Weakening Correctness and Linearizability for Concurrent Objects on Multicore Processors

Graeme Smith[1] and Lindsay Groves[2]

[1]School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, Australia
[2]School of Engineering and Computer Science,
Victoria University of Wellington, Wellington, New Zealand

Abstract. In this paper, we argue that there are two fundamental ways of defining correctness of concurrent objects on the weak memory models of multicore processors: we can abstract from concurrent interleaving and weak memory effects at the specification level, or we can abstract from concurrent interleaving only, leaving weak memory effects at the specification level. The first allows us to employ standard linearizability as the correctness criterion; a result proved in earlier work. The second requires a weakening of linearizability. We provide such a weakening and prove it sound and complete with respect to this notion of correctness.

1 Introduction

Libraries of efficient concurrent objects are central to developing concurrent programs. High-level concurrent algorithms utilise concurrent objects for sharing data between threads, e.g., concurrent queues and stacks, and for inter-thread synchronisation, e.g., locks [15]. Correctness of such objects is usually defined with respect to a sequential specification. For example, a concurrent queue being accessed concurrently by multiple threads should still behave essentially like a queue. The standard criterion for relating concurrent implementations of objects to their sequential specifications is *linearizability* [16]. In recent years, however, the question has arisen as to whether linearizability is the appropriate correctness criterion in the presence of weak memory models.[1] This has led to several proposals for weaker versions of linearizability which either change the sequential specification to allow weak memory effects [4, 14], or change the definition of linearizability itself [25, 8, 9, 11], as well as new approaches to verifying correctness unrelated to linearizability [21].

On sequentially consistent (SC) memory models, *observational refinement* [12] and *contextual refinement* [10] have been proposed as reference points with which to judge such correctness criteria. These take the view that an object implementation is correct if and only if a client program calling the object's

[1] In this paper, we refer solely to *hardware* weak memory models of multicore processors, e.g., x86-TSO [19], ARM [13, 20] and IBM POWER [22], and not *software* weak memory models that allow for compiler optimisations, e.g., C11 [3].

operations cannot distinguish the object from one that behaves according to the specification. Filipović et al. [12] prove that under certain assumptions observational refinement is equivalent to linearizability.

A similar reference point for judging correctness criteria on weak memory models, called *object refinement*, was recently proposed by Smith et al. [23]. In follow-on work, they prove the (somewhat surprising) result that the standard definition of linearizability for SC is also equivalent to object refinement [24]. Their proof holds for all current memory models, including SC, and without the assumptions required by Filipović et al. For SC, this suggests that object refinement captures exactly what was intended by linearizability.

The result for weak memory models, however, is quite strong. A number of efficient implementations of concurrent objects are incorrect under standard linearizability. This is due to the fact that under weak memory models the effect of an operation on one thread may be delayed from the perspective of other threads. In this paper, we suggest a relaxation of object refinement that allows more implementation flexibility, and discuss the consequences in terms of the verification and use of concurrent objects. In particular, we present a definition of *weak linearizability* that is sound and complete with respect to our relaxed notion of object refinement, and argue why both weak linearizability and standard linearizability are required in practice.

2 Correctness

The operations of a concurrent object may be called by multiple threads simultaneously. This results in an interleaving of their code in which interference must be handled using locks or, when efficiency is important, non-blocking techniques [18]. The behaviour of such interleaved code is difficult to reason about, and specifications usually abstract from interleaving by having operations which are atomic. For example, operations may be specified with a precondition/postcondition pair. An implementation C of a concurrent object is considered correct in this setting when any program P which calls C's operations behaves in a way consistent with calling the atomic operations of A:

$$\forall P \bullet P[A] \sqsubseteq P[C] \qquad (1)$$

where $P[A]$ and $P[C]$ denote the program P calling the atomic operations of A, and the potentially interleaving operations of C, respectively, and $\sqsubseteq$ denotes trace refinement [1, 2] where observations are of changes to *program variables*, i.e., variables declared as part of P.

2.1 Linearizability

The standard notion of correctness on SC is linearizability [16]. The semantics $[\![C]\!]$ of a concurrent object C is expressed as a prefix-closed set of *histories*, where each history is a sequence of events corresponding to *invocations* and *responses* of operations calls. The semantics $[\![A]\!]$ of an object specification A can similarly be expressed as a prefix-closed set of histories. Since operations are regarded as atomic (i) operation calls cannot overlap, i.e., an invocation is immediately followed by its matching response, and (ii) prefixes are restricted to *complete* histories in which the final event cannot be an invocation.

Linearizability relates histories of an object implementation, which may have *pending* invocations, i.e., invocations for which there is no response, to histories of an object specification which do not. To do this, it needs to *complete* the implementation histories. This can be done by adding a response when a pending invocation is deemed to have taken effect, and removing the invocation when it has not [16].

Let $ext(h)$ be the set of extensions of a history h formed by adding a sequence of responses for an arbitrary subset of the pending invocations, and $comp(h)$ complete h by removing all pending invocations. A concurrent object C is said to *linearize* with an object specification A when the following holds.

$$C \ lin \ A \mathrel{\widehat{=}} \forall\, h : [\![C]\!] \bullet \exists\, h' : [\![A]\!] \bullet \exists\, h^{+} : ext(h) \bullet comp(h^{+}) \sim h' \wedge \prec_{comp(h^{+})} \subseteq \prec_{h'}$$

where $h \sim h'$ denotes that h and h' are *thread equivalent*, i.e., when restricted to the events of any one thread they have the same sequence of invocations and responses, and $\prec_h$ captures the order of operations in a trace, where one operation comes before another if its response is before the other's invocation.

The intuition behind the definition is that operations which are overlapping in $comp(h^{+})$ are not ordered by $\prec_{comp(h^{+})}$ and, with $\prec_{h'}$ being a superset of $\prec_{comp(h^{+})}$, can occur in any order in h'. The implementation history h is said to linearize to h'.

Linearizability is *compositional*, allowing us to prove the correctness of a system of interacting objects by showing each component object is linearizable with respect to its specification [16].

3 Weak Memory Models

Weak memory models optimise performance by allowing the hardware to control accesses to shared memory. As a consequence, the effect of operations on one thread may be delayed from the perspective of another. On TSO, for example, writes to shared variables are placed in a FIFO store buffer, and only take effect in memory at a later time determined by the hardware, or when a fence is reached in the code [19]. Such fences may be added liberally by the programmer, but can negate the optimisations that the memory model offers.

POWER and ARM are weaker than TSO, allowing delayed writes to take effect out of FIFO order unless a specific dependency exists between them [13, 22]. They additionally support *non-multi-copy atomicity*.[2] This allows writes by one thread to be seen by other threads at different times, again determined by the hardware. This can be captured semantically by a *write list* [6] which orders

[2] This is no longer true for the latest version of ARMv8 [20].

all writes which have occurred in a program execution along with which threads have seen those writes.

3.1 Modelling Weak Memory Behaviour

Following [23], we model additional behaviours under weak memory models by adding an *effect* event for each program step and each operation call. An effect event indicates when the program step or operation takes place from the global perspective of all threads in the program. For a program step, this will be after the program step occurs, and for an operation after its invocation. The exact position of the effect event is determined by the memory model semantics.

For program steps and operations which write to shared variables, the effect will be when all threads have seen all of the writes. For example, on TSO the effect will be when the last write associated with the program step or operation takes place in the global memory. On non-multi-copy atomic processors, the effect will be when, for each updated variable, all threads have seen the write, or have seen a later write to the same variable.

Given a memory model M, we let $[\![C]\!]_M$ denote the set of histories of an implementation C under M, and $[\![A]\!]_M$ the set of histories of a specification A under M. These histories will include effects, as well as invocations and responses.

Returning to correctness as defined in (1), while the behaviour of $P[C]$ under M can be determined from $[\![C]\!]_M$, a choice needs to be made about the meaning of $P[A]$. On SC, the behaviour of $P[A]$ abstracts from interleaving of operations, as A's operations are regarded as atomic. On weak memory models, there are two fundamental possibilities:

1. $P[A]$ abstracts from the interleaving of operations (as on SC) *and* weak memory model effects.
2. $P[A]$ abstracts from the interleaving of operations but includes weak memory effects.

In [23], the first option was taken, resulting in correctness being equivalent to linearizability for *all* memory models [24]. As discussed in [24], a consequence of this choice is that any operation whose effect can influence the output of a later operation needs to be fenced to be correct on a weak memory model. This is quite strict as it disallows many implementations where fences are avoided to improve efficiency. In the remainder of this paper we consider the second option above by allowing weak memory effects in $P[A]$.

4 Weakening Correctness

Program behaviour under a weak memory model is modelled in terms of *traces*, which are sequences of (program) step, invocation, response and effect events:

$$\mathit{Event} \mathrel{\widehat{=}} \mathit{step}(PS) \mid \mathit{eff}(PS) \mid \mathit{inv}(Op, \mathit{Val}) \mid \mathit{res}(Op, \mathit{Val}) \mid \mathit{eff}(Op, \mathit{Val})$$

where PS is a program step by a particular thread, Op an operation call by a particular thread, and Val a set of values for inputs and outputs, including $\perp$ meaning no input or output.

Each event in a trace is unique (calls to the same operation, for example, being distinguished by annotating the operation name with its order of occurrence):

$$Trace \mathrel{\widehat{=}} \{t : \mathrm{seq}\, Event \mid (\forall\, i, j \leq \#t \bullet i \neq j \Rightarrow t_i \neq t_j) \wedge wf(t)\}$$

where wf, the well-formedness condition on traces, states that an invocation of an operation always occurs before the associated response and effect, a program step always occurs before its effect, and the output value of an operation's effect is the same as that of the corresponding response event.

$$\begin{aligned}
wf(t) \mathrel{\widehat{=}} {} & (\forall\, a : Op;\ out : Val \bullet \forall\, j \leq \#t \bullet t_j \in \{res(a, out), eff(a, out)\} \Rightarrow \\
& \qquad\qquad\qquad\qquad \exists\, in : Val;\ i < j \bullet t_i = inv(a, in)) \wedge \\
& (\forall\, p : PS \bullet \forall\, j \leq \#t \bullet t_j = eff(p) \Rightarrow \exists\, i < j \bullet t_i = step(p)) \wedge \\
& (\forall\, a : Op \bullet \exists\, out : Val \bullet \\
& \quad \forall\, i \leq \#t;\ o : Val \bullet t(i) = res(a, o) \vee t(i) = eff(a, o) \Rightarrow o = out)
\end{aligned}$$

A *sequential trace* is one where operations are atomic, i.e., all invocations have a matching response, and no events apart from effects may occur between the invocation and response of an operation.

$$\begin{aligned}
SeqTrace \mathrel{\widehat{=}} \{t : Trace \mid {} & (\forall\, a : Op;\ in : Val;\ i \leq \#t \bullet t_i = inv(a, in) \Rightarrow \\
& \qquad \exists\, out : Val;\ j \leq \#t \bullet t_j = res(a, out)) \wedge \\
& (\forall\, a : Op;\ in, out : Val;\ i < j < k \leq \#t \bullet \\
& \qquad t_i = inv(a, in) \wedge t_k = res(a, out) \Rightarrow \\
& \qquad\qquad \exists\, b : Op;\ out_b : Val \bullet t_j = eff(b, out_b))\}
\end{aligned}$$

The events of a trace t determine a set, $events(t)$, and the (total) order in which these events occur is denoted by a relation, $<_t$:

$$\begin{aligned}
& events(t) \mathrel{\widehat{=}} \{a : Event \mid \exists\, i \leq \#t \bullet t_i = a\} \\
& <_t \mathrel{\widehat{=}} \{(a, b) : Event \times Event \mid \exists\, i, j \leq \#t \bullet i < j \wedge t_i = a \wedge t_j = b\}
\end{aligned}$$

A program P has a set of events, $events(P)$, which it can undergo (according to the program text) and, for each memory model M, a partial order, $<_{P_M}$, which captures when an event can occur only after another under M (see [24] for further details). The traces of P are defined as follows:

$$[\![P]\!]_M \mathrel{\widehat{=}} \{t : Trace \mid events(t) \subseteq events(P) \wedge <_{P_M} \Subset <_t\}$$

where $<_{P_M} \Subset <_t$ specifies whether an order is *allowed* by P on M, formally defined as:

$$<_{P_M} \Subset <_t \ \mathrel{\widehat{=}}\ \forall (a, b) : <_{P_M} \bullet b \in events(t) \Rightarrow (a, b) \in <_t$$

That is, for any event b that occurs in trace t, if this event is constrained to come after another event a by $<_{P_M}$, then event a must also occur in t before event b.

The semantics of $P[C]$ under M are those traces t of $[\![P]\!]_M$ whose restriction to *object events* (invocation, responses and operation effects), denoted $t_{|o}$, is in $[\![C]\!]_M$.

$$[\![P[C]]\!]_M \mathrel{\widehat{=}} \{t : [\![P]\!]_M \mid t_{|o} \in [\![C]\!]_M\}$$

The semantics of $P[A]$ is given similarly:

$$[\![P[A]]\!]_M \mathrel{\widehat{=}} \{t : [\![P]\!]_M \mid t_{|o} \in [\![A]\!]_M\}$$

To motivate our definition of $[\![A]\!]_M$, consider an object with two operations: W, which writes to an object variable x, whose value is initially 0, and R, which reads x. One possible behaviour under a weak memory model such as TSO is:

$$\langle \mathit{inv}(n.W, 1), \mathit{res}(n.W, \bot), \mathit{inv}(m.R, \bot), \mathit{eff}(m.R, 0), \mathit{res}(m.R, 0), \mathit{eff}(n.W, \bot)\rangle$$

where the read by thread m returns 0 since the effect of the write by thread n has not yet taken effect. The operations do not overlap (the invocation of $m.R$ occurs after the response of $n.W$), modelling that the operations are atomic. However, weak memory effects are included, due to the effect of $n.W$ being delayed.

The above trace corresponds to a history of A, the specification of the object, where $m.R$ occurs before $n.W$. In general, a history h of A_M will be thread equivalent to a history h' of A such that the operations of h are ordered consistently with placing them between the corresponding invocations and effects of h. This can be defined in terms of the notation $h \sim h'$ and $\prec_h$ introduced to define linearizability in Section 2:

$$[\![A]\!]_M \mathrel{\widehat{=}} \{h : \mathit{SeqTrace} \mid h_{|o} = h \wedge \exists\, h' : [\![A]\!] \bullet h \sim h' \wedge \prec_{\mathit{trans}(h)} \subseteq \prec_{h'}\}$$

where $\mathit{trans}(h)$ replaces the effect of each operation in h by its response:

$$\begin{array}{lcl}
\mathit{trans}(\langle\,\rangle) & = & \langle\,\rangle \\
\mathit{trans}(\langle \mathit{inv}(a, \mathit{in})\rangle \frown h') & = & \langle \mathit{inv}(a, \mathit{in})\rangle \frown \mathit{trans}(h') \\
\mathit{trans}(\langle \mathit{res}(a, \mathit{out})\rangle \frown h') & = & \mathit{trans}(h') \\
\mathit{trans}(\langle \mathit{eff}(a, \mathit{out})\rangle \frown h') & = & \langle \mathit{res}(a, \mathit{out})\rangle \frown \mathit{trans}(h')
\end{array}$$

Object refinement, our notion of correctness, is then defined as:

$$P[A] \sqsubseteq_M P[C] \ \mathrel{\widehat{=}}\ \forall\, t : [\![P[C]]\!]_M \bullet \exists\, t' : [\![P[A]]\!]_M \bullet t'_{|global} = t_{|global}$$

where $t_{|global}$ is the restriction of t to *observable* program steps, i.e., those which change global variables.

5 Weak Linearizability

We now present a notion of weak linearizability, and prove it sound and complete with respect to our notion of object refinement defined in Section 4. This notion has been previously suggested for concurrent objects on TSO [8, 25]. It involves allowing an operation to linearize anywhere between its invocation and effect (i.e., when the operation's final write takes place in global memory on TSO). Here we show it corresponds to our notion of correctness not just for TSO, but for any weak memory model.

Let *ext* return the set of traces which extend a given trace with a sequence of responses such that the result is still a trace, i.e., responses are only added for pending invocations:

$$\mathit{ext}(t) \mathrel{\widehat{=}} \{t \frown \mathit{tr} : \mathit{Trace} \mid \forall\, i \leq \#\mathit{tr} \bullet \exists\, a : \mathit{Op};\ \mathit{out} : \mathit{Val} \bullet \mathit{tr}_i = \mathit{res}(a, \mathit{out})\}$$

and let *comp* return the trace obtained by removing all invocations from a given trace which have neither an effect nor a response:

$$\begin{array}{lcl} \mathit{comp}(\langle\,\rangle) & = & \langle\,\rangle \\ \mathit{comp}(\langle \mathit{inv}(a, \mathit{in})\rangle \frown t') & = & \mathit{comp}(t'),\ \text{if } \mathit{NoResp}(a, t') \\ \mathit{comp}(\langle e\rangle \frown t') & = & \langle e\rangle \frown \mathit{comp}(t'),\ \text{otherwise} \end{array}$$

where $\mathit{NoResp}(a, t) \mathrel{\widehat{=}} \nexists\, i \leq \#t;\ \mathit{out} : \mathit{Val} \bullet t_i \in \{\mathit{res}(a, \mathit{out}), \mathit{eff}(a, \mathit{out})\}$.

Weak linearizability, lin_M, is defined for any weak memory model M. By weak memory model, we mean a memory model where effects of program steps and operations may be delayed — this definition is not intended to be used with SC.

$$\begin{array}{l} C\ \mathit{lin}_M\ A \mathrel{\widehat{=}} \forall\, h : [\![C]\!]_M \bullet \exists\, h' : [\![A]\!] \bullet \\ \qquad\qquad\qquad \exists\, h^+ : \mathit{ext}(h) \bullet \mathit{comp}(h^+) \sim h' \wedge \prec_{\mathit{trans}(\mathit{comp}(h^+))} \subseteq \prec_{h'} \end{array}$$

Note that we linearize with respect to the original specification A which is independent of the memory model M.

Weak linearizability is proved to be compositional in [7]. The proofs of soundness and completeness below rely on the following lemmas proved in [24].

Lemma 1. *If the events of a trace t are events of a program P, then so are the events of any completion of t.*

$$\begin{array}{l} \forall\, P \bullet \forall\, t : \mathit{Trace} \bullet \forall\, t^+ : \mathit{ext}(t) \bullet \\ \qquad \mathit{events}(t) \subseteq \mathit{events}(P) \Rightarrow \mathit{events}(\mathit{comp}(t^+)) \subseteq \mathit{events}(P) \end{array}$$

Lemma 2. *If a trace t is allowed by a program P on memory model M, then so is any completion of t that only adds responses for operations whose effects have occurred.*

$$\begin{array}{l} \forall\, P, M \bullet \forall\, t : \mathit{Trace} \bullet \forall\, t \frown \mathit{tr} : \mathit{ext}(t);\ a : \mathit{Op};\ \mathit{out} : \mathit{Val} \bullet \\ \qquad (\forall\, i \leq \#\mathit{tr} \bullet \mathit{tr}_i = \mathit{res}(a, \mathit{out}) \Rightarrow \exists\, j \leq \#t \bullet t_j = \mathit{eff}(a, \mathit{out})) \wedge <_{P_M} \Subset <_t \Rightarrow \\ \qquad\qquad <_{P_M} \Subset <_{\mathit{comp}(t \frown \mathit{tr})} \end{array}$$

Our proofs also use the notion of a *matching trace*, which is formed from a trace t as follows. Firstly, t is extended with responses for exactly those pending invocations for which there is an effect, and the remaining pending invocations are removed. Secondly, each response is moved to immediately after any contiguous sequence of effects following its invocation.

5.1 Soundness

We now show that our notion of weak linearizability is sound with respect to our definition of object refinement.

Theorem 1. *If an object implementation C linearizes with an object specification A on memory model M, then C is an object refinement of A on M.*

$$C \ \mathit{lin}_M \ A \ \Rightarrow \ \forall P \bullet P[A] \sqsubseteq_M P[C]$$

Proof. Assume that $C \ \mathit{lin}_M \ A$ holds, and consider an arbitrary program P with $[\![P[C]]\!]_M \neq \varnothing$ (when $[\![P[C]]\!]_M = \varnothing$ the consequent is trivially true). We must show that for any trace, t, of $P[C]$ under weak memory model M, there is a corresponding trace, t', of $P[A]$ under M.

Since $t \in [\![P[C]]\!]_M$, there is an $h \in [\![C]\!]_M$ such that $t_{|o} = h$, and since $C \ \mathit{lin}_M \ A$, there is also an $h' \in [\![A]\!]$ and $h^+ \in \mathit{ext}(h)$ such that:

$$\mathit{comp}(h^+) \sim h' \ \wedge \ \prec_{\mathit{trans}(\mathit{comp}(h^+))} \subseteq \prec_{h'} \qquad \text{(S1)}$$

There may be several possible choices for h^+. We choose h^+ so that a response is added for each pending invocation whose effect occurs in t. This is always possible since we know that there exists at least one extension and related abstract history. Call them h_0^+ and h_0', respectively. h_0^+ cannot have less than the required responses. If it did, $\mathit{comp}(h^+)$ would be left with a pending invocation (with an effect) but no response. Hence, $\mathit{comp}(h^+) \sim h'$ would not hold. If h_0^+ has more than the required responses, since $[\![A]\!]$ is prefixed-closed, we can find an h' which is a subsequence of h_0' which does not have the additional operations corresponding to the extra responses.[3] This h' will satisfy (S1) for our chosen $h+$.

Let t' be the matching trace of t. Since a matching trace maintains the order of program steps and their effects we have:

$$t'_{|global} = t_{|global} \qquad \text{(S2)}$$

To complete the proof, it remains to show that $t' \in [\![P[A]]\!]_M$.

Since there are no overlapping operations on a given thread, the order of invocations and responses on a given thread in $\mathit{comp}(h^+)$ will be maintained in t'. Hence from (S1) we have:

$$t'_{|o} \sim h' \qquad \text{(S3)}$$

[3] Since there is at most one pending invocation per thread, such an h' will be in the prefix-closed set $[\![A]\!]$.

Since the relative order of invocations and effects in $\mathit{comp}(h^+)$ will be maintained in t', we have that $\mathit{trans}(t'_{|o}) = \mathit{trans}(\mathit{comp}(h^+))$. Hence from (S1) we have:

$$\prec_{\mathit{trans}(t'_{|o})} \subseteq \prec_{h'} \tag{S4}$$

Since invocations and responses are only separated by effects in t', provided that it satisfies the well-formedness property of traces and is in *Trace*, it will be in *SeqTrace*:

$$t' \in \mathit{Trace} \Rightarrow t' \in \mathit{SeqTrace} \tag{S5}$$

Therefore, from (S3), (S4) and (S5) and the definition of $\llbracket A \rrbracket_M$, we have:

$$t' \in \mathit{Trace} \Rightarrow t'_{|o} \in \llbracket A \rrbracket_M \tag{S6}$$

Given the definition of $\llbracket P[A] \rrbracket_M$ and (S6), to prove that $t' \in \llbracket P[A] \rrbracket_M$ we need to show that $t' \in \mathit{Trace}$, and that $t' \in \llbracket P \rrbracket_M$, i.e., $\mathit{events}(t') \subseteq \mathit{events}(P)$ and $<_{P_M} \Subset <_{t'}$. We prove each of these below, where t^+ is the trace t extended with the sequence of responses added to h to form h^+.

(i) $t' \in \mathit{Trace}$:
Since $t \in \llbracket P[C] \rrbracket_M$, it follows from Lemmas 1 and 2 that $\mathit{comp}(t^+) \in \llbracket P \rrbracket_M$ and hence is a *Trace*. Since t' has the same events as $\mathit{comp}(t^+)$, its events are unique. Also, since the construction of t' does not alter the relative order of program steps with their effects, nor invocations with their responses and effects, *wf* holds. Hence, $t' \in \mathit{Trace}$.

(ii) $\mathit{events}(t') \subseteq \mathit{events}(P)$:
Since $\mathit{comp}(t^+) \in \llbracket P \rrbracket_M$, it follows that $\mathit{events}(\mathit{comp}(t^+)) \subseteq \mathit{events}(P)$ from the definition of $\llbracket P \rrbracket_M$. Since t' has the same events as $\mathit{comp}(t^+)$, it follows that $\mathit{events}(t') \subseteq \mathit{events}(P)$.

(iii) $<_{P_M} \Subset <_{t'}$:
To prove this property we show that $\forall (a, b) : <_{P_M} \bullet\, b \in \mathit{events}(t') \Rightarrow (a, b) \in <_{t'}$. We consider four cases, according to whether a and b are program events or object events.

(a) The order between two *program events* (i.e., program steps or their effects) that is enforced by $<_{P_M}$ is maintained in t'.

This holds as the relative order of program events of t is unchanged in t'.

(b) If $<_{P_M}$ enforces that a program event has to occur before an *object event* (i.e., an invocation, response or effect of an operation) then this order is maintained by t'.

The relative order of program events with invocations and effects of t is unchanged in t'. Since an object's state is only accessed via its operations, there can be no synchronisation between a program event and the object's state while the operation is executing. Hence, if a program step must come before an operation's response, it must come before its invocation too. Since the response of an operation is not moved before its invocation, it is not moved before any program events which must come before it.

(c) If $<_{P_M}$ enforces that an object event has to occur before a program event then this order is maintained by t'.

The relative order of program events with invocations and effects of t is unchanged in t'. Since responses are only moved earlier in the trace, all program events after a response remain after the response.

(d) The order between two object events that is enforced by $<_{P_M}$ is maintained in t'.

The response of an operation *op* is not moved before its invocation, and its effect (on a weak memory model) cannot be constrained by P to occur before the response (since this would require the operation to either not write to shared variables or have a fence, and the operation's implementation is outside of P's control).
Also, since the implementation of object operations is outside P's control, synchronisation between operations cannot be enforced by P. Hence, if a response of an operation *op* must occur after another operation event e, other than *op*'s invocation, then *op*'s invocation must also occur after e. Hence, *op*'s response is not moved before e. □

5.2 Completeness

We now show that our notion of weak linearizability is complete with respect to our definition of object refinement. The completeness proof uses a notion of a *recording program*, which records each program step and operation call in a global variable g. For each program step PS on thread n we have:

$$PS;\ g := \text{“}n.PS\text{”}$$

and for each call on an operation Op on thread n we have:

$$l := in;\ x := Op(in);\ g := \text{“}n.Op(in, out)\text{”}$$

where l is a local variable used to hold the value of the input until after the operation. Since the changes to g will appear in the observable part of any trace of the recording program, when $t_{|global} = t'_{|global}$ for any traces t and t', we know that these traces have undergone the same program steps, operation calls and outputs of operation calls up until the last recording on each thread. These events may occur in a different order, however, since the recordings made by different threads can be interleaved.

The completeness proof uses the following lemma.

Lemma 3. *If P is a recording program and C is an object refinement of A, then for a given implementation trace $t \in [\![P[C]]\!]_M$ either the matching trace t' is in $[\![P[A]]\!]_M$, or there is a trace t'' in $[\![P[A]]\!]_M$ which is formed by moving one or more effects in t' earlier than in t.*

Proof Following the reasoning for matching traces in the proof of Theorem 1, t' will be in $[\![P]\!]_M$. Hence, the relative order of program steps and invocations in t' will be allowed by $P[A]$ on memory model M.

Since P is a recording program, if any effect allowed in t is not allowed by A, C will have a trace which is observably different to any of A. This contradicts the assumption that C is an object refinement of A. Hence, all effects in t are allowed by A. It remains for us to show that the effects may occur in the order of t' or some t'' formed from t' by moving one or more effects earlier.

We proceed using a proof by contradiction. Suppose that $[\![P[A]]\!]_M$ does not contain t' or any trace t'' formed from t' by moving one or more effects earlier. For C to be an object refinement of A, there must be a trace s in $[\![P[A]]\!]_M$ with the same events as t but with one or more effects occurring later than in t.

Let $\bar{t}$ denote the trace t with all responses removed, and u and v be non-empty sequences of events, and w be a possibly empty sequence of events. Let $\bar{t} = u \frown \langle \mathit{eff}_a \rangle \frown v \frown w$ and $\bar{s} = u \frown v \frown \langle \mathit{eff}_a \rangle \frown w$, i.e., where a single effect eff_a, of an event a, occurs later in s than in t. This implies that the occurrence of the last event of v, or when that event is an invocation the required output of its associated operation, requires that the effect of a has not occurred. From this we can deduce that:

(i) the last event of v is not a program step. If it were a program step then this program step would need to refer directly to a change made by a to a program variable (otherwise eff_a could occur before it). Since the program step is identical in the implementation (both use P), the implementation traces would also require eff_a to occur after v.

(ii) the last event of v is not an invocation. Let the last event of v be b. If b were an invocation then the associated operation's outputs in A would rely on a's effect not having occurred. On the other hand, the operation in C can produce the outputs in t after a has taken effect.

If a depends on earlier events of the same thread in t then there will be a dependency between those events that will prevent the effect of a occurring before the effects of the earlier events [6]. Hence, C allows the operation associated with b to produce the outputs in t when all such earlier events of a's thread have taken effect.

Therefore, it is possible to construct a recording program P' similar to P which forces a to take effect before b, e.g., eliding the recording events, P' could be:

$$...;\ a;\ \mathit{fence};\ z = 1;\ ...\ ||\ ...;\ \mathit{await}(z = 1);\ b;\ ...\ ||\ ...$$

where the fence forces a (and all earlier events in a's thread) to take effect, and the await instruction ensures that b does not occur until this has happened.

There will be a trace in $[\![P'[C]]\!]_M$ in which the operation invoked by b has the same output as in t. However, there will not be such a trace in $[\![P'[A]]\!]_M$. The operation invoked by b will behave differently due to the effect of a. This will result in a different recording, and hence implies C is not an object refinement of A.

(iiii) the last event of v is not an effect. The specification and implementation execute on the same memory model. Hence, any constraint on the ordering of effects in $P[A]$ will also hold in $P[C]$ [23].

Hence, the last step of v is neither a program step, invocation nor effect and we have a contradiction. The proof can be generalised to more than one effect event occurring later in s. □

We can now state and prove the completeness theorem.

Theorem 2. *If an object implementation C is an object refinement of a specification A on memory model M then C linearizes to A on M.*

$$(\forall P \bullet P[A] \sqsubseteq_M P[C]) \;\Rightarrow\; C \; \mathit{lin}_M \; A$$

Proof Assume the antecedent is true and that $[\![C]\!]_M \neq \varnothing$. Let h be a history in $[\![C]\!]_M$, and let P be a recording program which can generate a trace t corresponding to h, i.e., $t_{|o} = h$. Let $t^+ \in \mathit{ext}(t)$ be an extension of t which adds a response for exactly those pending invocations for which there is an effect in t.

Let t' be the matching trace of t. From Lemma 3 we know that there exists a trace s in $[\![P[A]]\!]_M$ such that $s = t'$ or s is formed from t' by moving one or more effects earlier. Therefore, from the definition of $[\![P[A]]\!]_M$, there exists an $h' \in [\![A]\!]$ such that:

$$s_{|o} \sim h' \;\wedge\; \prec_{\mathit{trans}(s_{|o})} \subseteq \prec_{h'} \tag{C1}$$

Since there are no overlapping operations on a given thread, the order of invocations and responses on a given thread in $\mathit{comp}(t^+)$ and s are the same.

$$\mathit{comp}(t^+)_{|o} \sim s_{|o} \tag{C2}$$

The relative ordering of invocations and effects in t' is the same as that in t. Hence, $\prec_{\mathit{trans}(\mathit{comp}(t^+)_{|o})} = \prec_{\mathit{trans}(t'_{|o})}$. Any trace formed from t' by moving effects earlier, will have less overlapping operations when *trans* is applied to it than t' when trans is applied to it. Therefore, we have:

$$\prec_{\mathit{trans}(\mathit{comp}(t^+)_{|o})} \subseteq \prec_{\mathit{trans}(s_{|o})} \tag{C3}$$

Hence, from (C1), (C2) and (C3), we have:

$$\mathit{comp}(t^+)_{|o} \sim h' \;\wedge\; \prec_{\mathit{trans}(\mathit{comp}(t^+)_{|o})} \subseteq \prec_{h'}$$ □

6 Chase-Lev Deque

In this section we consider the consequences of using our weakened definition of correctness with respect to a typical concurrent object: a version of the Chase-Lev work-stealing deque (double-ended queue) [5] developed specifically for ARM [17]. The code in Figure 1 corresponds to a version used in [6] which for simplicity eliminates returns from within a branch, and assumes the elements of the deque are integers. It also uses the fixes suggested in [6] for errors regarding the placement of control fences[4] present in [17].

```
put(v)
  int t;
  t=tail;
  tasks[t mod L]=v;
  fence;
  tail=t+1;
  return;
```

```
take
  int h,t,task;
  t=tail-1;
  tail=t;
  fence;
  h=head;
  if (h <= t)
    task=tasks[t mod L];
    if (h=t)
      if !CAS(head, h, h + 1) then
        task=empty;
      tail=tail+1;
  else
    task=empty;
    tail=tail+1;
  return task;
```

```
steal
  int h,t,task;
  h=head;
  fence;
  t=tail;
  if (h < t)
    cfence;
    task=tasks[h mod L];
    if !CAS(head, h, h+1)
      task=fail;
  else
    task=empty;
  return task;
```

Fig. 1. A version of Lê et al.'s work-stealing deque algorithm for ARM [17]

The deque is implemented as a circular array of size L with a head and tail pointer. Elements may be *put* on or *taken* from the tail by a worker thread, and additionally, other (thief) threads may *steal* an element from the head of the deque (in order to balance system workload). Since the put and take operations are executed by a single thread, there is no interference between these two operations.

The full details of the implementation can be found in [17, 6]. What is interesting for us is that the put operation has as assignment to tail which is unfenced, and hence may be delayed from the point of view of the thief threads.

A possible history of the implementation on ARM is:

$$\langle \mathit{inv}(\mathsf{n.put},\mathsf{v}), \mathit{res}(\mathsf{n.put},\bot), \mathit{inv}((\mathsf{m.steal}),\bot), \mathit{res}((\mathsf{m.steal}),\mathsf{empty})\rangle$$

This occurs when the effect of the put operation is delayed until after the steal operation has occurred. Specifically, tail is not updated until after the steal operation and hence h < t in steal is false.

[4] A control fence (`ctrl_isync` in ARM and denoted cfence in Figure 1) ensures that all branch instructions occurring before it take effect before any *loads*, i.e., reads of global variables, occurring after it.

This behaviour would not be available in a specification of the work-stealing deque. Hence, the implementation is not linearizable. To make it linearizable we would need to add a fence to the put operation, reducing efficiency. The behaviour is, however, weakly linearizable. Therefore, whether the implementation is considered correct depends on which of the two notions of correctness we assume.

There is an obvious trade-off. Abstracting from weak memory effects at the specification level simplifies high-level reasoning: a steal operation following a put may not return empty. If weak memory effects are included at the specification level, any reasoning involves understanding the details of the processor implementation: in this case, understanding that the steal may return empty. While this is not too much of a burden for this example, it may become so for more complex code.

However, abstracting from weak memory effects at the specification level disallows certain efficient implementations such as that in Figure 1, as fences are required for many operations.

Ultimately, the choice will reside with the developer of the program; specifically how confident they are with interpreting the behaviour of their program using the concurrent object under the weak memory models they wish it to run on. We envisage a library with a range of implementations for particular concurrent objects: one for the programmer who has limited knowledge of weak memory models, and other, more optimised, implementations aimed at programmers with a deeper understanding of particular weak memory models.

7 Conclusion

In this paper we have introduced a notion of object correctness which allows weak memory model effects at the specification level, and provided a weakening of linearizability which is sound and complete with respect to it. We have compared this correctness notion with a perhaps more standard one which abstracts from weak memory model effects at the specification level. While abstracting from weak memory model effects simplifies high-level reasoning, it also limits what is allowed in implementations regarded as being correct. In particular, it can disallow many implementations which limit the use of fences to obtain efficiency.

As not all programmers will be confident with interpreting the use of concurrent objects in the presence of weak memory models, it seems reasonable to expect libraries to have implementations satisfying both correctness notions for a given object specification.

Acknowledgement Thanks to Kirsten Winter for fruitful discussions on this topic. This work was supported by Australian Research Council Discovery Grant DP160102457.

References

1. R.-J.R. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 67–93. Springer, 1990.
2. R.-J.R. Back and J. von Wright. Trace refinement of action systems. In *CONCUR '94*, volume 836 of *LNCS*, pages 367–384. Springer, 1994.
3. M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66. ACM, 2011.
4. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In H. Seidl, editor, *ESOP 2012*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.
5. D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA'05*, pages 21–28. ACM Press, 2005.
6. R.J. Colvin and G. Smith. A wide-spectrum language for verification of programs on weak memory models. In K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, editors, *FM 2018*, volume 10951 of *LNCS*, pages 240–257. Springer, 2018.
7. J. Derrick and G. Smith. A framework for correctness criteria on weak memory models. In N. Bjørner and F.S. de Boer, editors, *FM 2015*, volume 9109 of *LNCS*, pages 178–194. Springer, 2015.
8. J. Derrick, G. Smith, and B. Dongol. Verifying linearizability on TSO architectures. In E. Albert and E. Sekerinski, editors, *iFM 2014*, volume 8739 of *LNCS*, pages 341–356. Springer, 2014.
9. S. Doherty and J. Derrick. Linearizability and causality. In *SEFM 2016*, volume 9763 of *LNCS*, pages 45–60. Springer, 2016.
10. B. Dongol and L. Groves. Contextual trace refinement for concurrent objects: Safety and progress. In K. Ogata, M. Lawford, and S. Liu, editors, *ICFEM 2016*, pages 261–278. Springer, 2016.
11. B. Dongol, R. Jagadeesan, J. Riely, and A. Armstrong. On abstraction and compositionality for weak-memory linearisability. In I. Dillig and J. Palsberg, editors, *VMCAI'18*, volume 10747 of *LNCS*, pages 183–204. Springer, 2018.
12. I. Filipović, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379 – 4398, 2010.
13. S. Flur, K.E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In R. Bodik and R. Majumdar, editors, *POPL 2016*, pages 608–621. ACM, 2016.
14. A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In M. Aguilera, editor, *DISC 2012*, volume 7611 of *LNCS*, pages 31–45. Springer, 2012.
15. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.
16. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
17. N.M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *PPoPP '13*, pages 69–80. ACM, 2013.
18. M. Moir and N. Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47:1–47:30, 2004.
19. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs 2009*, volume 5674 of *LNCS*, pages 391–407. Springer, 2009.

20. C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, 2018.
21. A. Raad, M. Doko, L. Rožić, O. Lahav, and V. Vafeiadis. On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.*, 3(POPL):68:1–68:31, 2019.
22. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. *SIGPLAN Not.*, 46(6):175–186, 2011.
23. G. Smith, K. Winter, and R.J. Colvin. Correctness of concurrent objects under weak memory models. In J. Derrick, B. Dongol, and S. Reeves, editors, *Refine 2018*, volume 282 of *EPTCS*, pages 53–67. Open Publishing Association, 2018.
24. G. Smith, K. Winter, and R.J. Colvin. A sound and complete definition of linearizability on weak memory models. *CoRR*, abs/1802.04954v2, 2019.
25. O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC2013*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.