N2436: Memory region stores flush and reloads force

Document #:	N2436
Date:	2019-09-22
Project:	WG14 Programming Language C
	WG14-WG21 liaison group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>
	Bob Steagall <bob.steagall.cpp@gmail.com>

There is an increasing need in modern code to work efficiently with shared memory where stores have side effects not visible to the current program's abstract machine, and whose contents may change unbeknownst to the current program's abstract machine. `volatile` will do the job, but it does not perform well. This paper proposes a pair of library functions with greatly improved performance, and which additionally provide bare minimum support for NV-DIMM memory.

Contents

1 Introduction

`volatile` is the canonical standard way to tell the C compiler to not eliminate loads nor stores to the thus qualified items, and thus not subject stores to dead store elimination, nor loads to reload elision. It additionally prevents the compiler from merging and reordering loads and stores to such regions, which makes qualifying everything with `volatile` an inefficient way of working with shared memory.

What would be ideal is for a mechanism with fewer unwanted side effects for telling the C compiler that stores to the shared memory region must be guaranteed to occur, and loads from the shared memory must always reload rather than reuse previously loaded values. Such stores can be reordered, merged and composed by the compiler so long as their final side effects are rendered;

similarly, reloads from such regions can be reordered and merged by the compiler in any way which is maximally efficient, so long as a complete reload is performed.

Directly mapped non-volatile storage devices are a special kind of shared memory where the `fdatasync()` operation may be efficiently implemented by ensuring that the CPU flushes modified cache lines to 'main memory', where the directly mapped storage device has been mapped into the C program. Directly mapped non-volatile storage is expected to become prevalent on high end and low power systems in the medium term, as it offers superior performance and power consumption.

2 Proposed design

There are two functions, one to force reloading of a region of memory, and another to flush any pending stores to a region of memory.

There are two strengths of use for each function, with a corresponding impact upon runtime performance:

1. Force the compiler to write out and reload state, with optional CPU memory fence to constrain reordering of loads and stores from the perspective of other CPUs.
2. Force both the compiler and the CPU to write out and reload state, with optional CPU memory fence to constrain reordering of loads and stores from the perspective of other CPUs.

Cache coherency is always preserved. If the CPU is required to reload cache lines from a region of memory by these functions, all modified cache lines for that region in all CPUs are written out beforehand. This would ruin the utility of forcing the CPU to reload main memory which has changed independently of CPU writes, so it is on the user to ensure that no modified cache lines exist for a region reloaded by the CPU i.e. flush all modified lines, tell the device to change the data, reload cache lines.

CPU store flush and reload is not atomic with respect to concurrent CPU store flush and reload upon overlapping bytes. Users should employ additional synchronisation to prevent concurrent CPU store flushes or reloads upon the same cache lines.

No guarantee is given that individual cache line flushes occur in any particular order before the CPU memory release fence i.e. if you flush four modified cache lines, main memory will receive those updates in an unknown order. If you do not wish this, flush each cache line individually with memory fence in between each.

No guarantee is given that individual cache line reloads occur in any particular order after the CPU memory acquire fence. If you reload four cache lines, they may be fetched from main memory in any order. If you do not wish this, reload each cache line individually with memory fence in between each.

3 Proposed API

A reference implementation for the proposed library APIs can be found at https://github.com/ned14/quickcpplib/blob/master/include/mem_flush_loads_stores.hpp, with support for x86, x64, ARM v7 and AArch64.

It has been in production use in the presented form for about six months, however two previous earlier API designs were in use for some years beforehand. This proposed API design reflects the experience gained in those years.

```
enum memory_flush
{
  memory_flush_none,    //!< No main memory flushing.

  memory_flush_retain,  //!< Flush modified cache line in CPU out to main
                        //!!< memory, but retain as unmodified in cache.

  memory_flush_evict    //!< Flush modified cache line in CPU out to main
                        //!< memory, and evict completely from all caches.
};

/*! \brief Ensures that reload elimination does not happen for a region of
memory, optionally synchronising the region with main memory.

\return The kind of memory flush actually used.
\param data The beginning of the byte array to ensure loads from.
\param bytes The number of bytes to ensure loads from.
\param kind Whether to ensure loads from the region are from main memory.
\param order The atomic reordering constraints to apply to this operation.

\note `memory_flush_retain` has no effect for reloads from main memory,
it is the same as doing nothing. Only `memory_flush_evict` evicts all the
cache lines for the region of memory, thus ensuring that subsequent loads
are from main memory. Note that if the cache line being reloaded is modified,
it will be flushed to main memory before being reloaded, thus destroying
any modified data there. You should therefore ensure that concurrent
actors never modify main memory with modified cache lines in your CPU.
*/
memory_flush mem_force_reload_explicit(volatile char *data,
                                       size_t bytes,
                                       memory_flush kind,
                                       memory_order order);

/*! \brief The same as `mem_force_reload_explicit()`, but with
`kind` set to `memory_flush_none`, and `order` set to `memory_order_acquire`.
This does not reload loads from main memory, and prevents reads and writes
to this region subsequent to this operation being reordered to before this
operation.
*/
memory_flush mem_force_reload(volatile char *data,
                              size_t bytes);

/*! \brief Ensures that dead store elimination does not happen for a region of
memory, optionally synchronising the region with main memory.

```

```
\return The kind of memory flush actually used.
\param data The beginning of the byte array to ensure stores to.
\param bytes The number of bytes to ensure stores to.
\param kind Whether to wait until all stores to the region reach main memory.
\param order The atomic reordering constraints to apply to this operation.

\warning On older Intel CPUs, due to lack of hardware support, we always execute
‘memory_flush_evict‘ even if asked for ‘memory_flush_retain‘. This can produce
some very poor performance. Check the value returned to see what kind of flush
was actually performed.
*/
memory_flush mem_flush_stores_explicit(volatile const char *data,
                                       size_t bytes,
                                       memory_flush kind,
                                       memory_order order);

/*! \brief The same as ‘mem_flush_stores_explicit()‘, but with
‘kind‘ set to ‘memory_flush_none‘, and ‘order‘ set to ‘memory_order_release‘.
This does not flush stores to main memory, and prevents reads and writes to
this region preceding this operation being reordered to after this operation.
*/
memory_flush mem_flush_stores(volatile const char *data,
                              size_t bytes);
```

4 Implementation cost

It is firstly important to note that from the perspective of the compiler, the semantics of the above functions could be exactly the same as for calling an `extern` function – the compiler must write out relevant state before the call, and reload relevant state after the call. The main difference is that compilers *could* treat these functions a bit more optimally, in that `mem_flush_stores()` does not require reloading state afterwards, and `mem_force_reload()` does not require writing out state beforehand. Also, the fact that `mem_flush_stores()` always equals an `atomic_thread_fence(memory_order_release)`, and `mem_force_reload()` always equals an `atomic_thread_fence(memory_order_acquire)`, could be useful to an optimiser.

On compilers without link time optimisation, a conforming implementation of these functions is trivially easy, just being `extern` is enough. The main memory flush support is also trivially easy to implement, on Intel CPUs one loops the `CLWB` opcode over the cache lines, on ARM64 it is the `dc cvac` opcode. Most CPUs have a similar operation – for those which do not, the functions return `memory_flush_none` to indicate that no main memory flushing was performed.

On compilers with link time optimisation, there is a trick which the reference library implementation uses – it calls an externally modifiable function pointer, thus forcing the compiler to flush and reload state even under link time optimisation. Current link time optimisation technology appears to be unable to discern that the function pointer will only ever have one value, and thus deindirect the function call, and thus optimise away the desired semantics.

5 Conclusion

Obviously built in support for these operations in the compiler would be much better again still than calling an `extern` function which in the default implementation, simply calls an atomic thread fence and nothing else. This is why these operations are proposed for standardisation: built in (i.e. inlined) understanding would be more efficient than calling a function which calls a single CPU opcode and returns.

One would also be guaranteed that future link time optimisation technology improvements will retain the desired semantics.

Finally there would be the hope that compilers could better optimise how much state to flush and reload based on the byte range supplied, as the current reference implementation is a sledgehammer which flushes and reloads all compiler state.

6 References

[Intel CLWB opcode] https://www.felixcloutier.com/x86/clwb

[ARM DC opcode] https://developer.arm.com/docs/ddi0595/latest/aarch64-system-instructions/dc-cvac