Implementing the UPC memory consistency model for shared-memory architectures

Dan Bonachea

Paul Hargrove, Jason Duell
Kathy Yelick, Chuck Wallace

http://upc.lbl.gov

Overview

- Describe implementation work for Berkeley UPC 2.0
  - full conformance to the current mem model proposal
  - on the large variety of SMP architectures we support
  - work also includes clusters of SMP (SMP+NIC), but for simplicity will focus on SMP case
- Basic Requirements for implementing memory model
- Survey of relevant features on modern processor arch.
- Pseudo-code level implementation of UPC memory ops

Basic Requirements for implementing UPC memory model

- UPC Memory model defines:
  - restrictions on the order in which memory operations may appear to execute
  - Assume vanilla symmetric memory model which was just added in UPC Spec v1.2pre5. Relevant properties for this discussion:
    - Enforce a global total order over all strict operations
    - All strict operations prevent apparent reordering of surrounding accesses issued by the same thread
      - no accesses may be moved past a strict operation in either direction
  - Ordering properties for correct implementation of **strict** accesses are *stronger* than what is required for simply maintaining single-thread sequential dependencies
    - ie stronger than what is guaranteed by sequential compilers and hardware
  - UPC implementations need to enforce limitations on sequential compiler optimizations and architectural reordering at program points near strict operations

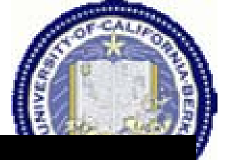
Basic Requirements for implementing UPC memory model

- Many traditional sequential compiler optimizations can **break** the memory model guarantees if applied to strict accesses
  - basically any optimization that causes strict accesses to be removed or reordered with respect to other accesses
  - eg. loop invariant code motion, copy propagation, common subexpression elimination, dead code elimination, etc. etc.
- Many modern processor/memory hardware features can also **break** the parallel memory model if applied to strict accesses
  - eg. Speculative Loads, Write combining, out-of-order write buffer completion, out-of-order cache invalidation processing, multi-bank memories supporting two or more outstanding split-phase writes
  - Notably, most processors allow loads to pass non-conflicting writes in the outgoing write buffer
  - Any of these can and often do break the semantics of strict accesses
    - *EVEN on a shared-memory system which is 100% fully "cache coherent"!!!*
  - To our knowledge, all modern architectures in use today exhibit one or more of these properties
    - need to use fence instructions to ensure consistency

Architectural Assumptions

- Assume: Shared memory (SMP) hardware with coherent caches
  - serial dependencies: every CPU sees it *own* read/writes in order
  - coherency: all CPU's agree on the value of any *single* mem. location which is not changing
  - no guarantees about the relative order in which updates to multiple memory locations are observed
- Assume: fully general architecture that requires both read and write fences
  - Write fence: compiler_opt_fence(); write_fence_instruction();
    - ensure all previous writes from this thread are "globally complete" before any subsequent reads or writes are issued
    - Prevents previous writes from appearing to move "down" past the fence
  - Read fence: compiler_opt_fence(); read_fence_instruction();
    - ensure any subsequent reads will see any previous stores from any thread which have "globally completed"
    - Prevents subsequent reads from appearing to move "up" past the fence
  - ReadWrite fence:
    - combination of both of the above
    - can be implemented more efficiently using single instruction on some arch

Implementing architectural read/write fence instructions



Architecture	Write fence instruction	Read fence instruction
Power/PowerPC	**sync**	**isync**
Alpha	**wmb**	**mb**
x86	**lock; addl $0,0(%%esp)** (any locked instruction is sufficient)	*none reqd.*
Athlon/Opteron	**mfence**	*none reqd.*
Itanium	**mf**	*none reqd.*
SPARC	**stbar**	*none reqd.*
MIPS	**sync**	*none reqd.*
PA-RISC	**SYNC**	*none reqd.*

Note: more is generally required when targeting architectures with separate scalar/vector units - outside the scope of this talk

Implementing compiler optimization fence

- Native/binary UPC compilers
  - Can directly enforce restrictions on compiler optimizations to preserve strict semantics
    - eg. Add dependency edges to precisely enforce the required conservatism
    - prevent the removal of any strict memory operations
    - prevent movement of any memory operations across a strict memory op
  - It *is* possible to do better and allow limited optimizations on strict ops
    - when fully parallel analysis can prove the program could not tell the difference
    - not worthwhile if we expect strict to be rare and usually represent real synch.
- Source-to-source UPC compilers (eg Berkeley UPC)
  - Need to apply the same optimization restrictions as native UPC compilers
    - for optimizations performed during source-to-source translation
  - *Also* need to prevent invalid optimizations on generated code
    - Most C compilers provide a method for doing this, in order to support related needs in system software (however, note volatile is generally *not* enough)
    - eg gcc: __asm__ __volatile__ ("": : : "memory")

      IntelC: __memory_barrier() intrinsic

      IBM xlc: #pragma reg_killed_by

Implementing the UPC memory model

- Relaxed Read or Local Read:
  - load
- Relaxed Write or Local Write:
  - store

Serial dependency preservation and coherency that we assume are provided by compilers and hardware already ensure we get conforming behavior for relaxed and local accesses (by design)

- Strict Write:
  - Write Fence: ensures previous relaxed writes are complete
  - store
  - ReadWrite Fence: ensures this write completes before any subsequent ops issued
- Strict Read:
  - ReadWrite Fence: ensures previous relaxed writes are complete
    prevent this read from being issued before previous ops complete
  - load
  - Read Fence: prevents subsequent relaxed reads from being issued before this read

Berkeley Lab

The University of California Berkeley
1868

Counter-example to constant propagation across strict ops

```
shared strict int a;
shared strict int b;
```

<u>Thread 0</u>

```
a = 10;
r1 = b;
c = a + 1;
```

<u>Thread 1</u>

```
a = 4;
b = 10;
```

SW(a, 10) → SW(a,4) → SW(b,10) → SR(b,10) → SR(a, ???)

must be 4