Abducing memory barriers

Richard Bornat[1,2,3] and Mike Dodds[1]
[1]Computer Laboratory, University of Cambridge, Cambridge, UK
[2]School of Electrical Engineering and Computer Science, Queen Mary, University of London, UK
[3]School of Engineering and Information Sciences, Middlesex University, London, UK
rb501@cam.ac.uk, richard@eecs.qmul.ac.uk, R.Bornat@mdx.ac.uk, Michael.Dodds@cl.cam.ac.uk

February 28, 2012

Abstract

A concurrent work-stealing algorithm is analysed to discover necessary positions for memory barriers. The analysis is targeted at the Power processor, and takes into account both instruction-execution reordering and delayed writes. Formal verifications of the effectiveness of the barriers under instruction-execution reordering are developed, and a semi-formal argument (using formal assertions with semantic reasoning) verifies barriers which deal with inter-thread propagation effects. The published algorithm had a number of suggested barriers to guard against reordering; it is shown that it needs one more barrier for that purpose, and two more to deal with propagation effects. The effect of missing barriers is described.

Programming is in a weird place. Our programs are mostly sequential but hardware is mostly parallel, even within a single processor. Caches, pipelines, weak memory and speculative execution conspire to rearrange execution in surprising ways. A single isolated program keeps its sequential meaning, but shared-variable concurrency becomes rather tricky. Special barrier instructions are needed to enforce some discipline, and it's hard to understand when and where they are needed. There's been a lot of study of the hardware [3, 26, 25] and some use of model-checking to verify barrier placement [13, 17, 5]; this paper looks at the problem from a programmer's point of view and uses verification.

Concurrent algorithms are often published as if they were to be executed on a sequentially-consistent machine and the placement of barriers is treated as if it were merely an implementation detail. This won't do: barriers are an essential part of an algorithm designed to run fast and concurrently. In figure 1, taken from [20], the authors have indicated where the barriers ought to go (though in private correspondence [30] they made it clear that these are only suggestions, not tested on any particular current processor; their experiments were on an architecture which doesn't reorder execution).

In this paper we discover the barriers that are needed by logical argument. We take each procedure and draw a dependency graph of its commands, showing which must follow what, even if the machine reorders execution. Then, by trying to make a formal proof that the graph preserves some global invariants, we find places where the proof won't go through. We fix those problems by inserting an extra ordering, provided by a barrier instruction. This is a sort of *abduction* [23] in that it involves guessing and verifying. We find all the suggested barriers and one extra. Then by considering orderings between different procedures we find two more barriers. Missing barriers cause bugs, and we describe the bugs that result.

Our technique works well for our chosen example. That is satisfying, but it is work in progress: our dependency graphs aren't formally tied to the semantics of the machine; when considering memory propagation effects we reason semi-formally; and it would be easy to guess too many barriers. For all that, we do find barriers, and it ought to be possible in the near future to dot all the logical i's and cross the formal t's.

Related work is either on automatically placing barriers [19, 2, 1] or removing redundant ones [29]. Most of it assumes TSO, the execution model of the Intel 86; for weaker architectures, such as IBM Power, the placement problem seems to be undecidable [7, 6]. It remains to be seen if automatic techniques can discover all and only the barriers needed for FIFO IWSQ on a weak architecture.

```
void put(Task tv) {
   Order write at 4 before write at 5
1: h := Hd;
2: t := Tl;
3: if t = h+Ta.size then { expand(); goto 1 };
4: Ta.array[t%Ta.size] := tv;
5: Tl := t+1
}

Task take() {
local Task tv;
1: h := Hd;
2: t := Tl;
3: if h=t then return EMPTY;
4: tv := Ta.array[h%Ta.size];
5: Hd := h+1;
6: return tv
}

void expand() {
   Order writes in 2 and 4 before write in 5
   Order write in 5 before write in put:5
1: s := Ta.size;
2: a' := new SizeTaskArray(2*s);
3: for i = Hd:Tl-1,
4: a'.array[i%a'.size] := Ta.array[i%Ta.size];
5: Ta := a'
}

Task steal() {
   Order read in 1 before read in 2
   Order read in 1 before read in 4
   Order read in 5 before CAS in 6
local Task tv;
1: h := Hd;
2: t := Tl;
3: if h=t then return EMPTY;
4: a := Ta;
5: tv := a.array[h%a.size];
6: if CAS(&Hd,h,h+1) then goto 1;
7: return tv;
}
```

Figure 1: Michael et al.'s FIFO IWSQ (from [20] with abbreviated names)

1 An example algorithm

Work stealing [4, 14] is a technique for scheduling tasks on multiprocessor machines. Each scheduler thread manages its own queue of tasks, but when the queue is empty it may try to steal a task from another. Michael et al. [20] describe an 'idempotent work stealing' algorithm (henceforth IWSQ): we analyse their

FIFO variant (shown in figure 1, with shortened names). Each queue is in its own global array, with its own Hd and Tl indices. Indices increase without limit and queue addressing is modulo the size of the array, giving wrap-around (circularity) within the array. Only the owner thread may write to the queue: it put()s at the tail and take()s from the head. If the queue gets too big the owner may expand() the array. A thread short on tasks may attempt to steal() from the head of another's queue.

IWSQ is designed to be more efficient than its predecessors by using fewer barriers in its common path (put() and take() by a queue owner), but there is a price to pay: because take() doesn't use a CAS,[1] a task may be stolen at the same time as the owner takes it from the queue. Tasks must therefore be idempotent – i.e. it mustn't matter that they are executed/used more than once. A practical example is packet routing. This is intended to be a practical algorithm; it is clearly worth ensuring that it is correct.

2 Relaxed execution rules and TIoSE

There has been much recent work on verifying concurrent algorithms [10, 11, 22, 16, 15, 9, 12] but it is all vulnerable to the objection that it doesn't take into account *relaxed memory* effects, which arise from interaction between in-thread execution reordering and inter-thread communication effects. Different threads[2] in a concurrent program running on a relaxed machine can have quite different views of what the values of shared variables are. The PowerPC is such a machine, and its semantics are mind-splittingly intricate [25]. But it has been explained to us, and we have found a way to use verification to deal with it.

Execution ordering first. Hardware designers are ambitious but they aren't mad: their machines preserve the illusion of sequential execution (henceforth TIoSE), within a single thread, that everything happens in the original *program order*. That requires some obvious constraints:

d: if one instruction computes a value which is used by another, then they must stay in program order;
c: instructions which have an external effect (e.g. writes to memory) and which follow a conditional branch must stay in program order with the branch;
m: instructions which access the same memory location must stay in program order.

These constraints give designers lots of wriggle room: there's no constraint, for example, that reads or writes to different locations have to happen in program order. Most strikingly the hardware can *speculate* execution of instructions out of order, provided that speculation is abandoned if a later conditional branch decides against those executions, and that results are only visible to instructions which follow in program order. But all that is hidden from the programmer under TIoSE.

Then communication effects. Communication takes place through the memory; caches take much of the strain so far as reads are concerned, but writes are still a problem. In the model of [25] a write happens in stages: first, the instruction is executed (possibly even speculated) and the value it writes is immediately available to reads in the same thread; then it is *committed* and the value is *propagated* to other threads. Memory organisation on Power is so weird that the model abstracts from memory and cache and describes reads taking values from particular writes. There's a global *coherence order* of writes to each individual location and threads read in coherence order,[3] but no necessary order on writes otherwise. In particular, writes to different locations may arrive at different threads in different orders. But the main thing is that a write is visible at one instant in the writing thread and only much later in other threads.

[1] CAS: compare and swap, an atomic action which reads a location, compares it with an expected value, and writes a new value if it read what was expected. Some variants return the value found in the location (hence 'swap'); in figure 1 CAS returns a Boolean to signal success or failure.

[2] Throughout this paper 'thread' means hardware thread: effectively a single program running on a single processor.

[3] An uncommitted write is necessarily later in coherence order than any committed write. So it's safe to read locally from your own writes.

TIoSE sorts out what happens in a single isolated thread, but execution reordering and delayed committal/propagation complicate communication. Barriers force some memory events to happen before others. A sync, the most expensive, forces all the reads and writes before it to finish – be committed and propagated – before any of the reads and writes after it can even start. In fact all the committed writes in other threads that the barring thread has read from must also have finished: sync is *cumulative*. The less expensive lwsync separates reads and writes before from writes afterwards (i.e. it doesn't separate writes from reads) and is also cumulative. Cheaper still is ctrl +isync which separates a single read from following reads and writes, and there are other, less powerful, barriers which we ignore.

3 Invariants and actions

In verifying concurrent programs, invariants are priceless, describing relationships between shared variables in an imaginary snapshot of the machine state. In relaxed memory they constrain the states of each thread: at every instant, taking into account the values of all the writes that have been propagated to it, the invariant must appear to hold. (2), for example, prescribes that we must never glimpse $Hd > Tl$ in any thread.

The logic we use is RGSep [28, 27], a development of separation logic [24, 21] and rely-guarantee [18]. In separation logic '$\mapsto$' is *points to*: so, for example, $Ta \mapsto Ar, Sz$ asserts that the variable Ta points to a two-word record in the heap; '$\star$' is *and separately*, which allows us to talk about different bits of the heap; '$\circledast$' is repeated '$\star$', a separated $\forall$; '$\wedge$' means *and at the same time* (no separation); and 'true' means as much space as you need to fill in the gaps.

(1) says that Ta points to a record which contains a pointer to an array and its size; the array is a record of separate tasks; within that record is the queue and possibly ('$\star\,\mathrm{true}$') some unused elements. For simplicity we have supposed single-word task values. To describe the queue we imagine the existence of a task sequence $\boldsymbol{Ts}$ which has length Tl, is indexed from 0 to $Tl-1$, and only changes by appending a value; the queue proper is $\boldsymbol{Ts}[Hd..Tl-1]$. (2) says that Hd never overtakes Tl; (3) that the array is never overfilled; (4) that Tl never decreases.

$$\exists Ar, Sz\,\big(\,Ta \mapsto Ar, Sz \star \big(\circledast_{i\in 0:Sz-1}(Ar+i \mapsto \_) \wedge (\circledast_{q\in Hd:Tl-1}(Ar + q\%Sz \mapsto \boldsymbol{Ts}_q) \star \mathrm{true})\big)\big) \quad (1)$$

$$Hd \leq Tl \quad (2)$$

$$\exists Ar, Sz(\,Ta \mapsto Ar, Sz \wedge Hd + Sz \geq Tl) \quad (3)$$

$$Tl = T \Rightarrow \Box(Tl \geq T) \quad (4)$$

In RGSep we impose a division between *local* memory, which is private to a thread and can't be seen or touched by other threads, and *shared memory*, which all threads can read or write. Communication is by writing to shared memory, causing what rely-guarantee calls *interference*. An *action*, the unit of interference, describes the effect of a single-word write.[4] We write $context : pre \rightsquigarrow post$ to describe the part of shared memory that doesn't change but is relevant to the action, and the pre- and post-values of the part of memory that does change.

The owner alone, with put(), can write to the array and alter Tl. To update the queue it's necessary to do both, but there are no locks in IWSQ so there must be two separate actions: (5) changes the array; (6) changes Tl and simultaneously updates the imaginary task sequence $\boldsymbol{Ts}$.

$$Ta \mapsto Ar, Sz \wedge Tl \leq j < Hd + Sz : Ar + j\%Sz \mapsto V \rightsquigarrow Ar + j\%Sz \mapsto V' \quad (5)$$

$$Ta \mapsto Ar, Sz \star Ar + Tl\%Sz \mapsto V : \begin{array}{l} Tl = T \wedge \boldsymbol{Ts} = TS \rightsquigarrow \\ Tl = T + 1 \wedge \boldsymbol{Ts} = TS.V \end{array} \quad (6)$$

[4] Single-word memory accesses are guaranteed to retrieve or deposit a complete value; multi-word reads and writes are piecewise. The actual operation of a single-word read or write is long-winded, but no thread ever retrieves a half-written single-word value. We would call them 'atomic' reads and writes, but that would be confusing in a relaxed-memory setting.

The owner, with take(), or a thief, with steal(), can increase Hd in (7). The owner can *decrease* Hd in (8), if a take() overlaps more than one steal(): the array element containing the restored member of the queue will have been unmodified during its absence, and the action won't make the array full.

$$Hd = H \rightsquigarrow Hd = H + 1 \quad (7)$$

$$Ta \mapsto Ar, Sz \wedge \big((Ar + H\%Sz \mapsto \boldsymbol{Ts}_H) \,\mathsf{S}\, Hd = H \big) \wedge H + Sz > Tl : Hd = H + 1 \rightsquigarrow Hd = H \quad (8)$$

In (9) the owner, with expand(), can switch task arrays by writing to Ta, but invariant (1) demands that the new array must already contain the queue proper, and Hd can't subsequently decrease to take in values which are not in both arrays at the instant of expansion.

$$Ta = T \wedge Hd = H \rightsquigarrow Ta = T' \wedge \Box(Hd \geq H) \quad (9)$$

Along with interference comes the notion of *stability*. An assertion in a thread is stable if other threads, interfere how they may, can't make it false. So the instruction <t:=Tl>, for example, will instantaneously produce the postcondition $t = Tl$; interference by action (6) can increase Tl and destroy the postcondition but the alternative $t \leq Tl$ is stable since Tl can't decrease and t is a local register.

4 Instruction-execution ordering

Figure 1 includes constraints on instruction ordering, justified in [20] by describing bad things that could happen if the orderings weren't obeyed. Programmers don't work like that: we shall abduce the barriers by trying to make an algorithm that produces good results.

4.1 Orderings in steal()

We begin with steal() since it has three ordering constraints in figure 1, making it the most intriguing of the four procedures. We recast it in figure 2(a) so that each instruction makes at most one access to memory: suffixes on line numbers (5a, 5b etc.) show where we have split a single line of the original; lower-case names denote registers; angle brackets show single-word memory operations. Tasks need not be single-word in IWSQ, so line 5d is not shown as a single-word read, but the address calculation assumes an array of single-word tasks; to allow for multi-word task reads wouldn't add much illumination and would considerably complicate assertions. There are no writes other than the one hidden in CAS. There are five jumps: two conditional branches, one goto and two returns.

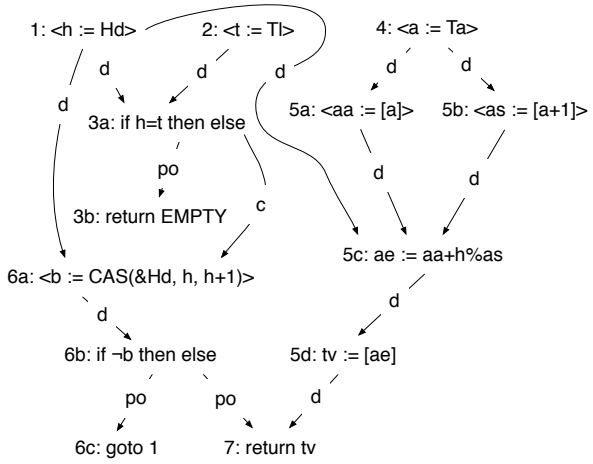
Figure 2(b) shows the important TIoSE dependencies of figure 2(a), using 'd' where one instruction produces a result used by another, 'c' where the effect of an instruction must follow a conditional branch, and 'po' where program order applies but we haven't considered what happens after a goto or return. There's something strange about the 'c' dependency: strictly it applies to the *committal* of a write, which must follow the branch, and the *execution* of the write can be speculated (though not in the case of a CAS). But we gloss over that distinction because in any case it marks a *proof dependency*: in reasoning about the effect of a command we can rely on the postconditions of the things it depends upon in either the 'd' or the 'c' sense.

It's immediately clear that lines 4, 5a and 5b float free of anything to do with Hd or Tl, and that the CAS is independent of anything to do with reading the array. Lines 1 and 2 are independent of each other, but line 1 has to precede quite a lot of others. Note that there isn't a barrier automatically associated with a CAS on Power: it is implemented as a special read followed by a special write; it is not speculated before a preceding branch; and the write is committed immediately if it succeeds (and it can fail spuriously, which messes up correctness arguments, but we shan't go there.)

Reasoning starts with the placing of assertions on dependencies, describing what you would see if you could take in at an instant all the thread's local memory and the values of all the writes that have been propagated to it. First consider line 7: we'd like to know that it returns a value that was, at some point

```
Task steal () {
    local Task tv;
1:  <h := Hd>;
2:  <t := Tl>;
3a: if h=t then {
3b:     return EMPTY
3c: };
4: <a := Ta>;
5a: <aa := [a]>;          // aa := a.array
5b: <as := [a+1]>;        // as := a.size
5c: ae := aa+h%as;        // ae := &(a.array[h%a.size])
5d: tv := [ae];           // tv := a.array[h%a.size]
6a: <b := CAS(&Hd, h, h+1)>;
6b: if !b then {
6c:     goto 1
6d: };
7:  return tv
}
```

(a) code with single-memory-access commands



(b) dependency graph

Figure 2: FIFO steal recast

during the steal () operation, the head value of the queue. Something like $tv = \boldsymbol{Ts}_h \wedge \Diamond(Hd = h)$ (where $\Diamond$ means 'at some instant since the procedure execution began'). The second conjunct is immediate, because $h = Hd$ holds immediately after line 1, which is an execution predecessor of line 7, and the first conjunct had surely better be a postcondition of line 5d. To consider line 5d we can derive from invariant (1)

$$\exists Ar, Sz, V(Ta \mapsto Ar, Sz \star Ar + h\%Sz \mapsto V \wedge (Hd \leq h < Tl) \Rightarrow V = \boldsymbol{Ts}_h) \tag{10}$$

– provided h is in the queue-index range, $\boldsymbol{Ts}_h$ is available in the array. If the procedure's instructions were executed sequentially, $h \neq t$ at line 3a, together with invariants (2) and (4), would ensure $h < Tl$ from line 4 onwards and then a successful CAS would tell us that $h = Hd < Tl$ at line 6a. From (10), plus invariant (7) for the tricky case when Hd first increases and then decreases between lines 1 and 6a, we could conclude that line 5d actually reads $\boldsymbol{Ts}_h$. But very little of that sequential ordering survives in figure 2(b) , and our job is to abduce how much of it we need to enforce.

After <h:=Hd> in the dependency graph we know instantaneously that $h = Hd \leq Tl$. Stably (under interference from other threads – see §3), because Hd can either increase or decrease we don't know at all the relationship between h and Hd. But Tl can only increase, so $h \leq Tl$ is stable. Similarly, after <t:=Tl> we know instantaneously $t = Tl$ and stably $t \leq Tl$. At line 3a, however, $(h \leq Tl \wedge t \leq Tl \wedge h \neq t) \not\Rightarrow h < Tl$. But if we abduce a barrier to enforce program order between lines 1 and 2 then we know instantaneously after line 2 that $h \leq t = Tl$, and stably $h \leq t \leq Tl$; then at line 3a $(h \leq t \leq Tl \wedge h \neq t) \Rightarrow h < Tl$. (In the other case when $h = t$ we know that at line 2 we had $h = t = Tl$ and then, since Tl can't decrease, we had $h = Hd = Tl$ at line 1, and that instant the queue was empty.)

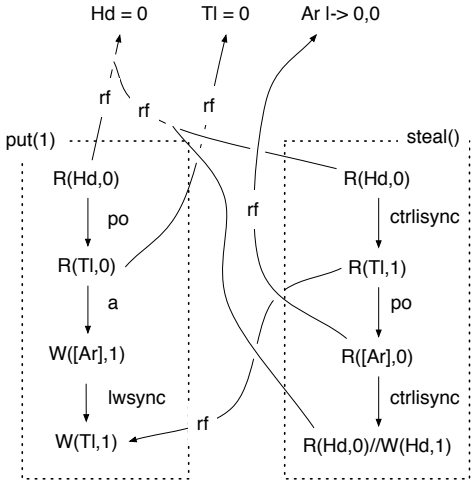
To be sure from (10) that we are reading $\boldsymbol{Ts}_h$ on line 5d we need two more barriers: one between lines 5d and 6a to ensure that the task is read before the CAS; and one between lines 2 and 5d, after we read t, to ensure that it is read when $h < t$.

The code with its barriers is in figure 3(a). A formal treatment of its dependency graph is in figure 4: the diagram is just a flattened dependency graph with ribbons instead of lines, and assertions written inside the ribbons.[5] In RGSep notation we draw boxes round assertions about shared memory, and leave local

[5] This style of diagram derives from 'ribbon proofs' [8, 31] but it is a distinct variant, designed to suit our dependency graphs.

```
Task steal () {
1:  <h := Hd>;
    --- Line 1 read/any read barrier ( ctrl +isync) ---
2:  <t := Tl>;
3a: if h=t then {
3b:     return EMPTY
3c: };
4:  <a := Ta>;
5a: <aa := [a]>;          // aa := a.array
5b: <as := [a+1]>;        // as := a.size
5c: ae := aa+h%as;        // ae := &(a.array[h%a.size])
    --- line 2 read/any read barrier ( ctrl +isync) ---
5d: tv := [ae];           // tv := a.array[h%a.size]
    --- line 5d read/any write barrier ( ctrl +isync) ---
6a: <b := CAS(&Hd, h, h+1)>;
6b: if !b then {
6c:     goto 1
6d: };
7:  return tv;
}
```

(a) code



(b) the empty steal bug without the line 2 / line 5d barrier

Figure 3: FIFO steal with barriers

memory outside the box. The commands and barriers are flattened out to fit their particular dependency ribbons (for reasons of space the line 2 / line 5d barrier is shown as a line of dashes). The postcondition of each command describes its instantaneous result, and '$\Rightarrow$ (stability)' shows the effects of interference.

The proof begins with the assertion that there is always an array element indexed by Hd which contains some value V and that $V = \boldsymbol{Ts}_{Hd}$ provided the queue isn't empty. <h:=Hd> picks up the value of Hd but that's unstable; what we know stably is that if $h = Hd < Tl$ then we can still read $\boldsymbol{Ts}_{Hd}$. That ribbon splits in two: the left one deals with h, t and Tl while the right one deals with the array. Once we read Ta we have to allow for interference from the owner, which might switch arrays with (9), so we could lose the association $Ta = A$; but if we do then we can be sure (a) that $Hd < Tl$ at the switch (the owner only expands a full array) and therefore (b) that $\boldsymbol{Ts}_{Hd}$ was definitely in the array before the switch, and therefore (c) that it's in the array rooted at A, which was the array before the switch. Then we read the array pointer and size, in two separate ribbons, while a third carries the facts about the array element we're hoping to read. Those three ribbons recombine, and we compute the address of the array element. We're still not sure, at this point, that there is a queue element to be read: this side of the diagram has taken no account of <t:=Tl>. Then the line 2 / line 5d barrier comes into play, bringing together a ribbon which asserts $H \leq T \leq Tl$ and one which asserts that we have the address of element H (modulo Sz). The two ribbons are used by tv :=[ae], which reads $\boldsymbol{Ts}_H$ provided $H = Hd < Tl$. Then a successful CAS, plus the assurance $h = H < T$ from the 'else' branch of the conditional, guarantees that we have $\boldsymbol{Ts}_h$.

TIoSE only guarantees that everything will work within a single thread which suffers no interference: We had to consider what happens when other threads interfere by changing Hd, Tl or Ta and fend them off by inserting barriers. The barrier between line 1 and line 2 also divides line 1 and line 4, so we can be sure when we execute <a:=Ta> that $h = H$, because $h = Hd < Tl$ is what we need to be sure that $\boldsymbol{Ts}_h$ is in the array. The barrier between line 2 and line 5d was necessary so that we could assert $H \leq T$ at the instant of reading the task; the later assurance that $H < T$ seals the deal.[6]

What remains is to deal with one goto and two returns. It is important to recognise that ordering is of

[6] Why not put the barrier between line 3a and line 5d? It would simplify the proof, but we did it this way because it feels more elegant and allows more speculation.

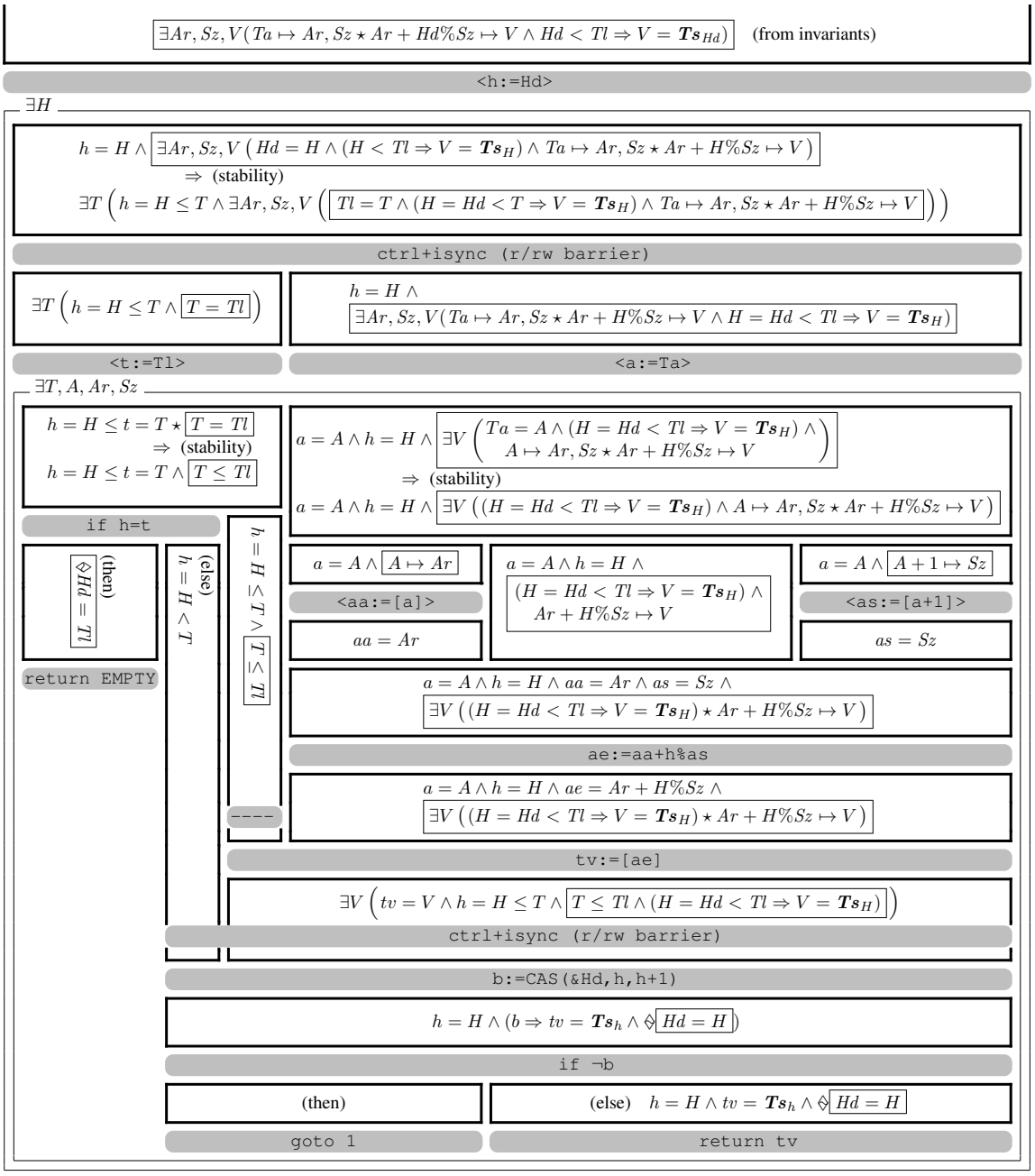


Figure 4: Formal proof of steal()

executions rather than instructions, so the goto doesn't simply loop back to the head of the diagram. What follows it is another instance of the whole diagram, and we have to be concerned about interactions between successive instances. Because of dependencies and the read/write barrier, only the CAS could migrate from the first into the second; because of the read/read and read/write barriers, only <h:=Hd> could migrate from the second into the first. Because they are accessing the same memory location, each stops the other going any further. We don't need any more barriers.

We expected to derive Michael et al.'s three barriers in steal() – read Hd before Tl; read Hd before

```
void put(Task tv) {
1:  <h := Hd>;
2:  <t := Tl>;
3a: <a := Ta>;
3b: <s := [a+1]>;
3c: if t = h+s then {
3d:     expand();
3e:     goto 1
3f: };
4a: <aa := [a]>;
4b: ae := aa+t%s;
4c: [ae] := tv;
--- write/write barrier (lwsync) ---
5a: <Tl := t+1>
5b: return
}
```

```
void expand() {
local Task tv;
1a: <a := Ta>;
1b: <aa := [a]>;
1c: <s := [a+1]>;
2a: a' := new SizedTaskArray(2*s);
2b: <aa' := [a']>;
2c: s' := 2*s;
3a: <i := Hd>;
3b: <t := Tl>;
3c: if i=t then {
3d:     goto 5a
3e: };
4a: ae := aa+i%s;
4b: tv := [ae];
4c: ae' := aa'+i%s';
4d: [ae'] := tv;
4e: i := i+1;
4f: goto 3c;    // od
5a: --- write/write barrier (lwsync) ---
5b: <Ta := a'>;
5c: return
}
```

```
Task take() {
local Task tv;
1: <h := Hd>;
2: <t := Tl>;
3a: if h=t then {
3b:     return EMPTY
3c: };
4a: <a := Ta>;
4b: <aa := [a]>;
4c: <s := [a+1]>;
4d: <ae := aa+h%s>;
4e: tv := [ae];
5: <Hd := h+1>;
6: return tv
}
```

Figure 5: put(), expand() and take() with barriers

Ta; read the task value before executing the CAS – but we found an extra one – read Tl before $[ae]$. Our abuction method, though, is very unlikely to be minimal: We can't say that because the invariants, pre- and post-conditions which we have chosen suggest particular barriers, those barriers are required; there might be a different, cheaper, set which would do the job. We can, however, show a bug without the barrier: with only program order between <t:=Tl> and <tv:=[ae]>[7] the reads can happen in any order, which means that the thief can read the task value before it reads Tl. A thread diagram in figure 3(b) shows the pattern of reads and writes, assuming initial values of 0 for index values and array elements and put(1) from the owner: despite the lwsync barrier in put (see §4.2), the thief reads an initial, invalid, task value from the array. Figure 4 shows that the extra barrier provably fixes that problem, preventing the bug.

In each case the argument which led to placing of a barrier was an *abduction*: there was something we needed to prove on a particular ribbon in the diagram, which we couldn't prove without connecting it to another ribbon using a barrier. We didn't have to say what nasty things could happen if the barrier was missing: nastiness arguments are particularly convoluted in the line 1 / line 4 ordering case, but by contrast it's clear when there is something we need for the proof which we can't guarantee without an ordering.

4.2 Orderings in put(), expand() and take()

There isn't room to show the analysis of put(), expand() and take() in as much detail as was given to steal(). The code with its barriers is in figure 5: We abduce the same orderings as in figure 1, produced by one write/write barrier (lwsync) in each of put() and expand() and no barriers at all in take(). In put() the barrier ensures that the write which puts the new task into the array must be finished before the index-update begins, which is necessary to ensure that action (6) preserves invariant (1); a concurrent programmer would know straight away that the barrier is needed, but the formal proof confirms that we need no more. The treatment of expand() is quite similar: its barrier ensures that the writes which create and initialise the new array are completed before its existence is publicised to the world by assignment to Ta. The proof of take() follows the TIoSE dependencies and, as in figure 1, it needs nothing more.

[7] On Power there is a control dependency between lines 2 and 5d, but in the case of a pair of reads that makes no difference.

5 Propagation effects

Interference between threads is caused by writes to shared locations. Threads must read in (global) coherence order from each location: a local uncommitted write is later in coherence order than any committed write since it hasn't been committed yet. So the hardware can read locally first from an uncommitted write and commit / propagate later. Different threads may therefore have different views of shared data.[8] Our reasoning so far has dealt with the local view, ensuring that invariants are preserved in each thread's view of shared data. This is valid when reasoning about the interference of other threads, which was what led to the three barriers in steal(). The barriers in put() and expand() are different: they were implicit in the specification, ordering actions to preserve invariants, but the specification was written that way because of shared data concerns: no thread should see the effect of a write to the array before it sees the effect of an update to Tl, for example. To argue that those barriers which preserve invariants in the view of the owner also preserve them in the views of thieves will require consideration of the way that writes are committed and propagated and the way that they interact with barriers. We shall see that the barriers we have placed do have effects that we require, but that we need more barriers still to make the algorithm work.

In this paper a *stable* assertion is a predicate on a thread's view of shared data whose truth can't be undermined by the interference of other threads – i.e. by the arrival of writes from other threads. In RGSep [28, 27] stable assertions are global: if $\boxed{P}$ holds in one thread and is stable, then $\boxed{P}$ holds in every thread. That's not so in a relaxed-memory setting. But some assertions are *universal* and hold in every thread: in our example all threads must agree about the value of elements of the task sequence, though they may disagree about its length. Both sync and lwsync barriers convert shared assertions to universal assertions because they are *cumulative*: every update that a thread makes, or reads from, before such a barrier will be committed and propagated to other threads before the barrier completes. If $\boxed{P}$ is stable before a cumulative barrier, based on the updates the barring thread has made or seen, then after the barrier all threads will have seen those updates and $\boxed{P}$ will be valid and stable in their views. We have to be careful about what it means for a barrier to 'complete': a sync completes before the next instruction execution, an lwsync before the next write.

This reasoning explains how the *message passing* paradigm of Power programming, using an lwsync between writing a message value and setting a flag or an index to publicise the message, works in logical terms. The lwsync in put() converts the stable assertion

$$\boxed{Ta = A \wedge Tl = T \wedge A \mapsto Ar, Sz \star \mathrm{true} \star Ar + T\%Sz \mapsto V} \quad (11)$$

into a universal assertion, and we can be sure that the update <Tl:=t+1>, given $t = T$, will effect action (6) in the view of *every* thread; it also ensures that it won't violate invariant (3) in any other thread's view. Similar reasoning shows that <Ta:=a'> in put() effects (9) everywhere and preserves (1).

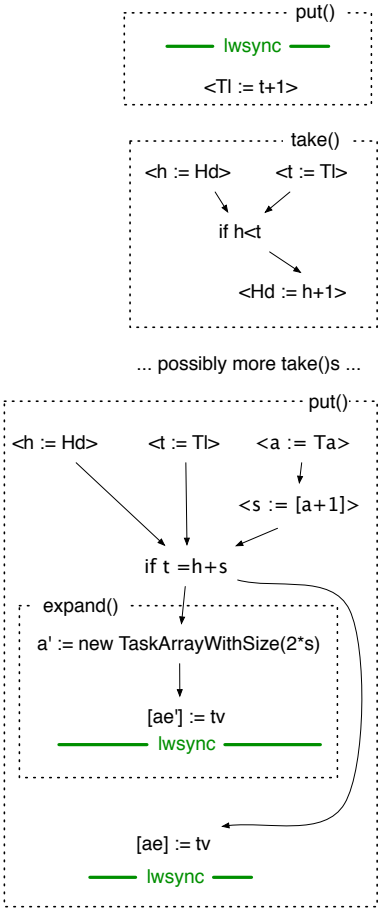


Figure 6: selected memory events in put(); take(); ...; put()

We have to be careful. lwsync provides a universal assertion which holds until the next write of the barring thread, which may undermine it. If <Tl:=t+1> is the next write

[8] RGSep [28, 27] deals with shared memory, but in the model of [25] the memory is abstracted away. There is, however, shared data.

propagated from the owner to a potential thief after the lwsync in put() then it will be seen in a state in which (11) holds; but because of out-of-order execution and committal / propagation effects it might not be propagated first. Suppose the owner executes put(); take(); ...; put() – at least one take(), possibly more, between two put()s, the second of which may invoke expand(), as in figure 6. None of the writes up to the next lwsync undermine (11): [ae]:=tv in a subsequent put() writes to a different location; the writes in expand() are to a different array; <Hd:=h+1> in take() is to a variable not mentioned. So whenever <Tl:=t+1> is committed and propagated, which must be before the end of the next lwsync, it will be received in a state described by (11), and similarly it won't violate (3).

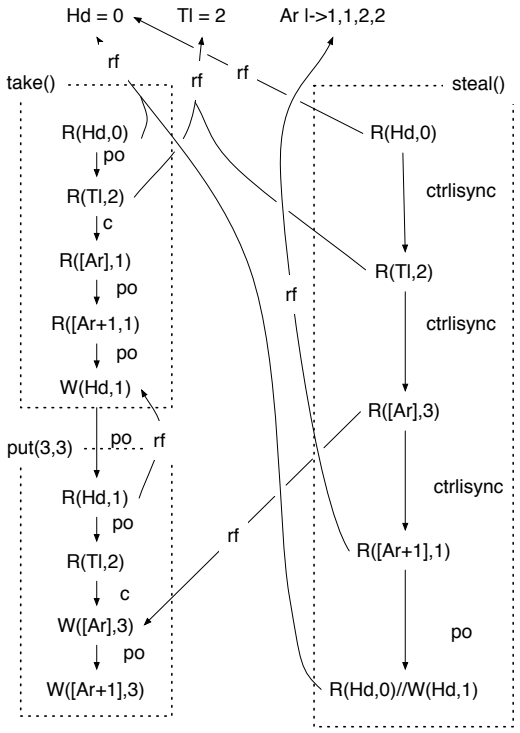


Figure 7: the overwriting bug

The case of <Ta:=a'> in expand() is similar, but simpler. The instructions which allocate and initialise the new array come before an lwsync; the whole of the new array, together with any uncommitted updates to Hd and Tl which make up the owner's view, will be propagated to all threads before the assignment <Ta:=a'> switches arrays; and the switching assignment itself, together with the [ae]:=tv which is part of the enclosing put(), will be propagated before <Tl:=t+1>; and so on round again.

So much for the lwsyncs; but we haven't finished. Between the lwsync of the first put() and the first lwsync of the last there are various writes but no barriers. Because committal and propagation is only constrained by barriers, those writes, in the view of other threads, are unordered, apart from updates to Hd in the take()s, which will be seen in program order. The write caused by <Tl:=t+1> can be propagated safely at any time, and likewise the writes which initialise and populate the new array in expand(); it remains to consider the writes caused by [ae]:=tv in put() when there is no expand() and the writes caused by <Hd:=h+1> in take(). Each of those writes are guarded by a conditional branch.

[ae]:=tv is only committed if $h + s > t$, which implies

$$Hd + [Ta + 1] > Tl \tag{12}$$

in the owner's view – i.e. the owner can see an empty slot in the array. But do other threads see the same slot as empty? (12) is a universal assertion after the first lwsync in our example execution, but <Tl:=t+1> undermines it. To ensure that [ae]:=tv is well-received by thieves requires

$$Hd + [Ta + 1] > Tl \Rightarrow \mathbb{U}(Hd + [Ta + 1] > Tl) \tag{13}$$

where $(U)(P)$ means 'P holds universally'. This is a universal assertion after the first lwsync in figure 6, because (12) is, and it isn't undermined by <Tl:=t+1>, which can only make the antecedent false. It is, however, undermined by <Hd:=h+1>, which may make the antecedent true without affecting the consequent. That can be remedied by inserting a conditional barrier: after <Hd:=h+1> in take() add "if h+s=t then lwsync".

Without that barrier the update to Hd can happen after the write to $[ae]$, and a thief can be stealing from a particular array element at the same instant that an owner is writing. Given an array of single-word tasks their writes will be serialised, and a thief could retrieve a task that isn't yet part of the queue, reading from the tail before Tl is updated. That might not matter, perhaps: but with multi-word task values it can retrieve part of the newly-written task and part of the old: see figure 7.

The other guarded update is <Hd:=h+1> in take(): it is only executed if $h < Tl$, which is required to ensure invariant (2) locally. Immediately after the first lwsync in figure 6 we have $H = Hd < Tl \Rightarrow \mathbb{U}(H+1 \leq Tl)$.

It's undermined by <Tl:=t+1> which may make the antecedent true but does not affect the consequent. This can be fixed with a conditional barrier if h=t then lwsync at the end of put(). Without that barrier a thief can go wrong, seeing $Hd > Tl$ and stealing an uninitialised value: see figure 8. But in this case we can do better: if we make (2) a local invariant in the owner only, and make thieves check $h \geq t$ instead of $h = t$, the bug is more cheaply solved (find that, automated methods!).

6 Coda

Abduction on dependency graphs found all of the orderings from [20], plus one extra. We have confidence that there are no more dependency orderings to be found, because of the formal proofs. Those proofs are not yet formally linked to a model of the Power processor, but that ought to be possible, and it should also be possible to mechanise the proofs and perhaps even the abduction.

A semi-formal treatment of propagation effects, using invariants but reasoning semantically, found two additional necessary orderings. We have some confidence that we have found all of these orderings too, by showing that unguarded actions are always well-received in other threads, and guarded actions sometimes need protection. We're fairly confident that there aren't more propagation bugs to be discovered in this algorithm, but it would be nice if this treatment could be made completely formal and eventually automated.

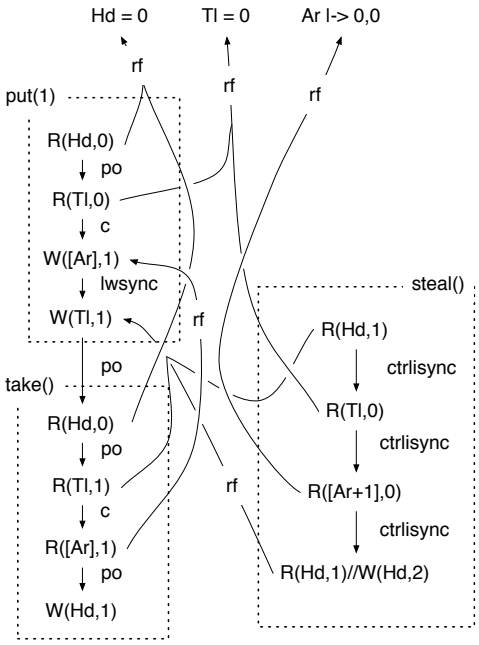


Figure 8: the overwriting bug

This paper reports work in progress. It may be that there are better ways of reasoning about the placing of barriers, but the history of our subject suggests that formal reasoning of some sort will be the way forward. It remains to be seen, when IWSQ is tested on Power, whether experiment finds the bugs that we have identified, or it finds some that we have missed, and whether it suggests a cheaper set of adequate barriers.

Acknowledgements

We owe to Mark Batty, Peter Sewell, Scott Owens, Susmit Sarkar and Jade Alglave all the knowledge we have of the arcane ways that Power processors treat our programs. Without their patient instruction we could have done nothing, but any mistakes and misunderstandings are our own. Mark, Peter, Scott and Susmit recognised the overwriting bug was a bug when it looked like a failure of analysis. Dino Distefano pointed out that we were abducing, not deducing. Ernie Cohen made us articulate what we meant by an invariant. Martin Vechev and Maged Michael's suggested orderings were a seminal inspiration, and they were kind enough to confirm that the 'bugs' we found can really happen (though we didn't ask them to confirm our suggested fixes). John Wickerson held our hands while we learnt to use his Latex/OCaml mechanism for drawing ribbon proofs. Without Peter O'Hearn's encouragement and support we should neither have started nor persisted.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-Example Guided Fence Insertion under Weak Memory Models. to appear (TACAS 2012). 1

[2] Jade Alglave and Luc Maranget. Stability in Weak Memory Models. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 50–66. Springer Berlin / Heidelberg, 2011. 1

[3] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP 2009: Workshop on Declarative Aspects of Multicore Programming*, January 2009. 1

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, 1998. 1

[5] Mohamed Atig, Ahmed Bouajjani, and Gennaro Parlato. Getting Rid of Store-Buffers in TSO Analysis. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 99–115. Springer Berlin / Heidelberg, 2011. 1

[6] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. What's Decidable about Weak Memory Models? to appear (ESOP 2012). 1

[7] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 7–18, 2010. 1

[8] Jules Bean. *Ribbon Proofs — A Proof System for the Logic of Bunched Implications*. PhD thesis, Queen Mary, University of London, 2006. 5

[9] Richard Bornat and Hasan Amjad. Explanation of two non-blocking shared-variable communication algorithms. *Formal Aspects of Computing*, pages 1–39, 2011. 2

[10] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, 2005. 2

[11] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in Separation Logic. In *Proceedings of MFPS XXI*. Elsevier ENTCS, May 2005. 2

[12] Matko Botinčan, Mike Dodds, and Suresh Jagannathan. Resource-sensitive synchronization inference by abduction. In *POPL*, 2012. 2

[13] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 12–21, 2007. 1

[14] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, 2005. 1

[15] Mike Dodds, Suresh Jagannathan, and Matthew J Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, 2011. 2

[16] Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 16–28, 2009. 2

[17] Thuan Huynh and Abhik Roychoudhury. Memory model sensitive bytecode verification. *Formal Methods in System Design*, 31:281–305, 2007. 1

[18] Cliff B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP '83*, pages 321–332, 1983. 3

[19] M Kuperstein, Martin T. Vechev, and E Yahav. Automatic inference of memory fences. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 111–119, 2010. 1

[20] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, 2009. 1, 4, 7, 11, v

[21] Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In L. Fribourg, editor, *CSL 2001*, pages 1–19. Springer-Verlag, 2001. LNCS 2142. 3

[22] Matthew J. Parkinson, Richard Bornat, and Peter O'Hearn. Modular verification of a non-blocking stack. *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 297–302, 2007. 2

[23] Charles Sanders Peirce. *Collected Papers of Charles Sanders Peirce*, volume V and VI. Belknap Press, 1935. 1

[24] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. 3

[25] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 175–186, 2011. 1, 2, 9

[26] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. (Research Highlights). 1

[27] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007. 3, 9

[28] Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, volume 4037 of *LNCS*, pages 256–271, August 2007. 3, 9

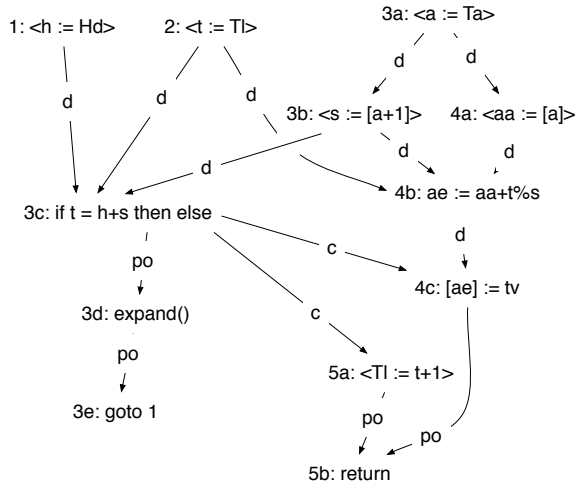
[29] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In Eran Yahav, editor, *Static Analysis*, volume 6887 of *Lecture Notes in Computer Science*, pages 146–162. Springer Berlin / Heidelberg, 2011. 1

[30] Martin Vechev and Maged M. Michael. IWSQ orderings not tested on Power. private communication, January 2012. 1

[31] John Wickerson, Mike Dodds, and Matthew J. Parkinson. Ribbon proofs for separation logic. to appear, 2012 . 5

```
void put(Task tv) {
1:  <h := Hd>;
2:  <t := Tl>;
3a: <a := Ta>;
3b: <s := [a+1]>;        // s := a.size
3c: if t = h+s then {
3d:     expand();
3e:     goto 1
3f: };
4a: <aa := [a]>;         // aa := a.array
4b: ae := aa+t%s;        // ae := &(a.array[t%a.size])
4c: [ae] := tv;          // a.array[t%a.size] := tv
5a: <Tl := t+1>;
5b: return
}
```

(a) code



(b) dependency graph

Figure 9: FIFO put

A Orderings in put

Analysis of steal is intricate because it could be interfered with by the owner via put, expand and take, as well as by other thieves. The owner is much less interfered with: thieves can only increase Hd with action (7).[9] Figure 9(a) shows the code for put; figure 9(b) shows the implied execution orderings.

This time there are two writes. The control dependencies between the conditional on line 3c and the writes (though strictly there's a dependency from lines 1 and 2 as well) is upon their *commitment* : i.e. propagation of the writes to other threads. Although the writes can be speculated before the branch, the proof dependency remains: we can rely on the result of the branch as a precondition on the write. The writes can't be propagated to other threads until the conditional is resolved, but they don't have to be propagated immediately, so writes to other addresses can overtake them. Although we only show a program-order dependency, it's obvious that there is an ordering issue between instruction executions in put and the enclosed expand(), dealt with that in §**??**.

The idea is clearly that line 4c should effect action (5) and line 5a action (6). That means there must be a precondition on line 5a that the array, at position $Tl\%Sz$, contains the next queue value $\boldsymbol{Ts}_{Tl}$ in order to preserve invariant (1).

We don't appear to need any more barriers. After line 1 we know instantaneously $h = Hd \leq Tl$ and stably $h \leq Hd \leq Tl$; after line 2 we know $t = Tl$, which is stable; after line 3a we know $a = Ta$, which is stable; after line 3b we know $a = Ta \wedge Ta \mapsto \_, s$, which is stable. So at line 3c if $t \neq h + s$ we know that $Tl < Hd + [Ta + 1]$, which is stable; if $t = h + s$ we know that the array was full because $Tl = Hd + [Ta + 1]$ held at the instant we read Hd. The revised code is in figure 5 and the formal proof in figure 10.

B Orderings in expand

Figure 11 shows expand() recast to use single-memory-access commands. As with put(), only the value of Hd is unstable, and it can only increase. The dependency graph is shown in figure 12: it's more complicated than any so far, because it shows two iterations of a loop. There are so many entangled dependencies that

[9] That's guaranteed because CAS(&Hd,h,h+1) is guaranteed to increase Hd by 1. It has nothing to do with barriers.

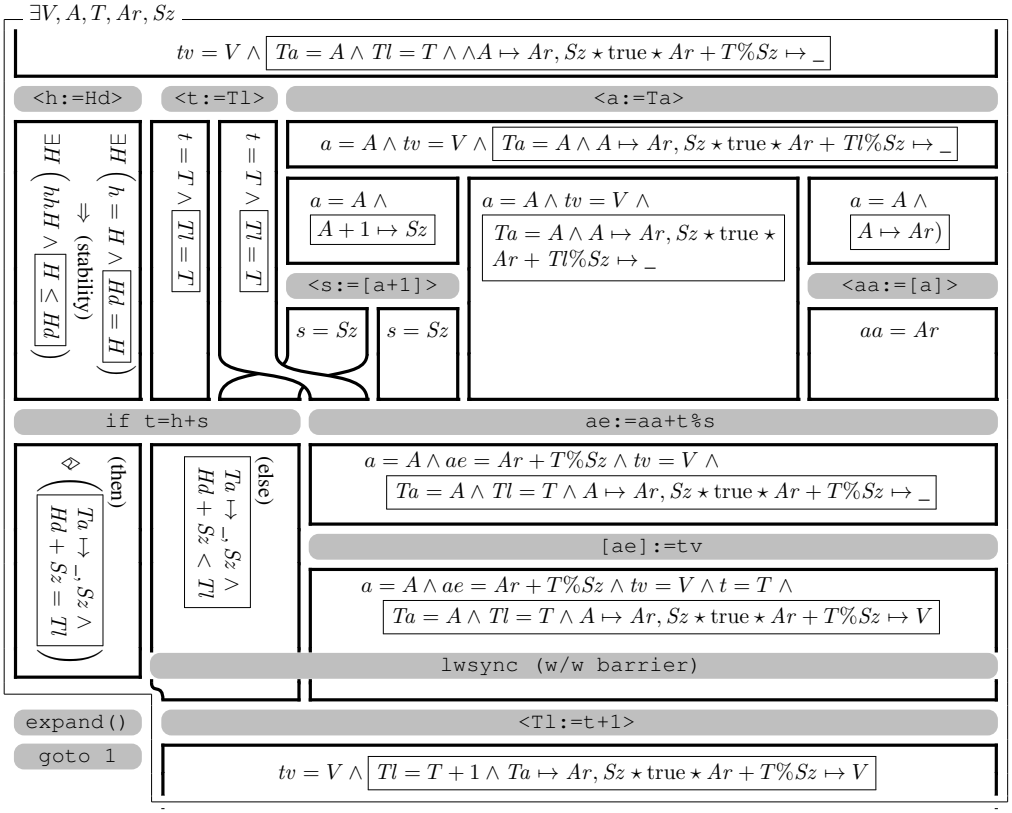


Figure 10: Formal proof of put

```
void expand() {
local Task tv;
1a: <a := Ta>;
1b: <aa := [a]>;
1c: <s := [a+1]>;
2a: a' := new SizedArray(2∗s);
2b: <aa' := [a']>;
2c: s' := 2∗s;
3a: <i := Hd>;      // for i in Hd..Tl-1 do
3b: <t := Tl>;
3c: if i=t then {
3d:     goto 5a
3e: };
4a: ae := aa+i%s;
4b: tv := [ae];     // tv := Ta.array[i%Ta.size]
4c: ae' := aa'+i%s';
4d: [ae'] := tv; // a'.array[i%a.size] := tv
4e: i := i+1;
4f: goto 3c;    // od
5a: <Ta := a'>;
5b: return
}
```

Figure 11: FIFO expand with single-memory-access commands

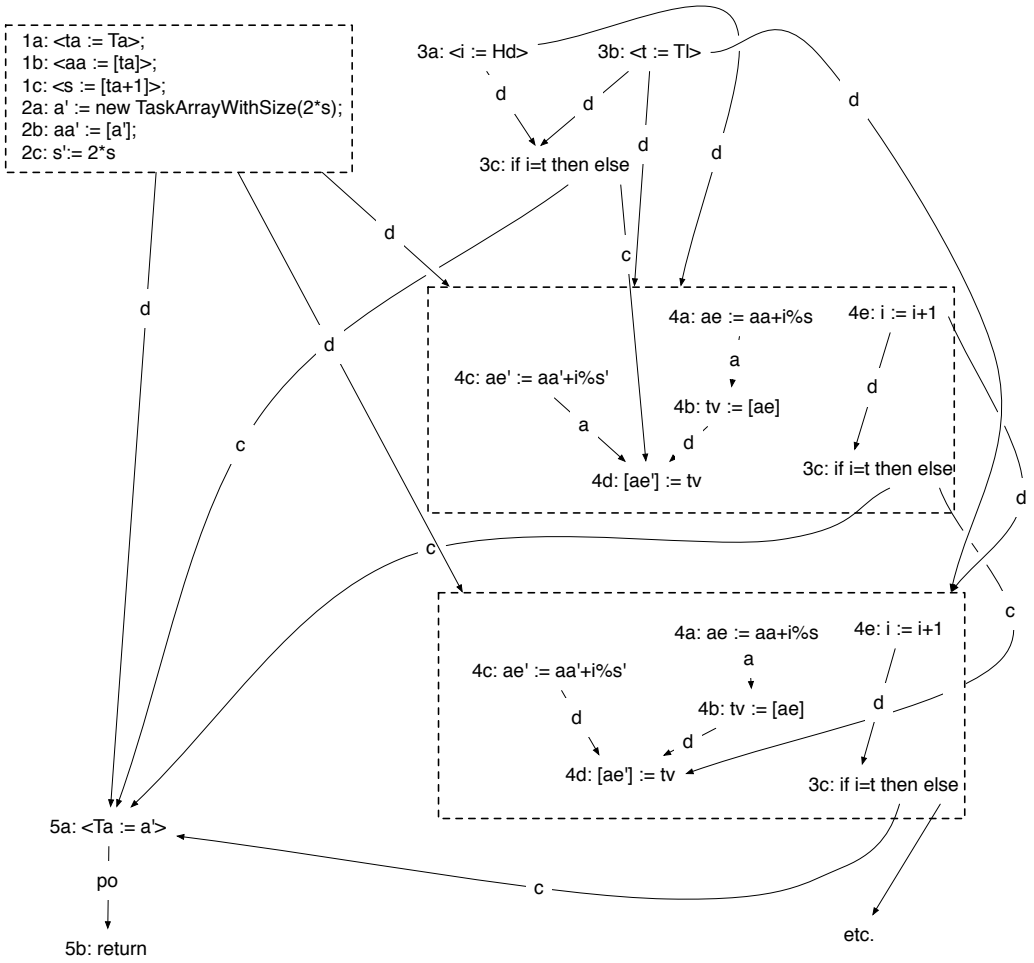


Figure 12: dependencies in FIFO expand

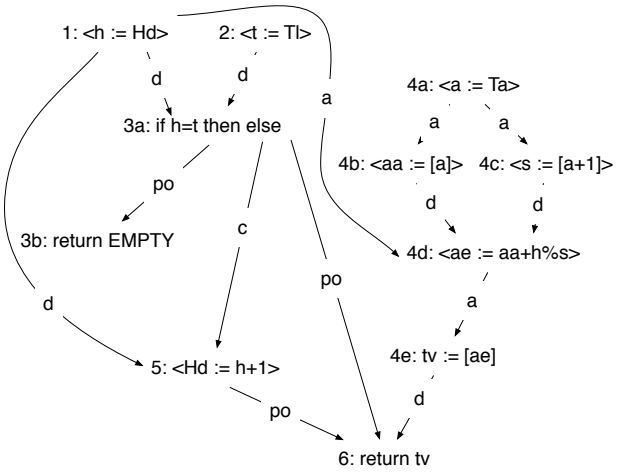
we've simplified by collecting the initialising assignments on lines 1a–2c into a dotted box at top left – each is data-dependent on its predecessor, so they will be executed in program order – and we've shown data dependencies from the box rather than its individual commands. Initialisation of the loop is top right, followed by the first test for termination; the first iteration is shown below, and the second below that. It's obvious that the first iteration data-depends on the initial assignment to i and the second iteration on the incremented value of i set during the first. There's not much ordering inside the loop, and in particular the write on line 4c has no successors, other than that it is a program-order predecessor of the return on line 5b.

All this complexity strains to produce very little. We don't have to order i:=Hd and t:=Tl because the environment can only increase Hd and if it does we will merely copy some unnecessary task(s) into the new array. We do need a write/write barrier somewhere between lines 2a and 4d on one side and line 5a on the other to ensure that when line 5a is executed the record which a points to is initialised, the queue values are in place in the array and invariant (1) is preserved. If we put the barrier after line 4c it will be executed on each execution of the loop; if we put it after the loop exit it's executed only once. Barriers are expensive instructions so we insert it at line 5a.

The modified code is shown in figure 5. The number of dependencies mean that a formal proof of the final dependency graph would be tangled and not very illuminating: apart from the loop structure it is extremely similar to the formal proof of put().

```
Task take() {
local Task tv;
1: <h := Hd>;
2: <t := Tl>;
3a: if h=t then {
3b:     return EMPTY
3c: };
4a: <a := Ta>;
4b: <aa := [a]>;     // aa := a.array
4c: <s := [a+1]>;    // s := a.size
4d: <ae := aa+h%s>; // ae := &a.array[h%a.size]
4e: tv := [ae];      // tv := a.array[h%a.size]
5: <Hd := h+1>;
6: return tv
}
```

(a) code



(b) dependency graph

Figure 13: FIFO take

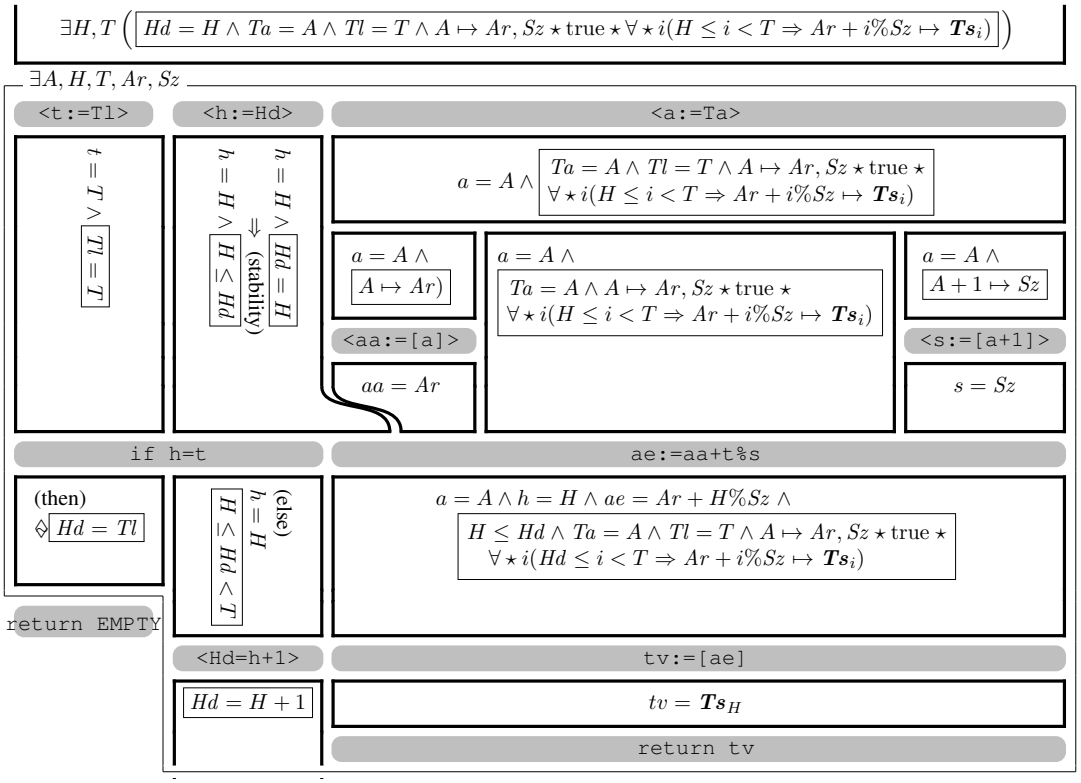


Figure 14: Formal proof of take

C Orderings in take

The single-memory access code for take is in figure 13(a) and the execution dependency graph in figure 13(b). It doesn't seem to need any barriers: before the write on line 5 we need to know that $h < Tl$, and we do, for the usual reasons (environment can't change Tl; at line 3a we have $h \neq t \Rightarrow h < t \wedge t = Tl$). The

program-order dependencies from line 3a to lines 3b and 6 are just to remind us that speculated executions of instructions in program order following the returns will be accepted or abandoned according to the result of the $h = t$ test.

The read on line 4e can happen before or after the write on line 5: indeed it can happen after the return, and we know that we will always get the same value, because only the owner can alter the array, and if it does so then the read will precede a subsequent write by memory ordering. No need for a barrier on that account.

The formal proof is in figure 14. The context for action (8), required in the proof of steal, is that the element indexed by Hd must have been in place beforehand whenever Hd is decreased. That isn't obvious from this proof, and to make it obvious would require lots more space. But we can argue as follows: if, at the instant when $Hd := h + 1$ is executed, $H + 1 < Hd$ then, since $Hd \leq Tl$, it follows that $H + 1 < Tl = T$; and then, from the fact that Tl is invariant throughout the execution, we know that $Ar + (H + 1)\%Sz \mapsto \boldsymbol{Ts}_{H+1}$, and similarly for any other values in the range $H + 1..Hd - 1$.

We have the same result as Michael et al.: no barriers in take.

D Thread diagrams in more detail

In the text we gave an abbreviated diagrams of interactions between owner and thief for the empty steal bug, and no diagrams at all for overwriting and overtaking. Here are the large versions of those diagrams, in figure 15 (cf. figure 3(b)), figrefoverwritebug and figure 17.

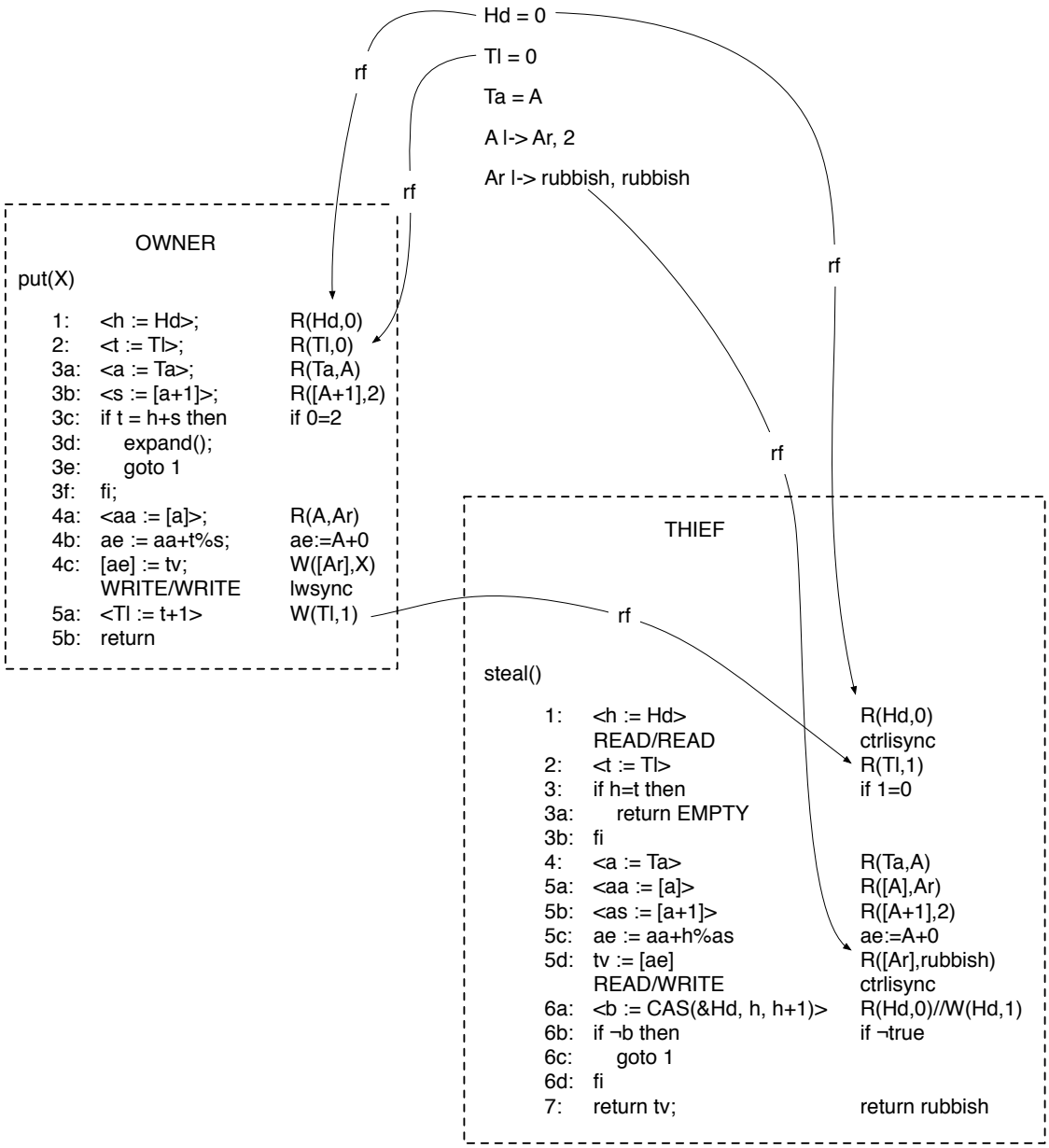


Figure 15: The 'empty steal' bug without a Tl/Ta barrier

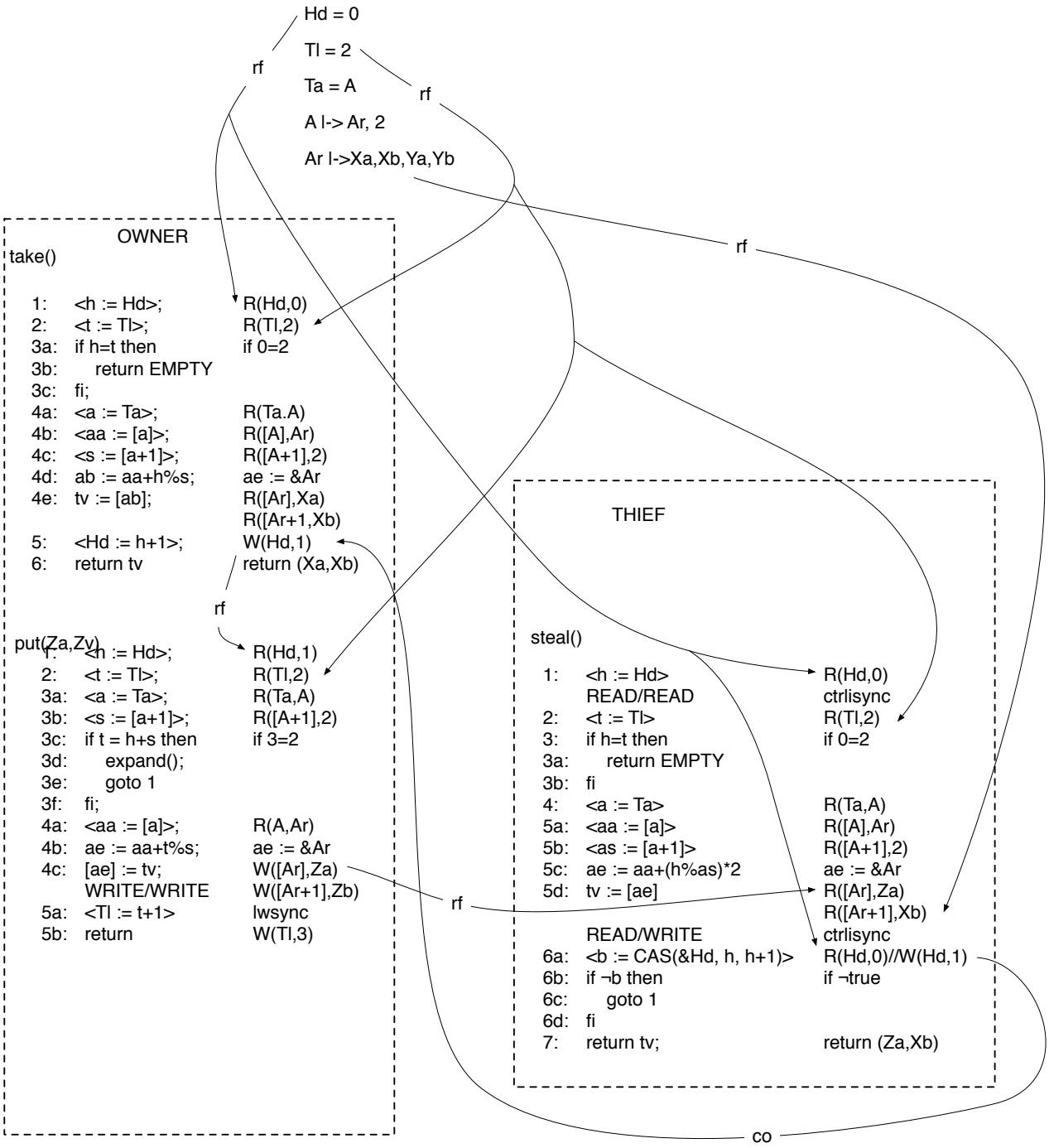


Figure 16: The overwrite bug without a barrier after <Hd:=h+1>

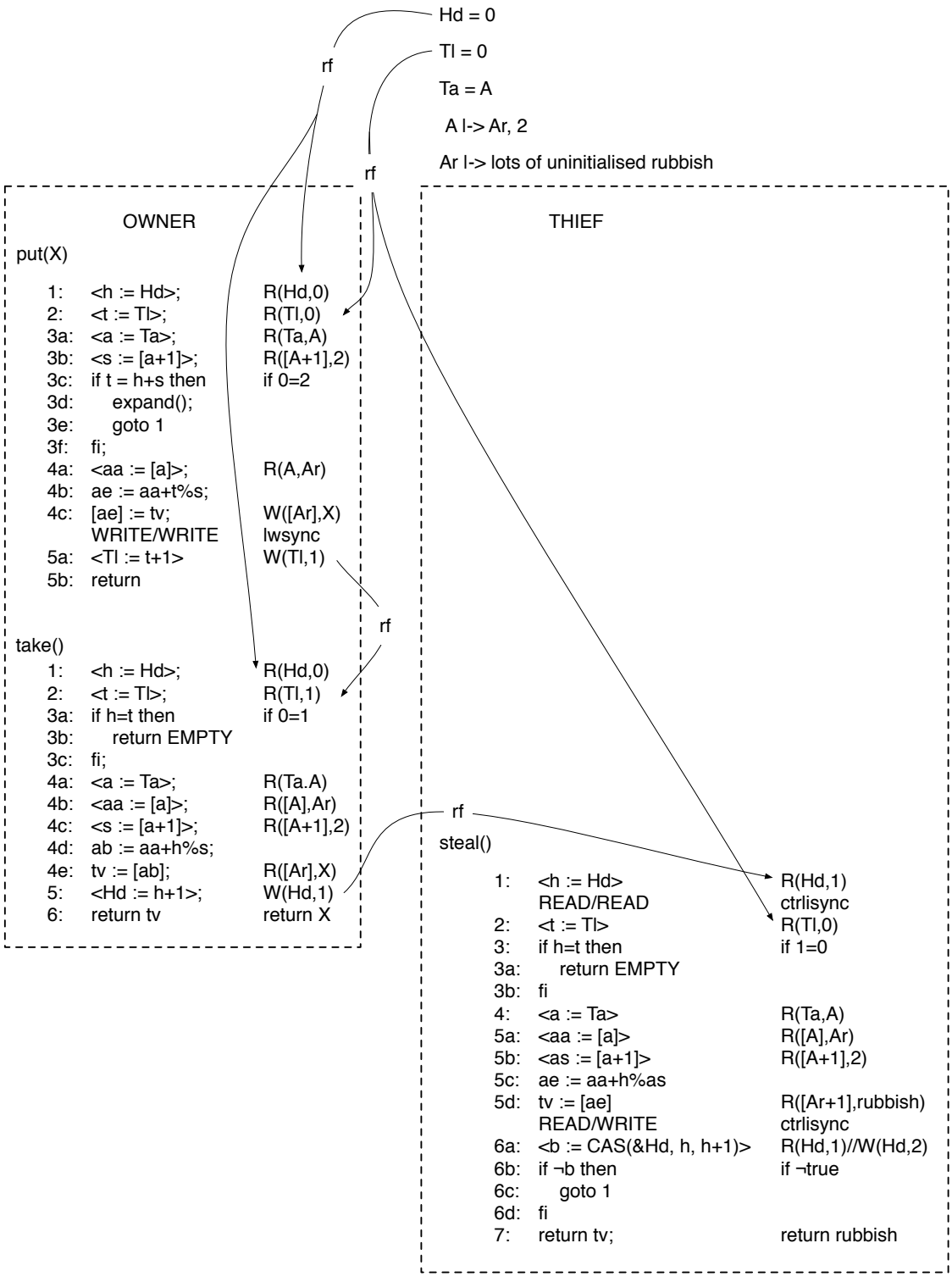


Figure 17: The overtaking bug with the lstinlineh=t test in steal()