓	-	✓	-	✓	-	-	✓	-	-	?
OPTIMIST 2012	-	✓	-	-	✓	-	✓	-	✓	✓	✓	✓	100
Unison	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	605

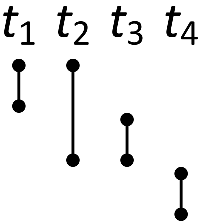
- Few global approaches
- Few register allocation subproblems
- Low scalability
- Unison: All subproblems, better scalability
  - key: constraint programming

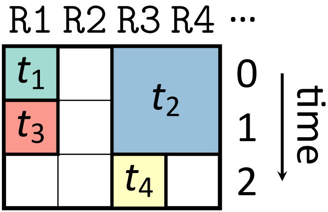
Model

- Register allocation
  - allocate temps to registers or memory
  - **register assignment**
  - spilling
  - coalescing
  - live range splitting
  - …
- Instruction scheduling
- Connection: temp live ranges

Register Assignment as Rectangle Packing

Register Assignment	Rectangle Packing
temp live ranges	rectangles
temp size (16 bits, 32 bits, ...)	rectangle width
interfering temps cannot share registers	rectangles cannot overlap

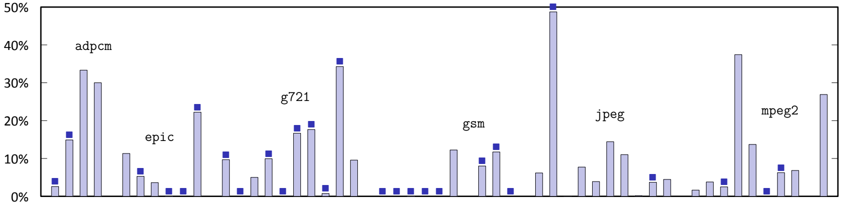




- Model based on Pereira and Palsberg, 2008

$$\text{no-overlap}(\langle r_{t_1}, r_{t_1}+1, ls_{t_1}, le_{t_1}\rangle, \langle r_{t_2}, r_{t_2}+2, ls_{t_2}, le_{t_2}\rangle, \ldots)$$

Speedup over LLVM 3.8



- 50 MediaBench functions
- Hexagon V4 processor
- Provably optimal (■) for 54% of the functions
- Compilation time: from seconds to minutes

Disclaimer

The case studies are generated with LLVM 3.8 for Hexagon V4

Disclaimer

`opt` is run with different optimization levels and some `llc` passes are disabled for simplicity

(`llc` is always run with `O3`)

Disclaimer

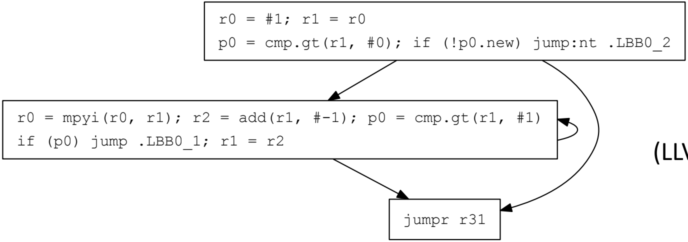
I am no expert on LLVM – just a humble user

Case Study: `fac`

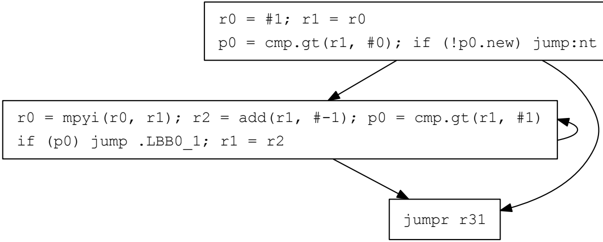
- Simple iterative factorial:

```
int fac(int n) {
  int f = 1;
  while (n > 0) {
    f = f * n;
    n--;
  }
  return f;
}
```

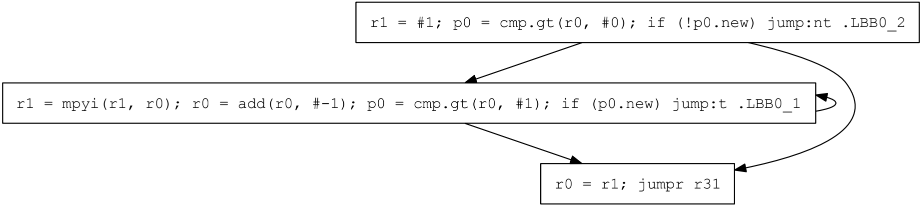
- Exposes opportunity for better coalescing
- Illustrates effect of integrated reasoning


r0 = #1; r1 = r0
p0 = cmp.gt(r1, #0); if (!p0.new) jump:nt .LBB0_2
r0 = mpyi(r0, r1); r2 = add(r1, #-1); p0 = cmp.gt(r1, #1)
if (p0) jump .LBB0_1; r1 = r2
jumpr r31

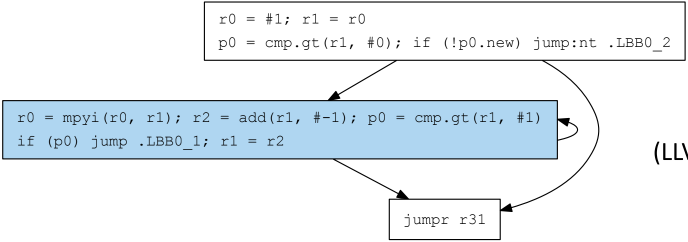
(LLVM)


r0 = #1; r1 = r0
p0 = cmp.gt(r1, #0); if (!p0.new) jump:nt .LBB0_2
r0 = mpyi(r0, r1); r2 = add(r1, #-1); p0 = cmp.gt(r1, #1)
if (p0) jump .LBB0_1; r1 = r2
jumpr r31

(LLVM)

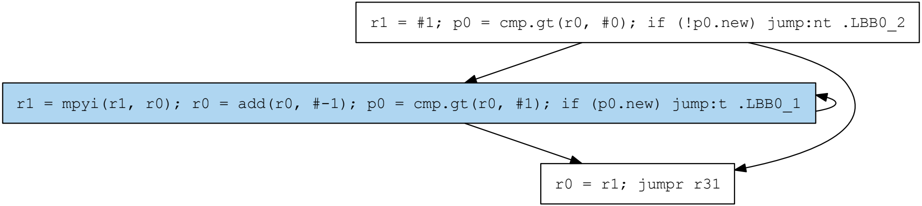

r1 = #1; p0 = cmp.gt(r0, #0); if (!p0.new) jump:nt .LBB0_2
r1 = mpyi(r1, r0); r0 = add(r0, #-1); p0 = cmp.gt(r0, #1); if (p0.new) jump:t .LBB0_1
r0 = r1; jumpr r31

(Unison)

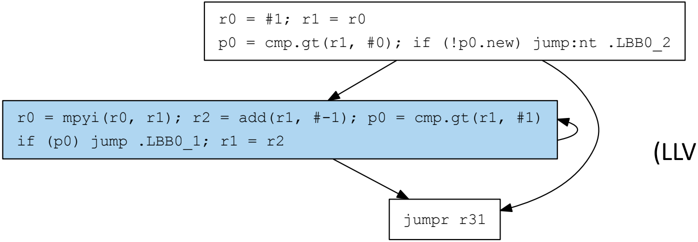


(LLVM)

LLVM's loop is twice as slow, why?

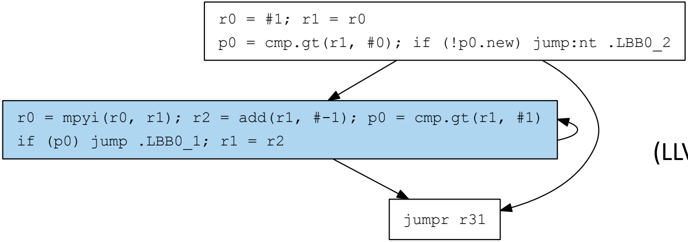


(Unison)



(LLVM)

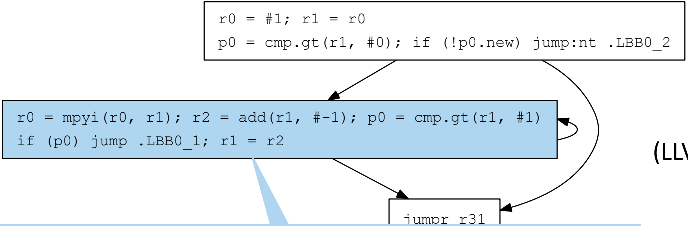
```
# After Simple Register Coalescing:
..
BB#1:
        %vreg2  = A2_addi %vreg10, -1
        %vreg9  = M2_mpyi %vreg9, %vreg10
        %vreg8  = C2_cmpgti %vreg10, 1
        %vreg10 = COPY %vreg2
        J2_jumpt %vreg8, <BB#1>
..
```



(LLVM)

```
# After Simple Register Coalescing:
..
BB#1:
        %vreg2  = A2_addi %vreg10, -1
        %vreg9  = M2_mpyi %vreg9, %vreg10
        %vreg8  = C2_cmpgti %vreg10, 1
        %vreg10 = COPY %vreg2
        J2_jumpt %vreg8, <BB#1>
..
```

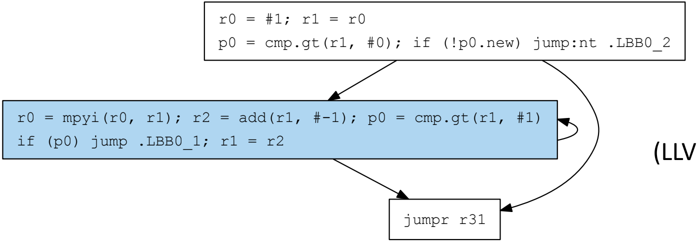
%vreg2 and %vreg10 not coalesced



(LLVM)

corresponding move requires a new bundle

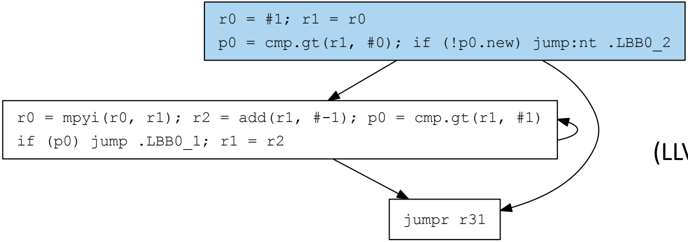
```
# After Simple Register Coalescing:
..
BB#1:
        %vreg2  = A2_addi %vreg10, -1
        %vreg9  = M2_mpyi %vreg9, %vreg10
        %vreg8  = C2_cmpgti %vreg10, 1
        %vreg10 = COPY %vreg2
        J2_jumpt %vreg8, <BB#1>
..
```



(LLVM)

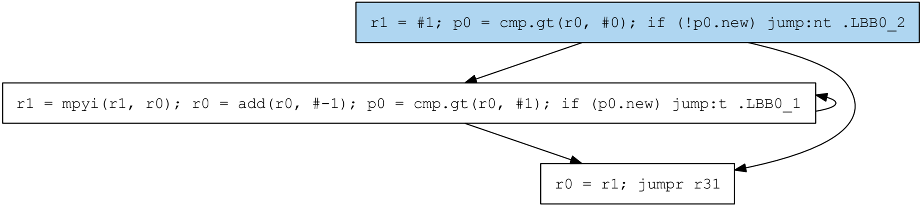
```
# After Simple Register Coalescing:
..
BB#1:
        %vreg9  = M2_mpyi %vreg9, %vreg10
        %vreg8  = C2_cmpgti %vreg10, 1
        %vreg2  = A2_addi %vreg10, -1
        %vreg10 = COPY %vreg2
        J2_jumpt %vreg8, <BB#1>
..
```

postponing A2_addi enables coalescing %vreg2, %vreg10



(LLVM)

Unison's initialization is one cycle faster, why?



(Unison)

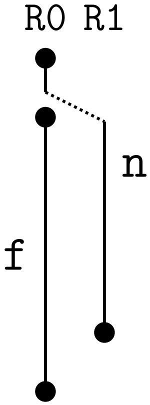
```
int fac(int n) {
  int f = 1;
  while(n > 0) {
    f = f * n;
    n--;
  }
  return f;
}
```

Calling convention: argument, return value in R0

```
int fac(int n) {
 int f = 1;
 while(n > 0) {
  f = f * n;
  n--;
 }
 return f;
}
```

However `n` and `f` interfere: move required

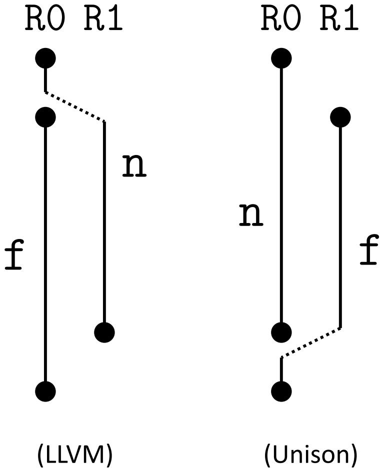
```
int fac(int n) {
  int f = 1;
  while(n > 0) {
    f = f * n;
    n--;
  }
  return f;
}
```



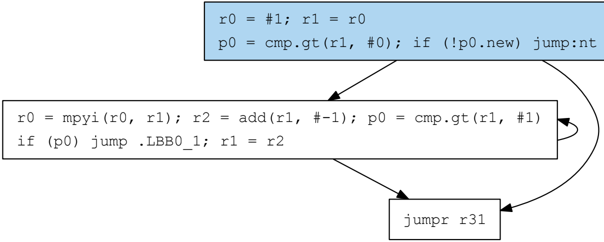
(LLVM)

LLVM moves `n` in the initialization block

```
int fac(int n) {
  int f = 1;
  while(n > 0) {
    f = f * n;
    n--;
  }
  return f;
}
```

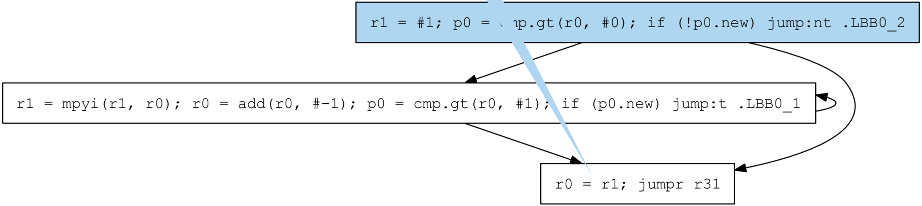


Unison moves `f` in the return block: does it matter?

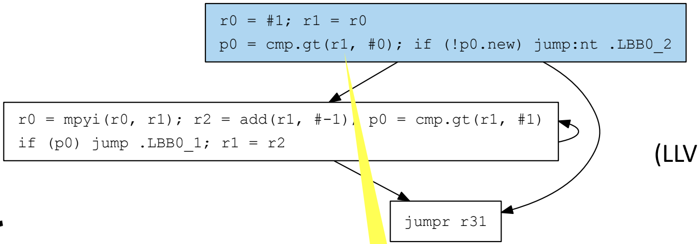


(LLVM)

It does! Unison's move can be scheduled in parallel

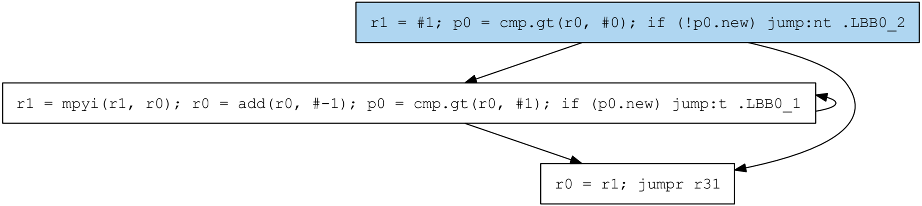


(Unison)



(LLVM)

What if `cmp.gt` could choose `r0` during scheduling?



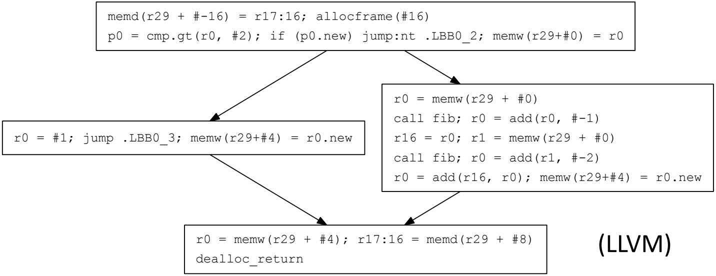
(Unison)

Case Study: `fib`

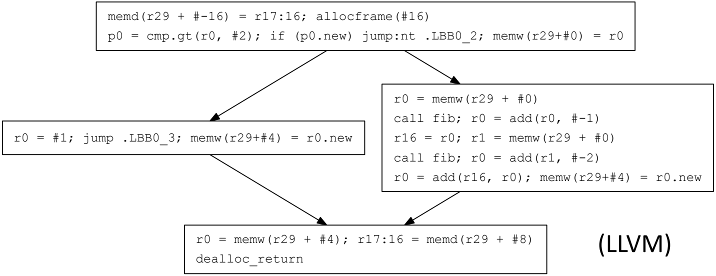
- Simple recursive *Fibonacci*:

```
int fib(int n) {
  if(n <= 2) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

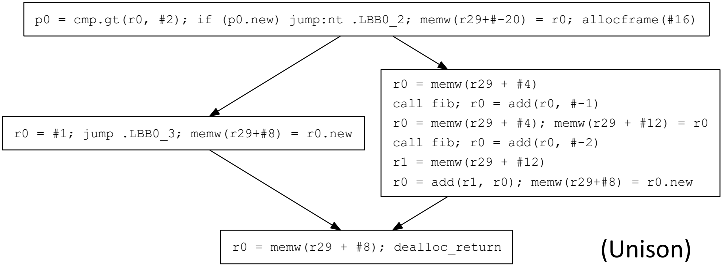
- Exposes opportunity for better spilling


memd(r29 + #-16) = r17:16; allocframe(#16)
p0 = cmp.gt(r0, #2); if (p0.new) jump:nt .LBB0_2; memw(r29+#0) = r0
r0 = #1; jump .LBB0_3; memw(r29+#4) = r0.new
r0 = memw(r29 + #0)
call fib; r0 = add(r0, #-1)
r16 = r0; r1 = memw(r29 + #0)
call fib; r0 = add(r1, #-2)
r0 = add(r16, r0); memw(r29+#4) = r0.new
r0 = memw(r29 + #4); r17:16 = memd(r29 + #8)
dealloc_return

(LLVM)


memd(r29 + #-16) = r17:16; allocframe(#16)
p0 = cmp.gt(r0, #2); if (p0.new) jump:nt .LBB0_2; memw(r29+#0) = r0
r0 = #1; jump .LBB0_3; memw(r29+#4) = r0.new
r0 = memw(r29 + #0)
call fib; r0 = add(r0, #-1)
r16 = r0; r1 = memw(r29 + #0)
call fib; r0 = add(r1, #-2)
r0 = add(r16, r0); memw(r29+#4) = r0.new
r0 = memw(r29 + #4); r17:16 = memd(r29 + #8)
dealloc_return

(LLVM)


p0 = cmp.gt(r0, #2); if (p0.new) jump:nt .LBB0_2; memw(r29+#-20) = r0; allocframe(#16)
r0 = #1; jump .LBB0_3; memw(r29+#8) = r0.new
r0 = memw(r29 + #4)
call fib; r0 = add(r0, #-1)
r0 = memw(r29 + #4); memw(r29 + #12) = r0
call fib; r0 = add(r0, #-2)
r1 = memw(r29 + #12)
r0 = add(r1, r0); memw(r29+#8) = r0.new
r0 = memw(r29 + #8); dealloc_return

(Unison)

```
memd(r29 + #-16) = r17:16; allocframe(#16)
p0 = cmp.gt(r0, #2); if (p0.new) jump:nt .LBB0_2; memw(r29+#0) = r0
```

```
r0 = #1; jump .LBB0_3; memw(r29+#4) = r0.new
```

```
r0 = memw(r29 + #0)
call fib; r0 = add(r0, #-1)
r16 = r0; r1 = memw(r29 + #0)
call fib; r0 = add(r1, #-2)
r0 = add(r16, r0); memw(r29+#4) = r0.new
```

```
r0 = memw(r29 + #4); r17:16 = memd(r29 + #8)
dealloc_return
```

(LLVM)

recursive case requires spilling, where to spill?

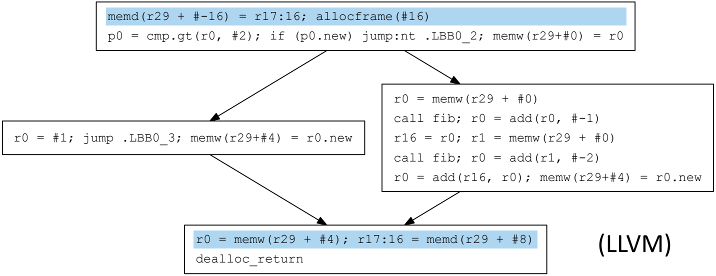
```
p0 = cmp.gt(r0, #2); if (p0.new) jump:nt .LBB0_2; memw(r29+#-20) = r0; allocframe(#16)
```

```
r0 = #1; jump .LBB0_3; memw(r29+#8) = r0.new
```

```
r0 = memw(r29 + #4)
call fib; r0 = add(r0, #-1)
r0 = memw(r29 + #4); memw(r29 + #12) = r0
call fib; r0 = add(r0, #-2)
r1 = memw(r29 + #12)
r0 = add(r1, r0); memw(r29+#8) = r0.new
```

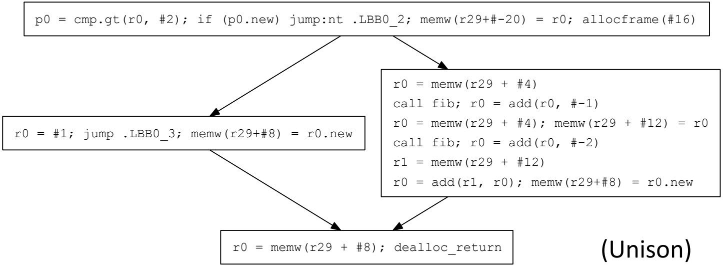
```
r0 = memw(r29 + #8); dealloc_return
```

(Unison)

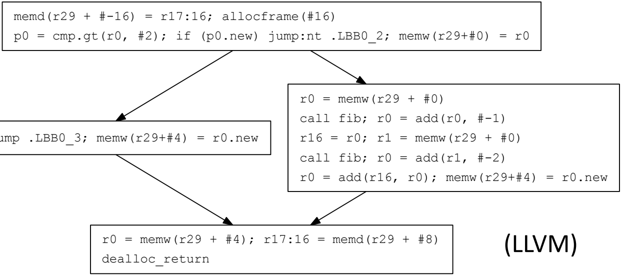


(LLVM)

LLVM frees a callee-saved register (always)

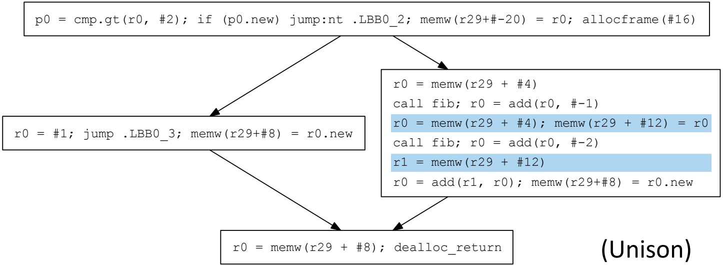


(Unison)

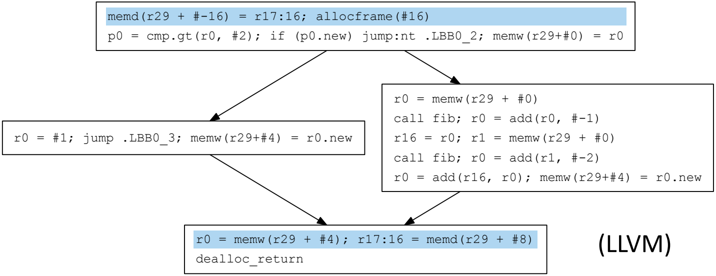


(LLVM)

Unison spills the value directly (recursive case)

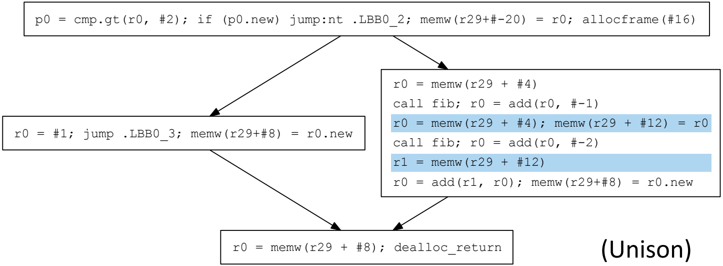


(Unison)

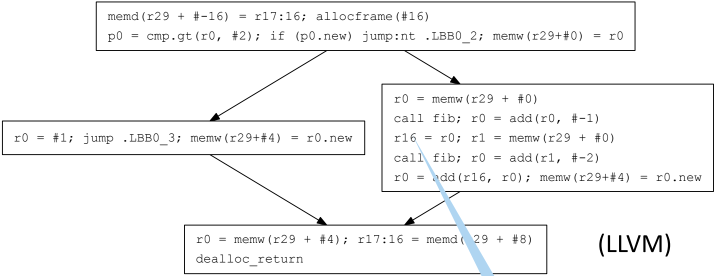


(LLVM)

LLVM spills twice as much as Unison, why?



(Unison)



(LLVM)

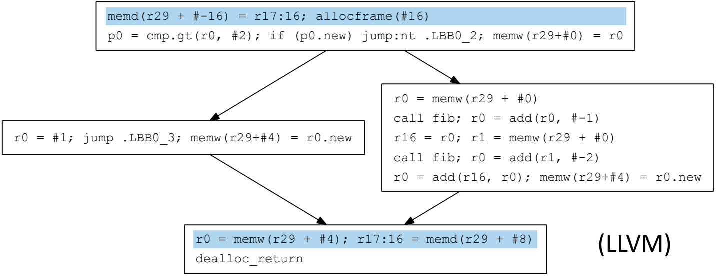
```
# After Virtual Register Rewriter:
..
BB#0:



        S2_storeri_io <fi#1>, 0,
        %P0 = C2_cmpgti %R0, 2
        J2_jumpt %P0, <BB#2>
..
BB#3:
        %R0 = L2_loadri_io <fi#0>, 0

        JMPret %R31
..
```

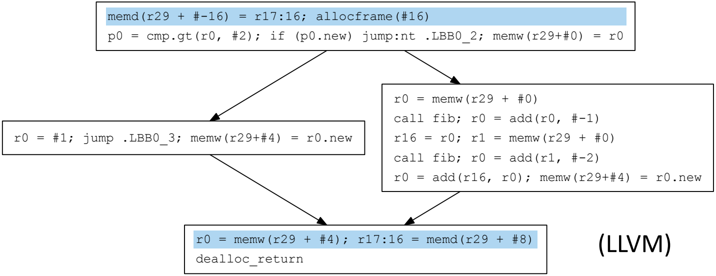
LLVM's register allocator thinks using `r16` has no cost



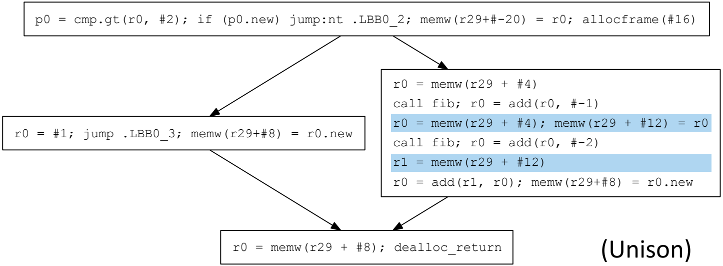
(LLVM)

```
# After Prologue/Epilogue Insertion & Frame Finalization:
..
BB#0:
        S2_allocframe 16
        S2_storerd_io %R29, 8, %D8
        S2_storeri_io %R29, 0, %R0
        %P0 = C2_cmpgti %R0, 2
        J2_jumpt %P0, <BB#2>
..
BB#3:
        %R0 = L2_loadri_io %R29, 4
        %D8 = L2_loadrd_io %R29, 8
        L4_return
..
```

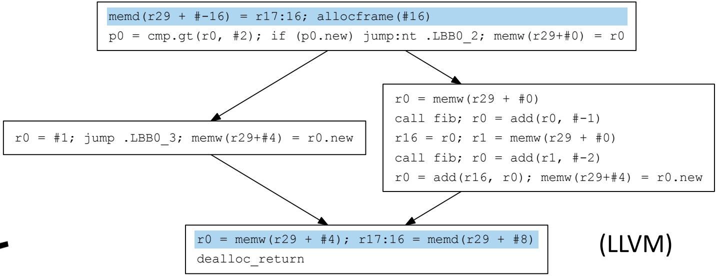
but `r16` is callee-saved and must be preserved


memd(r29 + #-16) = r17:16; allocframe(#16)
p0 = cmp.gt(r0, #2); if (p0.new) jump:nt .LBB0_2; memw(r29+#0) = r0
r0 = #1; jump .LBB0_3; memw(r29+#4) = r0.new
r0 = memw(r29 + #0)
call fib; r0 = add(r0, #-1)
r16 = r0; r1 = memw(r29 + #0)
call fib; r0 = add(r1, #-2)
r0 = add(r16, r0); memw(r29+#4) = r0.new
r0 = memw(r29 + #4); r17:16 = memd(r29 + #8)
dealloc_return

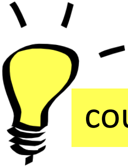
(LLVM)


p0 = cmp.gt(r0, #2); if (p0.new) jump:nt .LBB0_2; memw(r29+#-20) = r0; allocframe(#16)
r0 = #1; jump .LBB0_3; memw(r29+#8) = r0.new
r0 = memw(r29 + #4)
call fib; r0 = add(r0, #-1)
r0 = memw(r29 + #4); memw(r29 + #12) = r0
call fib; r0 = add(r0, #-2)
r1 = memw(r29 + #12)
r0 = add(r1, r0); memw(r29+#8) = r0.new
r0 = memw(r29 + #8); dealloc_return

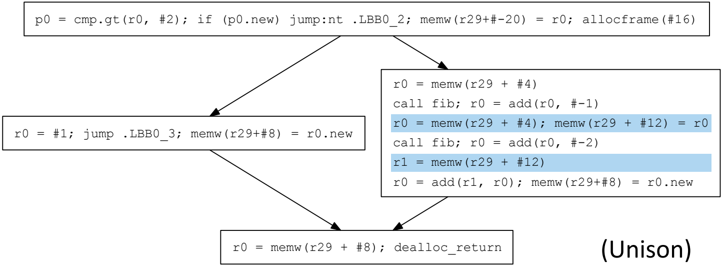
(Unison)



(LLVM)



could LLVM handle callee-saved spilling earlier?



(Unison)

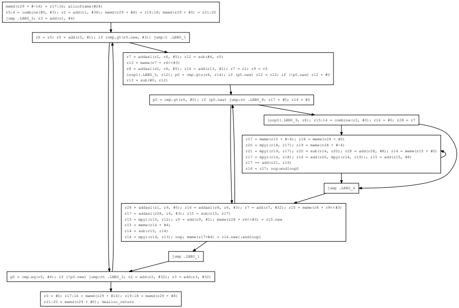
Case Study: `chol`

- Complex Cholesky decomposition (simplified)
- Illustrates the need for accurate information
- Exposes opportunity for better freq. estimation

Case Study: chol

```
typedef struct complex {
    int re;
    int im;
} complex;

int chol(const complex A[4][4], complex L[4][4]) {
  for (int i = 0; i < 4; i++) {
    int f = 1/L[i][i].re;
    for (int j = i+1; j < 4; j++) {
      complex q = {0, 0};
      for (int k = 0; k < i; k++) {
        q.re += L[i][k].re * L[j][k].re;
        q.im -= L[i][k].re * L[j][k].im;
        q.re += L[i][k].im * L[j][k].im;
        q.im += L[i][k].im * L[j][k].re;
      }
      L[j][i].re =  f * (A[i][j].re - q.re);
      L[j][i].im = -f * (A[i][j].im - q.im);
    }
  }
  return 0;
}
```


memd(r29 + #-16) = r17:16; allocframe(#24)
r5:4 = combine(#0, #3); r2 = add(r1, #36); memd(r29 + #8) = r19:18; memd(r29 + #0) = r21:20
jump .LBB0_3; r3 = add(r1, #4)
r6 = r5; r5 = add(r5, #1); if (cmp.gt(r5.new, #3)) jump:t .LBB0_1
r7 = addasl(r1, r6, #5); r12 = sub(#4, r5)
r13 = memw(r7 + r6<<#3)
r8 = addasl(r0, r6, #5); r14 = add(r13, #1); r7 = r2; r9 = r5
loop1(.LBB0_5, r12); p0 = cmp.gtu(r4, r14); if (p0.new) r12 = r13; if (!p0.new) r12 = #0
r13 = sub(#0, r12)
p0 = cmp.gt(r6, #0); if (p0.new) jump:nt .LBB0_8; r17 = #0; r14 = #0
loop0(.LBB0_9, r6); r15:14 = combine(r3, #0); r16 = #0; r28 = r7
r17 = memw(r15 + #-4); r18 = memw(r28 + #0)
r20 = mpyi(r18, r17); r19 = memw(r28 + #-4)
r21 = mpyi(r19, r17); r20 = sub(r14, r20); r28 = add(r28, #8); r14 = memw(r15 + #0)
r17 = mpyi(r14, r18); r14 = add(r20, mpyi(r14, r19)); r15 = add(r15, #8)
r17 += add(r21, r16)
r16 = r17; nop:endloop0
jump .LBB0_6
r28 = addasl(r1, r9, #5); r16 = addasl(r8, r9, #3); r7 = add(r7, #32); r15 = memw(r8 + r9<<#3)
r17 = addasl(r28, r6, #3); r15 = sub(r15, r17)
r15 = mpyi(r15, r12); r9 = add(r9, #1); memw(r28 + r6<<#3) = r15.new
r15 = memw(r16 + #4)
r14 = sub(r15, r14)
r14 = mpyi(r14, r13); nop; memw(r17+#4) = r14.new):endloop1
jump .LBB0_1
p0 = cmp.eq(r5, #4); if (!p0.new) jump:nt .LBB0_3; r2 = add(r2, #32); r3 = add(r3, #32)
r0 = #0; r17:16 = memd(r29 + #16); r19:18 = memd(r29 + #8)
r21:20 = memd(r29 + #0); dealloc_return

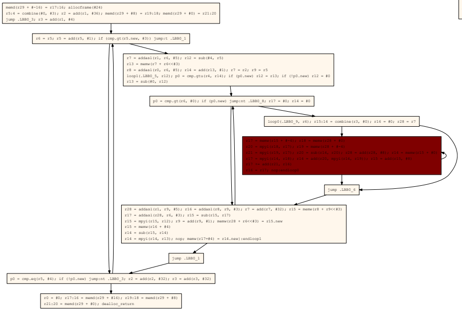
(LLVM)


memd(r29 + #-16) = r17:16; allocframe(#24)
r5:4 = combine(#0, #3); r2 = add(r1, #36); memd(r29 + #8) = r19:18; memd(r29 + #0) = r21:20
jump .LBB0_3; r3 = add(r1, #4)
r6 = r5; r5 = add(r5, #1); if (cmp.gt(r5.new, #3)) jump:t .LBB0_1
r7 = addasl(r1, r6, #5); r12 = sub(#4, r5)
r13 = memw(r7 + r6<<#3)
r8 = addasl(r0, r6, #5); r14 = add(r13, #1); r7 = r2; r9 = r5
loop1(.LBB0_5, r12); p0 = cmp.gtu(r4, r14); if (p0.new) r12 = r13; if (!p0.new) r12 = #0
r13 = sub(#0, r12)
p0 = cmp.gt(r6, #0); if (p0.new) jump:nt .LBB0_8; r17 = #0; r14 = #0
loop0(.LBB0_9, r6); r15:14 = combine(r3, #0); r16 = #0; r28 = r7
r17 = memw(r15 + #-4); r18 = memw(r28 + #0)
r20 = mpyi(r18, r17); r19 = memw(r28 + #-4)
r21 = mpyi(r19, r17); r20 = sub(r14, r20); r28 = add(r28, #8); r14 = memw(r15 + #0)
r17 = mpyi(r14, r18); r14 = add(r20, mpyi(r14, r19)); r15 = add(r15, #8)
r17 += add(r21, r16)
r16 = r17; nop:endloop0
jump .LBB0_6
r28 = addasl(r1, r9, #5); r16 = addasl(r8, r9, #3); r7 = add(r7, #32); r15 = memw(r8 + r9<<#3)
r17 = addasl(r28, r6, #3); r15 = sub(r15, r17)
r15 = mpyi(r15, r12); r9 = add(r9, #1); memw(r28 + r6<<#3) = r15.new
r15 = memw(r16 + #4)
r14 = sub(r15, r14)
r14 = mpyi(r14, r13); nop; memw(r17+#4) = r14.new):endloop1
jump .LBB0_1
p0 = cmp.eq(r5, #4); if (!p0.new) jump:nt .LBB0_3; r2 = add(r2, #32); r3 = add(r3, #32)
r0 = #0; r17:16 = memd(r29 + #16); r19:18 = memd(r29 + #8)
r21:20 = memd(r29 + #0); dealloc_return

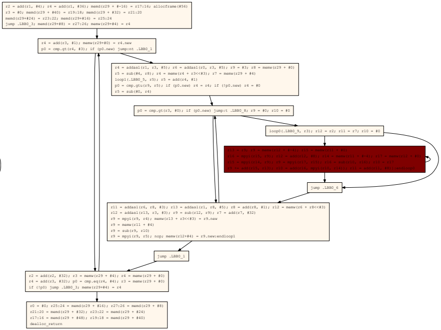
(LLVM)


r2 = add(r1, #4); r4 = add(r1, #36); memd(r29 + #-16) = r17:16; allocframe(#56)
r3 = #0; memd(r29 + #40) = r19:18; memd(r29 + #32) = r21:20
memd(r29+#24) = r23:22; memd(r29+#16) = r25:24
jump .LBB0_3; memd(r29+#8) = r27:26; memw(r29+#4) = r4
r4 = add(r3, #1); memw(r29+#0) = r4.new
p0 = cmp.gt(r4, #3); if (p0.new) jump:nt .LBB0_1
r4 = addasl(r1, r3, #5); r6 = addasl(r0, r3, #5); r9 = #3; r8 = memw(r29 + #0)
r5 = sub(#4, r8); r4 = memw(r4 + r3<<#3); r7 = memw(r29 + #4)
loop1(.LBB0_5, r5); r5 = add(r4, #1)
p0 = cmp.gtu(r9, r5); if (p0.new) r4 = r4; if (!p0.new) r4 = #0
r5 = sub(#0, r4)
p0 = cmp.gt(r3, #0); if (p0.new) jump:t .LBB0_8; r9 = #0; r10 = #0
loop0(.LBB0_9, r3); r12 = r2; r11 = r7; r10 = #0
r13 = r9; r9 = memw(r12 + #-4); r15 = memw(r11 + #0)
r16 = mpyi(r15, r9); r12 = add(r12, #8); r14 = memw(r11 + #-4); r17 = memw(r12 + #0)
r15 = mpyi(r14, r9); r9 = mpyi(r17, r15); r16 = sub(r10, r16); r10 = r17
r9 += add(r15, r13); r10 = add(r16, mpyi(r10, r14)); r11 = add(r11, #8)):endloop0
jump .LBB0_6
r11 = addasl(r6, r8, #3); r13 = addasl(r1, r8, #5); r8 = add(r8, #1); r12 = memw(r6 + r8<<#3)
r12 = addasl(r13, r3, #3); r9 = sub(r12, r9); r7 = add(r7, #32)
r9 = mpyi(r9, r4); memw(r13 + r3<<#3) = r9.new
r9 = memw(r11 + #4)
r9 = sub(r9, r10)
r9 = mpyi(r9, r5); nop; memw(r12+#4) = r9.new:endloop1
jump .LBB0_1
r2 = add(r2, #32); r3 = memw(r29 + #4); r4 = memw(r29 + #0)
r4 = add(r3, #32); p0 = cmp.eq(r4, #4); r3 = memw(r29 + #0)
if (!p0) jump .LBB0_3; memw(r29+#4) = r4
r0 = #0; r25:24 = memd(r29 + #16); r27:26 = memd(r29 + #8)
r21:20 = memd(r29 + #32); r23:22 = memd(r29 + #24)
r17:16 = memd(r29 + #48); r19:18 = memd(r29 + #40)
dealloc_return

(Unison)



(LLVM)



(Unison)

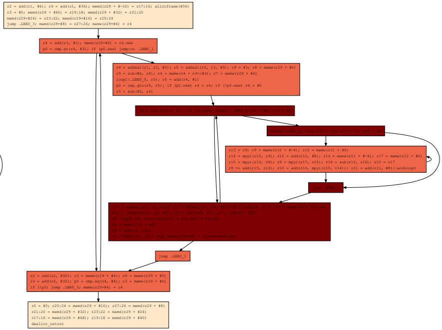
LLVM estimates the innermost loop dominates runtime

(LLVM)

(Unison)

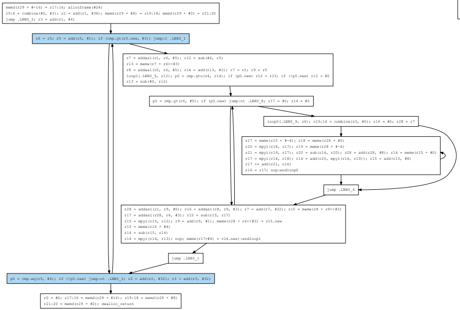
Unison optimizes that basic block (6 vs. 4 cycles)

(LLVM)

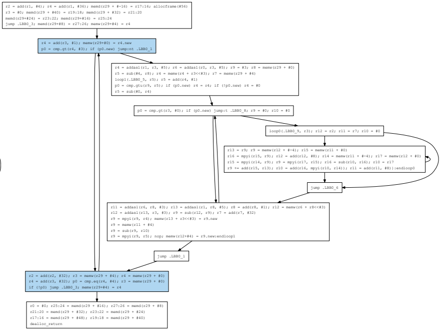


(Unison)

but in practice, other basic blocks are hotter



(LLVM)



(Unison)

so much that Unison’s code performs worse!

```c
typedef struct complex {
    int re;
    int im;
} complex;

int chol(const complex A[4][4], complex L[4][4]) {
  for (int i = 0; i < 4; i++) {
    int f = 1/L[i][i].re;
    for (int j = i+1; j < 4; j++) {
      complex q = {0, 0};
      for (int k = 0; k < i; k++) {
        q.re += L[i][k].re * L[j][k].re;
        q.im -= L[i][k].re * L[j][k].im;
        q.re += L[i][k].im * L[j][k].im;
        q.im += L[i][k].im * L[j][k].re;
      }
      L[j][i].re =  f * (A[i][j].re - q.re);
      L[j][i].im = -f * (A[i][j].im - q.im);
    }
  }
  return 0;
}
```

could LLVM's freq. estimation consider loop counts?

Unison Is Practical and Effective

- Integrated
  - register allocation
  - instruction scheduling
- Simple, optimal, slower
- For LLVM Users
  - traditional LLVM for compile/debug cycle
  - LLVM + Unison for release builds
- For LLVM developers
  - evaluation of heuristics
  - identification of improvement opportunities
    - coalescing, scheduling, spilling, frequency estimation
  - but: aggressive optimization requires accuracy

unison-code.github.io