Exercise 08

VU Performance-oriented Computing, Summer Semester 2024

Calvin Hoy

2024-05-20

(A) False Sharing

False sharing occurs when multiple objects, residing in an area of memory mapping to the same cache line, are read and written to exclusively by different CPU cores. Each time one core writes to its object, the whole cache line is invalidated and must be re-fetched from memory by the other core before it can access the other one, which can have major impacts on performance.

False sharing is avoided by padding data structures so that they each occupy a whole cache line. This can be done either manually by adding extra, unused members to the structure (often a byte array), or automatically by instructing the compiler to *align* each object to the cache line length.

The pull request in question (link) refactors some (presumably) old code with manually-padded data structures, removing the manual padding in favour of the `alignas` specifier introduced by C++11. It leaves in place the assumed cache line length of 64 bytes, however.

(B) Data Structure Selection

I found the following pull request in the Github repository for the Wii U emulator Cemu, which yielded a 60% FPS uplift in *The Legend of Zelda: Breath of the Wild* on the author's Steam Deck.

cemu-project/Cemu — #370 Linux/MacOS: Greatly improve performance (link)

The PR substitutes an `std::unordered_set` for a custom set data structure (`SparseBitSet`) in the code responsible for emulation of the Wii U's "Latte" chip — responsible among other things (link) for graphics processing.

Specifically, the relevant code deals with flushing the (emulated) CPU/GPU cache, and is by its nature called many times per frame, clearing the affected data structure every time. The reason for the change is that `.clear()` is a fairly expensive operation on an `std::unordered_set` with `libstdc++`, as it fully clears the allocated memory area.

The performance issue remedied by this PR did not exist on Windows, as `MSVC` implements `std::unordered_set` using a different underlying data structure for which `.clear()` is much cheaper.

Evaluation criteria

It should be noted that the PR changes not the data structure used by the code — a queue — but rather the *underlying* data structure used by the queue, from one that is implementation-defined and whose performance thus "cannot be blindly trusted" (sic.), to a known one. In this case, that is an array of 256 vectors.

Data type: `uint32_t`

The inserted values presumably represent memory addresses. These are comparable, hashable, countable… but all that is irrelevant.

Data quantity

Difficult to judge, but I assume the replacement data structure is named `SpareBitSet` for a reason.

I cannot entirely make sense of why the inserted values are split across multiple vectors by their 256-modulus. I assume this is informed by knowledge that many of the values inserted into the queue will be of a few common 256-moduli, and splitting them reduces the number of times that each vector needs to grow in size.

Access patterns

- Insertions accumulate over time
- Insertion order is arbitrary, always at the back
- Inserted items are frequently read back and deleted all at once
- Accesses occur across multiple threads (protected by a spinlock)
  - Insertions are made in multiple places across the code base
  - Reading+deletion occurs in a single place

These access patterns do lend themselves to a queue. Frequent full flushes of the full queue (as opposed to popping one or a few items at a time) explain why the custom data structure is more optimal than e.g. a linked list.

Target hardware

CPU (not GPU) code -> branching isn't *so* bad.