Memory Model Overview

Hans-J. Boehm

HP Labs

Hans.Boehm@hp.com

WG21/N2010=J16/06-0080

2006-04-21

Slides presented to concurrency sub-subgroup (slightly revised)

Talk Overview

- Status
- Very quick overview
- Discussion of consequences
  - This will impact compiler back-ends.
  - This will constrain future hardware.
  - I believe this is unavoidable if we want a tolerable programming model.

Current Status

- We currently have an informal proposal.
- The evolving version is at
  - http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/
- N1942 is close.
- Builds on Clark Nelson's sequence point proposal (N1944).
- Fundamental assumption:
  - Usability is more important than 5% performance.
- Now is the time to discuss the approach.
- It will take time to draft a formal proposal.
- Web site has companion atomics interface.

Proposal definitions

- Two operations *conflict* if they affect the same location, and at least one is a write.
- A *memory location* is a scalar object or a contiguous sequence of bit-fields.
- (Oversimplified) A memory access *happens-before* one in another thread if the second acquires a lock after the first released it.
- There is an (inter-thread) *data race* if there are two conflicting memory accesses by different threads, and neither happens-before the other.

Proposal Overview

- We define a *consistent execution*.
- Inter-thread visibility is defined using happens-before.
- If there is a consistent execution which
  - Sees the right input, and
  - Contains a data race

  then the semantics are undefined.
- Otherwise the program behaves according to one of its executions.
- We handle atomic operations fairly generally with a more complicated, and somewhat nonstandard definition of happens-before.

Library Issues

- Haven't looked at this in detail yet.
- Plan is to follow SGI STL:
  - Containers behave like scalars:
    - Two operations on a container conflict if one of them logically updates the container.
    - Allocation doesn't count as update.
    - User-invisible updates require internal locking.
    - Other locking is the clients reponsibility.
  - This seems to be the de facto standard.
    - except for I/O?
  - Basic_string and reference counting?
    - ABI change?

Positive Attributes

- Complexity of the proposal seems manageable.
  - In the absence of atomics, it seems as simple as anyone might have expected.
- We're known to be mathematically sloppy in only one place: The "depends-on" relation. And that's
  - probably fixable, if we really wanted.
  - not critical for mainstream optimization.
- Gives a sound foundation to thread in C++.
  - Simples, teachable rules for common case.
- Probably as easy to use as threads and locks can be.

Other impact on standard

- Everything, needs review.
- We need clean single-thread ordering semantics.
  - We need to know what "program order" is.
- Part of current discussion uses "sequence points", part doesn't.
- "Sequence points" define an order, and are not points.
- We don't agree on what it means.
- This needs to be fixed.

Implementation consequences

- Most optimizations are unaffected. Loads and stores can be eliminated, replicated, and moved between atomic/synchronization calls.
- But
  - Some fairly fundamental ones are affected.
  - And (like Java & CLI) we impose hardware constraints:
    - Required for multiprocessors.
    - May need restartable critical sections on uniprocessors.

Optimization Restrictions

No speculative or unnecessary stores.

- Stores to struct/class members may not unnecessarily overwrite adjacent members.
  - Intel Example:

```
struct {char a; int b:9; int c:7; char d;}
```

  - A store to `b` must be implemented as 2 byte stores.
- Speculative register promotion often illegal:

```
for (T *p = q; p != 0; p = p -> next)
    if (p -> data > 0) ++count;
```

  - Standard register promotion of `count` becomes illegal.

Optimization Restrictions 2

- Some kinds of code hoisting are problematic.
- Stores may not be advanced across potentially nonterminating loops.
  - Example:

    ```
    for (T*p = q; p != 0; p = p -> next) ++count;
    x = 42;
    ```
  - Uncommon? But analysis is commonly wrong.

Architectural Implications

- Byte stores must be well-supported.
  - Required for Java/CLI.
  - Very old DEC Alpha machines won't work.
    - And compilers should limit support.
  - Others?
- Atomic operations may be optional, but require more:
  - Atomic loads/stores for most scalars.
  - Compare-and-swap (ll/sc) highly desirable.
  - Cheap way to enforce "causal ordering":
    - happens-before *is transitive*.

Questions:

- Any concerns?
- Single thread performance will take a small hit.
  - Low single-digit SPECcpu performance?
  - Except for "no threads" compiler option.
  - Is this OK? Other options?
- Are the architectural constraints OK?
  - We do have (bad?) options for location defn.
- Is the library approach OK?
- Atomic operations issues?