Predication and Speculation

Compilers for High Performance Computers

Stefano Petrilli
stefano.petrilli@upc.edu

Jakob Eberhardt
jakob.eberhardt@estudiantat.upc.edu

December 10, 2024

Modern Processors

- Superscalar

Modern Processors

- Superscalar
- Deep Pipelines

Modern Processors

- Superscalar
- Deep Pipelines
- Out of Order

Modern Processors

- Superscalar
- Deep Pipelines
- Out of Order
- **How do we keep the pipeline busy?**

Misprediction Penalty vs Optimal Load

Architecture	Misprediction Penalty	Optimistic Load Cost
Sapphire Rapids	14	5
Alder Lake-P	14	5
Ice Lake	14	5
Broadwell	16	5
Haswell	16	5
Cortex A57	14	4
Cortex R52	8	1
Cortex M4	2	2

Table: Penalty assumptions used in LLVM for different architectures.

Compiler-Controlled Speculation

- During runtime, we only know the pipeline
- During compilation, we know the global picture

Speculative Execution

- Make assumptions about future control flow
- **Execute before we know we need the result**
- This includes moving instructions across branches
- **Risk:** May alter the program's execution & result

Restrictions for Speculation

```
1      ldr r1, [address]
2      ldr r2, [address2]
3      beq taken, r1, #0
4      sdiv r2, #5, r1
5      b end
6  taken:
7      add r1, r2, #2
8  end:
```

```
1      ldr r1, [address]
2      ldr r2, [address2]
3      sdiv r2, #5, r1
4      beq taken, r1, #0
5      b end
6  taken:
7      add r1, r2, #2
8  end:
```

Listing: If we could schedule `sdiv` earlier, we likely increase ILP. However, it may overwrite `r2` used in `taken` or throw an exception

If we want to move an instruction I above its branch Br:

- **Restriction 1**: The destination register of I is not used as a source if Br is taken
- **Restriction 2**: Instruction I will not cause an exception which will alter the program execution if Br is taken.

Computing the Average of Absolute Values

- Go through all nodes
- If `wt` is negative, subtract it from `weight`
- Else, add it to `weight`
- Compute avg if we had at least one node

```
1  avg = 0;
2  weight = 0;
3  count = 0;
4
5  while (ptr != NULL) {
6       count++;
7       if(ptr->wt < 0) {
8            weight -= ptr->wt;
9       } else {
10           weight += ptr->wt;
11      }
12      ptr = ptr->next;
13 }
14
15 if(count !=0) {
16      avg = weight / count;
17 }
```

Steps

Typical Ingredients

1. Identify *trace*
2. Superblock creation
3. Dependency graph (based on architectural model)
4. List scheduling

Control Flow Profile

Profiling

- Collect execution data
- Facilitate optimization decisions
- E.g. in LLVM [5] with `-fprofile-instr-generate`

- The loop part BB2 to BB5 is interesting
- 90% of the weights are positive

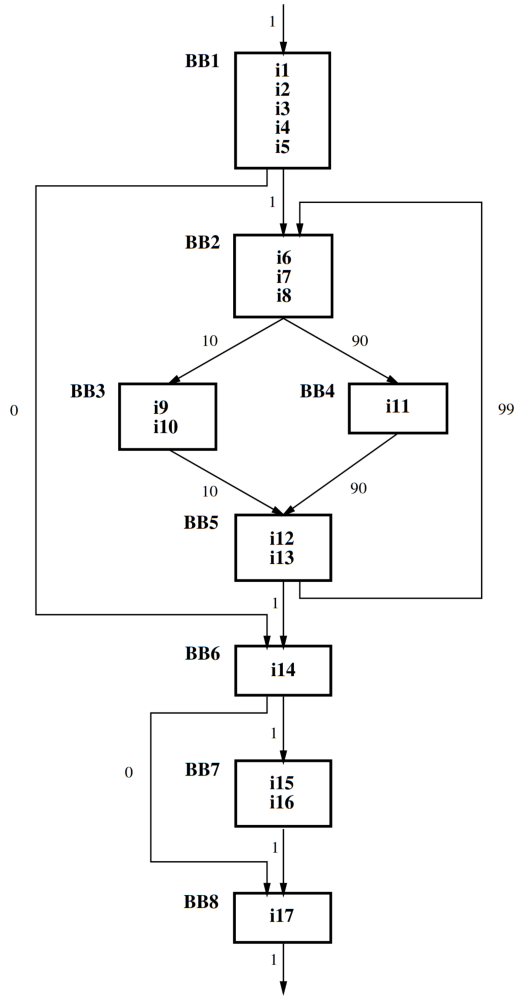


Figure: Weighted Control Flow Graph with profiling data of Chang [2]

Loop Part as a Superblock

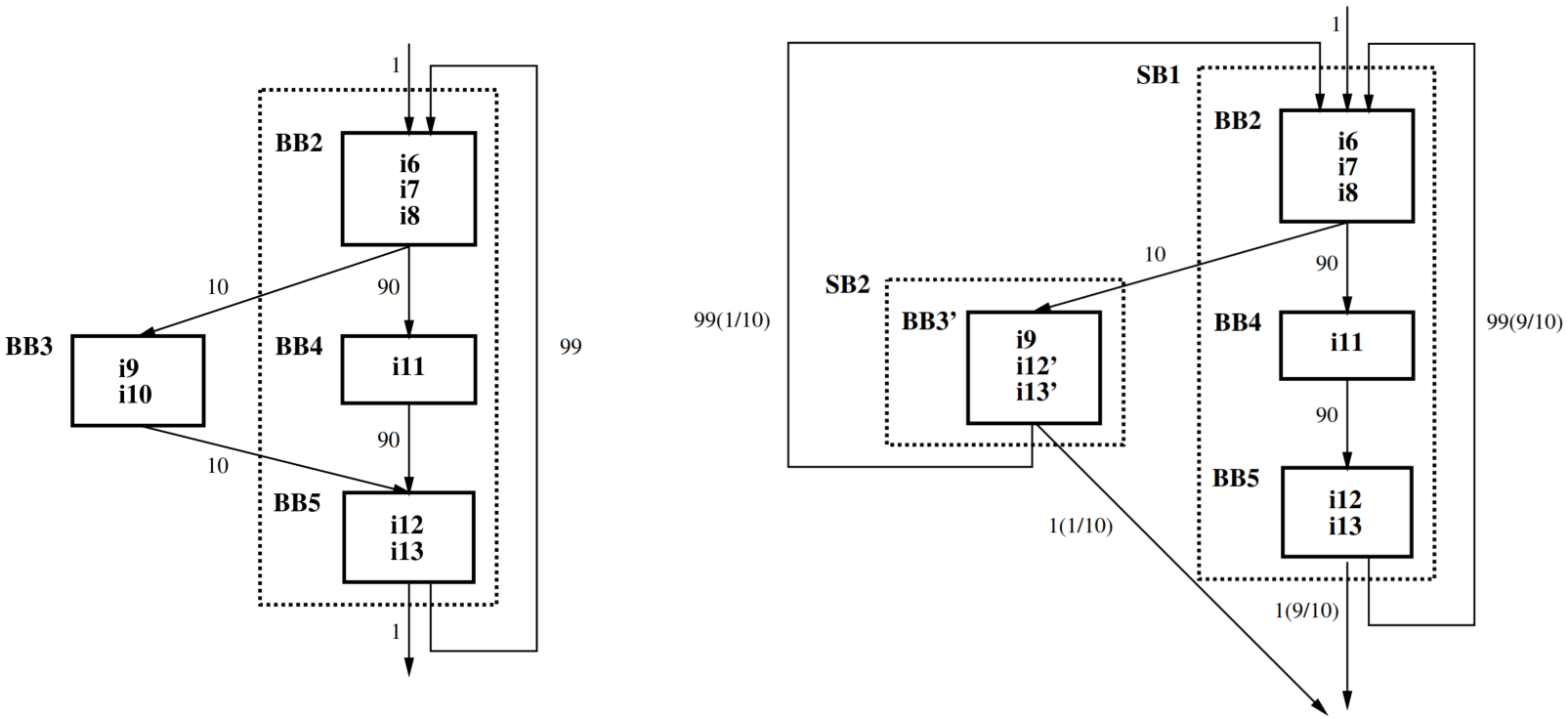


Figure: Loop before and after Superblock creation [2]. BB5 is duplicated in BB3'

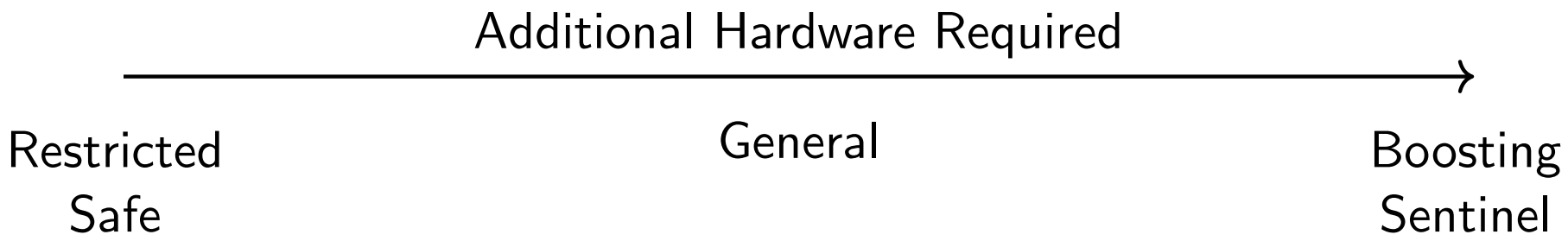
BB2 → BB4 → BB5 is the hot part (trace)

- Superblocks reduce bookkeeping [3]
- No need to consider side entrances when scheduling a superblock

Models for Speculative Scheduling

Different Code Percolation Models

- Enable different levels of speculation
- And hence different levels of performance
- Can alter the result of the program
- Require additional hardware



Scheduling our Superblock (Report page 22)

```
1      ldr r1, _ptr        ; Initialize
2      mov r7, 0
3      mov r2, 0
4      mov r3, 0
5      beq L3, r1, 0
6  L0:
7      add r2, r2, 1       ; I1: Increment count
8      ldr r4, 0[r1]       ; I2: Load ptr->wt into r4
9      btl L1, r4, 0       ; I3: If wt < 0, branch to L1
10     add r3, r3, r4      ; I4: Add wt to weight
11     ldr r5, 4[r1]       ; I5: Load ptr->next into r5
12     beq L3, r5, 0       ; I6: If next is NULL, jump to L3
13     add r2, r2, 1       ; I7: Increment count
14     ldr r6, 0[r5]       ; I8: Load next->wt into r6
15     btl L1X, r6, 0      ; I9: If wt < 0, branch to L1X
16     add r3, r3, r6      ; I10: Add wt to weight
17     ldr r1, 4[r5]       ; I11: Move ptr to ptr->next->next
18     bne L0, r1, 0       ; I12: Loop back to L0 if ptr != NULL
19 L3:
20     beq L4, r2, 0       ; If count == 0, skip division
21     div r7, r3, r2
22     str _avg, r7
23 L4:
24 ; ...
25 L1X:
26     mov r1, r5          ; Adjust ptr = ptr->next
27     mov r4, r6          ; Move wt to r4 for subtraction
28 L1:
29     sub r3, r3, r4      ; Subtract wt from weight
30     ldr r1, 4[r1]       ; Move ptr to ptr->next
31     bne L0, r1, 0       ; Loop back to L0 if ptr != NULL
```

Listing: The loop part of the superblock was unrolled once. We assume a one-cycle latency for ALU instructions and a two-cycle load delay.

Restricted Code Percolation

Properties of Restricted Code Percolation

- *Avoid errors* category [1]
- Compiler cannot move excepting instructions above branches
- Other instructions which fulfill Restriction 1 can be moved
  - E.g. after renaming
- Requires no additional hardware
- No extra load on DCache

Safe Code Percolation Extension [1]

- Restriction 2 can be relaxed
  - If there is a proof that the instructions will never cause an exception
- Typically speculative loads by check allocation boundaries

Restricted Code Percolation

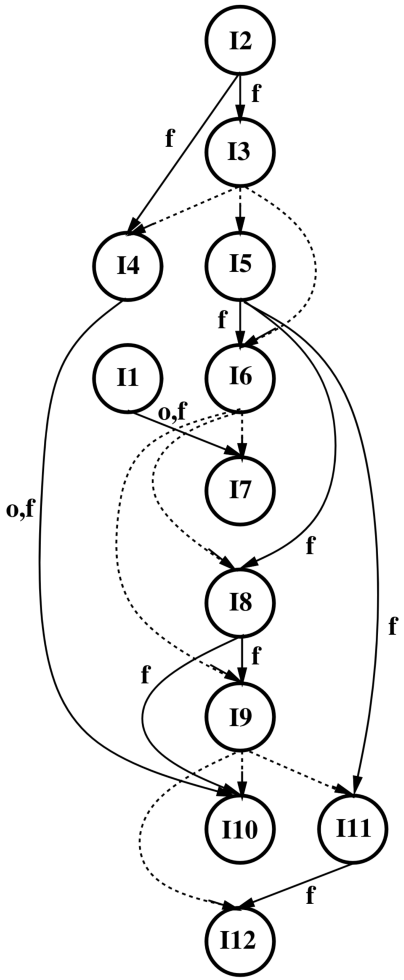


Figure: Nine dashed control dependencies and twelve data dependencies (**f**low, **o**utput) under restricted percolation [2].

	U 1	U 2	U 3	U 4
C1	I-1 add	I-2 ldr		
C2				
C3	I-3 btl	I-4 add	I-5 ldr	
C4				
C5	I-6 beq	I-7 add	I-8 ldr	
C6				
C7	I-9 btl	I-10 add	I-11 ldr	
C8				
C9	I-12 bne			

Figure: The schedule tableau is quite sparse. Loads have to stay in their home block. After `C1`, all data dependencies for `I7` are available.

But I7's destination r2 is in live-out(I6)

```
1     ldr r1, _ptr       ; Initialize
2     mov r7, 0
3     mov r2, 0
4     mov r3, 0
5     beq L3, r1, 0
6  L0:
7     add r2, r2, 1      ; I1: Increment count
8     ldr r4, 0[r1]      ; I2: Load ptr->wt into r4
9     btl L1, r4, 0      ; I3: If wt < 0, branch to L1
10    add r3, r3, r4     ; I4: Add wt to weight
11    ldr r5, 4[r1]      ; I5: Load ptr->next into r5
12    beq L3, r5, 0      ; I6: If next is NULL, jump to L3
13    add r2, r2, 1      ; I7: Increment count
14    ldr r6, 0[r5]      ; I8: Load next->wt into r6
15    btl L1X, r6, 0     ; I9: If wt < 0, branch to L1X
16    add r3, r3, r6     ; I10: Add wt to weight
17    ldr r1, 4[r5]      ; I11: Move ptr to ptr->next->next
18    bne L0, r1, 0      ; I12: Loop back to L0 if ptr != NULL
19 L3:
20    beq L4, r2, 0      ; If count == 0, skip division
21    div r7, r3, r2
22    str _avg, r7
23 L4:
24 ; ...
25 L1X:
26    mov r1, r5         ; Adjust ptr = ptr->next
27    mov r4, r6         ; Move wt to r4 for subtraction
28 L1:
29    sub r3, r3, r4     ; Subtract wt from weight
30    ldr r1, 4[r1]      ; Move ptr to ptr->next
31    bne L0, r1, 0      ; Loop back to L0 if ptr != NULL
```

General Code Percolation

Properties of General Code Percolation

- *Ignore errors* category
- Lifts Restriction 2 by simply ignoring exceptions
- Requires non-excepting counterpart instructions

- E.g., if a load accesses an invalid address, its destination will be garbage
- May cause an actual exception later if the result is used

General Code Percolation

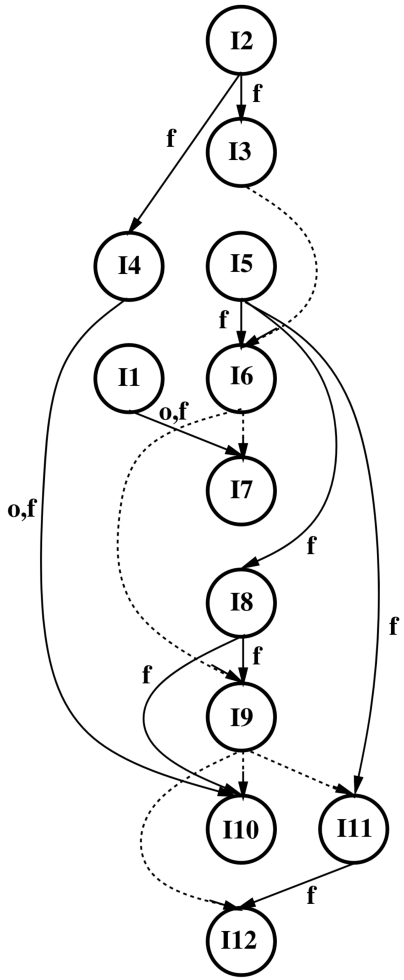


Figure: Dependence Graph for General Code Percolation with six control dependencies [2].

	U 1	U 2	U 3	U 4
C1	I-1 add	I-2 ldr	I-5 ldr	
C2				
C3	I-3 btl	I-4 add	I-8 ldr	I-11 ldr
C4	I-6 beq			
C5	I-7 add	I-9 btl	I-10 add	I-12 bne

Figure: By lifting Restriction 2, we are able to schedule the loads earlier at the cost of eventual errors. E.g., `I8` which is now above `I6` (`next == null`) will cause errors.

Boosting Code Percolation

Properties of Boosting Code Percolation

- *Resolve errors*
- Lifts Restriction 1 & 2 with additional hardware support
- Temporarily hold side effects of a boosted instruction
  - Until its branch is executed

Hardware for Boosting

- Superblock → Boosted instruction is in the not-taken part
- 1 to N bits → above how many branches the instruction was moved
- Not taken while boosted instructions are in the pipeline → remove bit
- Taken while boosted instructions are in the pipeline → squash them

Shadow Register File & Shadow Store Buffer [6]

- If boosted instruction writes back before its branch → keep its result
  - And delay exception handling until we know the branch outcome
- If branch is not taken → copy result into real register or store buffer
  - If there was an exception, re-execute not-taken part in-order
- If branch is taken → squash shadow values, ignore exceptions and go to branch target

Boosting Code Percolation

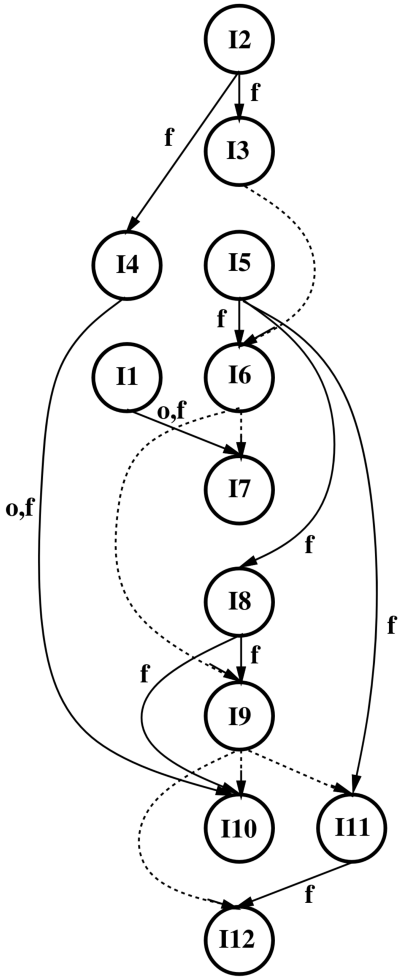


Figure: If we boost above one branch, we have three control dependencies left [2].

	U 1	U 2	U 3	U 4
C1	I-1 add	I-2 ldr	I-5 ldr	
C2	I-7 add			
C3	I-3 btl	I-4 add	I-8 ldr	I-11 ldr
C4	I-6 beq			
C5	I-9 btl	I-10 add	I-12 bne	

Figure: I7 can be scheduled in C2. The incremented `count` variable will be copied into the real register once branch I6 commits and was not taken.

Practical Study

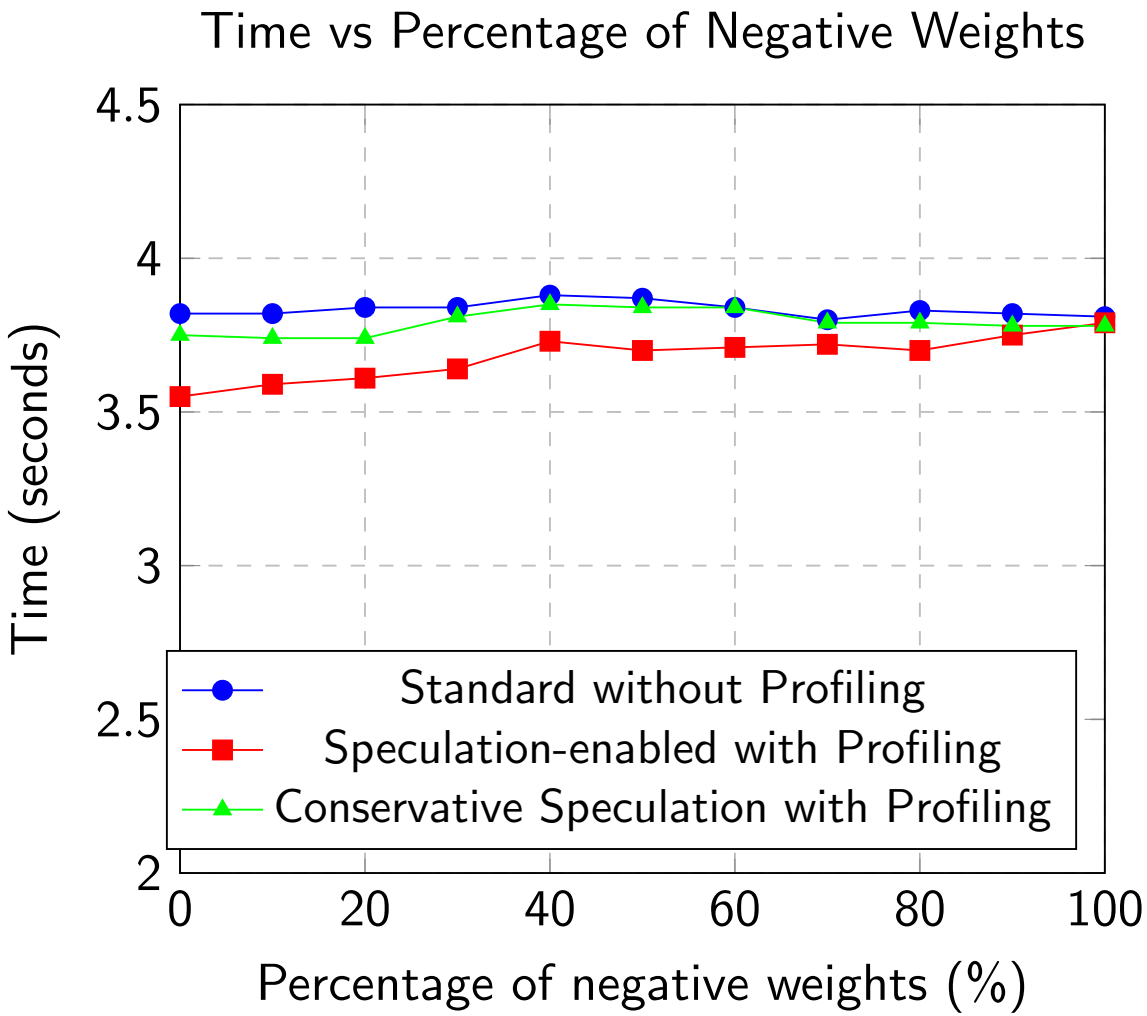


Figure: 99% positive weights during profiling, increasing percentage of negative weight during benchmarking.

Predication

Many branches are hard to predict.

Predicated Instructions

MOVGT R1 R0

Predicated Instructions

MOVGT R1 R0

Predicated Instructions

MOVGT R1 R0

Predicated Instructions

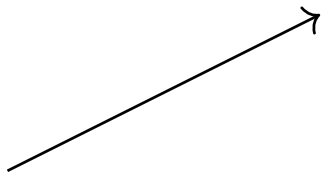
MOVGT R1 R0

Minimal Predication Example

```
if (x > 0) y = 10;
else y = 20;
```

Minimal Predication Example

```
if (x > 0) y = 10;
else y = 20;
```

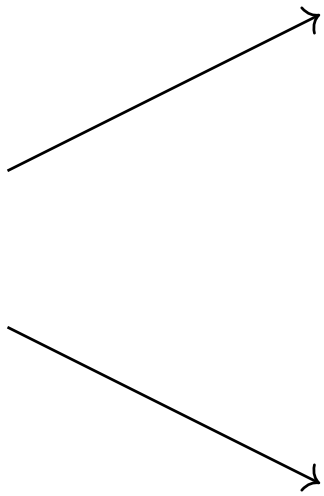


Before If-Conversion:

```
    cmp r0, #0
    bgt greater
    mov r1, #20
    b end
greater:
    mov r1, #10
end:
```

Minimal Predication Example

```
if (x > 0) y = 10;
else y = 20;
```



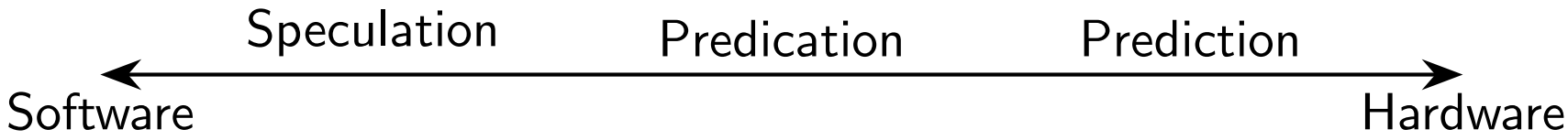
Before If-Conversion:

```
    cmp r0, #0
    bgt greater
    mov r1, #20
    b end
greater:
    mov r1, #10
end:
```

With If-Conversion:

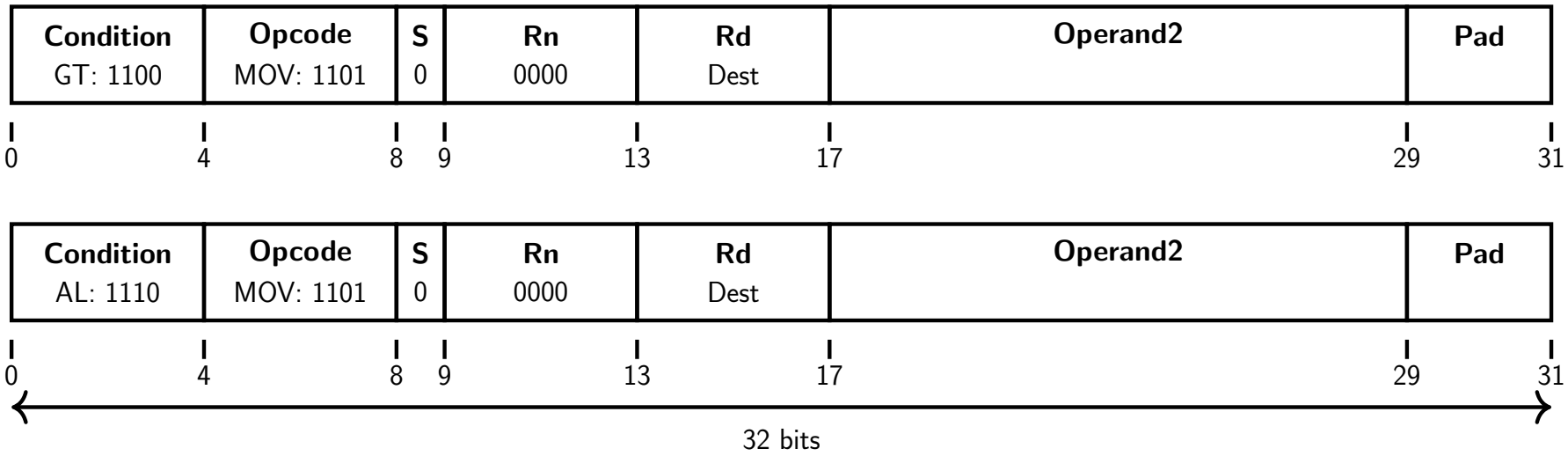
```
    cmp r0, #0
    movgt r1, #10
    movle r1, #20
```

Predication



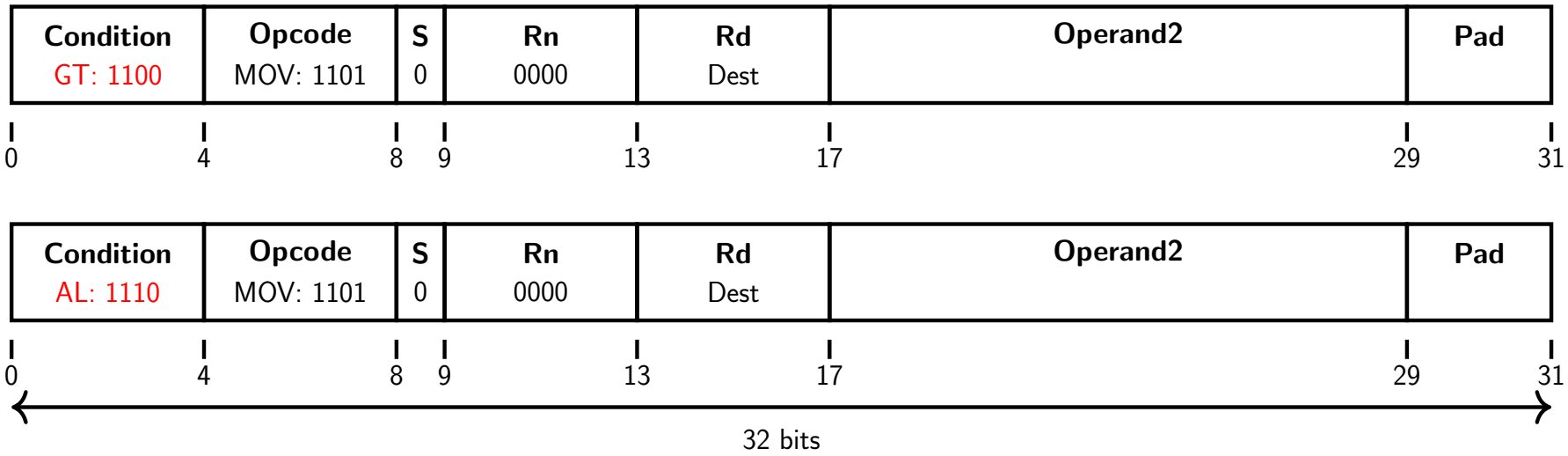
ARMv7 Predication

MOVGT vs MOV



ARMv7 Predication

MOVGT vs MOV



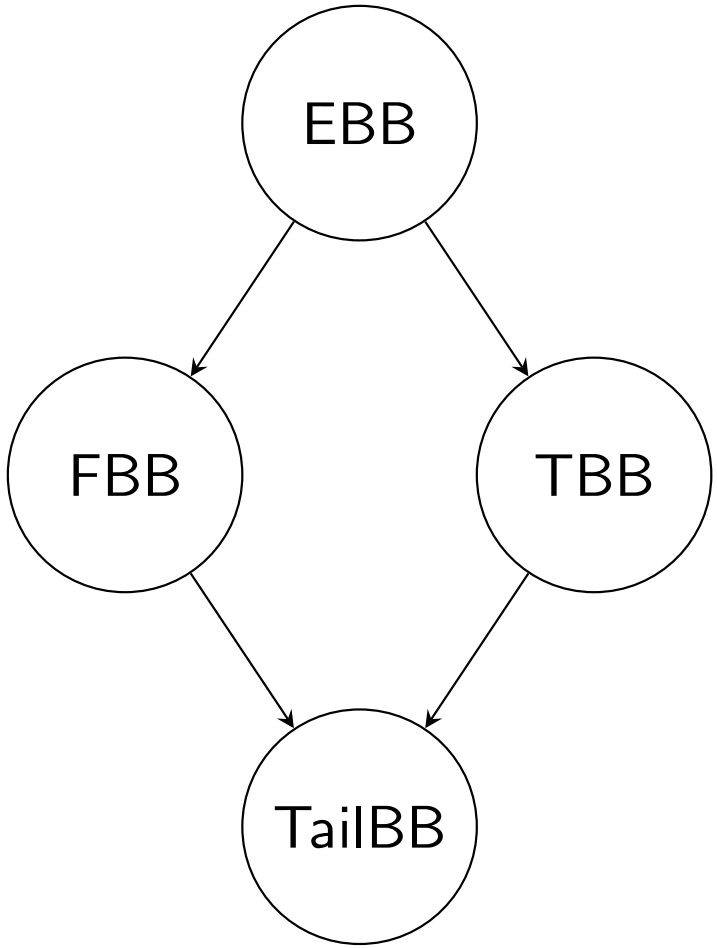
If-converting a real function

If-converting a real function

```
1  #include <cstdint>
2
3  uint32_t
4  computeLoan(bool isHouseLoan, int principal)
5  {
6    //EBB
7    uint32_t baseRate = principal * 2 / 100;
8
9    if (isHouseLoan) {
10     // TBB
11     baseRate = principal * 5 / 100;
12   } else {
13     // FBB
14     baseRate = principal * 7 / 100;
15   }
16
17   // TailBB
18   return (baseRate + 5) / 10 * 10;
19 }
```

If-converting a real function

```
1  #include <cstdint>
2
3  uint32_t
4  computeLoan(bool isHouseLoan, int principal)
5  {
6    //EBB
7    uint32_t baseRate = principal * 2 / 100;
8
9    if (isHouseLoan) {
10     // TBB
11     baseRate = principal * 5 / 100;
12   } else {
13     // FBB
14     baseRate = principal * 7 / 100;
15   }
16
17   // TailBB
18   return (baseRate + 5) / 10 * 10;
19 }
```



If-converting a real function

```
computeLoan(bool, int):
    push    {r11, lr}
    mov     r11, sp
    sub     sp, sp, #16
    and     r0, r0, #1
    strb    r0, [r11, #-1]
    str     r1, [sp, #8]
    ldr     r0, [sp, #8]
    lsl     r0, r0, #1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldrb    r0, [r11, #-1]
    tst     r0, #1
    beq     .LBB0_2
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_3
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    b       .LBB0_3
.LBB0_2:
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_2
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
.LBB0_3:
    ldr     r0, [sp, #4]
    add     r0, r0, #5
    ldr     r1, .LCPI0_4
    bl      __aeabi_uidiv
    ldr     r1, .LCPI0_4
    mul     r0, r0, r1
    mov     sp, r11
    pop     {r11, pc}
```

If-converting a real function

```
computeLoan(bool, int):
    push    {r11, lr}
    mov     r11, sp
    sub     sp, sp, #16
    and     r0, r0, #1
    strb    r0, [r11, #-1]
    str     r1, [sp, #8]
    ldr     r0, [sp, #8]
    lsl     r0, r0, #1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldrb    r0, [r11, #-1]
    tst     r0, #1
    beq     .LBB0_2
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_3
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    b       .LBB0_3
.LBB0_2:
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_2
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
.LBB0_3:
    ldr     r0, [sp, #4]
    add     r0, r0, #5
    ldr     r1, .LCPI0_4
    bl      __aeabi_uidiv
    ldr     r1, .LCPI0_4
    mul     r0, r0, r1
    mov     sp, r11
    pop     {r11, pc}
```

```
computeLoan(bool, int):
    push    {r11, lr}
    mov     r11, sp
    sub     sp, sp, #16
    and     r0, r0, #1
    strb    r0, [r11, #-1]
    str     r1, [sp, #8]
    ldr     r0, [sp, #8]
    lsl     r0, r0, #1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldrb    r0, [r11, #-1]
    ldr     r3, .LCPI0_3
    ldr     r4, .LCPI0_2
    cmp     r0, #1
    mov     r1, r3
    movne   r1, r4
    ldr     r0, [sp, #8]
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldr     r0, [sp, #4]
    add     r0, r0, #5
    ldr     r1, .LCPI0_4
    bl      __aeabi_uidiv
    ldr     r1, .LCPI0_4
    mul     r0, r0, r1
    mov     sp, r11
    pop     {r11, pc}
```

If-converting a real function

```
computeLoan(bool, int):
    push    {r11, lr}
    mov     r11, sp
    sub     sp, sp, #16
    and     r0, r0, #1
    strb    r0, [r11, #-1]
    str     r1, [sp, #8]
    ldr     r0, [sp, #8]
    lsl     r0, r0, #1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldrb    r0, [r11, #-1]
    tst     r0, #1
    beq     .LBB0_2
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_3
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    b       .LBB0_3
.LBB0_2:
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_2
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
.LBB0_3:
    ldr     r0, [sp, #4]
    add     r0, r0, #5
    ldr     r1, .LCPI0_4
    bl      __aeabi_uidiv
    ldr     r1, .LCPI0_4
    mul     r0, r0, r1
    mov     sp, r11
    pop     {r11, pc}
```

```
computeLoan(bool, int):
    push    {r11, lr}
    mov     r11, sp
    sub     sp, sp, #16
    and     r0, r0, #1
    strb    r0, [r11, #-1]
    str     r1, [sp, #8]
    ldr     r0, [sp, #8]
    lsl     r0, r0, #1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldrb    r0, [r11, #-1]
    ldr     r3, .LCPI0_3
    ldr     r4, .LCPI0_2
    cmp     r0, #1
    mov     r1, r3
    movne   r1, r4
    ldr     r0, [sp, #8]
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldr     r0, [sp, #4]
    add     r0, r0, #5
    ldr     r1, .LCPI0_4
    bl      __aeabi_uidiv
    ldr     r1, .LCPI0_4
    mul     r0, r0, r1
    mov     sp, r11
    pop     {r11, pc}
```

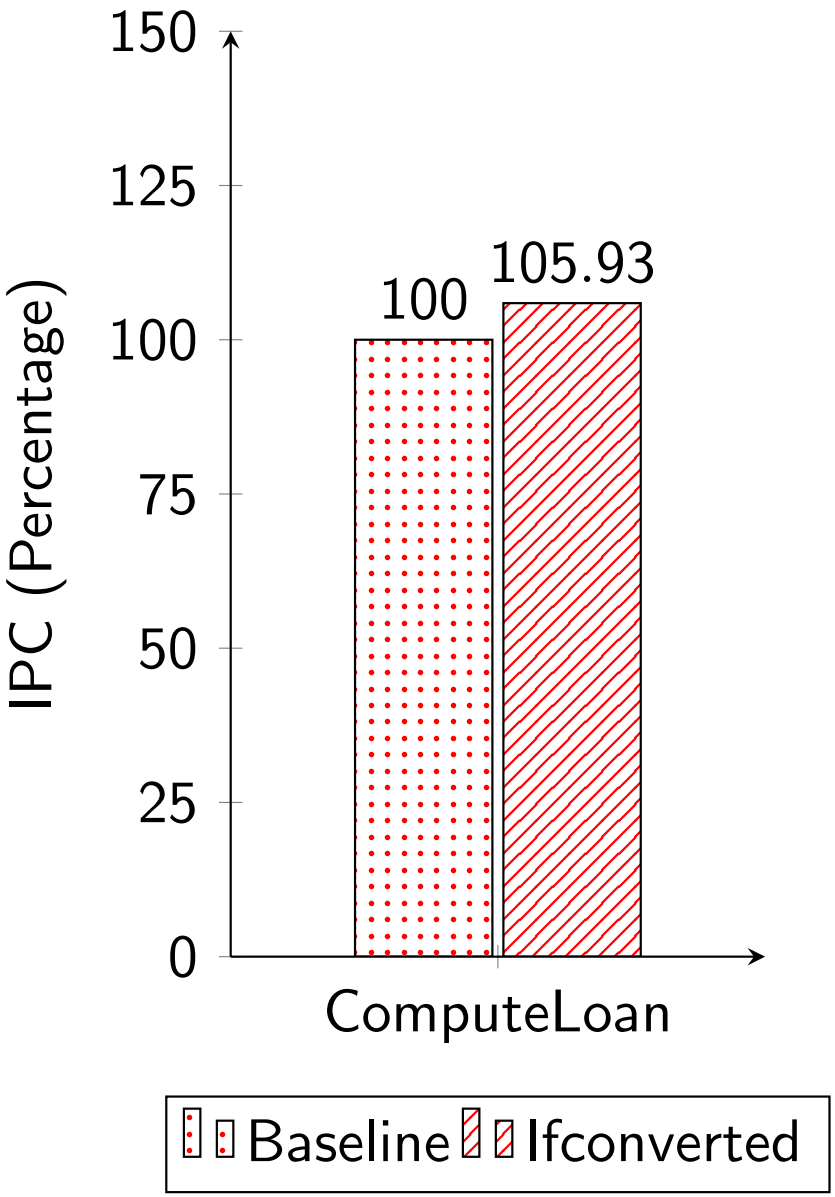
If-converting a real function

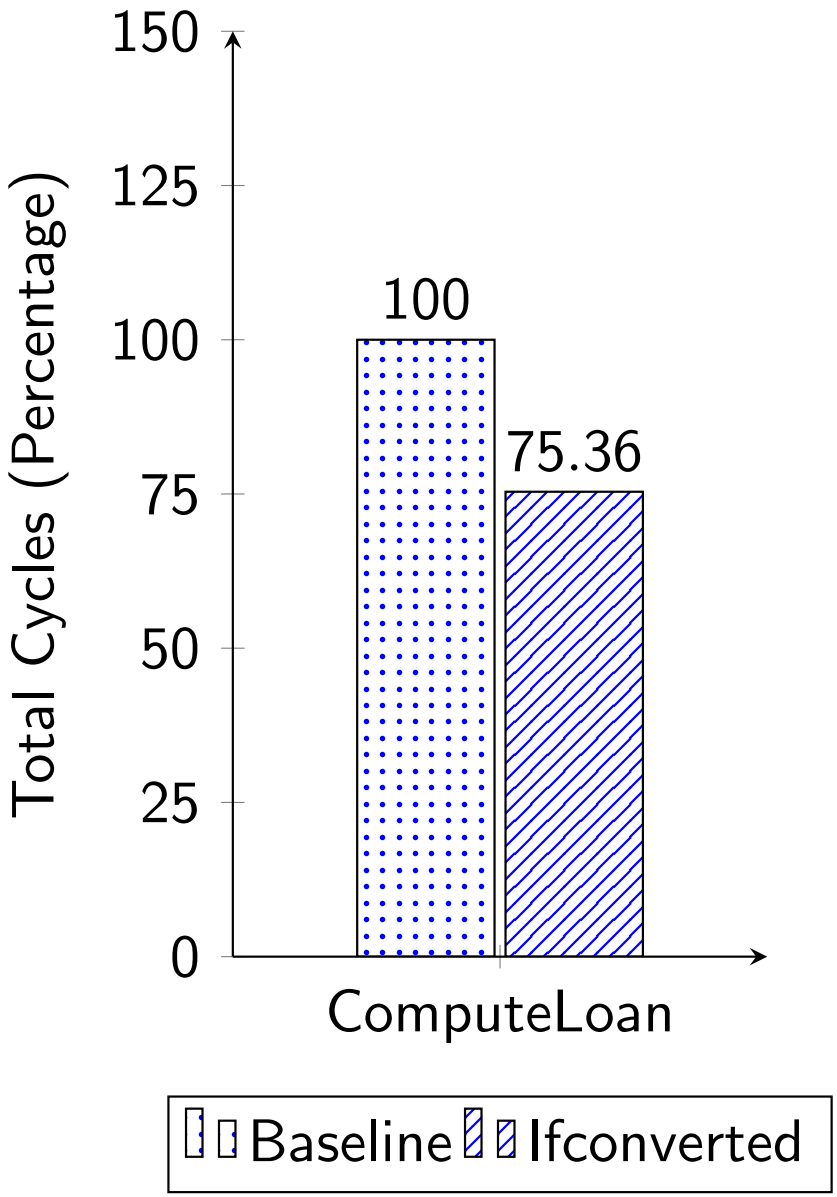
```
computeLoan(bool, int):
    push    {r11, lr}
    mov     r11, sp
    sub     sp, sp, #16
    and     r0, r0, #1
    strb    r0, [r11, #-1]
    str     r1, [sp, #8]
    ldr     r0, [sp, #8]
    lsl     r0, r0, #1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldrb    r0, [r11, #-1]
    tst     r0, #1
    beq     .LBB0_2
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_3
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    b       .LBB0_3
.LBB0_2:
    ldr     r0, [sp, #8]
    ldr     r1, .LCPI0_2
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
.LBB0_3:
    ldr     r0, [sp, #4]
    add     r0, r0, #5
    ldr     r1, .LCPI0_4
    bl      __aeabi_uidiv
    ldr     r1, .LCPI0_4
    mul     r0, r0, r1
    mov     sp, r11
    pop     {r11, pc}
```

```
computeLoan(bool, int):
    push    {r11, lr}
    mov     r11, sp
    sub     sp, sp, #16
    and     r0, r0, #1
    strb    r0, [r11, #-1]
    str     r1, [sp, #8]
    ldr     r0, [sp, #8]
    lsl     r0, r0, #1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldrb    r0, [r11, #-1]
    ldr     r3, .LCPI0_3
    ldr     r4, .LCPI0_2
    cmp     r0, #1
    mov     r1, r3
    movne   r1, r4
    ldr     r0, [sp, #8]
    mul     r0, r0, r1
    ldr     r1, .LCPI0_0
    bl      __aeabi_idiv
    str     r0, [sp, #4]
    ldr     r0, [sp, #4]
    add     r0, r0, #5
    ldr     r1, .LCPI0_4
    bl      __aeabi_uidiv
    ldr     r1, .LCPI0_4
    mul     r0, r0, r1
    mov     sp, r11
    pop     {r11, pc}
```

If-converting a real function, benchmarks

If-converting a real function, benchmarks





Pros and Cons of Predication

Pros

- ↑ Reduces Code Size

Pros and Cons of Predication

Pros

- ↑ Reduces Code Size
- ⇑ Removes Branches

Pros and Cons of Predication

Pros

- ↑ Reduces Code Size
- ⇑ Removes Branches
  - Removes branch misprediction penalty
  - Improves IPC
  - Makes execution time predictable

Pros and Cons of Predication

Pros

- ↑ Reduces Code Size
- ⇑ Removes Branches
  - Removes branch misprediction penalty
  - Improves IPC
  - Makes execution time predictable

Cons

- ↑ Increases Code Size

Pros and Cons of Predication

Pros

- ↑ Reduces Code Size
- ⇑ Removes Branches
  - Removes branch misprediction penalty
  - Improves IPC
  - Makes execution time predictable

Cons

- ↑ Increases Code Size
- ↑ Sometimes we need to execute more instructions

Pros and Cons of Predication

Pros

- ↑ Reduces Code Size
- ⇑ Removes Branches
  - Removes branch misprediction penalty
  - Improves IPC
  - Makes execution time predictable

Cons

- ↑ Increases Code Size
- ↑ Sometimes we need to execute more instructions
  - Increases the I-Cache pressure
  - Increase pressure on registers

Pros and Cons of Predication

Pros

- ↑ Reduces Code Size
- ⇑ Removes Branches
  - Removes branch misprediction penalty
  - Improves IPC
  - Makes execution time predictable

Cons

- ↑ Increases Code Size
- ↑ Sometimes we need to execute more instructions
  - Increases the I-Cache pressure
  - Increase pressure on registers
- ⇑ Makes the architecture more complex
  - More area and power consumption

Pros and Cons of Predication

Pros

- ↑ Reduces Code Size
- ⇑ Removes Branches
  - Removes branch misprediction penalty
  - Improves IPC
  - Makes execution time predictable

Cons

- ↑ Increases Code Size
- ↑ Sometimes we need to execute more instructions
  - Increases the I-Cache pressure
  - Increase pressure on registers
- ⇑ Makes the architecture more complex
  - More area and power consumption
- ⇑ Does not interact positively with OOO

Questions & References I

[1] Roger A. Bringmann, Scott A. Mahlke, and Wen-mei W. Hwu. "A study of the effects of compiler-controlled speculation on instruction and data caches". In: *HICSS (1)*. 1995, pp. 211–220. URL: https://doi.org/10.1109/HICSS.1995.375392.

[2] P.P. Chang et al. "Three architectural models for compiler-controlled speculative execution". In: *IEEE Transactions on Computers* 44.4 (1995), pp. 481–494. DOI: 10.1109/12.376164.

[3] Wen-mei Hwu et al. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation". In: *The Journal of Supercomputing* 7 (May 1993), pp. 229–248. DOI: 10.1007/BF01205185.

Questions & References II

[4] Scott A. Mahlke et al. "Sentinel scheduling: a model for compiler-controlled speculative execution". In: *ACM Trans. Comput. Syst.* 11.4 (Nov. 1993), pp. 376–408. ISSN: 0734-2071. DOI: 10.1145/161541.159765. URL: https://doi.org/10.1145/161541.159765.

[5] LLVM Project. *LLVM Project - Profile Data Tool*. https://llvm.org/docs/CommandGuide/llvm-profdata.html. Accessed: 27-Nov-2024.

[6] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. "Boosting beyond static scheduling in a superscalar processor". In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ISCA '90. Seattle, Washington, USA: Association for Computing Machinery, 1990, pp. 344–354. ISBN: 0897913663. DOI: 10.1145/325164.325160. URL: https://doi.org/10.1145/325164.325160.

Backup: Speculation Code Example

```
1     ldr r1, _ptr        ; Initialize
2     mov r7, 0
3     mov r2, 0
4     mov r3, 0
5     beq L3, r1, 0
6  L0:
7     add r2, r2, 1       ; I1: Increment count
8     ldr r4, 0[r1]       ; I2: Load ptr->wt into r4
9     btl L1, r4, 0       ; I3: If wt < 0, branch to L1
10    add r3, r3, r4      ; I4: Add wt to weight
11    ldr r5, 4[r1]       ; I5: Load ptr->next into r5
12    beq L3, r5, 0       ; I6: If next is NULL, jump to L3
13    add r2, r2, 1       ; I7: Increment count
14    ldr r6, 0[r5]       ; I8: Load next->wt into r6
15    btl L1X, r6, 0      ; I9: If wt < 0, branch to L1X
16    add r3, r3, r6      ; I10: Add wt to weight
17    ldr r1, 4[r5]       ; I11: Move ptr to ptr->next->next
18    bne L0, r1, 0       ; I12: Loop back to L0 if ptr != NULL
19 L3:
20    beq L4, r2, 0       ; If count == 0, skip division
21    div r7, r3, r2
22    str _avg, r7
23 L4:
24 ; ...
25 L1X:
26    mov r1, r5          ; Adjust ptr = ptr->next
27    mov r4, r6          ; Move wt to r4 for subtraction
28 L1:
29    sub r3, r3, r4      ; Subtract wt from weight
30    ldr r1, 4[r1]       ; Move ptr to ptr->next
31    bne L0, r1, 0       ; Loop back to L0 if ptr != NULL
```

Sentinel Scheduling

Properties of Sentinel Scheduling [4]

- *Resolve errors*
- Lifts Restriction 2
- Split excepting instructions into two:
- Operation-part which can be moved
- Sentinel-part remains in home block and checks for exceptions

Hardware for Sentinel Scheduling

- Additional bit in opcode to mark speculative instructions
- Registers need exception tag
- Sentinel instruction (a `mov`)