Forth concurrency for the 21st century

Andrew Haley

redhat

Where are we?

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

redhat

Where are we going?

Shared-memory multiprocessors are the dominant technology for servers and desktops

Today we have four cores per die

According to Moore's law, in ten years we'll have 100 cores per die

But they may not be very much faster than the cores we have today

We need language support so that normal human beings can program these beasts

redhat

Where were we?

Chuck Moore's Forth multi-tasking design, from early 1970s:

Round-robin scheduler

Non-preemptive

Because of the lack of preemption, this design is very easy to use, because

You don't have to lock data structures unless there is a PAUSE or I/O

redhat

However

This design doesn't work for shared-memory multiprocessors, where there are several cores working on the same memory at the same time

redhat

Where are we now?

Almost no concurrency support in the Forth language standard

Some Forths use language support from OS: POSIX threads

GET and RELEASE primitives using mutexes for shared data structures

Difficult and unreliable to program, and doesn't scale well: deadlocks, races, etc.

The heart of the problem is that no-one knows how to organize and maintain large systems that rely on locking

Locks are not *composable*

redhat

Where are we?

Alternatively, lock-free data structures using CompareAndSet

But almost no-one in the world knows how to program them, and even those few people make mistakes

The principal difficulty is that synchronization primitives such as CompareAndSet work on only a single word, and this often forces a complex and unnatural structure on algorithms

Even a lock-free queue is an order of magnitude more complex

Lock-free structures are not composable either

redhat

In summary

Locks are hard to manage effectively

CompareAndSet operates on only one word at a time, resulting in complex algorithms

It is difficult to compose multiple calls to multiple objects into atomic units

redhat

Transactions and Atomicity

Wouldn't it be nice if we could say

```
begin-atomic
  x @ if  x foo  then
  true y !
end-atomic
```

Everything between **`begin-atomic`** and **`end-atomic`** is in an uninterruptable transaction – as long as we don't do any I/O

The code in **`foo`** also executes as part of this transaction

Programming this model would be just like the “old” Forth round-robin multitasker

redhat

Transactions and Atomicity

In a simple single-core system, **`begin-atomic`** and **`end-atomic`** don't have to do anything except ensure that no task switch occurs

In the case of a round-robin scheduler, they don't have to do anything at all

redhat

Transactional memory

For every atomic block, there are two possibilities

The transaction *commits*, so its results become visible outside the atomic block

The transaction *aborts*, and it leaves the program's state unchanged

If the Transactional Memory (TM) system detects a collision between transactions, it aborts one or more of them and re-executes those that have failed

This process of re-execution is not visible to the program

redhat

Types of Transactional Memory system

Consistent and Inconsistent

- Inconsistent TMs can lead to e.g. segfaults and exceptions

Fine-grained and coarse-grained

- Fine-grained TMs work on cells. Each time a cell in memory is accessed, the TM makes sure no other transaction has altered the cell since this transaction began
- Coarse-grained TMs work on entire objects in memory, but Forth has no idea what an object is

TM for Forth must be *consistent* and *fine-grained*

redhat

Types of Transactional Memory system

Deferred or direct update

- In a *direct-update* TM, writes are done immediately to memory. The system must record the original value of an object so that if a transaction has to be aborted it can be restored
- In a *deferred-update* TM, the system updates an object in a location private to the transaction, and only writes the real object when the transaction succeeds

redhat

TL2

Dice, Shalev, and Shavit, Transactional Locking II, DISC 2006

The front-running Software Transactional Memory system – IMO

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.811&rep=rep1&type=pdf

redhat

TL2

Uses commit-time locking and a global version clock

Fine-grained, consistent, deferred-update

redhat

TL2

Uses commit-time locking and a global version clock

Fine-grained, consistent, deferred-update

A global version-clock is incremented once by each transaction that writes to memory, and is read by all transactions.

Every cell in memory has a corresponding lock that contains a version number

Transactions start by reading the global version-clock and validating every location read against this clock

This guarantees that only consistent memory views are ever read

redhat

TL2

Writing transactions maintain a *write set.* This is the set of (address,value) pairs to be committed at the end of the transaction

Writing transactions also maintain a *read set.* This is the set of addresses that have been read during the transaction

Writing transactions need a read set but read-only ones do not

redhat

TL2

Writing Transactions

- Sample the global version-clock
- Run through a speculative execution
- Lock the write-set
- Increment global version-clock
- Validate the read-set
- Commit and release the locks

redhat

TL2

Low-Cost Read-Only Transactions

- Sample the global version-clock
- Run through a speculative execution

This is very fast

redhat

TL2

Low Contention Global version-clock Implementation

Tricky, but works. Read the paper

redhat

STM in Forth

Save the top N elements of the data stack (on the return stack)

Execute the transaction

If the transaction committed, throw away the saved stack items

If the transaction aborted, restore the top N elements of the data stack (from the return stack) and retry the transaction

If the transaction threw an exception, throw away the saved stack items and re-throw the exception

redhat

STM in Forth

```
'transaction @ if
    execute  // We're already in a transaction
else
    'transaction !
    10 depth min n>r  // Save 10 cells
    begin  nr@ drop
        ['] do-transaction catch
        dup retrytx = while
            drop  repeat
    throw
    nrdrop
    0 'transaction !
then
```

redhat

STM in Forth

We only need two new primitives, **`begin-atomic`** and **`end-atomic`**

But for good read-only transaction performance, we also want **`begin-readonly-atomic`**

`Begin-atomic` selects a wordlist that contains transactional versions of `:` `;` `@` `!`

Partial-word writes such as **`c!`** are hard but can be defined by using transactional `@` and `!`

Don't use words such as **`cmove`** in a transaction

Don't do any I/O in a transaction

redhat

STM in Forth

STM is going to be the only game in town

Although atomic transactions have been used in databases forever, Software Transactional Memory is very new. The key papers only date from a few years ago

Future processors will have hardware support for TM

GCC will soon have STM, but it's going to be a while before it's in Standard C

STM is not even on Java's radar

The **`atomic`** Forth primitives scale beautifully from the smallest embedded system to the largest multi-CPU server

Forth could be one of the first languages with STM support.

redhat

STM in Forth

There is no law that says Forth must be trailing-edge!

Questions?