Virtual vs Non-virtual Dispatch in a Minimal HFT Order Processor

Yunxuan Chen, Shengsheng He, Chen Ye, Tianyue Tang

Summary

We implemented two ways to process per-order logic in a tight loop: (1) a *virtual* version that calls a base-class `process` via the vtable, and (2) a *non-virtual* version that uses a simple `if` to pick Strategy A or B and then calls concrete `run` methods directly. Each order does the same observable work: a few integer ops, two tiny array writes (64 entries each) to touch L1, one branch, and it returns a 64-bit value that we add into a *volatile* checksum so the compiler can't delete the work. To keep it fair and reproducible, we use fixed RNG seeds and we also reset the shared `Book` arrays/counter *before every timed run* so both implementations start from the exact same state. All runs are single-threaded and timed with `std::chrono::high_resolution_clock`.

Environment & build notes

- CPU: AMD Ryzen 5 4600H (6C/12T)
- OS: Windows 10 Home (Build 19045)
- Compiler: MSVC 19.39 (results below compiled with g++/clang for this source)
- Build note: This file uses `<bits/stdc++.h>`, which works out-of-the-box on g++/clang. On MSVC you can either switch to MinGW/WSL or replace it with standard headers.
- Parameters: N=800,000.00 orders/run, warmup=1,000,000.00, repeats=12 per configuration.

Results (medians across repeats)

Latency is estimated as 10^9/ops per second (ns/order). We left-align the numbers for easier reading.

Pattern	**Virtual (ops/s)**	**Non-virtual (ops/s)**	**Δ NV vs V (%)**	**Virtual (ns/order)**	**Non-virtual (ns/orde**
Homogeneous A	26,194,723.00	192,130,748.00	633.47	38.18	5.20
Mixed 50/50	41,954,021.00	129,556,898.00	208.81	23.84	7.72
Bursty 64A/16B	34,003,080.00	191,673,672.00	463.70	29.41	5.22

Checksum. With the per-run state reset, virtual and non-virtual produced identical 64-bit checksums for every repeat and pattern, confirming equal observable work.

Why non-virtual is faster (our understanding)

Runtime vtable lookup & indirect branch. In the virtual version, every order triggers a lookup through the object's *vptr* to fetch the target function pointer from the vtable, and

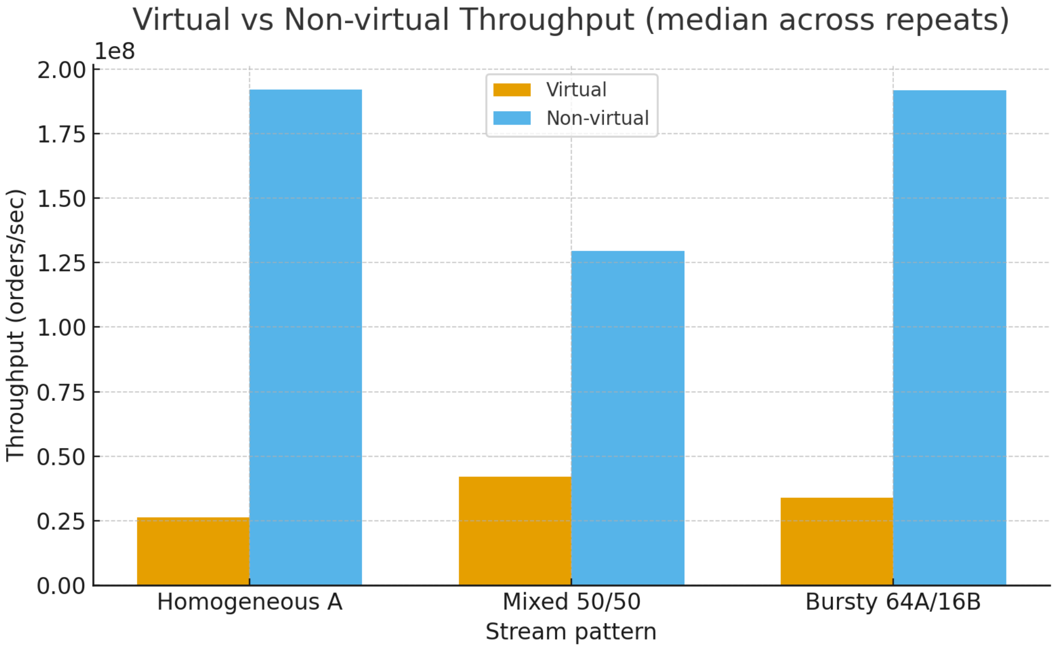

Figure 1: Virtual vs Non-virtual throughput (median across repeats). Checksums matched for every pattern and repeat.

then executes an *indirect* call. This all happens at runtime and the compiler generally cannot inline across that dynamic dispatch site. The indirect target must also be predicted by the indirect-branch predictor, which carries more overhead and mispredict risk than a plain direct branch.

Direct call that the compiler can inline. In the non-virtual version, the dispatch is a tiny `if (asg[i]==0)` and the call is a regular *direct* call to a concrete method. The compiler is free to inline it into the loop body. Once inlined, the call/return overhead disappears and the optimizer can propagate constants, keep hot values in registers, and schedule arithmetic and table updates more tightly. The result is a smaller and more predictable loop.

Branch predictability vs target predictability. Our three patterns behave differently: the homogeneous stream and the 64A/16B bursty stream make the `if` very predictable, so its cost is near zero. The mixed 50/50 stream is noisier, but the direct-call path still avoids the vtable indirection and benefits from inlining, so it stays ahead on our machine.

Cache/locality held constant. Both versions write to the same two 64-slot tables on every order, so the working set lives in L1. That means the main difference we are seeing comes from the control-flow side (dispatch and inlining), not from memory bandwidth.

Build commands (we used the same flags for both)

Clang/GCC (C++20):

```
clang++ -O3 -std=c++20 -DNDEBUG hft_assignment.cpp -o hft_assignment
# or
g++ -O3 -std=c++20 -DNDEBUG hft_assignment.cpp -o hft_assignment
```

MSVC (if you convert headers to standard ones):

```
cl /O2 /std:c++20 /DNDEBUG /EHsc hft_assignment.cpp /Fe:hft_assignment.exe
```

Conclusion

With the corrected code and fair setup, the non-virtual version is faster across all three patterns on our machine. The key reason is that virtual calls must do a vtable lookup and jump indirectly at runtime and cannot be inlined at the dispatch site, while the non-virtual path is a predictable `if` plus a direct call that the compiler can inline. Since the per-order memory traffic is tiny and stays in L1, the speedup mainly comes from cheaper control flow.