CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3659

Dekker's mutual exclusion algorithm made RW-safe

Peter A. Buhr[1,*,†], David Dice[2] and Wim H. Hesselink[3]

[1]*Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada*
[2]*Oracle Labs, Burlington, MA, USA*
[3]*Department of Computing Science, University of Groningen, 9700 AK Groningen, The Netherlands*

SUMMARY

Dekker's algorithm was thought to be safe in an environment *without* atomic reads or writes where bits flicker or scramble during simultaneous operations. A counter-example is presented showing Dekker's algorithm is unsafe without atomic read. A modification to the original algorithm is presented making it RW-safe, allowing threaded systems to be built on low cost/power hardware without atomic read/write. Correctness is verified by means of invariants and UNITY logic. A performance comparison is made for several two-thread software mutual-exclusion algorithms to see if the RW-safe Dekker is competitive. A subset of the two-thread solutions are then compared in two N-thread tournament algorithms. The performance results show that the additional checks in the RW-safe Dekker do not disadvantage the algorithm in comparison with other two-thread algorithms. The RW-safe N-thread tournament algorithms are competitive with the hardware-assisted Mellor-Crummey and Scott algorithm. Copyright © 2015 John Wiley & Sons, Ltd.

Received 10 May 2015; Revised 5 August 2015; Accepted 9 August 2015

KEY WORDS: Dekker's algorithm; software solution; mutual exclusion; performance experiment

1. INTRODUCTION

The problem of mutual exclusion was proposed in 1965 by Dijkstra [1, 2] and is formulated as follows. Start with a system of N concurrent threads or processes communicating by shared memory. From time to time these threads need exclusive access to a shared resource, called a *critical section* (CS). When a thread is in a CS , other threads needing that resource must wait. *Mutual exclusion* (MX) is the design of an entry and exit protocol that ensures there is never more than one thread in a CS . In a traditional setting, all threads are in an infinite loop

```
thread(p) :
  loop
    NCS ;       // noncritical section
    entry ;     // entry protocol
    CS ;        // critical section
    exit ;      // exit protocol
  endloop .
```

NCS stands for the noncritical section, where threads cannot use data associated with the CS. Threads may remain forever in the NCS. Over the years, many software MX algorithms have been proposed. Recently, we performed an investigation of 20 of these algorithms [3]; the algorithms were implemented and their performance compared under both zero and high contentions.

*Correspondence to: Peter A. Buhr, Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1, Canada.
†E-mail: pabuhr@uwaterloo.ca

When threads are waiting in the entry protocol, it is required that some waiting thread eventually enters the CS, called *collective progress* (absence of indefinite postponement or livelock). A stronger requirement is that, when a thread needs to enter the CS, it eventually enters, called *individual progress* (absence of starvation or lockout-freedom). All mutual exclusion algorithms have a minimal of one bit of information for each thread, shared amount N participating threads. A thread sets its bit to indicate intent to enter the critical section, after which it scans the other thread bits to determine when it is safe to enter. An efficient mutual-exclusion algorithms attempts to minimize the amount of scanning and retry, while maintaining collective progress and individual progress. Organizing the mutual-exclusion algorithm in a tournament structure (Section 7) helps minimizes scanning and retry.

The manuscript EWD123 [2, pp. 17,18] contains the first correct mutual exclusion algorithm for $N = 2$ threads. This algorithm is attributed to Th. J. Dekker. Figure 1 shows a version with minor modifications and a structured version. (The explanation of **volatile** and `Fence()` are postponed to Section 8.2.) The two threads are numbered 0 and 1. Variable p ranges over the thread identifiers. The algorithm uses the shared variables `turn` and `flag`. The condition `flag`$[p] = 1$ implies that thread p is interested in the CS (we have toggled the values of `flag` in comparison with [2]). The condition `turn` $= p$ means that, when both threads are interested, thread p has priority and thread $1 - p$ waits, that is, `turn` contains the next p that should enter the CS, breaking a tie on simultaneous arrive (prevent livelock) and forcing alternation of waiting threads (prevent starvation). The command **await** B for a condition B is defined to be the atomic command that tests B and proceeds only when B holds. It is therefore equivalent to

while $\neg B$ **do** skip **endwhile**.

While Dekker's algorithm is 50 years old, it still has surprises. We believed Dekker's algorithm to be RW-safe, that is, that it works without atomic read or write (Section 2); however, our assumption turned out to be false. We discovered this fact while attempting to build RW-safe N-thread solutions for mutual exclusion in other work. The contributions of this work are as follows:

1. A counter-example showing Dekker's algorithm is RW-unsafe;
2. A construction of a Dekker's algorithm variant that is RW-safe;
3. A correctness proof of the new Dekker variant;

```
var turn : volatile integer ;
var flag : array [0 : 1] of volatile integer := (0, 0) ;
```

```
loop
    NCS;

L1:flag[p] := 1 ;
    Fence() ;
L2:if flag[1 − p] = 1 then
        if turn = p then
            goto L2 ; endif
        flag[p] := 0 ;
        Fence() ;   // optional
        await turn = p ; goto L1 ;
    endif
    CS;
    turn := 1 − p ;
    flag[p] := 0 ;
endloop .
```

(a) Variant

```
loop
    NCS;
    loop
        flag[p] := 1 ;
        Fence() ;
        if flag[1 − p] = 0 then break ; endif
        if turn = p then
            await flag[1 − p] = 0 ; break ; endif
        flag[p] := 0 ;
        Fence() ;   // optional
        await turn = p ;
    endloop
    CS;
    turn := 1 − p ;
    flag[p] := 0 ;
endloop .
```

(b) Structured

Figure 1. Dekker's algorithm.

DOI: 10.1002/cpe

4. A performance comparison of several two-thread software algorithms for mutual exclusion; and
5. A performance comparison of two N-thread tournament algorithms using the different two-thread algorithms.

The RW-safeness is discussed in Section 2, with an explanation of why the original Dekker's algorithm is RW-unsafe and how to modify Dekker's algorithm to be RW-safe. Safety and the invariants needed for progress of the RW-safe algorithm are verified in Section 3. Progress is proved with UNITY logic [4] in Section 4. In Section 5, the proofs are extended for our variant of Dekker's original algorithm. Section 6 discusses ordering issues between arriving threads. Section 7 introduces N-thread tournament algorithms. Section 8 explains the implementation of the algorithm in a shared-memory system, how it is tested along with competing algorithms, and evaluates the results. The conclusions are in Section 9.

2. RW-SAFENESS

A shared variable is called *atomic* [5, p. 88] if read and write operations on it behave as if they never overlap but always occur in some total order that refines the *precedence* order, where an operation precedes another iff it terminates before the other starts. *Safeness* is a weaker property of a shared variable. A shared variable is safe, if, under the assumption that write operations on the variable never overlap, every read operation that does not overlap with any write operation returns the most recently written value, and every read operation that does overlap with some write operation returns an arbitrary value of the correct type. When a write operation on a variable is in progress during a read, the value of the variable is said to *flicker* because different simultaneous reads return arbitrary values. When a write operation on the shared variable is in progress during another write, the value of the variable is said to *scramble* because subsequent reading can return an arbitrary value, that is, not a value written by either write.

An algorithm is called *W-safe* if it is immune to scrambling; it is called *RW-safe* if it is immune to both flickering and scrambling [3].[‡] *R-safeness* is not a viable concept because the values read are always suspect because of scrambling. An algorithm is *RW-unsafe*, if it is not RW-safe, that is, it requires atomic write and read operations.

> Any (software mutual exclusion) algorithm based upon atomic operations cannot be considered a fundamental solution to the mutual exclusion problem. [6, p. 314]

Atomicity of memory write can fail on a multiprocessor when multiple CPUs simultaneously read/write to the same memory location without proper memory-bus arbitration. For example, low-cost low-power devices exist with multiple computational units, such as CPUs, DSPs, and antennas (cell phones, cameras, printers, music players, small appliances, toys, and even singing greeting cards), but to reduce cost/power, atomicity (conflict resolution) is not always provided so simultaneous read/write causes read to flicker and simultaneous writes cause scrambled bits.

> Multi-port memory is widely used in multiprocessor systems because it can provide parallel access to the shared memory. ... Most of these products do not consider the conflicts caused by accessing to the same data, while a few products only provide simple conflict mechanisms, such as busy control scheme. [7, p. 251]

As well, atomicity of memory write can fail on a uniprocessor if assignment is performed in parts. For example, a 32-bit integer constant may be assigned as two 16-bit values because 16-bit values fit in the immediate field of an instruction. Also, larger 64-bit values, for example, 64-bit address, may be assigned as two 32-bit values by the memory-bus hardware. Hence, simultaneous assignment to the same memory location can result in the bits being scrambled, generating an arbitrary value.

[‡]The noun corresponding to this concept of *safe* should be *safeness* [5, p. 88], because it applies to the property of atomic hardware, whereas the noun *safety* applies to the property of concurrent algorithms that 'nothing bad happens', meaning no deadlock or error state.

DOI: 10.1002/cpe

```
for ( ;; ) {
    for ( int i = 0; i < 100; i += 1 ) flag[p] = i % 2;
    flag[p] = 1;
    if ( flag[1 - p] == 0 ) break;
    if ( turn == p ) {
        await( flag[1 - p] == 0 ); break };
    for ( int i = 0; i < 100; i += 1 ) flag[p] = i % 2;
    flag[p] = 0;
    await( turn == p );
}
CS;
for ( int i = p; i < 100; i += 1 ) turn = i % 2;
turn = 1 - p;
for ( int i = 0; i < 100; i += 1 ) flag[p] = i % 2;
flag[p] = 0;
```

Figure 2. Dekker's algorithm in C language with flickering (not RW-safe).

Similarly, if a read is performed during a memory write where the assignment is performed in parts or without proper memory-bus arbitration, reads see flickering values.

The assumption that write operations on a variable never overlap is obviously satisfied when the variable is written by only one thread. In this case, it is called an *output variable* of the thread. The assumption can also be implied by invariants of the program.

2.1. Dekker's algorithm

Both Dekker algorithms in Figure 1 are RW-unsafe because of the following scenario.[§] Assume thread 0 is performing a flickering write of `flag`[0] := 0 in the exit protocol. Thread 1 reads a `flag`[0] = 0 flicker, goes through the CS, sets `turn` := 0, goes through NCS, reads a `flag`[0] = 1 flicker, and starts waiting for `turn` = 1. Finally, thread 0 concludes its exit protocol by setting `flag`[0] = 0, goes to NCS, and dies or falls asleep. Now thread 1 can never pass the await statement. This scenario violates the requirement of individual progress.

However, to give a sense of how robust Dekker's algorithm is in practice, Figure 2 shows a C language implementation of the algorithm with aggressive flickering before each assignment to simulate hardware without safeness. This program has been run with 32 parallel threads for many CPU hours without failure, whereas most software solutions fail immediately with flickering. Indeed, Dekker's algorithm allows flickering at all the assignments, but only under contention, which is why it appears to be RW-safe.

Dekker's algorithm has W-safeness because there are no simultaneous writes. The variables in array `flag` are written by one task and read by the other, so there is never any simultaneous write by both tasks to these variables (output variables). The variable `turn` has the potential for simultaneous write, but the assignment to `turn` occurs in the exit protocol, and the waiting task cannot make progress out of the entry protocol until *both* assignments in the exit protocol are finished (assignment order is important). Therefore, a waiting task cannot race through the CS to perform a simultaneous write of `turn` with the other task in the exit protocol, because the other task has completed the exit protocol.

Doran and Thomas [9, Figure 1, p. 207] refactored Dekker's algorithm in Figure 3 into a variant without **goto** or loop statements, excluding busy-waiting loops. However, this variant is RW-unsafe. Like Dekker's algorithm, the Doran and Thomas algorithm allows flickering of `turn` at line 19, and flickering of `flag` at the lines 11, 14, and 16, but *not* at line 20. The failure scenario is the same as for Dekker.

[§]A prior paper [8, p. 1152], incorrectly states that Dekker's algorithm is RW-safe.

DOI: 10.1002/cpe

```
thread(p : {0,1}) :
   loop
10    NCS;
11    flag[p] := (flickering) 1 ;
      Fence() ;
12    if flag[1 − p] then
13       if turn ≠ p then
14          flag[p] := (flickering) 0 ;
            Fence() ;   // optional
15          await turn = p ;
16          flag[p] := (flickering) 1 ;
            Fence() ;
         endif
17       await ¬ flag[1 − p] ;
      endif
18    CS ;
19    turn := (flickering) 1 − p ;
20    flag[p] := (NO FLICKERING) 0 ;
   endloop .
```

Figure 3. Loop-free Dekker variant (not RW-safe).

```
      loop
10       NCS;
         loop
11          flag[p] := (flickering) 1 ;
            Fence() ;
12          if flag[1 − p] = 0 then break ; endif
13          if turn = p then
14                await flag[1 − p] = 0 ; break ; endif
15          flag[p] := (flickering) 0 ;
            Fence() ;   // optional
16          await turn = p ∨ flag[1 − p] = 0 ;
         endloop
17       CS;
18       if turn = p then
19          turn := (flickering) 1 − p ; endif
20       flag[p] := (flickering) 0 ;
      endloop .
```

Figure 4. Dekker's algorithm with RW-safeness.

2.2. *Dekker's algorithm with RW-safeness*

The aim of this paper is to present a small modification of the structured Dekker's algorithm in Figure 1(b), and to prove it is correct and RW-safe. Figure 4 shows the new variant of Dekker's algorithm with RW-safeness. It uses the same variables as before. Line numbers are added for later reference. The second disjunct of the condition at line 16 and the test at line 18 are the only changes to the structured Dekker algorithm. The assignments to the shared variables in the lines 11, 15, 19, and 20 are marked with 'flickering' to indicate that reading during such an assignment may return arbitrary values.

The aforementioned RW-safe failure shows that waiting for turn = p is too rigid. Therefore, the waiting condition has been weakened by replacing it with turn = p ∨ flag [1 - p] = 0, where the new condition enables progress should the other thread not return.

DOI: 10.1002/cpe

Mutual exclusion holds in the region 17–19.

$$MX: \ 0 \,\mathbf{in}\, \{17 \ldots 19\} \ \wedge \ 1 \,\mathbf{in}\, \{17 \ldots 19\} \ \Rightarrow \ false.$$

That is, it is never the case that both threads (0, 1) are in the region 17–19. In the following, we prove this without using the variable `turn`. Therefore, assignments to `turn` in line 19 can never overlap, and the `flag`s are output variables, implying W-safeness.

The aforementioned failure scenario shows the first disjunct in Figure 4 at line 16 is too strong. Furthermore, if the first disjunct is removed, leaving only the second disjunct, the following scenario shows that individual progress fails. Initially, `turn` = 0 and the threads are at line 10. Thread 0 enters and goes into the CS, thread 1 enters and starts waiting at line 16. Now thread 0 repeatedly exits the CS, sets `turn` := 1, performs the NCS, enters again, finds `flag[1]` = 0, and goes to the CS. Thread 1 remains at line 16 because the test on `turn` has been removed and thread 1 repeatedly finds `flag[1]` = 1, as it happens to read `flag[1]` only when thread 0 is in the CS, violating individual progress. Such unfortunate scheduling is allowed by weak fairness.

If the test in line 18 is removed, the following scenario violates individual progress. Thread 0 enters, goes through the CS, and starts writing `turn` while `flag[0]` = 1. Thread 1 enters, arrives at line 16, and starts waiting because it finds `turn` = 0. Thread 0 repeatedly does the following. It sets `turn` := 1, exits, goes through the NCS, enters, finds `flag[1]` = 0, goes through the CS, and starts writing `turn` in line 19 (line 18 being removed). Thread 1 remains at line 16 because it only tests the waiting condition when thread 0 is at line 19, reading `turn` = 0 (flicker value) and `flag[0]` = 1. Again weak fairness allows such unfortunate scheduling.

3. VERIFICATION OF SAFETY

The correctness of the algorithm of Figure 4 is verified by interpreting it as a *transition system*, with a global state (that comprises the values of all shared and private variables, including program counters), in which the threads perform *steps* that cannot be subdivided. Hence, there is no interference within steps. Thread p obtains a private variable $pc.p$ (program counters) initially set to line 10. Each line number stands for a single transition. The abstract commands NCS and CS are regarded as skip in the transition system. Finally, it is simpler in the proof to treat array `flag` as a Boolean array, where the value 1 (i.e., interested in CS) is replaced by $true$ and 0 is replaced by $false$.

Following [10], a flickering assignment `x` := (flickering) E is modeled as a nondeterministic choice

$$\ell: \quad (\texttt{x} := \text{arbitrary} \ ; \ \textbf{goto} \ \ell \parallel \texttt{x} := E),$$

where $\parallel$ indicates a nondeterministic choice between its two operands with lower binding than the semicolon of sequential composition. We assume liveness for this definition so the repetition terminates.

According to the principle of *single critical reference*, an atomic command reads or writes at most one shared variable (not both) [11, (3.1)], [12, p. 273]. Figure 4 violates this principle in line 16 by inspecting both `turn` and `flag[1 - p]`. This violation is allowed, however, because it concerns a disjunction of two conditions that can be evaluated at different moments. When either of the conditions evaluates to true, the thread can pass the **await** statement.

The mutual exclusion predicate MX holds initially because both threads are at line 10. If MX is falsified by either thread, say p, then it must be true that p enters CS from line 12 or line 14, while the other thread, say $q = 1 - p$, is in 17–19. We express this as MX is threatened only by the steps 12 and 14. MX is preserved because of the additional invariant

$$Ip1: p \,\mathbf{in}\, \{12 \ldots 14\} \cup \{17 \ldots 19\} \Rightarrow \texttt{flag}[p].$$

Indeed, when q is in 17–19, $Ip1$ implies that `flag`$[q]$ holds, so that thread p at 12 or 14 cannot go to line 17. Predicate $Ip1$ holds initially at line 10 and does not change while stepping into or out of $\{12 \ldots 14\} \cup \{17 \ldots 19\}$ and is therefore invariant. This concludes the proof of mutual exclusion (MX).

DOI: 10.1002/cpe

To show progress, the following invariants are needed:

$$\begin{aligned} &\mathit{Jp0}: \ p \ \textbf{in} \ \{10 \ldots 20\}, \\ &\mathit{Jp1}: \ p \ \textbf{in} \ \{10, 16\} \Rightarrow \neg \, \mathtt{flag}[p], \\ &\mathit{Jp2}: \ p \ \textbf{at} \ 14 \wedge \mathtt{turn} \neq p \Rightarrow 1 - p \ \textbf{at} \ 19. \end{aligned}$$

For *Jp0*, p is always in the range $\{10 \ldots 20\}$. For *Jp1*, p starts with $\mathtt{flag}[p]$ set to false and resets it to false at lines 15 and 20. For *Jp2*, the only way p can be at line 14 and not equal to `turn` is when the other thread is at line 19 and flickering `turn`. Indeed, if the other thread is flickering `turn`, nothing can be said about turn.

4. PROGRESS

The proof of progress is based on bounded UNITY as proposed in [13]. We distinguish between the *environment* steps, which advance the program context, and the *forward steps*, which advance the algorithm goal. The environment steps are the steps of lines 10 and 17, and the flickering steps of lines 11, 15, 19, and 20 in which $pc.p$ does not change. The forward steps are the steps with precondition $pc.p \notin \{10, 17\}$ that modify $pc.p$.

A thread is *enabled* if it can do a forward step, where *enabledness* is expressed by the condition $ena(p)$. For the new Dekker's algorithm, the enabledness condition of thread p is defined as

$$\begin{aligned} \mathit{ena(p)}: \quad & p \ \textbf{in} \ \{10 \ldots 20\} \setminus \{10, 17\} \\ & \wedge \, (pc.p = 14 \Rightarrow \neg \, \mathtt{flag}[1 - p]) \\ & \wedge \, (pc.p = 16 \Rightarrow \neg \, \mathtt{flag}[1 - p] \vee \mathtt{turn} = \mathtt{p}). \end{aligned}$$

Indeed, thread p is enabled iff it is inside the entry and exit protocol, and it is *not* blocked at an **await** statement.

4.1. Bounded unity

Progress towards the CS can only be expected for executions in which all threads perform enough forward steps. In bounded UNITY, this is quantified as follows. An execution fragment is called a *round* if, for every thread p, it contains a step of p or a state in which p is not enabled. Progress from some condition P to some condition Q is quantified by defining that P leads to Q within n rounds using notation $P \ \mathbf{LT} \ \langle n \rangle \, Q$. That is, if execution starts in a state where P holds, then some state where Q holds will occur in at most n rounds.

The challenge is to show that each thread from any point in the entry protocol reaches the CS in a bounded number of rounds. Of course, if the other thread is in the CS, the thread in the entry protocol cannot proceed to the CS until the other thread has left. It is assumed every thread in the CS leaves within a bounded number of rounds, say cs

$$p \ \textbf{at} \ 17 \ \mathbf{LT}\langle cs \rangle \ p \ \textbf{at} \ 18. \tag{1}$$

In bounded UNITY (as in UNITY [4, p. 65]), leads-to properties are proved by means of the operators **unless** and **ensures**. Here, P **unless** Q means that when $P \ \wedge \ \neg \, Q$ holds, this remains valid unless Q becomes valid (but Q may never become valid, and Q becoming valid does not necessarily imply that P ceases to hold). P **ensures** Q means that P **unless** Q holds and that there is some thread that, whenever $P \wedge \neg \, Q$ holds, is enabled and establishes Q when executed. It follows that P leads-to Q within one round

$$(P \ \textbf{ensures} \ Q) \ \Rightarrow \ P \ \mathbf{LT}\langle 1 \rangle \ Q.$$

The next important rule is transitivity

$$(P \ \mathbf{LT}\langle k \rangle \ Q) \wedge (Q \ \mathbf{LT}\langle m \rangle \ R) \ \Rightarrow \ (P \ \mathbf{LT}\langle k + m \rangle \ R).$$

Finally, the PSP (progress–safety–progress) rule

DOI: 10.1002/cpe

$$(P\ \mathbf{LT}\langle k\rangle\, Q) \wedge (A\ \mathbf{unless}\ M) \Rightarrow (P \wedge A)\mathbf{LT}\langle k\rangle (Q \wedge A) \vee M$$

shows that **unless** can be distributed across $\mathbf{LT}\langle k\rangle$.

4.2. *Progress of the algorithm*

First, the new algorithm has *unbounded overtaking*, just as Dekker's original one. If thread p is at line 16, that is, with `flag[p] =` $false$ because of $Jp1$, thread q can unboundedly often enter CS via line 12. Only when thread p sets `flag[p]` at line 11 is the other thread directed to line 13. If p remains at line 16, it does no steps (although it is enabled); therefore, no rounds have passed during this unbounded overtaking.

Next, both lines 12 and 14 in the entry protocol lead to 17 (CS). It is easier to prove line 14 reaches the CS because at line 12 it is unknown if `flag[1-p] = 0` holds, so p may have to go to line 13. At line 14, this complication is absent. The proof from line 14 is subdivided into four cases depending on where the other thread is located at 11–16, 10, 18–20, or 17.

For case 1, it is sufficient to prove

$$p\ \mathbf{at}\ 14 \wedge\ 1 - p\ \mathbf{in}\ \{11 \ldots 16\}\ \mathbf{LT}\langle 5\rangle\, p\ \mathbf{at}\ 17. \tag{2}$$

That is, if p is at line 14, it takes a maximum of four steps by q in $\{11 \ldots 16\}$ and one step by p for p to reach line 17. This is proved by noting *Ip1* implies `flag[p]` holds, and *Jp2* implies `turn =` p and that q is not at line 14. Therefore, it follows that thread q can go along the locations 11, 12, 13, 15, and 16 (four steps). While at 16 it becomes blocked, which can be expressed by the variant function

$$\begin{aligned} \mathit{vf0}(q) = (\ & pc.q = 16\ ?\ 0 \\ :\ & pc.q = 15\ ?\ 1 \\ :\ & pc.q = 13\ ?\ 2 \\ :\ & pc.q = 12\ ?\ 3 \\ :\ & pc.q = 11\ ?\ 4 \\ :\ & 0\ ). \end{aligned}$$

The predicate

$$\mathit{uvf0}(p, n):\quad pc.p = 14 \wedge 1 - p\ \mathbf{in}\ \{11 \ldots 16\}\ \wedge\ vf0(1 - p) \leqslant n$$

then satisfies

$$uvf0(p, 0)\ \mathbf{ensures}\ p\ \mathbf{at}\ 17$$

because

$$\begin{aligned} uvf0(p, 0) \Rightarrow\ & p\ \mathbf{at}\ 14 \\ \Rightarrow\ & q\ \mathbf{at}\ 16 \text{ by } uvf0, vf0 \\ \Rightarrow\ & \texttt{flag}[p], \neg\, \texttt{flag}[q] \text{by } \mathit{Jp1} \\ \Rightarrow\ & \texttt{turn} = p \text{ by } \mathit{Jp2} \\ \Rightarrow\ & ena(p), \neg\, ena(q) \\ \Rightarrow\ & \mathbf{ensures} p\ \mathbf{at}\ 17 \end{aligned}$$

and

$$\mathit{uvf0}(p, n + 1)\ \mathbf{ensures}\ \mathit{uvf0}(p, n)\ \vee\ p\ \mathbf{at}\ 17$$

because q is enabled to decrease *vf0*, and a step of p establishes p at 17.

DOI: 10.1002/cpe

By transitivity and induction, this gives

$$uvf0(p,4)\ \mathbf{LT}\langle 5\rangle\ p\ \mathbf{at}\ 17.$$

This proves Formula (2) because $uvf0(p,4)$ is equivalent to the left-hand side of (2).

For case 2, it is sufficient to prove

$$p\ \mathbf{at}\ 14\ \wedge\ 1-p\ \mathbf{at}\ 10\ \mathbf{ensures}\ p\ \mathbf{at}\ 17\ \vee\ 1-p\ \mathbf{in}\ \{11\ldots 16\}.$$

By Formula (2) and transitivity, it follows from this formula that

$$p\ \mathbf{at}\ 14\ \wedge\ 1-p\ \mathbf{at}\ 10\ \mathbf{LT}\langle 6\rangle\ p\ \mathbf{at}17. \tag{3}$$

For case 3, it follows from Formula (3) that

$$p\ \mathbf{at}\ 14 \wedge 1-p\,\mathbf{in}\,\{18\ldots 20\}\ \mathbf{LT}\langle 8\rangle\ p\ \mathbf{at}17 \tag{4}$$

because thread q reaches line 10 from 18, 19, and 20 in two rounds. If p is at 14, and q is at 18, then *Jp2* implies that p = `turn` making the condition at 18 false. Therefore, q steps directly from 18 to 20 and 20 to 10 in two rounds. If q is at 19 or 20, at most two rounds are necessary.

For case 4, Formula (4) with p and (1) with $1-p$ are combined by transitivity

$$p\ \mathbf{at}\ 14\ \wedge\ 1-p\ \mathbf{at}\ 17\ \mathbf{LT}\langle cs+8\rangle p\ \mathbf{at}17, \tag{5}$$

which states $1-p$ in the CS implies it then steps into the exit protocol, which steps p into the CS. Combining Formulas (2), (3), (4), and (5), covering all possible locations for $1-p$, and eliminating it by disjunction gives

$$p\ \mathbf{at}\ 14\ \mathbf{LT}\langle cs+8\rangle p\ \mathbf{at}\ 17. \tag{6}$$

This concludes the treatment of progress for thread p at line 14.

We now generalize line 14 to the whole entry protocol, that is, lines 11–16. In order to use Formula (6), we observe that line 14 is reached by thread p only if `turn = p` because of the test in line 13. The value of `turn`, however, can flicker when there is a thread at line 19. We therefore introduce the condition

$$ST(p):\quad \texttt{turn} = p\ \wedge\ 1-p\ \mathbf{not}-\mathbf{at}\ 19.$$

This stands for *stable turn*: it can be falsified only when thread p itself is at line 19, because $1-p$ cannot enter line 19 when *ST(p)* holds.

Indeed, when thread p is in the entry protocol and *ST(p)* holds, p reaches line 14 within five rounds

$$p\ \mathbf{in}\ \{11\ldots 16\}\ \wedge\ ST(p)\ \mathbf{LT}\langle 5\rangle p\ \mathbf{at}14. \tag{7}$$

This result is proved in the same way as Formula (2) with

$$\begin{aligned} vf1(p) = (\ & pc.p = 14\ ?\ 0\\ & :\ pc.p = 15\ ?\ 5\\ & :\ pc.p = 16\ ?\ 4\\ & :\ pc.p = 11\ ?\ 3\\ & :\ pc.p = 12\ ?\ 2\\ & :\ pc.p = 13\ ?\ 1\\ & :\ 0\ ) \end{aligned}$$

giving the number of steps to reach line 14, and predicate *uvf1(p, n)*

$$uvf1(p,n):\quad p\ \mathbf{in}\ \{11\ldots 16\}\ \wedge\ ST(p)\ \wedge\ vf1(p) \leqslant n.$$

DOI: 10.1002/cpe

As the maximum value of *vf1* is 5, this gives Formula (7). Combining Formula (7) with (6) by transitivity gives

$$p \textbf{ in } \{11 \ldots 16\} \ \wedge \ ST(p) \ \textbf{LT}\langle cs + 13\rangle \, p \textbf{ at } 17. \tag{8}$$

We now note that *ST(p)* is established by thread $1 - p$ in the forward step 19. If $1 - p$ is in the exit protocol, ST is true in at most two steps

$$1 - p \textbf{ in } \{17 \ldots 19\} \ \textbf{LT}\langle cs + 2\rangle ST(p). \tag{9}$$

If p is in the entry protocol, any step that causes p to leave the entry protocol results in p being in the CS

$$p \textbf{ in } \{11 \ldots 16\} \textbf{ unless } p \textbf{ at } 17. \tag{10}$$

Combining Formulas (9) and (10) with the PSP-rule gives

$$(1 - p \textbf{ in } \{17 \ldots 19\}) \wedge (p \textbf{ in } \{11 \ldots 16\}) \ \textbf{LT}\langle cs + 2\rangle (ST(p) \wedge p \textbf{ in } \{11 \ldots 16\}) \vee (p \textbf{ at } 17).$$

Combining the PSP result with Formula (8) by transitivity gives

$$1 - p \textbf{ in } \{17 \ldots 19\} \ \wedge \ p \textbf{ in } \{11 \ldots 16\} \ \textbf{LT} \ \langle 2 \cdot cs + 15\rangle \ p \textbf{ at } 17. \tag{11}$$

The $2 \cdot cs + 15$ rounds are needed as follows: first, according to Formula (9), $cs + 2$ rounds may be necessary to force thread $1 - p$ to establish $ST(p)$, which enables the precondition of Formula (8), allowing p to reach 17 in at most $cs + 13$ rounds, where cs is the number of rounds sufficient for traversing the CS.

We now prove

$$p \textbf{ in } \{11 \ldots 16\} \ \wedge \ 1 - p \textbf{ in } \{11 \ldots 16\} \ \textbf{LT} \ \langle 10\rangle \ p \textbf{ at } 17 \ \vee \ 1 - p \textbf{ at } 17. \tag{12}$$

Here, both threads are in the entry protocol, and the claim is that within 10 rounds one reaches the CS. Which thread reaches the CS depends on the value of `turn`, where `turn = p` or `turn = 1 - p`. The goal is to prove Formula (12) with the conjunct `turn = p` added to the left-hand side, and then `turn = 1 - p`, using the following predicate

$$\textit{uvf1a(p, n):} \ p \textbf{ in } \{11 \ldots 16\} \ \wedge \ 1 - p \textbf{ in } \{11 \ldots 16\} \ \wedge \ \texttt{turn} = p \ \wedge \ \textit{vf1(p)} \leqslant n.$$

First, consider the case `turn` = p, that is, in Formula (12)

$$p \textbf{ in } \{11 \ldots 16\} \ \wedge \ 1 - p \textbf{ in } \{11 \ldots 16\} \ \wedge \ \texttt{turn} = p \ \textbf{LT} \ \langle 10\rangle \ p \textbf{ at } 17 \ \vee \ 1 - p \textbf{ at } 17. \tag{13}$$

This case is proved by means of the predicate *uvf1a*, where *uvf1a(p, 0)* implies

$$p \textbf{ at } 14 \ \wedge \ 1 - p \textbf{ in } \{11 \ldots 16\}.$$

Therefore, by Formula (2)

$$\textit{uvf1a(p, 0)} \ \textbf{LT} \ \langle 5\rangle \ p \textbf{ at } 17 \ \vee \ 1 - p \textbf{ at } 17.$$

Next, we observe that *uvf1a(p, n+1)* ensures

$$\textit{uvf1a(p, n)} \ \vee \ p \textbf{ at } 17 \ \vee \ 1 - p \textbf{ at } 17.$$

This case holds because a step of p decreases *vf1(p)* keeping p in lines 11–16 or establishes p at 17, and similarly, a step of q keeps q in lines 11–16 or establishes q at 17. By transitivity and induction, these two results combine to give

$$\textit{uvf1a(p, 5)} \ \textbf{LT}\langle 10\rangle \ p \textbf{ at } 17 \ \vee \ 1 - p \textbf{ at } 17.$$

DOI: 10.1002/cpe

This implies Formula (13) because the precondition of (13) implies *uvf1a(p, 5)*.

By swapping the threads p and $1-p$ in Formula (13), giving

$$p \textbf{ in } \{11 \ldots 16\} \ \wedge \ 1-p \textbf{ in } \{11 \ldots 16\} \ \wedge \ \texttt{turn} = 1-p \ \textbf{LT} \langle 10 \rangle \ p \textbf{ at } 17 \ \vee \ 1-p \textbf{ at } 17. \quad (14)$$

Combining Formulas (13) and (14) by disjunction to remove `turn`, results in Formula (12).

Using the PSP rule on (12) and (10), we obtain

$$p \textbf{ in } \{11 \ldots 16\} \ \wedge \ 1-p \textbf{ in } \{11 \ldots 16\} \ \textbf{LT} \langle 10 \rangle (1-p \textbf{ at } 17 \wedge p \textbf{ in } \{11 \ldots 16\}) \vee p \textbf{ at } 17.$$

This combines with (11) by transitivity and disjunction to

$$p \textbf{ in } \{11 \ldots 16\} \ \wedge \ 1-p \textbf{ in } \{11 \ldots 16\} \ \textbf{LT} \langle 2 \cdot cs + 25 \rangle \ p \textbf{ at } 17.$$

This formula has the same postcondition as Formula (11). We can therefore take the disjunction of the preconditions of this formula and (11), and take the maximum of the number of required rounds, giving

$$p \textbf{ in } \{11 \ldots 16\} \ \wedge \ 1-p \textbf{ in } \{11 \ldots 19\} \ \textbf{LT} \langle 2 \cdot cs + 25 \rangle \ p \textbf{ at } 17. \quad (15)$$

When thread $1-p$ is and remains at line 10, then $\neg \, \texttt{flag}[q]$ by *Jp1*, so thread p in 11–16 can reach line 17 in five steps using a similar variant function to *vf1*. It is here that the *second* disjunct of the guard of line 16 is used. As thread $1-p$ may move to line 11, we have

$$p \textbf{ in } \{11 \ldots 16\} \ \wedge \ 1-p \textbf{ at } 10 \ \textbf{LT} \langle 2 \cdot cs + 30 \rangle \ p \textbf{ at } 17. \quad (16)$$

Just one more step is needed for $1-p$ at line 20 to move to 10

$$p \textbf{ in } \{11 \ldots 16\} \ \textbf{LT} \langle 2 \cdot cs + 31 \rangle \ p \textbf{ at } 17. \quad (17)$$

In words, this means

Theorem 1
The entry protocol takes not more than $2 \cdot cs + 31$ rounds.

Remark 1
The following is a scenario in which thread p needs at least $2 \cdot cs$ rounds for its entry protocol. This scenario starts with $\texttt{turn} \neq p$. The other thread q arrives at line 11 after p does, but it executes lines 11 and 12 before p and therefore arrives at CS first. After execution of CS, thread q sets `turn` := p, and in the same round, it goes through NCS, and lines 11, 12 to CS again. In the next round, thread p observes that $\texttt{turn} = p$, and goes to line 11, but it can enter CS only after q has executed CS a second time. This shows that the summand $2 \cdot cs$ in Theorem 1 is best possible. It may well be that the constant 31 can be improved.

Remark 2
It is easy to see and verify that the exit protocol takes at most three rounds (lines 18, 19, and 20), and combined with Formula (1) giving

$$p \textbf{ in } \{17 \ldots 20\} \ \textbf{LT} \langle cs + 3 \rangle \ p \textbf{ at } 10$$

and then combined with Formula (17) giving

$$true \ \textbf{LT} \langle 3 \cdot cs + 34 \rangle \ p \textbf{ at } 10.$$

DOI: 10.1002/cpe

5. DEKKER'S ALGORITHM ITSELF

Because of the similarities between the original algorithm and new Dekker's algorithm, much of the previous proof can be adapted to prove correctness of Dekker's original algorithm given atomic reads and writes. The first step is to strip down the algorithm of Figure 4 by removing the second disjunct of the guard of line 16 and the flickering of `flag` in line 20. In view of the failure scenario described at the start of Section 2.1, this is the smallest possible change that shows the essence of Dekker's algorithm.

In order to prove the correctness of this modification, it is possible to use the proof script for the RW-safe algorithm in Figure 4. This script can be applied to the modified original version. Then it turns out that only the proof of Formula (16) fails, that is, the point where the first disjunction is used. This failure can be repaired by means of the additional invariant

$$Jp3: \ p \ \mathbf{in} \ \{13 \dots 16\} \ \wedge \ 1 - p \ \mathbf{at} \ 10 \ \Rightarrow \ \texttt{turn} = p.$$

Jp3 is not threatened by modification of `turn`, because `turn` is modified only in line 19 with the postcondition that some thread is at line 20. *Jp3* is therefore threatened only by a thread going from 12 to 13 or from 20 to 10. If thread p is at 12 and $1 - p$ is at 10, then thread p does not go to 13 because of *Jp1*. Predicate *Jp3* is preserved by step 20 because of the new invariant

$$Jp4: \ p \ \mathbf{at} \ 20 \ \Rightarrow \ \texttt{turn} = 1 - p.$$

Predicate *Jp4* is threatened only by the steps 18 and 19, and the flickering at 19 of a thread different from p. It is preserved because we can strengthen both MX and *Ip1* to include line 20:

$$MXa: \ p \ \mathbf{in} \ \{17 \dots 20\} \ \wedge \ 1 - p \ \mathbf{in} \ \{17 \dots 20\} \Rightarrow false,$$
$$Ip1a: \ p \ \mathbf{in} \ \{12 \dots 14\} \ \cup \ \{17 \dots 20\} \Rightarrow \texttt{flag}[p].$$

It is the flickering of `flag` in line 20, that makes *Ip1a* (and hence the other new invariants) invalid for the algorithm of Figure 4.

Finally, if the flickering of `turn` is removed at line 19, the test of line 18 becomes superfluous. When the innocent flickering of `flag` in lines 11 and 15 is removed, the algorithm in Figure 1(b) is obtained.

6. FIRST-COME FIRST-SERVE

The strongest fairness condition among threads is the first-come, first-served (FCFS) [6, p. 330]. FCFS is the same for two threads as for N: if thread p passes the *doorway* before thread q enters the doorway, p must execute the critical section before q. The problem is precisely defining the doorway; normally, it is the wait-free initial fragment of the entry protocol. However, the unstructured (Figure 1(a)), structured (Figure 1(b)), and RW-safe (Figure 4) versions each start with an unbounded waiting loop, and hence, no doorway exists. In the Doran variant (Figure 3), there appears to be a well-defined doorway: lines 11–14. However, the algorithm does not satisfy FCFS, because of the following scenario.

- Assume `turn` = 1;
- Threads 0 and 1 both execute 11 and 12;
- Thread 1 goes to 14, set `flag`[p] := *false*, then to 15, and starts waiting;
- Thread 0 goes to 17, 18, 19, sets `turn` = 1, then 20 and 10;
- Then thread 0 enters the doorway while thread 1 is waiting; and
- Thread 0 goes to 12, and enters the CS because thread 1 is not scheduled.

This unbounded overtaking (Section 4) violates FCFS because thread 0 enters the doorway while thread 1 is waiting, and yet, thread 0 enters the CS before thread 1.

DOI: 10.1002/cpe

7. N-THREAD TOURNAMENT

Tournament approaches are used to construct simple N-thread mutual-exclusion algorithms with a reduced amount of scanning and retry. In general, tournament approaches produce the fastest N-thread mutual-exclusion algorithms (see for comparison [3]). Furthermore, tournament approaches rely on a two-thread mutual-exclusion algorithm, so they provide a perfect testbed for the new Dekker algorithm.

A tournament uses divide-and-conquer, where only one of every D threads progresses to the next round, and the others busy wait. A tournament solution is based on an auxiliary solution (MX) for D threads, where D is a small number > 1; usually $D = 2$. A D-ary tree is constructed with height $\lceil \log_D N \rceil$, and each thread is assigned a starting position at one of the N leaves of the tree. To reach the CS, a thread repeatedly performs the entry protocol of MX, in competition with the siblings of its current node. When it wins the competition, it moves to the parent of its current node. When it reaches the root of the tree, it accesses the CS. After finishing the CS, a thread retraces its path from root to leaf (starting position) executing the exit protocols of MX at each node.

Because of discrete subdivisions (D), some mechanism must exist for non-powers of D players. There are two options:

1. Create a minimal tree with only N start nodes so some paths from leaf to root are shorter than others.
2. Round N to the next power of D resulting in non-existent threads marked as not participating (do not want in). These extra entries must be tested but never cause contention.

Both approaches introduce unfairness because of the unbalanced competition, that is, some threads do not have to compete with as many threads.

Figure 5(a) shows the maximal binary tree ($D = 2$) approach for N threads by rounding N to the next power of 2, and Figure 5(b) shows the minimal binary tree. For all trees, threads start at the leaves of the tree, and $\lceil \log_2 N \rceil$ levels of internal nodes to implement the tournament. Each node is a two-thread solution for mutual exclusion, and each thread is assigned to a particular leaf where it begins the mutual exclusion process. The two-thread algorithms ensures an arriving thread is guaranteed to make progress for the internal MX algorithm; that is, the loser at each node eventually becomes the winner and continues to the next level. Therefore, each thread eventually reaches the root of the tree and enters the CS. Tournament algorithms allow unbounded overtaking unless there is synchronization among the nodes of the tree. That is, a fast thread can enter the CS arbitrarily often down one branch, while a slow thread is working down another branch.

Whether a tournament algorithm is RW-safe depends on the MX algorithm, as it contains all communication between the threads. Because the new Dekker's algorithm is the only known two-thread solution with RW-safeness, it can now be used to create a class of RW-safe tournament algorithms. Then, the RW-safe and RW-unsafe tournament algorithms can be compared for performance differences to determine the cost of RW-safeness.

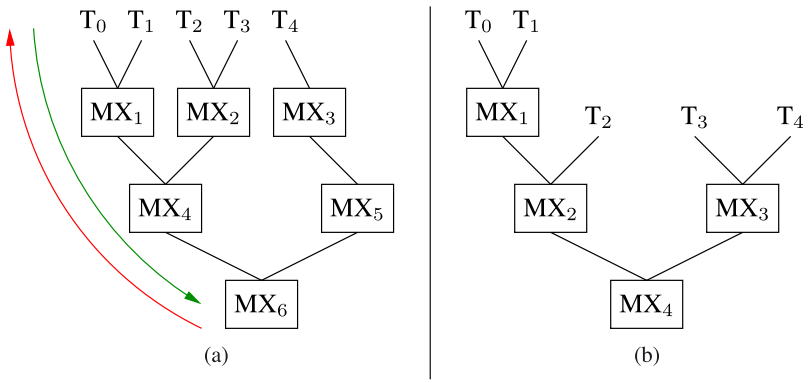

Figure 5. N-thread tournament tree, $N = 5$, $D = 2$: (a) maximal and (b) minimal tree sizes.

DOI: 10.1002/cpe

8. PERFORMANCE EXPERIMENT[¶]

Two performance experiments, maximal contention and minimal contention, are used to compare the performance differences among several two-thread solutions and two tournament algorithms using different two-thread algorithms at each node. The maximal-contention experiment tests the algorithms in the worst case scenario where all (but one) threads are spinning in the lock. The minimal-contention experiment tests the algorithms in the best case scenario where the lock has been provisioned for N threads, but only one thread is active, that is, the thread does not have to wait in the lock. These extremes give a general sense of how a locking algorithm performs. Often an algorithm is designed to optimize only one of these scenarios. Finally, a standard atomic *hardware-assisted* algorithm is presented to contrast performance with software solutions based solely on atomic read/write (or RW-safeness with no atomic requirement). The code to reproduce the experiments is publicly available [14], and the algorithm implementations are checked by the Relacy Race Detector [15].

8.1. Experimental setup

The performance experiments have a test harness that creates T pthread worker threads and uses the algorithms constructed for N threads, where $1 \leqslant T \leqslant N$, and $N = 2$ for two-thread solutions and $N \leqslant 32$ for N-thread solutions. The *maximal contention* experiment uses $T = N$; the *minimal contention* experiment uses $T = 1$. After thread creation, the harness blocks for a fixed period, t, and then sets a global stop flag to indicate the experiment is over. The T threads repeatedly attempt entry into the following self-checking CS until the stop flag is set:

```
void CriticalSection( const unsigned int id )   {
    static volatile unsigned int CurrTid;       // current thread in CS
    CurrTid = id;                               // RACE
    for ( int = 1; i <= 100; i += 1 )           // delay
        if ( CurrTid != id ) abort();           // mutual exclusion violation?
}
```

The shared variable `CurrTid` holds the id of the thread currently in the CS, and it is initialized at the beginning of *CriticalSection* to the calling thread's id. A thread then loops 100 times pretending to perform the CS, but at the end of each iteration, the thread compares its id with the one in `CurrTid` for any change. If there is a change, mutual exclusion has been violated, and the program is stopped.

During the t seconds of the experiment, each thread counts the number of times it enters the CS. The higher the aggregate count, the better an algorithm, as it is able to process more requests for the CS per unit time (throughput). When the stop flag is set, a worker thread stops entering the CS and atomically adds its subtotal entry-counter to a global total entry-counter. When the harness unblocks after t seconds, it busy waits until all threads have noticed the stop flag and added their subtotal to the global counter, which is then stored. Five identical experiments are performed, each lasting 20 s. The median value of the five results is plotted. The thread identifiers are randomized across the processors to avoid inadvertent correlations between thread start-up order and geographic placement on the system, which can influence false sharing in the lock data structures.

The *minimal contention* experiment measures the cost of *fast* access within an algorithm as N increases. To ensure the single thread exercises all aspects of an algorithm, it is assigned different start-points on each access to the CS by randomly changing its thread id. The randomness is accomplished using approximately 64 pseudo random thread ids, where 64 is divided by N to obtain R repetitions, for example, for $N = 5$, $R = 64/5 = 12$. Each of the 12 repetitions is filled with five random values in the range, $0..N-1$, without replacement, for example, 0 3 4 1 2. There are no consecutive thread ids within a repetition, but there may be between repetitions. The thread cycles through this array of ids during an experiment.

[¶]This section is based on [3, § 21], as the same experimental setup is used.

DOI: 10.1002/cpe

The performance experiments were run on three different multicore hardware systems to determine if there is consistency across platforms:

1. Supermicro AS-1042G-TF with four sockets, each containing an AMD Abu Dhabi 6380 16 core 2.5 GHz, equals 64 cores, running Linux v3.13.0-49, compiling with gcc 4.9.2;
2. Supermicro SYS-8017R-TF+ with four sockets, each containing an Intel Xeon 8 core 2.6 GHz E5-4620V2, equals 32 cores, turbo-boost on, running Linux v3.13.0-49, compiling with gcc 4.9.2; and
3. Oracle single-socket SPARC T2+ with eight cores, each core has two pipelines, and each pipeline supports (multiplexes) four logical processors, running Solaris 10, compiling with gcc 4.9.2.

Only two sockets are used on the AMD, and all four on the Intel for the 32 core experiments; hence, NUMA effects with respect to accessing shared data occur. The SPARC T2+ is a single-node UMA-architecture.

The major architecture difference between the two ×86 (Intel/AMD) servers are as follows: interconnect, QPI versus HyperTransport, and cache-coherence protocol (MESI/MESIF versus MOESI). As well, Intel processors have hyper-threading, but it is not used.

The major difference between the ×86 and SPARC architectures is communication cost, because of the single versus multi-socket architectures. This difference in communication cost has a significant effect on software solutions for mutual exclusion because the algorithms communicate heavily via shared memory. Therefore, when communication cost is high, placement of threads is crucial to reduce the cost of shared access, that is, pack cores then sockets. However, the default scheduling policies for both Linux and Solaris assumes low sharing among threads (which is normally the correct assumption) and hence, scatter threads across sockets and cores to prevent inter-core resource competition, such as increased cache pressure, which slows execution. Because the single-socket SPARC has very low communication costs, thread placement is largely irrelevant; hence, Solaris placement policy does not affect testing the software algorithms on the SPARC T2+ (but would on other SPARC processors). Unfortunately, given the high communication costs on the ×86 machines, the thread placement policy of Linux is the worst possible approach to demonstrate differences among the software algorithms and results in unusual behavior and significant jitter. Therefore, for the ×86 experiments, threads are explicitly pinned (using affinity) to cores and then sockets, which clears up irregularities in the data and more closely represents single-socket processors on embedded systems, where these algorithm might be used.

To contrast different two-thread solutions in a realistic test with other software algorithms, the tournament algorithms TaubenfeldBuhr and PetersonBuhr [3] are used to exercise two two-thread algorithms to compare performance with new RW-safe Dekker algorithm. In addition, Lamport's bakery is included as it was the first software algorithm with RW-safeness and FCFS service, and hence starvation free. Finally, one atomic hardware-assisted algorithm is included to give a sense of overall performance. The algorithm is Mellor-Crummey and Scott (MCS) [16], which augments a software solution with an atomic fetch-and-store and compare-and-assign instruction. MCS is *fast* (constant time for one thread) and FCFS, and hence starvation free. Faster hardware-assisted locks exist but most have starvation. A significant effort was spent to optimize each algorithm, including the use of *fastpath* for branch prediction [17].

8.2. Implementation correctness

Figure 1 shows the necessary implementation additions for preventing incorrect optimizations by the compiler and hardware. The variant of the algorithm in Figure 4 has the same additions in identical locations. These additions are discussed in detail.

The **volatile** qualifiers control static compiler transformations, such as movement, elision, and caching of fetched values into registers. For example, at line 15, if the compiler caches `turn` or `flag`[1 - *p*] into registers for the **await** loop, changes to these variables in memory are not seen, resulting in no progress. The **volatile** qualifier prevents the caching, forcing a variable to be loaded

DOI: 10.1002/cpe

on each reference. Because the **volatile** qualifier is associated with a variable rather than its specific usages, the compiler may restrict some valid optimizations, resulting in reduced performance.

The two fences control dynamic hardware transformations. The implementations are run on ×86 and SPARC processors, both using a TSO memory model, which relaxes write to read program order [18]. Hence, reads can be moved before writes for disjoint variables. For a sequential program, this is a legitimate optimization. For a concurrent program, it violates program order, increasing the possibilities for interference beyond what is considered in the verification discussed in Sections 3 and 4.

In general, fences are inserted after every write to a shared variable that is followed by a read from a shared variable. In cases where we can reason about safety, some fences are optional and removed to boost performance by allowing the hardware to perform its optimizations. For the purposes of verification, some optional fences are retained so there is only a single algorithm to verify, instead of multiple versions based on possible code movement. For the first fence of Figure 4, a scenario is presented to show that removal of the fence can lead to violation of mutual exclusion. For the second fence, its insertion is optional, but verification in Sections 3 and 4 only applies to the implementation if the fence is present. Similarly, the second optional fence in Figure 1(b) is required for verification in Section 5.

The first fence prevents the read of `flag`[1 - p] at line 12 from moving before the read of `flag`[p] at line 11, as in:

```
        temp = flag[1 - p] ; // move read before disjoint write
11      flag[p] := (flickering) 1 ;
12      if temp = 0 then break ; endif
```

This relaxation allows the scenario where p_1, p_2 reach line 10(a) simultaneously and read that the other thread does not want into the CS. Then both threads write their intents at line 11 and enter the CS, violating mutual exclusion. If the threads read each others intents *after* being assigned, one or both sees the others intent set.

The second fence prevents the read of `flag`[1 - p] at line 15 from moving before the write of `flag`[p] at line 24, as in:

```
        temp = flag[1 - p] ; // move read before disjoint write
15      flag[p] := (flickering) 0 ;
16      while temp ≠ 0 ∧ turn ≠ p do temp := flag[1 - p] endwhile ;
```

where `temp` is reloaded on each iteration of the **await**. This fence is optional because postponing `flag`[p] := 0 only keeps other threads waiting longer, but eventually the store occurs. Similar arguments can be applied to the second fence in Figures 1(a), (b), and 3. The third fence in Figure 3 is required because it provides the same effect as the first fence with respect to variable `flag`. A slight performance gain comes from removing the optional fences in the algorithm implementations, so all performance experiments are performed with optional fences removed.

8.3. Experimental results

Figure 6(a), (c), and (e) shows the entry count results from the maximal performance experiment for each two-thread software algorithm on the ×86 and SPARC architectures, respectively. Figure 6(b), (d) and (f) shows the entry count results from the minimal performance experiment, that is, an access with no contention, one thread, but $N = 2$. Seven two-thread software algorithms are examined.

- Peterson [19, Figure 1, p. 115] and Tsay [20, Figure 1, p. 397], which are examples of write-race solutions using atomic write to prevent livelock and starvation. Write-race solutions cannot be RW-safe because of the atomic write but are often shorter and easier to understand because of the simple mechanism to break ties and prioritize entry.
- Dekker's original and structured from Figure 1, Doran from Figure 3, RW-safe Dekker from Figure 4, which are examples of alternation (e.g., `turn`) to prevent livelock and starvation.
- Kessels [21, p. 137], which is an example of a read-race solution using atomic reads to prevent livelock and starvation. Read-race solutions cannot be RW-safe because of the atomic read.

DOI: 10.1002/cpe

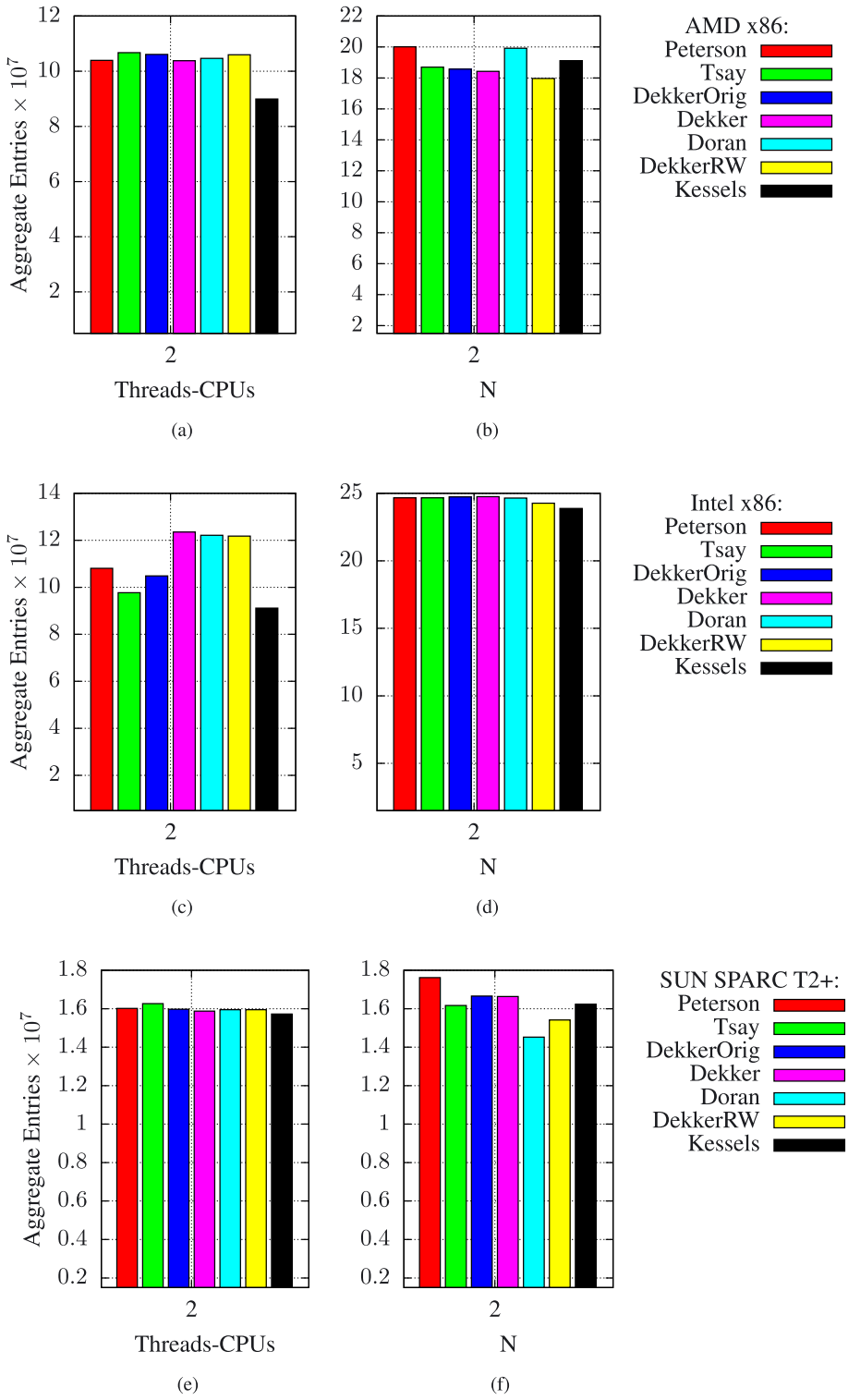

Figure 6. Critical-section entry-counts, 20 s, measure of algorithm performance, higher value is better: (a) max contention: $T = N = 2$; (b) min contention: $T = 1$, $N = 2$; (c) max contention: $T = N = 2$; (d) min contention: $T = 1$, $N = 2$; (e) max contention: $T = N = 2$; and (f) min contention: $T = 1$, $N = 2$.

DOI: 10.1002/cpe

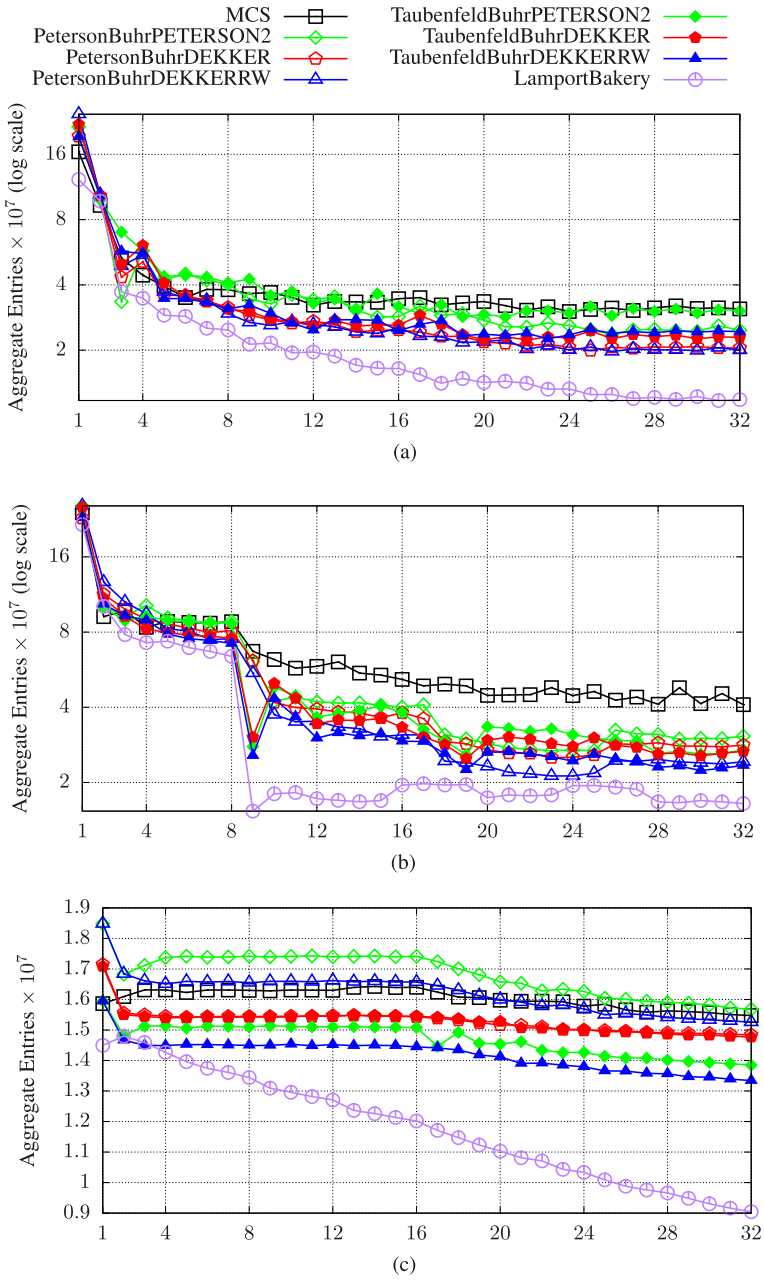

Figure 7. Critical-section entry-counts, maximal contention: $T = N = 1..32$, 20 s, algorithm performance, higher value is better: (a) AMD ×86: Threads-CPUs; (b) Intel ×86: Threads-CPUs; and (c) SUN SPARC T2+: Threads-CPUs.

To reduce the number of lines in the tournament graphs, the following two-thread algorithms are *not* shown: Tsay, Doran, and Kessels. All two-thread algorithms were tested, and all results were in tight groupings illustrated by the remaining two-thread algorithms: Peterson, structured Dekker and RW-safe Dekker.

Figure 7(a), (b), and (c) shows the entry count results from the maximal performance experiment for each algorithm on the AMD ×86, Intel ×86 and SUN SPARC T2+ architectures, respectively. The graph for ×86 uses a log scale because the results range by an order of magnitude, which compresses the results, making it difficult to see differences among them. Figure 8(a), (b), and (c) plots the relative standard deviation, $rc_v = \frac{\sigma}{\mu} \times 100$, for the maximal performance experiment, which is a percentage of the coefficient of variation (c_v) representing a normalized measure of dispersion of

DOI: 10.1002/cpe

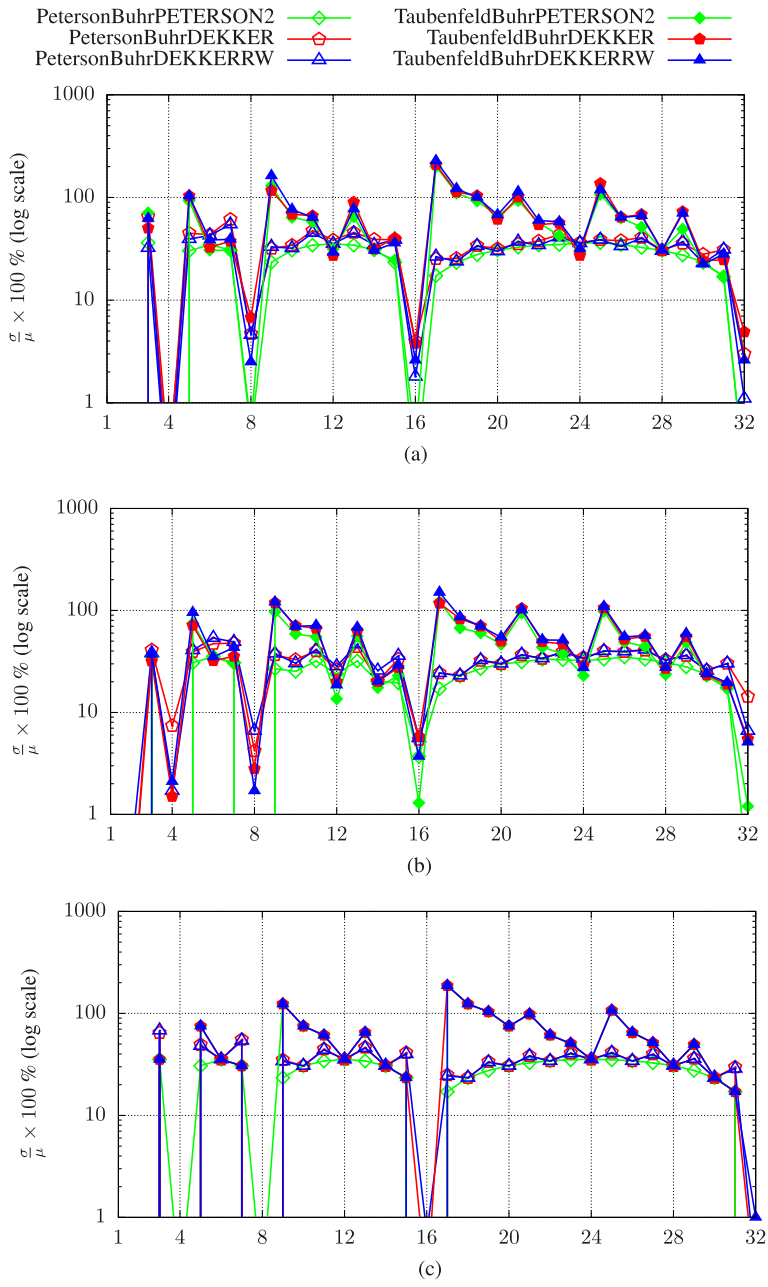

Figure 8. Relative standard deviation, maximal contention: $T = N = 1..32$, 20 s, fairness among threads, where 0% is prefect fairness. Algorithms not shown have less than 1% unfairness, for example, FCFS algorithms: (a) AMD ×86: Threads-CPUs; (b) Intel ×86: Threads-CPUs; and (c) SUN SPARC T2+: Threads-CPUs.

fairness for each algorithm. If an algorithm is perfectly fair, then the count values for each thread are essentially equal (modulo small differences at start-up and close-down), resulting in an rc_v of essentially zero. The more entry counts differ, the higher the percentage of unfairness. The graph is in log scale to spread out the curves into ranges: 1–10%, 10–100%, and 100–1000%. Figure 9(a), (b), and (c) shows the entry count results from the minimal performance experiment, that is, an access with no contention, one thread, but $N = 1..32$. As N increases, more steps are required by most of the algorithm even when only one thread is accessing the CS. For example, MCS is $O(1)$ with six accesses (three reads, three writes), but the tournament algorithms take $O(\log N)$, resulting in steps at powers of 2.

DOI: 10.1002/cpe

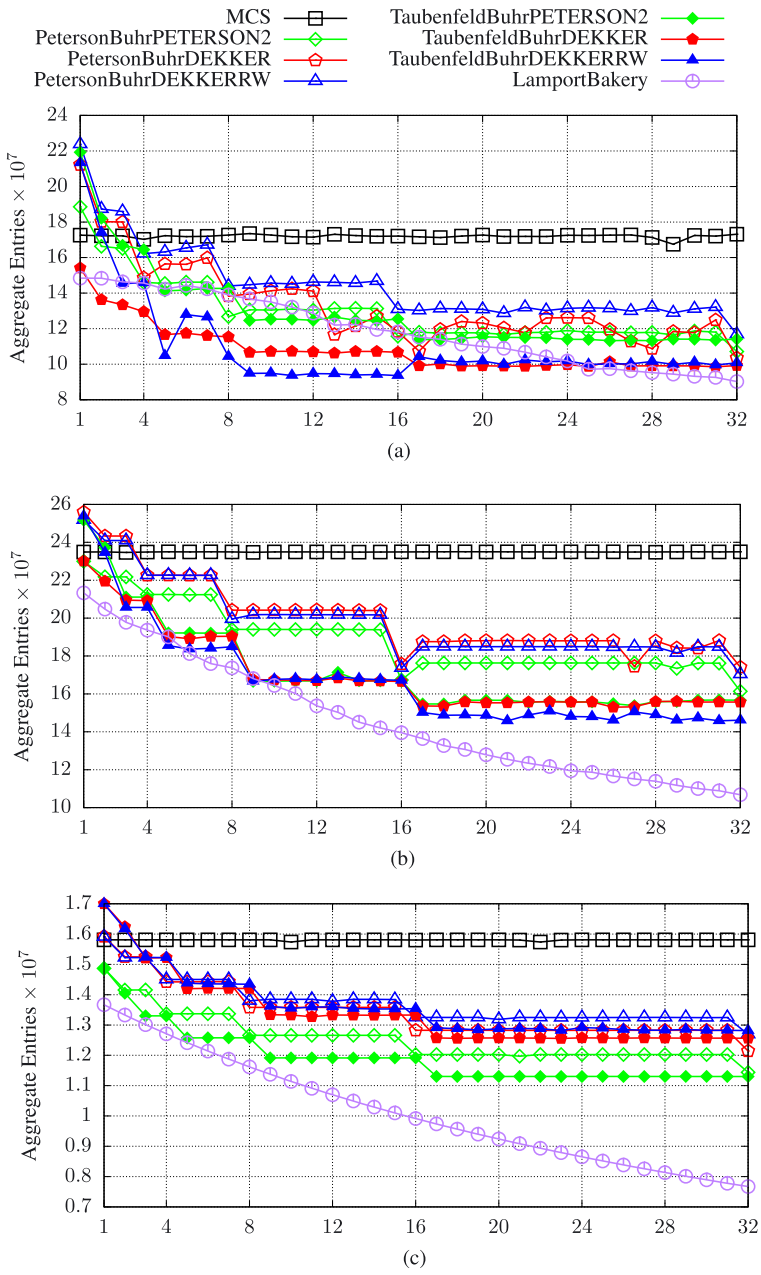

Figure 9. Critical-section entry-counts, minimal contention: $T = 1$, $N = 1..32$, 20 s, higher value is better: (a) AMD ×86: N; (b) Intel ×86: N; and (c) SUN SPARC T2+: N.

8.4. Experimental evaluation

Figure 6 shows maximal and minimal experimental results for the seven two-thread algorithms. Results are discussed based on the different computer environments.

AMD ×86 Performance is very similar for both maximal contention and minimal contention. The only outlier is Kessels' algorithm for maximal contention, which is slowest because of the complex entry-protocol for the read-race. Without any cache pressure in the minimal contention experiment, performance is similar.

Intel ×86 Performance for maximal contention indicates the Dekker-based algorithms have a slight advantage, while Kessels' race algorithm is slowest. Performance for minimal contention is very similar.

DOI: 10.1002/cpe

SUN SPARC T2+ Performance is very similar for both maximal contention and minimal contention.

Figure 7 is the maximal ($T = N$) contention results for the two tournament algorithms PetersonBuhr and TaubenfeldBuhr using three different two-thread algorithm, structured Dekker, RW-safe Dekker and Peterson, plus MCS and Lamport bakery.

AMD ×86 Performance is in three groups. First is MCS and both tournaments using Peterson 2-thread, where TaubenfeldBuhr is equal to MCS and PetersonBuhr is equal until 16 processors. Second is the tournaments using both Dekker 2-thread algorithms, which exhibit similar results. Third is LamportBakery, which is slowest across the entire range of processors because of significant scanning and retry in the entry protocol. Notice the small perturbations at or after powers of two processors, where the 2-ary tournament algorithms change tree height.

Intel ×86 Performance is in three groups. First is MCS, which is equal with the tournaments until 8 processors and about 20–30% faster to 32 processors. Second is the tournaments from 8 to 32 processors with a slight advantage for Peterson followed by structured Dekker and RW-safe Dekker. Third is LamportBakery, which is slowest across the entire range of processors.

SUN SPARC T2+ Performance is in three groups, noting that PetersonBuhr tournament does better than TaubenfeldBuhr in most cases. First is a tight group of PetersonBuhr with Peterson two-thread and RW-safe Dekker, plus MCS. Second is PetersonBuhr and TaubenfeldBuhr with structured Dekker, with TaubenfeldBuhr with Peterson two-thread and RW-safe Dekker slightly below. Third is LamportBakery, which is slowest across the entire range of processors.

There is little or no effect at the 16-core NUMA boundary for the AMD and Intel experiments. There are some effects at the powers-of-two boundaries because of the binary trees used in the tournament algorithms. However, nothing special is performed in the algorithms to be NUMA aware, so the algorithms and/or the experiment structure is immune to NUMA issues.

Figure 8 is the fairness results of the maximal ($T = N$) contention results for the two tournament algorithms PetersonBuhr and TaubenfeldBuhr using three different two-thread algorithm, structured Dekker, RW-safe Dekker and Peterson, plus MCS and Lamport bakery. MCS and Lamport bakery are FCFS, implying perfect fairness, and hence are not displayed. All three graphs are similar. (Note, some curves are so similar, one may not appear to be present in the graph.) For tournament algorithms, the rc_v is zero for powers of 2 and rises and falls in between these values depending on the non-power of 2 mechanism used. PetersonBuhr has four smooth bumps (between 1–4, 4–8, 8–16, and 16–32 processors) with a small amount of unfairness in the range 10–30% because of the minimal binary tree. TaubenfeldBuhr have more unfairness right after a power of 2, which then diminishes as N is increased to the next power of 2 (saw-tooth pattern), and hence are in the range 10–200%.

Figure 9 is the minimal ($T = 1, N = 1..32$) contention results for the two tournament algorithms PetersonBuhr and TaubenfeldBuhr using three different two-thread algorithm, structured Dekker, RW-safe Dekker and Peterson, plus MCS and Lamport bakery. The graphs all have the following pattern: MCS is flat because its fastpath is seven accesses regardless of N; tournament algorithms logarithmically step-down (reduced performance) at powers of two as the depth of the tree increases; Lamport bakery linearly decreases as scanning is a function of N. The only anomaly is that the AMD results are more compressed than the Intel and SPARC, with some jitter. Again, there is little or no effect at the 16-core NUMA boundary for the AMD and Intel experiments. All of the stepping is at powers-of-two boundaries because of the binary trees used in the tournament algorithms.

The key observation is that the additional checks in RW-safe Dekker do not disadvantage the algorithm in comparison with other two-thread algorithms. As well, the N-thread tournament algorithms are competitive with the hardware-assisted MCS algorithm, with only atomic RW or no atomic RW.

DOI: 10.1002/cpe

9. CONCLUSION

It is interesting that there are still subtle nuances in the original two-thread solution to mutual exclusion. The lack of RW-safeness for Dekker's algorithm was a surprise, and finding a RW-safe version was a challenge. The new RW-safe version can be used in N-thread tournament algorithms to produce high-performance RW-safe N-thread solutions. The RW-safe tournament algorithms are only slightly slower than the best RW-unsafe algorithms, which are competitive with the hardware-assisted MCS algorithm. All of which are faster than Lamport's bakery algorithm, which was the first algorithm with RW-safeness. These new RW-safe N-thread algorithms can be used in low cost/power hardware environments without basic atomic read/write or complex atomic instructions, allowing embedded system developers to construct threaded software on a single or multicore system with pre-emptive scheduling.

ACKNOWLEDGEMENTS

We thank Aaron Moss for his detailed comments on the proofs. Peter Buhr was funded by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. Dijkstra EW. Solution of a problem in concurrent programming control. *Communications of the ACM* 1965; **8**:569.
2. Dijkstra EW. Co-operating sequential processes. In *Programming Languages*, Genuys F (ed.). NATO Advanced Study Institute, Academic Press: London, New York etc., 1968; 43–112. Also EWD123 (1965).
3. Buhr PA, Dice D, Hesselink WH. High-performance N-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience* 2014; **27**:1–51. (Available from: http://dx.doi.org/10.1002/cpe.3263) [Accessed on March 2015].
4. Chandy KM, Misra J. *Parallel Program Design: A Foundation*. Addison–Wesley: Boston, MA, USA, 1988.
5. Lamport L. On interprocess communication. Parts I and II. *Distributed Computing* 1986; **1**:77–101.
6. Lamport L. The mutual exclusion problem. Parts I and II. *Journal of the ACM* 1986; **33**:313–348.
7. Zuo Wang, Jiaxing Li. An intelligent multi-port memory. *Symposium on Intelligent Information Technology Application Workshops, Shanghai, China,* IEEE Computer Society, Los Alamitos, CA, USA, December 2008; 251–254.
8. Buhr PA, Harji AS. Concurrent urban legends. *Concurrency and Computation: Practice and Experience* August 2005; **17**(9):1133–1172.
9. Doran RW, Thomas LK. Variants of the software solution to mutual exclusion. *Information Processing Lettters* July 1980; **10**(4/5):206–208.
10. Anderson JH, Gouda MG. Atomic semantics of nonatomic programs. *Information Processing Lettters* 1988; **28**: 99–103.
11. Owicki S, Gries D. An axiomatic proof technique for parallel programs. *Acta Informatica* 1976; **6**:319–340.
12. Apt KR, de Boer FS, Olderog ER. *Verification of Sequential and Concurrent Programs*. Springer: New York, 2009.
13. Hesselink WH. Mutualexclusion by four shared bits with not more than quadratic complexity. *Science of Computer Programing* 2015; **102**(0):57–75.
14. Buhr PA, Dice D, Hesselink WH, December 2013. Concurrent-locking, (Available from: https://github.com/pabuhr/concurrent-locking) [Accessed on March 2014].
15. Vyukov D. Relacy race detector, 2009, (Available from: http://www.1024cores.net/home/relacy-race-detector) [Accessed on March 2014].
16. Mellor-Crummey JM, Scott ML. Algorithm for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* February 1991; **9**(1):21–65.
17. Lalja DJ. Reducing the branch penalty in pipelined processors. *Computer* July 1988; **21**(7):47–55.
18. Adve SV, Gharachorloo K. Shared memory consistency models: a tutorial, DEC Western Research Laboratory: 250 University Avenue, Palo Alto, California, 94301, U.S.A., 1995; **29**: 66 –76. (Available from: http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf) [Accessed on December 1996].
19. Peterson GL. Myths about the mutual exclusion problem. *Information Processing Letters* 1981; **12**:115–116.
20. Tsay YK. Deriving a scalable algorithm for mutual exclusion. In *Distributed Computing*, Kutten S (ed.)., LNCS. Springer: Berlin Heidelberg, 1998; 394–407.
21. Kessels JLW. Arbitration without common modifiable variables. *Acta Informatica* 1982; **17**:135–141.

DOI: 10.1002/cpe