Optimization and Performance Analysis of Tree Traversal Algorithms: BFS and DFS

The research is made to evaluate the optimization and performance characteristics of Breadth-First Search (BFS) and Depth-First Search (DFS) tree traversal algorithms. I have implemented and analyzed both unoptimized and optimized versions of these algorithms, focusing on memory locality, branch prediction, and compiler optimizations. The results reveal significant performance improvements for BFS, while interestingly, DFS optimizations showed mixed results where leaning on having a negative timing result.

1. Overview

My experiment focuses on both BFS and DFS, including its pre-order, in-order, and post-order variants for the DFS

- Implement and optimize BFS and DFS algorithms for large tree structures.
- Analyze the impact of these optimizations on performance, memory locality, and branch prediction.
- Evaluate how different compiler optimization levels affect the generated assembly code.

2. Methodology:

System Specifications:

- Operating System: Linux
- Programming Language: C
- Architecture: x86-64

Cache Hierarchy:

- L1d: 896 KiB (24 instances)
- L1i: 1.3 MiB (24 instances)
- L2: 32 MiB (12 instances)
- L3: 36 MiB (1 instance)

To accurately measure the differences between different version of the search algorithm, I have implemented both unoptimized and optimized versions of BFS and DFS (pre-order, in-order, post-order) algorithms and analyze it with perf, valgrind and system timing command .

Test Environment:

- Tree Size: largest is 4,618,081 nodes for primary testing. I have chosen tree depth varies at 3 and up to 17 on different test cases and have a consecutive 3 runs in order to measure the average time more accurately.

```
#define TREE_DEPTH 17
#define NUM_RUNS 3
#define CHILDREN_PER_NODE 3
```

- This accumulates to additional tests with ~1 million and ~10 million nodes
- Compiler: GCC with -O1 and -O3 optimization levels

3. Results and Analysis:

Performance Overview: Below is the timing result measured with -std=c17 -march=haswell -O3 on depth 17

- BFS: 68.68% improvement (201.923 ms to 63.247 ms)
- DFS Pre-order: 0.58% improvement (32.566 ms to 32.378 ms)
- DFS In-order: 1.07% slower (34.831 ms to 35.203 ms)
- DFS post-order: 28.44% slower (34.508 ms to 44.322 ms)

Memory Locality Analysis:

The BFS optimization algorithm significantly improved memory locality, particularly as I increase from smaller to larger tree structures. Notably for lower three_depth size, the difference in time is barely visible, so I had to go over the cache size of the system in order to measure the timing. The key optimization for my method was by replacing the linked-list queue with an array-based queue.

- This change improved cache utilization, especially L1-dcache load misses reduced from 0.78% to 0.51%
- However, based on perf output, the LLC performance is worse, with LLC load miss rate increased from 69.23% to 84.32%
- This suggests that while the optimized BFS improved L1 cache usage, it led to less localized memory access patterns for larger data sets that don't fit in L1/L2 caches.

However, for DFS variants, the memory locality seems to either not improve or actually perform worse as I increase the number of nodes. Below is a timing test for every algorithm at tree_depth 15:

```
Tree created with 4299751 nodes

Total memory used by tree: 1549681944 bytes
Unoptimized versions:
                    BFS took 157.140 ms (avg of 3 runs)
          DFS pre-order took  27.930 ms (avg of 3 runs)
           DFS in-order took  29.733 ms (avg of 3 runs)
         DFS post-order took  29.402 ms (avg of 3 runs)

Optimized versions:
         BFS (optimized) took  48.968 ms (avg of 3 runs)
DFS pre-order (optimized) took  28.442 ms (avg of 3 runs)
 DFS in-order (optimized) took  30.764 ms (avg of 3 runs)
DFS post-order (optimized) took  39.335 ms (avg of 3 runs)
```

In this case, the performance results suggest that these changes in the DFS did not significantly improve memory locality for the given tree structure but seems to be worse.

- Another observation is that while tree depth is below 8, DFS optimized version still seems to perform faster, until it hits 9 and above.

Branch Predictability:

- Based on the output of perf, both optimized and unoptimized versions maintained excellent branch prediction rates (< 1% miss rate). However, there was a slight increase in branch miss rate in the optimized version of this latest run. (0.81% to 0.84%).
  - ```
    <not counted>      cpu_atom/branches/
      52,305,381       cpu_core/branch-misses/         #    0.84% of all branches
    ```
- For BFS, the main loop structure remained relatively simple and predictable for the CPU's branch predictor, while for the DFS variants, especially the post-order traversal, has more complex branching patterns in their iterative implementations. This is likely why DFS perform worse than before while theoretically it should be faster.

Compiler Optimization Analysis:

Finally, I proceed to analyze the assembly code generated by GCC using Compiler Explorer, with -O1 and -O3 optimization levels. As expected, the -O3 optimization produced more efficient assembly. Below are the key differences in the BFS optimized code (-O3 vs -O1):

- O3 made better use of registers, keeping more values in registers rather than memory.

```
movl $1024, %r15d
movl $1, %r12d
```

O3

```
movq $1024, 8(%rsp)
movl $1, %r12d
```

O1

- O3 optimized the final call to `free` by using a **jmp** instead of a **call.**
```

- Also, O3 combines addition and multiplication into one instruction (**leaq (%r12,%r12), %r15) ,** which is more efficient compared to -O1 implementation:
    - **leaq (%rax,%rax), %rdx**
    - **movq %rdx, 8(%rsp)**
- While the main loop structure was similar, -O3 has better register allocation within the loop. Specifically, -O1 keeps the process function pointer in %r15 while O3 keeps it in %r14 : This allows -O3 to use %r15 for the queue capacity. Additionally, -O1 stores the queue capacity in memory (8(%rsp)) but O3 keeps it in %r15 , reducing memory accesses .
- -O3 also use better reallocation by saving registers. An example from assembly versions, where -O1 has to use %rax and %rdx for temporary calculations, hence If the capacity needs to be updated, there's an additional memory write (movq %rdx, 8(%rsp)).
- While in-O3, throughout the loop, the capacity is accessed directly from the register %r15, saving one less register compared to O1:

```
-O1:
leaq    (%rax,%rax), %rdx
movq    %rdx, 8(%rsp)
salq    $4, %rax
movq    %rax, %rsi
```

```
-O3:
leaq    (%r12,%r12), %r15
salq    $4, %rsi
```

    - When updating the capacity, it uses leaq (%r12,%r12), %r15, which doubles the value and stores it directly in %r1, no memory access needed.

I have also checked out the DFS implementations under both optimizations, however, the differences between -O1 and -O3 were less noticeable. This suggests that the recursive DFS implementations were already well-optimized at lower optimization levels, explaining the minimal performance changes in my runs.

4. Summary

BFS Optimization Success:

The significant performance improvement in BFS (68.68%) can be attributed to the array-based queue implementation, which greatly enhanced cache utilization, particularly for L1 cache. This optimization aligns well with the cache hierarchy of the processor, allowing for more efficient memory access patterns.

DFS Mixed/Negative Results:

The minimal improvements and slight degradations in DFS performance highlight the complexity of optimizing recursive algorithms. The iterative implementations, while reducing stack usage, turned out to introduce more complex control flows that may have negatively impacted branch prediction and cache usage.

Memory Locality Trade-offs:

While the BFS optimization improved L1 cache usage, the increased LLC miss rate suggests a trade-off in memory access patterns. This indicates that for very large tree structures that don't fit in lower-level caches, the optimized BFS might face challenges in maintaining its performance advantage.

Optimization impact

The analysis of assembly code generated at different optimization levels (-O1 vs -O3) revealed that the BFS optimization allowed for more effective compiler optimizations, particularly in register usage and instruction selection. However, the DFS implementations, especially the recursive versions, showed fewer real improvements with higher optimization levels, suggesting they were already well-optimized at lower levels.