ESet 与 std::set 性能对比分析报告

实验设定：

- **硬件平台：** Apple MacBook Air M3 芯片 (16GB)
- **编译器：** clang++ (C++20 标准)
- **对比重点：** ESet 与标准库 std::set
- **关键优化策略：** 编译器的优化级别 (O0 对比 O2)，以及对 ESet 实施强制内联

---

1. 第一轮测试：无优化 (O0) 条件下的性能评估

- **编译选项：** `-O0` (无编译器优化)
- **核心发现：**
  - 在此编译条件下，测试结果显示 **ESet 在插入 (Insert)、查找 (Find) 和删除 (Erase) 操作上均显著优于 std::set。**
    - **插入 (Insert)：** ESet 性能领先约 23% 至 39%。
    - **查找 (Find)：** ESet 性能领先约 22% 至 38%。
    - **删除 (Erase)：** ESet 优势最为显著，性能提升约 77% 至 79%。
  - Range Query 为 ESet 特有功能。

关键数据摘要 (随机数据, 10万元素级, O0):

- *Insert:* std::set 约 20.2ms, ESet 约 15.4ms (ESet 性能提升 23.73%)
- *Erase:* std::set 约 2.8ms, ESet 约 0.63ms (ESet 性能提升 77.69%)

---

2. 第二轮测试：标准 O2 优化下的初步对比 (ESet 未作针对性调整)

- **编译选项：** `-O2`
- **核心发现：**
  - **启用 O2 优化后，观察到 std::set 的性能获得大幅提升，并在该阶段反超 ESet。** 此现象表明标准库的实现更能从通用编译器优化中获益。
  - **查找 (Find)：** std::set 表现出极高的性能 (通常在 20-30纳秒量级)，ESet 相比之下性能略低或基本持平。
  - **删除 (Erase)：** 在此阶段，ESet 的删除性能相对较低，较 std::set 慢约 300%。
  - **插入 (Insert)：** 两者表现互有优劣，std::set 在处理大规模随机数据插入时表现出一定优势。
  - **预热 (Warmup) 影响：** 测试中注意到，若不进行预热运行，std::set 的性能优势有所减弱。

关键数据摘要 (随机数据, 100万元素级, O2, ESet未优化):

- *Insert:* std::set 约 314ms, ESet 约 349ms (ESet 性能落后 11.08%)
- *Find:* std::set 26ns, ESet 20ns (ESet 性能领先 23.08%)
- *Erase:* std::set 约 0.46ms, ESet 约 1.84ms (ESet 性能落后 300.96%)

---

3. 第三轮测试：O2 优化与 ESet 强制内联结合策略

- **ESet 的改进措施：** 于 ESet 的关键函数中增添 `__attribute__((always_inline)) inline` 指令。
- **编译选项：** `-O2`
- **核心发现：**
  - **经过强制内联优化，ESet 性能获得显著提升，与 std::set 的整体性能已非常接近，在多项测试中表现出互有优劣的特性。**
  - **删除 (Erase)：** ESet 此前的删除性能问题得到有效解决，与 std::set 基本持平 (差异通常在 ±1% 之内)。
  - **插入 (Insert)：** 两者性能非常接近，差异一般在 ±10% 范围。
  - **查找 (Find)：** 两者均表现出很高的性能，常出现耗时相同的情况；在处理部分随机或重复数据时，ESet 甚至展现出更快的速度。
  - **迭代 (Iteration)：** 在部分迭代相关的测试中，ESet 显示出一定的优势。
  - **数据特性对性能的影响：** 初步观察表明，ESet 在处理有序数据时可能表现更佳，而 std::set 或在随机数据处理上略有优势。

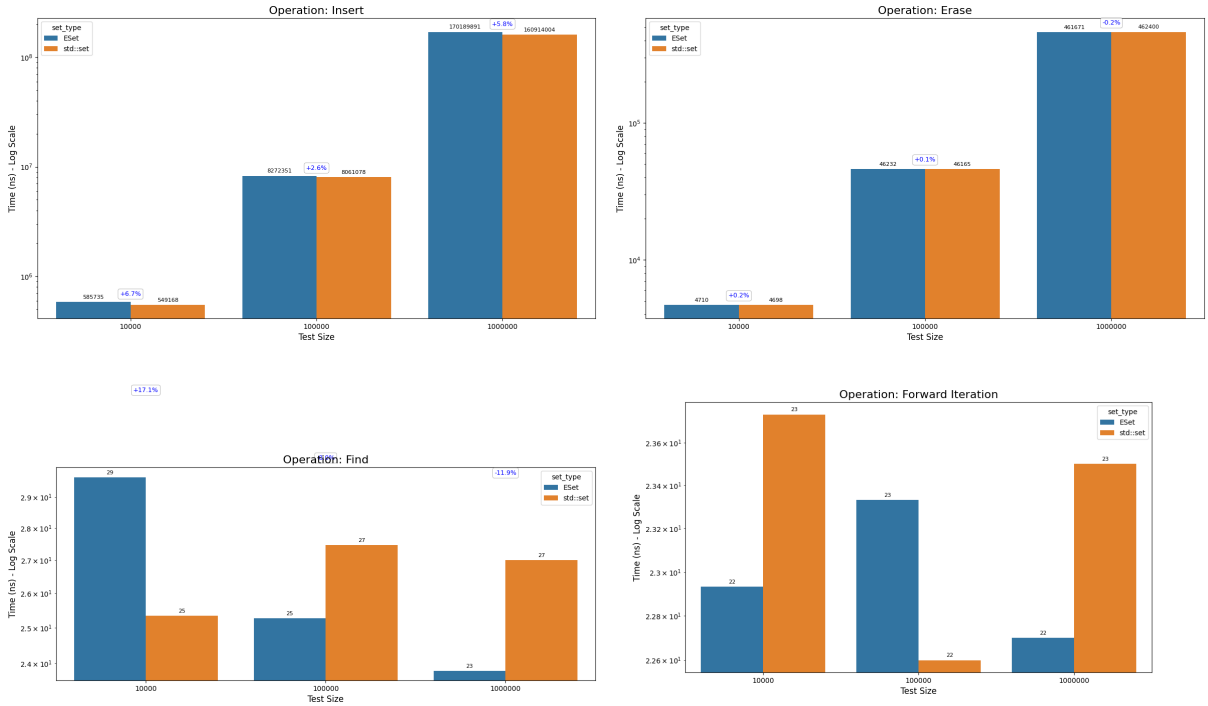
关键数据摘要 (随机数据, 100万元素级, O2, ESet强制内联):

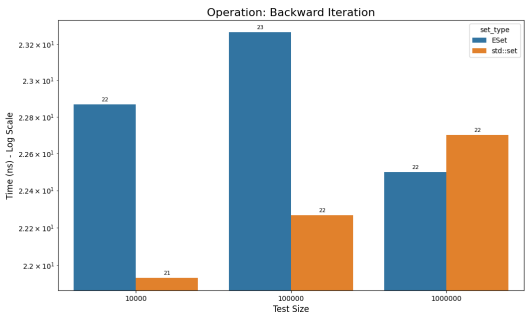
- *Insert:* std::set 约 261ms, ESet 约 276ms (ESet 性能落后 5.63%)
- *Find:* std::set 33ns, ESet 33ns (0.00%)
- *Erase:* std::set 约 0.4611ms, ESet 约 0.4615ms (ESet 性能落后 0.09%)

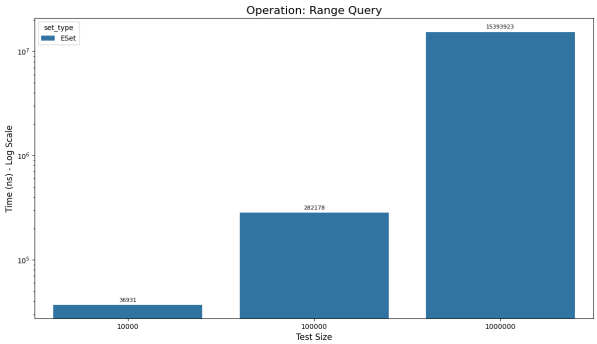
---

4. 图表分析 (基于第三轮测试数据)

以下为五次测试平均性能数据图表，分别展示了插入、删除、查找、正向迭代、反向迭代以及范围查询操作的性能对比：







图表显示，在插入和删除任务中，ESet 和 std::set 的性能表现得极为接近。在其余的任务中，尽管存在百分比差异，但由于绝对的耗时差距非常小（通常只有几纳秒），因此实际上的性能差距不大。

---

5. 结论

1. **编译器优化的关键作用：** 分析表明，标准编译器优化（例如 O2）对于 std::set 性能的提升作用非常关键。ESet 在不进行优化时有良好的表现，但在标准的 O2 优化下，若不进行针对性的代码调整，其部分性能指标会不及 std::set。
2. **强制内联的积极效应：** 对 ESet 的关键函数应用强制内联后，其在 O2 优化下的性能得到巨大改善。这使得 ESet 在多数测试场景下都能够与 std::set 相媲美，甚至在某些特定情况下反超。
3. **ESet 的性能特征总结：**
    - 经过优化后，ESet 与 std::set 在核心的插入、查找、删除操作上，性能已非常接近。
    - 此外，观察到 ESet 在处理有序数据时可能具有一定的性能优势。

附录：

-测试代码如下：

```
#include "include/Eset.hpp"
#include <algorithm>
#include <chrono>
#include <climits>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <random>
#include <set>
#include <sstream>
#include <vector>

using namespace std;
using namespace chrono;

// 测试配置
const vector<size_t> TEST_SIZES = {10'000, 100'000, 1'000'000};
const int RANDOM_SEED = 111;
const int WARMUP_RUNS = 1;
const int TEST_RUNS = 5;
```

```cpp
// 生成测试数据
vector<int> generate_test_data(size_t n, const string &type) {
  vector<int> data(n);
  mt19937 rng(RANDOM_SEED);

  if (type == "random") {
    uniform_int_distribution<int> dist(1, INT_MAX);
    generate(data.begin(), data.end(), [&]() { return dist(rng); });
  } else if (type == "sorted") {
    iota(data.begin(), data.end(), 1); // 1, 2, 3...
  } else if (type == "reverse") {
    iota(data.rbegin(), data.rend(), 1); // n, n-1,...1
  } else if (type == "duplicate") {
    uniform_int_distribution<int> dist(1, 100); // 大量重复
    generate(data.begin(), data.end(), [&]() { return dist(rng); });
  }

  return data;
}

// 精确计时模板
template <typename Func> auto measure_time(Func &&func, int runs = 1) {
  for (int i = 0; i < WARMUP_RUNS; ++i)
    func();

  nanoseconds total{0};
  for (int i = 0; i < runs; ++i) {
    auto start = high_resolution_clock::now();
    func();
    auto end = high_resolution_clock::now();
    total += duration_cast<nanoseconds>(end - start);
  }
  return total / runs;
}

// 迭代器性能测试
template <typename SetType>
pair<nanoseconds, nanoseconds> benchmark_iterator_operations(SetType &s) {
  // 正向遍历
  auto forward_time = measure_time(
      [&] {
        volatile size_t count = 0;
        for (auto it = s.begin(); it != s.end(); ++it) {
          count++;
        }
      },
      TEST_RUNS * 3); // 更多次测试减少误差

  // 反向遍历
  auto backward_time = measure_time(
      [&] {
        volatile size_t count = 0;
        for (auto it = s.end(); it != s.begin();) {
          --it;
```

```
            count++;
          }
        },
        TEST_RUNS * 3);

    return make_pair(forward_time, backward_time);
}

// 基本性能测试
template <typename SetType>
tuple<nanoseconds, nanoseconds, nanoseconds, nanoseconds, nanoseconds>
benchmark_all_operations(const vector<int> &data) {
    SetType s;

    // 插入测试
    auto insert_time = measure_time(
        [&] {
          for (const auto &x : data)
            s.emplace(x);
        },
        TEST_RUNS);

    // 查找测试
    auto find_time = measure_time(
        [&] {
          for (const auto &x : data)
            s.find(x);
        },
        TEST_RUNS);

    // 删除测试
    auto erase_time = measure_time(
        [&] {
          for (const auto &x : data)
            s.erase(x);
        },
        TEST_RUNS);

    // 迭代器测试
    auto [forward_time, backward_time] = benchmark_iterator_operations(s);

    return make_tuple(insert_time, find_time, erase_time, forward_time,
                      backward_time);
}

// 范围查询测试
template <typename SetType>
nanoseconds benchmark_range_query(const vector<int> &data) {
    SetType s;
    for (const auto &x : data)
      s.emplace(x);

    if constexpr (requires { s.range(0, 0); }) {
      auto time = measure_time(
```

```
        [&] { volatile size_t cnt = s.range(data.front(), data.back()); },
        TEST_RUNS);
    return time;
  }
  return nanoseconds(0);
}

// 对比两种结构的性能
void compare_benchmarks(const vector<int> &data, const string &data_type,
                        ofstream &csv, size_t test_size) {
  cout << "\n=== Comparing std::set and ESet (" << data.size() << "
elements, "
       << data_type << " data) ===\n";

  // 测试 std::set
  auto [std_insert, std_find, std_erase, std_forward, std_backward] =
      benchmark_all_operations<set<int>>(data);
  auto std_range = benchmark_range_query<set<int>>(data);

  // 测试 ESet
  auto [eset_insert, eset_find, eset_erase, eset_forward, eset_backward] =
      benchmark_all_operations<ESet<int>>(data);
  auto eset_range = benchmark_range_query<ESet<int>>(data);

  // 输出对比表格
  cout << setw(20) << "Operation" << setw(15) << "std::set" << setw(15)
       << "ESet" << setw(15) << "Difference (%)" << endl;
  cout << string(65, '-') << "\n";

  auto print_result = [&](const string &name, nanoseconds std_time,
                          nanoseconds eset_time) {
    double diff = (double(eset_time.count()) / std_time.count() - 1) *
100;
    cout << setw(20) << name << setw(15) << std_time.count() << setw(15)
         << eset_time.count() << setw(14) << fixed << setprecision(2) <<
diff
         << " %\n";

    // 写入CSV文件
    csv << test_size << "," << data_type << ",std::set," << name << ","
        << std_time.count() << "\n";
    csv << test_size << "," << data_type << ",ESet," << name << ","
        << eset_time.count() << "\n";
  };

  print_result("Insert", std_insert, eset_insert);
  print_result("Find", std_find, eset_find);
  print_result("Erase", std_erase, eset_erase);
  print_result("Forward Iteration", std_forward, eset_forward);
  print_result("Backward Iteration", std_backward, eset_backward);

  if (std_range.count() > 0) {
    print_result("Range Query", std_range, eset_range);
  } else {
```

```
    cout << setw(20) << "Range Query" << setw(15) << "N/A" << setw(15)
         << eset_range.count() << " ns\n";
    csv << test_size << "," << data_type << ",std::set,Range Query,N/A\n";
    csv << test_size << "," << data_type << ",ESet,Range Query,"
        << eset_range.count() << "\n";
  }
}

int main() {
  ofstream out("output/test4.txt");
  cout.rdbuf(out.rdbuf());

  ofstream csv("output/test4_data.csv", ios::app);
  bool is_first_run = csv.tellp() == 0;
  if (is_first_run) {
    csv << "test_size,data_type,set_type,operation,time_ns\n";
  }

  cout << "===== Enhanced ESet vs std::set Benchmark =====\n";
  cout << "Config: Warmup=" << WARMUP_RUNS << ", TestRuns=" << TEST_RUNS
       << "\n\n";

  for (size_t size : TEST_SIZES) {
    cout << "\n■■■■■ TEST SIZE: " << size << " ■■■■■\n";

    // 随机数据测试
    auto random_data = generate_test_data(size, "random");
    compare_benchmarks(random_data, "random", csv, size);

    // 有序数据测试
    auto sorted_data = generate_test_data(size, "sorted");
    compare_benchmarks(sorted_data, "sorted", csv, size);

    // 重复数据测试
    if (size <= 100'000) { // 避免太大测试集
      auto dup_data = generate_test_data(size, "duplicate");
      compare_benchmarks(dup_data, "duplicate", csv, size);
    }
  }

  cout.rdbuf(nullptr);
  return 0;
}
```