Optimization

Arnav Gupta

December 8, 2024

Contents

1 Optimization

Computer with infinite memory and speed requires no optimizations to use less memory or run faster (space and time).

With finite resources, optimization is useful/necessary to conserve resources and for good performance. Most programs not written in optimal order or in minimal form.

General forms of optimizations:

- **reordering**: data and code are reordered to increase performance in certain contexts
- **eliding**: removal of unnecessary data, data accesses, and computation
- **replication**: processors, memory, data, and code are duplicated because of limitations in processing and communication speed (speed of light)

Optimized program must be isomorphic to original (same result for fixed input).

Kinds of optimizations are restricted by execution environment.

2 Sequential Optimizations

Most programs are sequential, even concurrent programs are:

- large, sections of sequential code per thread connected by
  - small sections of concurrent code where threads interact (protected by synchronization and mutual exclusion — SME)

Sequential execution presents simple semantics for optimization, where operations occur in program order (sequentially).

Dependencies result in partial ordering among a set of statements (precedence graph):

- **data dependency**: reordering reads and writes
- **control dependency**: statements cannot be reordered if the earlier line determines if the later line is executed

To achieve better performance, compiler/hardware make changes:

1. reorder disjoint (independent) operations (variables have different addresses)
2. elide unnecesary operations (transformation/dead code)
   (a) remove immediate change, no loop body, tail recursion, etc
3. execute in parallel if multiple functional units (adders, floating units, pipelines, cache)

Very complex reordering, reducing, and overlapping of operations allowed.

Overlapping implies micro-parallelism, but limited capacity in sequential execution.

3 Memory Hierarchy

Memory hierarchy includes registers (on CPU), memory, and possibly cache, where replication must occur across all layers.

Optimizing data flow along hierarchy defines a computer's speed.

Hardware aggressively optimizes data flow for sequential execution.

Having basic understanding of cache is essential to understanding performance of both sequential and concurrent programs.

3.1 Cache

Since CPU faster than memory, copy data from memory into registers. But not many registers, so move highly accessed data within a program from memory to registers for as long as possible and then back to memory.

Can quickly run out of registers as more data accessed, so must rotate data from memory through registers dynamically (compiler attempts ot keep highly used variables in regsters).

This does not handle highly accessed data among programs (threads), since each context switch saves and restores most registers to memory (registers are private and cannot be shared). Solution is to use hardware cache (automatic registers) to stage data without pushing to memory and allow sharing of data among programs.

Caching transparently hides latency of accessing main memory. Cache loads in 64/128/256 bytes, called **cache line**, with addresses multiple of line size.

When cache full, data evicted (remove old cache lines to bring in new based on LRU). When program ends, addresses are flushed from memory hierarchy.

3.2 Cache Coherence

Multi-level caches used, each larger but with diminishing speed and cost.

Data reads logically percolate variables from memory up memory hierarchy, making cache copies to registers.

Necessary to eagerly move reads up memory hierarchy to take advantage of temporal locality.

Data writes from registers to variables logically percolate down memory hierarchy though cache copies to memory.

Advantageous to lazily move writes down memory hierarchy since that way later reads see this.

If OS moves program to another process, all caching info is invalid and the program's data hierarchy reforms.

Unlike registers, all cache values are shared across the computer. Hence, a variable can be replicated in a large number of locations.

Without cache coherence for shared variable, there is madness. With cache coherence for shared variable (snooping or directory-based), variables are updated everywhere.

Cache coherence: hardware protocol ensuring update of duplicate data

Cache consistency: addresses when processor sees update (bidirectional synchronization)

Prevent flickering and scrambling during simultaneous R/W or W/W with update and acknowledge.

Eager cache-consistency means data changes appear instantaneous by waiting for acknowledge from all cores (complex/expensive).

Lazy cache-consistency allows reader to see own write before acknowledgment (concurrent programs read stale data):

- writes eventually appear in (largely) same order as written
- critical section works as writes to shared variable appear before write to lock release (so other threads see write to lock after write to shared variable)
- otherwise, spin (lock) until write appears

Cache thrashing: if threads continually read/write same memory locations, they invalidate duplicate cache lines, resulting in excessive cache updates (updated value bounces from one cache to the next)

False sharing: since cache lines contain multiple variables, cache thrashing can occur inadvertently

Fix false sharing by separating variables on same line with sufficient storage (padding) to be in next cache line. Difficult for dynamically allocated variables as memory allocator positions storage.

4 Concurrent optimizations

In sequential execution, **strong memory ordering**: reading always returns last value written

In concurrent execution, **weak memory ordering**: reading can return previously written value or value written in future:

- happens on multi-processor because of scheduling and buffering
- notion of current value becomes blurred for shared variables unless everyone can see values assigned simultaneously

SME control order and speed of execution, otherwise non-determinism causes random results or failure.

Sequential sections accessing private variables can be optimized normally, but not across concurrent boundaries.

Concurrent sections accessing shared variables can be corrupted by sequential optimizations, so restrict optimizations to ensure correctness.

For correctness and performance, identify concurrent code and only restrict its optimization. How to restrict depends on what sequential assumptions are implicitly applied by hardware and compiler.

4.1 Disjoint Reordering

$R_x \rightarrow R_y$ allows $R_y \rightarrow R_x$, so reordering disjoint reads does not cause problems.

$W_x \rightarrow R_y$ allows $R_y \rightarrow W_x$ can cause problems.

$R_x \rightarrow W_y$ allows $W_y \rightarrow R_x$ can cause problems.

$W_x \rightarrow W_y$ allows $W_y \rightarrow W_x$ can cause problems.

Compiler uses all these reorderings to break mutual exclusion:

- moves lock entry/exit after/before critical section because entry/exit variables not used in critical section

- double-check locking for singleton-pattern can help
  - both checks check for change in storage at different parts of code (inside and outside lock)
- can fail if writes and reads related to malloc are reordered

4.2 Eliding

For high-level language, compiler decides when/which variables are loaded into registers and for how long.

Elide reads (loads) by copying (replicating) value into a register. So, variable logically disappears for duration in register (can cause livelock).

Elide meaningless sequential code: can cause task to miss signal by not delaying.

4.3 Replication

Benefit to reorder R/W since faster instructions can then happen at once.

Modern processors increase performance by executing multiple instructions in parallel (data flow, precedence graph) on replicated hardware:

- internal pool of instructions taken from program order
- begin simultaneous execution of instructions with inputs
- collect results from finished instructions
- feed results back into instruction pool as inputs
- so, instructions with independent inputs execute out-of-order

From sequential perspective, disjoint reordering is unimportant, so hardware starts many instructions simultaneously.

From concurrent perspective, disjoint reordering is important.

5 Memory Model

Manufacturers define set of optimizations performed implicitly by processor.

Memory model is defined by set of optimizations.

Atomic consistent (AT) has events occur instantaneously, so slow or impossible to optimize.

Sequential consistency (SC) accepts all events cannot occur instantaneously, so may read old values. Still strong enough for software mutual exclusion (often considered minimum model for concurrency).

No hardware supports just AT/SC.

6 Preventing Optimization Problems

All optimization problems result from races on shared variables.

If shared data is protected by locks (implicit or explicit):

- locks define sequential/concurrent boundaries
- boundaries must preclude optimizations that affect concurrency

Race free: synchronization and mutual exclusion preclude races

Race free does have races, since races are internal to locks, which lock programmer must deal with.

Two approaches:

- ad hoc: programmer manually augments all data races with pragmas to restrict compiler/hardware optimizations (not portable but often optimal)
- formal: language has memory model and mechanisms to abstractly define races in program (portable but often baroque and suboptimal)

Data access/compiler: `volatile` qualifier

- force variable loads and stores to/from registers (at sequence points)
- created for `longjmp` or force access for memory-mapped devices
- for architectures with few registers, all variables are pretty much implicitly volatile
- Java `volatile` and C++11 `atomic` are stronger, since they prevent eliding and disjoint reordering
  - C++11 `atomic` automatically fences shared variables, but can be suboptimal

Program order/compiler (static): disable inlining or use `asm("" ::: "memory");`

Memory order/runtime (dynamic): `sfence`, `lfence`, `mfence`, which guarantee previous stores and/or leads are completed before continuing.

Atomic operations test-and-set often imply fencing.

Cache is normally invisible and does not cause issues (except for dynamic memory allocation).

Mechanisms to fix issues are specific to compiler or platform: difficult, low-level, diverse semantics, and not portable.

Locks built with these features ensure SC for protected shared variables: no user races and strong locks, so SC memory model.