Forth concurrency for the 21st century, Part 2:

What are we going to do about *volatile*?

Andrew Haley

redhat

Where are we, *recap*

- Moore's law has not been cancelled: every 18 months, the number of transistors per unit area doubles
- However, clock speeds have not been increasing for several years, and if anything have got slightly slower
- Performance can still be increased with new pipeline and cache designs, but not by much.
- There seems to be a 4 GHz barrier
- Future machines will have more and more cores

redhat

Motivation

- VFX forth compiled

```
counter @ dup 2 +
```

into machine code equivalent to

```
counter @ 2 +  counter @ swap
```

- This breaks a common idiom for a shared counter:

```
begin
   counter @ dup N +
   counter compare&swap
until
```

redhat

Motivation

- I sent an email to Steve, who replied:

  > “It's not a bug ... it's the volatile problem!”
  > “Note that this is only a problem for x86 and hosted systems.”

- So, there is a volatile problem. And that's official!

redhat

Profane languages and *volatile*

- C has *volatile*. Is it the first language to have it? I think so

- Hans Boehm points out that there are only three portable uses for it. Summarizing,

  * marking a local variable in the scope of a *setjmp* so that the variable does not rollback after a *longjmp*.

  * variables may be "externally modified", but the modification in fact is triggered *synchronously* by the thread itself, e.g. because the underlying memory is mapped at multiple locations

  *A volatile *sigatomic_t* may be used to communicate with a signal handler in the same thread

redhat

Profane languages and *volatile*

- Java has *volatile*. But its meaning is totally different from that of C's *volatile*
- Java's *volatile* is a memory barrier

redhat

Memory barriers and data races

- Current processors feature *out-of-order execution*. From the point of view of a thread, memory accesses appear to occur in program order. However, from the point of view of another thread, there is no guarantee that memory writes from another thread will occur in the same order, or that its stores into memory will *ever* be visible!
- The new C++ standard takes the view that all access to data shared between threads that are not explicitly protected by locks or atomic operations are undefined
- Because some Forth systems are implemented in C, they will necessarily have the same problem

redhat

data races

- Thread 1:

```
99 result !  1 ready !
```

- Thread 2:

```
begin  pause  ready @ until
result @  ...
```

- The value in `result` is undefined

redhat

load acquire and *store release*

- What would it take to make this work?
- Thread 1:

```
99 result !  1 ready volatile!
```

- Thread 2:

```
begin  pause  ready volatile@ until
result @  ...
```

redhat

load acquire and *store release*

- **`volatile!`** and **`volatile@`** must be *memory barriers*
- Every store to memory that *happens before* a **`volatile!`** is visible to another thread after
- that other thread does a **`volatile@`**
- These operations are known as ***load acquire*** and ***store release***

redhat

load acquire and *store release*: x86

- On x86, *load acquire* is:

  mov memory -> register

- *store release* is:

  mov register -> memory

- Compilers don't have to do anything special because on the x86 there is a guarantee that all threads see stores in the same order. Other processors don't have such guarantees, so naive programmers may accidentally write non-portable code

redhat

load acquire and *store release*: ARM

- On ARM, *load acquire* is:

  ld memory -> register
  dmb (flush all locally cached loads)

- *store release* is:

  dmb (force all pending stores)
  mov register -> memory

- There are other equivalent sequences

redhat

load acquire and *store release*: PowerPC

- On ppc, *load acquire* is:

  ld memory -> register;
  lwsync (flush all locally cached loads)

- *store release* is:

  lwsync (force all pending stores)
  ld register -> memory

- There are other equivalent sequences

redhat

Forth and the future

- The C an C++ standard committees used to think that they didn't have to worry about these things. They have now changed their minds, because threading issues can't be left to libraries

- If Forth has a future on multi-core systems these issues must be addressed. We should start thinking about them now