Mohammad Iqbal

CSC 210

12/09/2024

**Assembly Analysis and Optimization Report for QuickSort**

**Overview**

This report compares two assembly implementations of a simple swap function, which swaps the values of two integer variables by using their pointers. The first implementation is compiler-generated, and the second is an optimized version designed to minimize redundancies and enhance performance.

This report compares two assembly implementations of the partition function, a core routine commonly used in quicksort algorithms. The first implementation is compiler-generated, while the second is optimized for performance and clarity by reducing stack operations and maximizing register usage.

This report compares two assembly implementations of the quickSort function, an essential recursive algorithm for sorting arrays. The first implementation is compiler-generated, while the second is optimized to improve efficiency by leveraging registers and reducing stack usage.

**Compiler-Generated Assembly Analysis**

**Assembly Breakdown**

The first implementation involves redundant instructions and unnecessary stack usage. Key steps include:

1. **Prologue:**
   * Standard stack setup using pushq %rbp and movq %rsp, %rbp.
2. **Stack Storage:**
   * The function stores the pointers (%rdi and %rsi) in stack memory (-24(%rbp) and -32(%rbp)).
   * It fetches the values from these memory locations multiple times.
3. **Value Swapping:**
   * Values are loaded into registers (%eax and %edx) after dereferencing the pointers.
   * The first value is temporarily stored in a stack variable (-4(%rbp)) before being used to complete the swap.

**Weaknesses**

* **Redundant Memory Access:**
  + The implementation frequently stores and retrieves data from the stack, even though it is unnecessary for a simple swap operation.
* **Performance Overhead:**
  + Stack operations (pushq, popq) and multiple memory dereferences increase execution time.
* **Register Underutilization:**
  + The function does not fully utilize available registers to streamline the operations.

**Optimized Assembly Analysis**

**Assembly Breakdown**

The optimized implementation eliminates stack operations and uses registers exclusively for the entire process:

1. **Elimination of Prologue/Epilogue:**
   * The prologue (pushq %rbp) and epilogue (popq %rbp) are omitted, removing stack overhead.
2. **Direct Register Usage:**
   * The function directly loads pointers into registers (%rax for the first pointer and %rdx for the second).
   * Values are loaded into registers (%ecx for the first value and %ebx for the second) without using temporary stack storage.
3. **Efficient Swapping:**
   * The second value is directly stored at the first pointer's location.
   * The first value is directly stored at the second pointer's location.

**Strengths**

* **Reduced Instructions:**
  + The optimized implementation reduces the number of instructions by removing unnecessary memory accesses and stack manipulations.
* **Improved Register Usage:**
  + The function efficiently uses general-purpose registers (%rax, %rdx, %ecx, %ebx) for intermediate data storage and manipulation.
* **Lower Latency:**
  + By avoiding stack operations and reducing memory dereferences, the function improves execution speed.

**Performance Comparison**

A screenshot of a computer

Description automatically generated

**Conclusion**

The optimized swap function significantly outperforms the compiler-generated version in terms of efficiency and execution speed. By eliminating unnecessary stack operations and focusing on direct register usage, the optimized code reduces the instruction count and minimizes memory overhead.

**Compiler-Generated Assembly Analysis**

**Assembly Breakdown**

1. **Prologue and Stack Usage:**
   * The compiler-generated implementation establishes a stack frame using pushq %rbp and adjusts the stack pointer (subq $32, %rsp).
   * Arguments (%rdi, %esi, %edx) are stored in stack variables (-24(%rbp), -28(%rbp), -32(%rbp)), and local variables are also allocated on the stack.
2. **Pivot Selection:**
   * The pivot value is fetched from the array using an indexed addressing mode:

movl -32(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

movq -24(%rbp), %rax

addq %rdx, %rax

movl (%rax), %eax

movl %eax, -4(%rbp)

1. **Partitioning Loop:**
   * The function iterates over the array and compares each element with the pivot:
     + Values smaller than the pivot are swapped using the swap function (\_Z4swapRiS\_).
     + Indices and counters are updated on the stack (-8(%rbp), -12(%rbp)).
2. **Final Swap:**
   * After completing the loop, the pivot is swapped into its correct position.
3. **Epilogue:**
   * Stack is restored (leave) and control is returned (ret).

**Weaknesses**

* **Excessive Stack Usage:**
  + The implementation relies heavily on stack storage for arguments, loop counters, and intermediate variables, causing unnecessary memory operations.
* **Register Underutilization:**
  + Registers are underused, with only temporary values loaded into them during operations.
* **Performance Overhead:**
  + Frequent stack access and function calls (\_Z4swapRiS\_) add latency.

**Optimized Assembly Analysis**

**Assembly Breakdown**

1. **Direct Register Usage:**
   * All function arguments are directly stored in registers:
     + %r8 for the array pointer.
     + %r9d for the starting index.
     + %r10d for the ending index.
2. **Pivot Selection:**
   * The pivot value is loaded into %r11d without intermediate stack storage.
3. **Partitioning Loop:**
   * Indices and counters are stored in registers (%r12d, %r13d), and values are compared directly in registers.
   * Swap operations call \_Z4swapRiS\_ with minimal overhead, as the pointers are already in registers (%rdi, %rsi).
4. **Final Swap:**
   * The pivot is swapped into its correct position using efficient register addressing.
5. **Return:**
   * The function directly returns the result in %eax without any epilogue setup.

**Strengths**

* **Elimination of Stack Usage:**
  + No stack frame is established, reducing memory operations.
* **Efficient Register Management:**
  + All intermediate values and loop counters are managed in registers, maximizing performance.
* **Reduced Instruction Count:**
  + The optimized version requires fewer instructions by avoiding stack setup and teardown.

**Performance Comparison:**

A screenshot of a computer

Description automatically generated

**Conclusion**

The optimized partition function demonstrates significant improvements in execution speed and efficiency compared to the compiler-generated version. By eliminating unnecessary stack operations and leveraging registers for all intermediate values, the optimized implementation reduces latency and instruction overhead. This optimization is particularly impactful in recursive algorithms like quicksort, where the partition function is invoked repeatedly.

**Compiler-Generated Assembly Analysis**

**Assembly Breakdown**

1. **Prologue and Stack Setup:**
   * Establishes a stack frame using pushq %rbp and adjusts the stack pointer (subq $32, %rsp).
   * Stores function arguments (%rdi, %esi, %edx) in stack variables (-24(%rbp), -28(%rbp), -32(%rbp)).
2. **Base Case Check:**
   * Compares the starting index (-28(%rbp)) with the ending index (-32(%rbp)) to determine if the recursion should terminate:

movl -28(%rbp), %eax

cmpl -32(%rbp), %eax

jge .L12

1. **Partitioning:**
   * Calls the partition function (\_Z9partitionPiii) to partition the array and determine the pivot index. The result is stored in a stack variable (-4(%rbp)).
2. **Recursive Calls:**
   * Makes two recursive calls to quickSort:
     + Left side: Processes elements before the pivot.
     + Right side: Processes elements after the pivot.
   * Arguments for the recursive calls are passed via stack variables.
3. **Epilogue:**
   * Restores the stack and returns control to the caller.

**Weaknesses**

* **Excessive Stack Usage:**
  + Arguments and intermediate variables are stored on the stack, leading to frequent memory operations.
* **Limited Register Usage:**
  + Only temporary values are loaded into registers, while function arguments and results rely heavily on stack variables.
* **Performance Overhead:**
  + Frequent stack setup and teardown, along with multiple memory accesses, slow down the implementation.

**Optimized Assembly Analysis**

**Assembly Breakdown**

1. **Direct Register Usage:**
   * Arguments are directly assigned to registers:
     + %r14 for the array pointer.
     + %r15d for the starting index.
     + %edx (stored temporarily on the stack) for the ending index.
2. **Base Case Check:**
   * Compares the start and end indices directly using registers (%r15d and edx), avoiding stack variables.
3. **Partitioning:**
   * Calls the partition function with the array pointer and indices passed directly in registers (%rdi, %rsi, %rdx). The result is stored in a register (%eax).
4. **Recursive Calls:**
   * Left side and right side of the array are processed recursively, with arguments passed through registers:
     + Left side: The ending index is set to pivot - 1 using leal -1(%rax), %edx.
     + Right side: The starting index is set to pivot + 1 using leal 1(%rax), %ecx.
5. **Epilogue:**
   * Minimal stack usage is maintained, and the stack frame is restored with a simple leave and ret.

**Strengths**

* **Reduced Stack Usage:**
  + Most arguments and results are stored and passed via registers, minimizing memory accesses.
* **Improved Register Utilization:**
  + Registers like %r14, %r15d, %eax, and %edx handle data efficiently, avoiding redundant stack operations.
* **Lower Latency:**
  + By reducing stack setup/teardown and memory operations, the optimized implementation runs faster.

**Performance Comparison**

A screenshot of a computer

Description automatically generated

**Conclusion**

The optimized quickSort function demonstrates significant improvements over the compiler-generated version by relying on register usage for argument passing, condition checks, and intermediate computations. The reduced dependency on stack variables lowers memory access overhead, making the optimized version more suitable for performance-critical applications, particularly when dealing with large datasets.

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

\_Z9partitionPiii:

.LFB2277:

.cfi\_startproc

endbr64

movq %rdi, %r8

movl %esi, %r9d

movl %edx, %r10d

movl %r10d, %eax

cltq

leaq 0(,%rax,4), %rdx

movq %r8, %rax

addq %rdx, %rax

movl (%rax), %eax

movl %eax, %r11d

movl %r9d, %eax

subl $1, %eax

movl %eax, %r12d

movl %r9d, %eax

movl %eax, %r13d

jmp .L6

.L8:

movl %r13d, %eax

cltq

leaq 0(,%rax,4), %rdx

movq %r8, %rax

addq %rdx, %rax

movl (%rax), %eax

cmpl %eax, %r11d

jl .L7

addl $1, %r12d

movl %r13d, %eax

cltq

leaq 0(,%rax,4), %rdx

movq %r8, %rax

addq %rax, %rdx

movl %r12d, %eax

cltq

leaq 0(,%rax,4), %rcx

movq %r8, %rax

addq %rcx, %rax

movq %rdx, %rsi

movq %rax, %rdi

call \_Z4swapRiS\_

.L7:

addl $1, %r13d

.L6:

cmpl %r10d, %r13d

jl .L8

movl %r10d, %eax

cltq

leaq 0(,%rax,4), %rdx

movq %r8, %rax

addq %rax, %rdx

movl %r12d, %eax

cltq

addq $1, %rax

leaq 0(,%rax,4), %rcx

movq %r8, %rax

addq %rcx, %rax

movq %rdx, %rsi

movq %rax, %rdi

call \_Z4swapRiS\_

movl %r12d, %eax

addl $1, %eax

ret

.cfi\_endproc

\_Z9partitionPiii:

.LFB2277:

.cfi\_startproc

endbr64

pushq %rbp

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

movq %rsp, %rbp

.cfi\_def\_cfa\_register 6

subq $32, %rsp

movq %rdi, -24(%rbp)

movl %esi, -28(%rbp)

movl %edx, -32(%rbp)

movl -32(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

movq -24(%rbp), %rax

addq %rdx, %rax

movl (%rax), %eax

movl %eax, -4(%rbp)

movl -28(%rbp), %eax

subl $1, %eax

movl %eax, -12(%rbp)

movl -28(%rbp), %eax

movl %eax, -8(%rbp)

jmp .L6

.L8:

movl -8(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

movq -24(%rbp), %rax

addq %rdx, %rax

movl (%rax), %eax

cmpl %eax, -4(%rbp)

jl .L7

addl $1, -12(%rbp)

movl -8(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

movq -24(%rbp), %rax

addq %rax, %rdx

movl -12(%rbp), %eax

cltq

leaq 0(,%rax,4), %rcx

movq -24(%rbp), %rax

addq %rcx, %rax

movq %rdx, %rsi

movq %rax, %rdi

call \_Z4swapRiS\_

.L7:

addl $1, -8(%rbp)

.L6:

movl -8(%rbp), %eax

cmpl -32(%rbp), %eax

jl .L8

movl -32(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

movq -24(%rbp), %rax

addq %rax, %rdx

movl -12(%rbp), %eax

cltq

addq $1, %rax

leaq 0(,%rax,4), %rcx

movq -24(%rbp), %rax

addq %rcx, %rax

movq %rdx, %rsi

movq %rax, %rdi

call \_Z4swapRiS\_

movl -12(%rbp), %eax

addl $1, %eax

leave

.cfi\_def\_cfa 7, 8

ret

.cfi\_endproc

**A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated**

**Additional Considerations in Register and Stack Usage for swap, partition, and quickSort**

**Careful Management of Registers**

In the implementation of swap, partition, and quickSort, registers are heavily used to improve efficiency by avoiding unnecessary memory accesses. However, due to the interdependence of these functions, especially their recursive and nested nature, careful attention must be given to how registers are allocated and preserved:

1. **Register Overlap Risks:**
   * Registers used in swap and partition can interfere with those in quickSort if not carefully managed.
   * For instance, quickSort invokes partition, which in turn may call swap. If the same registers are reused across these functions without saving and restoring their values, critical data may be overwritten.
   * Registers like %rax, %rdx, %rcx, %rdi, and %rsi are often used for argument passing and return values in function calls, increasing the chances of overlap.
2. **Caller-Saved vs. Callee-Saved Registers:**
   * **Caller-Saved Registers** (e.g., %rax, %rcx, %rdx): The calling function (quickSort or partition) must assume that these registers will be overwritten during a function call and should save their values on the stack if needed.
   * **Callee-Saved Registers** (e.g., %rbx, %rbp, %r12-%r15): These registers must be preserved by the called function (partition or swap), which means saving their current state on the stack and restoring it before returning.
3. **Function Calls and ABI Compliance:**
   * The x86-64 System V calling convention dictates that arguments are passed in specific registers (%rdi, %rsi, %rdx, etc.) and results are returned in %rax. Recursive calls within quickSort and nested calls to partition and swap require careful adherence to this convention to prevent register corruption.

**Unavoidable Stack Usage**

Despite optimizations, stack usage in quickSort cannot be completely avoided due to its recursive nature:

1. **Recursive Calls:**
   * Each recursive call to quickSort requires maintaining the state of the current frame, including local variables (e.g., start and end indices) and return addresses. These must be pushed onto the stack to preserve the execution context.
2. **Partitioning:**
   * The partition function modifies indices and array elements, which often necessitates temporary storage on the stack when registers are insufficient or need to be preserved for subsequent operations.
3. **Stack Frames:**
   * Each recursive call adds a new stack frame. While minimizing stack usage is possible by reducing local variable storage, the stack is inherently used for tracking the recursion depth.
4. **Memory Safety:**
   * Excessive recursion depth can lead to stack overflow, so implementations must consider practical constraints on input size or adopt techniques like tail recursion optimization or iterative approaches where feasible.

**Strategies to Mitigate Risks**

1. **Preserving Registers:**
   * Explicitly save caller-saved registers on the stack in quickSort before invoking partition or swap.
   * Use push and pop instructions or memory storage to preserve and restore critical values.
2. **Efficient Stack Usage:**
   * Combine local variables into a single stack allocation where possible.
   * Avoid redundant data storage, and reuse stack space efficiently across recursive calls.
3. **Balance Between Stack and Register Usage:**
   * Registers should be prioritized for frequently accessed data, while less critical or transient values are stored on the stack.
   * This balance ensures performance is optimized without compromising correctness due to register overlap.

**Conclusion**

While registers are crucial for optimizing swap, partition, and quickSort, their overlapping usage across these functions requires careful management to avoid corruption and maintain correctness. Additionally, due to the recursive nature of quickSort, stack usage cannot be entirely eliminated. By strategically combining register preservation and efficient stack usage, the implementation can achieve a balance between performance and reliability. This consideration is especially important in deeply recursive algorithms like quickSort, where both stack depth and register safety are critical concerns.

A black screen with white text

Description automatically generated

A black background with white text

Description automatically generated

A screenshot of a computer program

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Elements | Compiler Generated | Optimized |
| 0 | 0 | 0 |
| 1000 | 4.622417 | 2.4727767 |
| 2000 | 18.00897 | 8.3572133 |
| 3000 | 33.60747 | 15.218233 |
| 4000 | 42.7575 | 21.270567 |
| 5000 | 66.21823 | 30.569367 |
| 6000 | 102.9207 | 43.939367 |
| 7000 | 125.861 | 59.422167 |
| 8000 | 163.91 | 73.0461 |
| 9000 | 202.0993 | 90.116167 |
| 10000 | 250.9937 | 111.793 |