Cross-Architecture Translation Analysis: x86 → ARM64

1. C Code to Assembly Mapping (GD ARM64 Version)

Function Signature and Register Mapping

```c
int* func0(const char* paren_string, int* returnSize)
```

ARM64 Register Assignments:

- `x0` (x20) = `paren_string` (input string pointer)
- `x1` (x19) = `returnSize` (pointer to return size)
- `x0` (reused) = `all_levels` (return array pointer)
- `w8` = `level` (current nesting level)
- `w22` = `max_level` (maximum level seen)
- `w21` = `count` (number of complete groups)
- `w9` = `chr` (current character)

Line-by-Line C to ARM64 Assembly Mapping

C Lines: Variable Initialization

```c
int* all_levels = NULL;
int level = 0, max_level = 0, i = 0, count = 0;
```

```assembly
mov x19, x1          ; save returnSize pointer
mov x20, x0          ; save paren_string pointer
mov w8, #0           ; level = 0
mov w22, #0          ; max_level = 0
mov w21, #0          ; count = 0
mov x0, #0           ; all_levels = NULL
```

C Line: `for (i = 0; paren_string[i] != '\0'; i++)`

assembly

```
LBB0_3:
ldrb  w9, [x20]        ; chr = paren_string[i]
; ... process character ...
add x20, x20, #1     ; i++ (increment pointer)
```

C Line: `if (chr == '(')`

assembly

```
cmp w9, #40          ; compare chr with '(' (ASCII 40)
b.eq  LBB0_7          ; branch if equal
```

C Line: `level += 1; if (level > max_level) max_level = level;`

assembly

```
LBB0_7:
cmp w8, w22          ; compare level with max_level
csinc w22, w22, w8, lt ; if level > max_level, max_level = level
add w8, w8, #1       ; level += 1
```

C Line: `else if (chr == ')')`

assembly

```
cmp w9, #41          ; compare chr with ')' (ASCII 41)
b.ne  LBB0_1          ; branch if not equal
```

C Line: `level -= 1;`

assembly

```
subs  w8, w8, #1      ; level -= 1 (and set flags)
b.ne  LBB0_2          ; if level != 0, continue
```

C Lines: `all_levels = realloc(...); all_levels[count++] = max_level;`

```assembly
add w23, w21, #1     ; temp = count + 1
sbfiz x1, x23, #2, #32 ; x1 = temp * sizeof(int)
bl  _realloc         ; call realloc
str w22, [x0, w21, sxtw #2] ; all_levels[count] = max_level
mov x21, x23         ; count = temp
mov w22, #0          ; max_level = 0
```

2. x86 to ARM64 Translation Analysis

Register Mapping Comparison

Variable	x86	ARM64 GD	ARM64 Pred
`paren_string`	`%rbx`	`x20`	`x20` ✓
`returnSize`	`%r14`	`x19`	`x19` ✓
`all_levels`	`%rax`	`x0`	`x0` ✓
`level`	`%ecx`	`w8`	`w8` ✓
`max_level`	`%r12d`	`w22`	`w22` ✓
`count`	`%r15d`	`w21`	`w21` ✓
`chr`	`%edx`	`w9`	`w9` ✓

3. Vertical Comparison: GD vs Predicted

Critical Differences Identified

Line	GD (Correct)	Predicted (Incorrect)	Issue Type
39	`add w23, w21, #1`	`sbfiz x23, x21, #2, #32`	**CRITICAL: Wrong operation order**
40	`sbfiz x1, x23, #2, #32`	`add x1, x23, #4`	**CRITICAL: Incorrect size calculation**

42-44	`str w22, [x0, w21, sxtw #2]`
	`mov x21, x23`
	`mov w22, #0`
		`mov w8, #0`
		`mov w22, #0`
		`str w22, [x0, x23]`
		`add w21, w21, #1`
			CRITICAL: Wrong store logic

4. Root Cause Analysis: Translation Failures

Error 1: Incorrect Operation Sequencing

x86 Source:

```assembly
movslq  %r15d, %r13      ; r13 = count (sign extend)
incl  %r15d            ; count++
leaq  4(,%r13,4), %rsi ; rsi = (count * 4) + 4 = (count+1)*4
```

ARM64 GD (Correct Translation):

```assembly
add w23, w21, #1         ; temp = count + 1
sbfiz x1, x23, #2, #32     ; x1 = temp * 4
```

ARM64 Predicted (Incorrect Translation):

```assembly
sbfiz x23, x21, #2, #32    ; x23 = count * 4 (WRONG!)
add x1, x23, #4          ; x1 = (count * 4) + 4
```

Problem: The prediction model misunderstood the x86 `leaq 4(,%r13,4), %rsi` instruction. It translated it as two separate operations instead of recognizing it calculates `(count+1)*4`.

Error 2: Store Operation Logic Error

x86 Source:

```assembly
movl  %r12d, (%rax,%r13,4) ; all_levels[count] = max_level
xorl  %ecx, %ecx           ; level = 0
xorl  %r12d, %r12d         ; max_level = 0
```

ARM64 GD (Correct):

```assembly
str w22, [x0, w21, sxtw #2] ; all_levels[count] = max_level
mov x21, x23                ; count = temp (count+1)
mov w22, #0                 ; max_level = 0
```

ARM64 Predicted (Incorrect):

```assembly
mov w8, #0          ; level = 0 (correct)
mov w22, #0         ; max_level = 0 (correct)
str w22, [x0, x23]  ; all_levels[count*4] = 0 (WRONG VALUE & INDEX!)
add w21, w21, #1    ; count++ (should be done before store)
```

Critical Issues:

1. **Wrong Value**: Stores `0` (max_level after reset) instead of the actual max_level
2. **Wrong Index**: Uses `x23` (which is `count*4`) as byte offset instead of element index
3. **Wrong Timing**: Increments count after the store instead of before

5. Why the Prediction Failed

Cross-Architecture Complexity

The prediction model failed to properly handle several x86→ARM64 translation challenges:

1. Complex x86 Addressing Modes

x86's `leaq 4(,%r13,4), %rsi` is a single instruction that:

- Takes a register value (`%r13`)
- Multiplies by 4
- Adds 4
- Stores result in destination

ARM64 requires multiple instructions for this, and the model got the sequence wrong.

2. Instruction Reordering Misunderstanding

x86 code:

```assembly
movslq  %r15d, %r13     # temp = count
incl  %r15d           # count++
movl  %r12d, (%rax,%r13,4) # store at old count position
```

The model failed to recognize that:

- The store uses the OLD count value (before increment)
- The increment happens for the NEXT iteration
- ARM64 needs different sequencing due to addressing mode differences

3. Register Usage Pattern Confusion

The model correctly mapped most registers but failed when temporary calculations were needed. It confused:

- Temporary values vs. final values
- Calculation order vs. usage order

6. Connection to O2 Optimization

These errors reveal fundamental issues with cross-architecture compiler optimization translation:

Optimization Strategy Mismatch

- **x86 O2**: Uses complex addressing modes to minimize instructions
- **ARM64 O2**: Uses more instructions but simpler addressing for better pipeline efficiency
- **Prediction Error**: Tried to mimic x86 structure instead of using ARM64-optimal patterns

Data Flow Analysis Failure

- **O2 Optimization** performs sophisticated data flow analysis to determine when values are live
- **Prediction Model** failed to track that `max_level` needed to be stored BEFORE being reset to 0

Architecture-Specific Instruction Selection

- **x86**: Complex instructions like `leaq` can do multiplication + addition in one step
- **ARM64**: Separate shift/add instructions are often more efficient
- **Prediction**: Incorrectly tried to preserve x86 instruction boundaries

7. Summary

The prediction contains **3 critical logical errors** that make it functionally incorrect:

1. **Wrong realloc size calculation** - allocates incorrect memory size
2. **Wrong stored value** - stores 0 instead of the actual max_level
3. **Wrong array indexing** - uses byte offset instead of element index

These errors stem from the model's failure to understand:

- x86 complex addressing mode semantics
- Proper instruction reordering for different architectures
- ARM64-specific optimization patterns

This demonstrates why cross-architecture code translation is extremely challenging - it requires deep understanding of both architectures' instruction semantics, addressing modes, and optimization

strategies.