WG21/N2272 **J16/07-0132** **2007-05-01**

Nick Maclaren
University of Cambridge Computing Service,
New Museums Site, Pembroke Street,
Cambridge CB2 3QH, England.
Email: nmm1@cam.ac.uk
Tel.: +44 1223 334761
Fax: +44 1223 334679

Optional Sequential Consistency

1.0. Introduction

This addresses the possibility referred to in N2177 that we could allow a parameter to the atomics template to change the behaviour from sequential consistency (SC, Lamport) to, say, cache coherent causal consistency (CCCC or C4, for which we need a reference, which may be provided by N2226). This is the generally-accepted "weak C4", and not the poorly-understood "strong C4".

This document refers **only** to the simpler, "high-level" atomics.

It does not reopen the question of whether they should specify sequential consistency by default, and it assumes that.

I believe that the following assertions are true:

1: Most programmers starting with shared-memory parallelisation initially think in terms of sequential consistency.

2: Most programmers capable of writing correct threaded code can be taught C4 in a few minutes and do not find it "unnatural".

3: Most programmers get caught out by the difference between SC and C4 only when they are being too clever by half – and there are innumerable other "gotchas" in C++ waiting for such people.

4: Most of the very few programmers capable of **using** the difference will want the "low-level" atomics, anyway, as it gives them more control and potentially more performance.

5: It would be a very bad idea for some "high-level" atomics to support SC and others to support C4 *in the same execution of the same program*, as no ordinary programmer would be able to work out the consequences. I can't, for one.

6: Most (not all) tightly coupled shared memory architectures support SC in the hardware, but only some of those actually deliver it to the program for unflagged 'atomic' operations (e.g. loads or stores of an aligned `int`), despite what the documentation implies. When C++0X is standardised, not delivering SC to the program will become a clear bug.

7: It is expensive to deliver SC on some tightly coupled shared memory architectures, and very expensive when emulating shared memory in software on large distributed memory systems. Note that this expense is more about scalability to lots of processors than a simple constant factor.

8: With the above model, the effect on C++ is **exactly** equivalent to selecting between garbage collection and manual memory management, as proposed in N2128, but does not interact with that (or anything else, outside the atomics).

Therefore I believe that we **should** provide an option, and this paper proposes how to do that – if people agree that an option should be provided. Lawrence Crowl quite correctly points out that

it specifies nothing that an implementor could not provide as an extension. However, by providing an option, we will encourage all implementors to be compatible, if they choose to use this feature.

1.1. The Proposal

This uses the attribute syntax proposed in N2236, not the keywords used in N2128, though all names are obviously arbitrary, using code like:

```
#include <atomic>
register [[atomics_need_SC]];
#include <atomic>
register [[atomics_need_C4]];
#include <atomic>
register [[atomics_need_PDQ];[†]
#include <atomic>
```

1: The C++ standard specifies the meaning of `atomics_need_SC` and (using suitable wording) `atomics_need_C4`. Any other specification is implementation-defined.

2: The default in C++0X is `atomics_need_SC`, and a conforming implementation must support that. It must accept `atomics_need_C4`, but may treat it as `atomics_need_SC`.

3: An implementation must provide the semantics of **all** of the models specified in the program for **all** of the uses (i.e. it must implement the closure of the models used anywhere in the code). Note that I am not assuming that SC is the strongest model, but that will be the normal case[‡].

4: An implementation is expected to issue a diagnostic and terminate if it starts in a weak model and loads a dynamic library that needs a stronger or incompatible one. This is likely to be stated only in an informative footnote, as dynamic loading is outside C++0X.

The consequence is that neither an implementor nor a programmer need do anything except assume SC, but that an implementation **may** allow a programmer to request another model in a standard-defined fashion and a programmer **may** specify that he needs only a weaker model (or even that he needs a stronger one) in conforming (but non-portable) code.

It will make it fairly easy for implementors to provide alternative models that can be used in clean programs. This could be a great help in getting real experience with different models for any future extensions or changes.

A programmer that needs SC and wants to future-proof his code can specify `atomics_need_SC` explicitly and be reasonably sure that any changes in C++1X will not affect him. While we should not assume that C++1X will change the default, we should not lock it out unnecessarily.

† I have no views on whether an implementation should be required to use `__atomics_need_PDQ` or whether attributes starting `atomics_need_` should be extensible.

‡ Some real-time systems might want to support strict time-based models, and this specification allows that without putting anything extra into the standard.