String Filter Function: x86 → ARM64 Translation Analysis

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

Function Signature and Register Mapping

```c
char **func0(char **strings, int size, const char *substring, int *out_size)
```

ARM64 Register Assignments:

- `x0` (x21) = `strings` (input string array)
- `x1` (x24) = `size` (array size, used as counter)
- `x2` (x20) = `substring` (search string)
- `x3` (x19) = `out_size` (pointer to output size)
- `x22` = `out` (result array pointer)
- `w23` = `count` (number of matches found)
- `w25` = temporary for `count + 1`

Line-by-Line C to ARM64 Assembly Mapping

C Lines: Variable Initialization

```c
char **out = NULL;
int count = 0;
```

```assembly
mov x22, #0           ; out = NULL
mov w23, #0           ; count = 0
```

C Line: `for (int i = 0; i < size; i++)`

assembly

```
mov w24, w1          ; initialize counter with size
LBB0_3:              ; loop header
; ... loop body ...
add x21, x21, #8     ; strings++ (next pointer)
subs x24, x24, #1    ; size-- (decrement counter)
b.eq LBB0_6          ; if counter == 0, exit loop
```

C Line: `if (strstr(strings[i], substring) != NULL)`

assembly

```
ldr x0, [x21]        ; load strings[i]
mov x1, x20          ; load substring
bl  _strstr          ; call strstr
cbz x0, LBB0_2       ; if result == NULL, continue loop
```

C Line: `out = (char **)realloc(out, sizeof(char *) * (count + 1));`

assembly

```
add w25, w23, #1     ; temp = count + 1
sbfiz x1, x25, #3, #32 ; x1 = temp * 8 (sizeof(char*))
mov x0, x22          ; load current out pointer
bl  _realloc         ; call realloc
mov x22, x0          ; out = realloc result
```

C Lines: `out[count] = strings[i]; count++;`

assembly

```
ldr x8, [x21]        ; load strings[i]
str x8, [x0, w23, sxtw #3] ; out[count] = strings[i]
mov x23, x25         ; count = temp (count + 1)
```

2. x86 to ARM64 Translation Analysis

Register Mapping Comparison

Variable	x86	ARM64 GD	ARM64 Pred
`strings`	`%r12`	`x21`	`x22` ✗
`size`	`-56(%rbp)`	`x24`	`x24` ✓
`substring`	`-64(%rbp)`	`x20`	`x21` ✗
`out_size`	`-48(%rbp)`	`x19`	`x19` ✓
`out`	`%rbx`	`x22`	`x20` ✗
`count`	`%r14d`	`w23`	`w23` ✓
`i`	`%r15`	implicit	implicit ✓

3. Vertical Comparison: GD vs Predicted

Critical Differences Identified

Line	GD (Correct)	Predicted (Incorrect)	Issue Type
19-22	`mov x20, x2`<br>`mov x21, x0`<br>`mov w23, #0`<br>`mov x22, #0`	`mov x21, x2`<br>`mov x22, x0`<br>`mov w23, #0`<br>`mov x20, #0`	**CRITICAL: Register assignment confusion**
26-30	`add x21, x21, #8`<br>`subs x24, x24, #1`<br>`b.eq LBB0_6`<br>`ldr x0, [x21]`<br>`mov x1, x20`	`add x22, x22, #8`<br>`subs x24, x24, #1`<br>`b.eq LBB0_6`<br>`ldr x0, [x22]`<br>`mov x1, x21`	**CRITICAL: Wrong registers used**
34-38	`add w25, w23, #1`<br>`sbfiz x1, x25, #3, #32`<br>`mov x0, x22`<br>`mov x22, x0`<br>`str x8, [x0, w23, sxtw #3]`	`add w23, w23, #1`<br>`sbfiz x1, x23, #3, #32`<br>`mov x0, x20`<br>`mov x20, x0`<br>`str x8, [x0, w23, sxtw #3]`	**CRITICAL: Wrong increment timing**

4. Root Cause Analysis: Translation Failures

Error 1: Register Assignment Confusion

x86 Source Pattern:

```assembly
movq   %rdi, %r12        ; strings -> %r12
movq   %rdx, -64(%rbp)   ; substring -> stack
xorl   %ebx, %ebx        ; out = NULL
```

ARM64 GD (Correct Translation):

```assembly
mov x21, x0          ; strings -> x21
mov x20, x2          ; substring -> x20
mov x22, #0          ; out = NULL
```

ARM64 Predicted (Incorrect Translation):

```assembly
mov x22, x0          ; strings -> x22 (WRONG!)
mov x21, x2          ; substring -> x21 (WRONG!)
mov x20, #0          ; out = x20 (WRONG!)
```

Problem: The prediction model swapped the register assignments for critical variables, causing all subsequent operations to use wrong data.

Error 2: Increment Timing Error

x86 Source:

```assembly
movslq  %r14d, %r13     ; r13 = count (old value)
incl  %r14d            ; count++ (increment for next)
leaq  8(,%r13,8), %rsi ; size = (old_count + 1) * 8
; ... realloc ...
movq  %rax, (%rbx,%r13,8) ; store at old_count index
```

ARM64 GD (Correct):

```assembly
add w25, w23, #1        ; temp = count + 1
sbfiz x1, x25, #3, #32   ; size = temp * 8
; ... realloc ...
str x8, [x0, w23, sxtw #3] ; store at old count
mov x23, x25            ; count = temp
```

ARM64 Predicted (Incorrect):

```assembly
add w23, w23, #1        ; count++ (TOO EARLY!)
sbfiz x1, x23, #3, #32   ; size = new_count * 8
; ... realloc ...
str x8, [x0, w23, sxtw #3] ; store at NEW count (WRONG INDEX!)
```

Critical Issue: The prediction increments `count` BEFORE using it as the store index, causing an off-by-one error that writes past the allocated array bounds.

5. Detailed Error Impact Analysis

Error Impact 1: Complete Data Flow Corruption

Due to register swapping:

- `strings` pointer is stored in `x22` instead of `x21`
- `substring` pointer is stored in `x21` instead of `x20`
- `out` array pointer is stored in `x20` instead of `x22`

This means:

- String loads use the wrong base address
- `strstr` calls receive wrong arguments
- Array stores target wrong memory locations

Error Impact 2: Array Bounds Violation

The incorrect increment timing causes:

```c
// What should happen:
out[count] = strings[i];  // store at index 'count'
count++;                  // increment for next

// What prediction does:
count++;                  // increment first
out[count] = strings[i];  // store at index 'count+1' (WRONG!)
```

This creates a buffer overflow where the first element is never written, and writes occur one position beyond the allocated space.

6. Why the x86→ARM64 Translation Failed

Complex x86 Addressing Mode Misinterpretation

x86 Pattern:

```assembly
leaq  8(,%r13,8), %rsi    ; rsi = (r13 * 8) + 8 = (count + 1) * 8
```

Correct ARM64 Translation:

```assembly
add w25, w23, #1        ; temp = count + 1
sbfiz x1, x25, #3, #32   ; x1 = temp * 8
```

Incorrect Prediction:

```assembly
add w23, w23, #1        ; count = count + 1 (immediate increment)
sbfiz x1, x23, #3, #32   ; x1 = count * 8 (using incremented value)
```

Register Pressure Handling Confusion

x86 uses stack spills for some variables while ARM64 has more registers available. The prediction model failed to properly map the register allocation strategy between architectures.

Instruction Scheduling Misunderstanding

The model didn't understand that x86's complex addressing allows calculating the new size while preserving the old index value for the store operation.

7. Connection to O2 Optimization

Architecture-Specific Optimization Strategies

- **x86 O2**: Heavy use of complex addressing modes and stack spills
- **ARM64 O2**: More registers available, simpler addressing modes preferred
- **Translation Error**: Model tried to preserve x86 instruction ordering instead of generating optimal ARM64 patterns

Data Dependency Analysis Failure

- **O2 Optimization** performs sophisticated analysis of when values are needed
- **Prediction Model** failed to recognize the data dependency between array indexing and counter increment timing

8. Summary

The prediction contains **2 catastrophic errors** that make it completely non-functional:

Error 1: Register Assignment Chaos (Lines 19-22)

- **Impact**: Complete corruption of data flow
- **Cause**: Misunderstanding of x86 register allocation vs ARM64 register usage

- **Result**: Function operates on completely wrong data

Error 2: Array Index Off-by-One (Lines 34-38)

- **Impact**: Buffer overflow and incorrect array contents
- **Cause**: Misinterpreting x86 complex addressing mode timing
- **Result**: Memory corruption and undefined behavior

Root Cause: Cross-Architecture Semantic Gap

The prediction model failed to understand that successful cross-architecture translation requires:

1. **Algorithmic understanding** rather than instruction-by-instruction mapping
2. **Architecture-specific optimization patterns** rather than preserving source patterns
3. **Data flow analysis** to maintain correctness across different instruction sequences

This example perfectly demonstrates why cross-architecture code translation is one of the most challenging problems in compiler technology - it requires deep semantic understanding of both the algorithm and the target architecture's optimal implementation patterns.