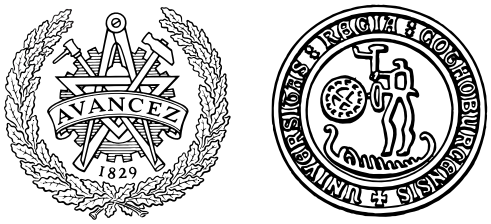
Technical Report no. 2005:16

Reactive Spin-locks: A Self-tuning Approach

Phuong Hoai Ha, Marina Papatriantafilou, Philippas Tsigas

CHALMERS | GÖTEBORG UNIVERSITY



Department of Computing Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2005

Department of Computing Science and Engineering
Division of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Technical Report no. 2005:16
ISSN: 1652-926X

Göteborg, Sweden, September 2005

Abstract

Reactive spin-lock algorithms that can automatically adapt to contention variation on the lock have received great attention in the field of multiprocessor synchronization, since they can help applications achieve good performance in all possible contention conditions. However, in existing reactive spin-locks the reaction relies on (i) some fixed *experimentally tuned thresholds, which may get frequently inappropriate in dynamic environments like multiprogramming/multiprocessor systems, or (ii) known probability distributions of inputs.*

This paper presents a new reactive spin-lock algorithm that is completely self-tuning, which means no experimentally tuned parameter nor probability distribution of inputs are needed. The new spin-lock is based on both synchronization structures of applications and a competitive online algorithm. Our experiments, which use the Spark98 kernels and the SPLASH-2 applications as application benchmarks, on a multiprocessor machine SGI Origin2000 and on an Intel Xeon workstation show that the new self-tuning spin-lock helps the applications with different characteristics nearly achieve the best performance in a wide range of contention levels.

1 Introduction

Multiprocessor systems aim at supporting parallel computing environments, where processes are running concurrently. In such parallel processing environments the interferences among processes are inevitable. Many concurrent processes may cause high traffic on the system bus (or network), high contention on memory modules and high load on processors; all these slow down process executions. These interferences generate a variable and unpredictable environment to each process. Such a variable environment consequently affects interprocess-synchronization methods like spin-locks. Some complex spin-locks such as MCS queue-lock are good for high-load environments, whereas others such as *test-and-test-and-set* lock are good for low-load environments [10]. This fact raises a question on constructing reactive spin-locks that can adapt to load variation in their surrounding environment so as to achieve good performance in all conditions.

There exist reactive spin-lock algorithms in the literature [2, 10]. Spin-lock using the *test-and-test-and-set* operation with exponential backoff ($TTSE$) [2] is an example: every time a waiting process reads a busy lock, i.e. there is probably high contention on the lock, it will double its backoff delay in order to reduce the contention. Another reactive spin-lock that can switch from spin-lock using $TTSE$ to a complex local-spin queue-lock when the contention is considered high was suggested in [10].

However, these reactive spin-locks suffer some drawbacks. First of all, their reactive schemes rely on either some experimentally tuned thresholds or known probability distributions of some inputs. Such *fixed* experimental threshold-values may frequently become inappropriate in variable and unpredictable environments such as multiprogramming systems. Assumption on known probability distributions of some inputs is not usually feasible. Further, the reactive spin-locks do not adapt to synchronization characteristics of applications and thus they are inefficient for different applications. We observe that characteristics of applications such as delays inside/outside the critical sections have a large impact on which spin-lock will help the applications achieve the best performance. Lim's reactive spin-lock [10], which switches to $TTSE$ [2] when contention is low and to MCS queue-lock [11] when contention is high, was showed inefficient to some real applications [8]. A good reactive spin-lock should not only react to the contention variation on lock, but also adapt to a variety of applications with different characteristics.

These issues motivated us to design a new reactive spin-lock that requires neither experimentally tuned thresholds nor probability distributions of inputs. The new spin-lock moreover adapts itself to applications, keeping its good performance on different applications.

We classify spin-locks into two categories: *arbitrating locks* such as ticket-locks and queue-locks and *non-arbitrating locks* such as TAS locks. *Arbitrating locks* are locks that identify who is the next lock holder in advance. The rest of spin-locks are *non-arbitrating locks*.

Arbitrating locks and non-arbitrating locks each have their own advantages. Arbitrating locks prevent processors from causing bursts in network traffic as well as high contention on the lock. This is because they avoid the situation that many processors concurrently realize the lock available and thus concurrently try to acquire the lock [2, 1, 6, 8, 11]. Although the advantages of arbitrating spin-locks have been studied so widely, the following advantages of non-arbitrating spin-locks have not been studied deeply. Non-arbitrating locks have two interesting properties: i) tolerance to crash failures in the lock-competing phase, the *Entry section* in Figure 1, and ii) ability of exploiting *locality/cache* and the underlying system supports such as page migration [9]. The lock holder can re-acquire the lock and re-use the exclusive shared data many times before the lock is acquired by another processor, saving time used for transferring the lock and the shared data from one to another. From experiments we observe that the non-arbitrating locks is favored by applications with the critical section much larger than the non-critical section (cf. Figure 1) to exploit locality/cache whereas the arbitrating locks is favored by ones with the critical section much smaller than the non-

while $true$ **do** *Noncritical section*; *Entry section*; *Critical section*; *Exit section*; **od**

Figure 1. The structure for parallel applications

critical section to avoid bursts both in network traffic and in memory contention. This implies that characteristics of a specific application can decide which kind of locks helps the application achieve better performance. (Further discussions on the advantages of both lock categories continue in Section A.1.)

1.1 Contributions

We designed and implemented a new reactive spin-lock with the following properties:

- It is completely self-tuning: neither experimentally tuned parameters nor probability distributions of inputs are needed. The new reactive scheme automatically adjusts its backoff delay reasonably according to load on the lock as well as characteristics of applications. The scheme is built on a competitive online algorithm. What it needs from the system is only the ratio of the latency of remote memory reference to the latency of level 1 cache reference, which is available in documents about the system architecture.

- It combines the advantages of both arbitrating and non-arbitrating spin-locks. In order to achieve this property, the new spin-lock does not use *strict* arbitrations like ticket-locks, but instead introduces a *loose* form of arbitration. This allows the spin-lock to be able to exploit locality. Combining a *loose* arbitration with a suitable reactive backoff scheme helps the new spin-lock achieve the advantages of the both categories.

In addition to proving the correctness of the new spin-lock, in order to test its feasibility we ran experiments using Spark98 kernels [12] and SPLASH-2 applications [15] as application benchmarks on an SGI Origin2000, a well-known commercial ccNUMA system, and on a popular workstation with two Intel Xeon processors. These experiments showed that in a wide range of contention levels the new reactive spin-lock performed nearly as well as the best, which was manually tuned for each benchmark on each system.

The synchronization primitives related to our algorithms are *fetch-and-add (FAA)* and *compare-and-swap (CAS)*, which are available in most recent systems either in hardware like Intel, Sun machines or in software like SGI machines. The definitions of the primitives are described in Figure 2.

The rest of this paper is organized as follows. Section 2 describes the problem and then models it as an online problem. Section 3 presents a new competitive algorithm for reactive spin-locks. Section 4 presents correctness proofs of the new spin-lock. Section 5 presents a heuristic for the new reactive spin-lock to adapt to synchronization characteristics of applications. Section 6 presents the performance evaluation of the new reactive spin-lock and compares the spin-lock with the representatives of arbitrating and non-arbitrating spin-locks using the application benchmarks. Finally, Section 7 concludes this paper. Another section that describes our problem analysis, which led and motivated this work, can also be found in the Appendix. We think this section can be of interest by itself.

2 Problem and model

The theoretical model of parallel applications in our research is typically described as a set of threads with the structure shown in Figure 1 [1]. We consider a system with P sequential processes running on P processors. We assume that each process runs on one processor, which is common in recent systems such as SGI Origin2000. In this case, we do not need to switch the process state from spinning to blocking in the *Entry section* (cf. Figure 1), i.e. there is no context-switching cost in the spin-lock overhead [7].

First of all, we determine the upper/lower bounds of backoff delays between two consecutive spins. Let "delay base" $base_l$ of a lock l be the average interval in which the lock holder keeps the lock locally before yielding it to another process. In order to obtain a high probability of spinning a free lock, a backoff delay $delay_i$ between two consecutive spins of a process p_i on the lock l should not be smaller than $base_l$, $base_l \leq delay_i$. On the other hand, according to Anderson [2] the upper bound for backoff delays should equal the number of processes potentially interested in acquiring the lock so that the backoff has the same performance as statically assigned slots when there are many spinning processes. This implies $delay_i \leq P \cdot base_l$, where P is the number of processes potentially interested in acquiring the lock. In conclusion,

$$base_l \leq delay_i \leq P \cdot base_l \tag{1}$$

where $delay_i$ is a time-varying measure.

Secondly, we look at the problem of how to compute a reasonable $delay_i$ for the next backoff every time a waiting process p_i observes a busy lock. In the $TTSE$ spin-lock

TAS(x) $atomically\{$ $oldx \leftarrow x$; $x \leftarrow 1$; **return** $oldx$; $\}$ /* *init:* $x \leftarrow 0$ */
FAA(x, v) $atomically$ $\{$ $oldx \leftarrow x$; $x \leftarrow x + v$; **return**$(oldx)$ $\}$
CAS(x, old, new) $atomically$ $\{$ **if**$(x = old)$ **then** $x \leftarrow new$; **return**$(true)$; **else return**$(false)$; $\}$

Figure 2. Synchronization primitives, where x is a variable and v, old, new are values.

[2], the backoff delay $delay_i$ is doubled up to some limit every time a waiting process reads a busy lock. In fact, the backoff scheme in the $TTSE$ spin-lock comes from Ethernet's backoff scheme for networks with characteristics different from spin-locks. In networks the cost to a collision is equal and independent of the number of processes whereas in the spin-locks the cost depends on the number of participating processes [2]. Therefore, the backoff scheme in $TTSE$ spin-lock is not competitive and its performance heavily relies on how well its base/limit values are chosen.

In the rest of this section we analyze the problem and then model it as an online game between a malicious adversary and a player.

Let "delay surplus" $surplus_i$ of a process p_i be

$$surplus_i = (P \cdot base_l - delay_i) \qquad (2)$$

We have $0 \leq surplus_i \leq (P-1) \cdot base_l$. Like $delay_i$, $surplus_i$ is a time-varying measure.

Definition 2.1. *A* load-rising (resp. load-dropping) transaction phase *is a maximal sequence of processes' subsequent visits at the lock with monotonic non-decreasing (resp. non-increasing) contention level on the lock*[1]*. A load-rising phase ends when a decrease in contention is observed. At that point, a load-dropping phase begins.*

Our goal is to design a reactive non-arbitrating spin-lock whose backoff delay (or delay in short) is dynamically and optimally adjusted to contention variation on the lock. This implies that we need to minimize two opposite factors: i) the delay between a pair of lock release and lock acquisition due to the backoff and ii) the communication bandwidth used by spinning processes as well as the load on the lock.

This is an online problem. Whenever a spinning process p_i observes a load increase on the lock, it has to decide whether it should increase its $delay_i$ now. If it increases its delay too soon, it will waste time on a long backoff delay when the lock becomes available. If it does not increase its delay in time, it will cause the same problems as spin-lock using TTS such as high network traffic, high contention on the lock, which consequently delay the lock holder to release the lock. If the process knew in advance how contention on the lock would vary in the whole competing period, it would have been able to find an optimal solution. However, there is no way for processes to know that information, the information about the future in an unpredictable environment.

We are interested in designing a deterministic online algorithm against a malicious adversary for the spin-lock problem. In such kind of problems, randomization cannot improve competitive performance [5]. For deterministic online algorithms the adversary with the knowledge of the algorithms generates the worst possible input to maximize the competitive ratio. The adversary creates transaction phases that fool the player, a process competing for the lock, to increase/decrease his delay incorrectly. This makes the player end up with a bad result whereas the adversary still achieves the best result.

Figure 3 illustrates how the adversary can create such transaction phases. Assume that the adversary designs A as an optimal load-point to increase the delay and B as an optimal load-point to decrease the delay. Since the adversary has both knowledge of the deterministic algorithm used by the player and full control on creating load inputs, the malicious adversary can add a sequence of load-rising points $\cdots \leq a_1 \leq a_2 \leq \cdots \leq a_n < A$ that fools the player to increase his delay up to the maximum before the load reaches A (i.e. to fool the player to increase his delay too soon). When the player observes a load increase on the lock, he will increase his delay according to his deterministic algorithm, and eventually his delay reaches the maximum at some point a_i before the load reaches point A.

The goal of online/offline algorithms is to maximize $\mathcal{P} = \sum_{t \in T_j} \Delta surplus_{i,t} \cdot l_t$ for each transaction phase T_j, where l_t is the load at time $t \in T_j$ and $\Delta surplus_{i,t}$ is the additional amount of surplus that the player/process p_i spends at load l_t. The idea behind this goal is to put a longer delay at a higher contention level reasonably. For the game in Figure 3, the adversary achieves the best value $\mathcal{P}$ at A since he will use all his surplus "budget", $(P-1) \cdot base_l$, at the suitable load-point A where l_t becomes maximum in the load-rising transaction phase T_j. That means the player increases his delay too soon, wasting time on a long backoff delay when the lock becomes available.

Similarly, the adversary can fool the player on the load-dropping phase from A to B by adding a sequence of load-dropping points $b_1 \geq b_2 \geq \cdots \geq b_m > B$. When the player observes a load decrease on the lock, he decreases his delay, and eventually his delay reaches the minimum at

[1] The contention level on a lock is measured by the number of processes that are competing for the lock, cf. Section 3.

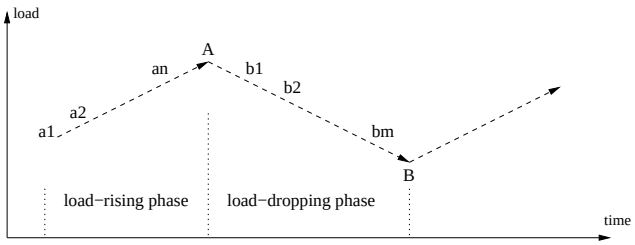


Figure 3. The transaction phases of contention variations on the lock.

some point b_j before the load reaches point B. That means the player decreases his delay too soon, causing high network traffic and high contention on the lock.

Lastly, we determine upper/lower bounds of load l_t on the lock. Load on the lock is the number of processes currently waiting for the lock, i.e. $l_t \leq P$. On the other hand, a process needs to delay only if it could not acquire the lock, so we have $1 \leq l_t \leq P$.

In summary, the spin-lock problem can be described as the following online game. With known upper/lower bounds of load l_t on the lock,$1 \leq l_t \leq P$, the player (a process i_i) needs to spend his initial delay surplus (e.g. $(P-1) \cdot base_l$) at l_t efficiently. Load l_t are unfolded on-the-fly and when a new value l_t is observed, a new period starts. Given a current load value, the player has to decide how much of his delay surplus should be spent at the current load, i.e. how much his current backoff delay should be lengthened at the current load.

3 The algorithm

In order to play against the malicious adversary, the player needs a *competitive* online algorithm for computing his backoff delay. When load on the lock increases, the player has to reduce his delay surplus, $surplus$, by exchanging it with another assest called $savings$. When load on the lock decreases, he increases $surplus$ by exchanging this $savings$ back to $surplus$.

The idea of our spin-lock algorithm is as follows. During a load-rising phase T_j, when the player observes a load increase on the lock, he increases his delay *just enough* to keep a bounded competitive ratio even if the load suddenly drops to the minimum in the next observation. The amount of time by which the player's delay increases is computed similarly to the *threat-based* method of [5]. The online algorithm for computing the delay can be described by the following rules:

- The delay is increased only when load on the lock is the highest so far in the present transaction phase.
- When increasing delay, increase *just enough* to keep the competitive ratio $c = P - \frac{P-1}{P^{1/(P-1)}}$, even if the load drops to the minimum in the next observation.

The amount of time by which the delay should increase is:

$$\Delta delay = \Delta surplus = initSurplus \cdot \frac{1}{c} \cdot \frac{load - load^{-}}{load - 1} \quad (3)$$

where $initSurplus$ is $surplus$ at the beginning of a load-rising transaction phase, $load$ is the present load on the lock observed by the player, and $load^{-}$ is the highest load on the lock before the present observation (cf. procedure $Surplus2Savings$ in Figure 4).

The online algorithm is presented via pseudocode in Figure 4. Every time a new load-rising transaction phase starts, the value $initSurplus$ is set to the last value of $surplus$ in the previous transaction phase (lines C2, C3). At the beginning of a transaction, load on the lock is initialized to $counter$ and $delay = counter \cdot base_l$, where $counter$, a sort of ordering tickets, shows how many processes are competing for the lock. The $counter$ is obtained when the process reads the lock at the first time (line A1). Each process chooses an initial $surplus$ with respect to its own ticket/counter (line A2)

$$initSurplus = (P - counter) \cdot base_l \quad (4)$$

This helps the new spin-lock partly prevent processes from concurrently observing a free lock, the worst situation for non-arbitrating spin-locks.

Symmetrically, in a load-dropping phase the amount of time by which the player's delay should decrease is computed by applying the same method with only one change, namely that the value of load on the lock $load$, which is decreasing, is replaced by the inverse $\frac{1}{load}$ (cf. procedure $Savings2Surplus$).

Finally, we briefly explain the whole spin-lock algorithm via pseudocode in Figure 4. In order to know the load on a lock, we need a counter to count how many processes are concurrently competing for the lock. If we used a separate counter, we would generate additional bottleneck beside the lock. Therefore, we used a single-word variable to contain both the lock and the counter (cf. $LockType$ in Figure 4).

A process p_i calls procedure $Acquire(L)$ when it wants to acquire lock L. The structure of the procedure is similar to the spin-lock using TTS except for the ways to compute the delay and to update the lock. First, p_i increases both values $\langle lock, counter \rangle$ by 1 (line $A1$). The lock L has been occupied if $L.lock \neq 0$. When spinning the lock locally (line A5), if p_i observes a free lock, i.e $L.lock = 0$, it will try to acquire the lock by increasing only field $L.lock$ by 1 (field $L.counter$ is kept intact, line A7). It will successfully

acquire the lock if no other processes have acquired the lock in this interval, i.e. $cond.lock = 0$ (line A8).

Process p_i calls procedure $Release()$ when releasing the lock. The procedure has to do two tasks atomically: i) reset the $lock$ field and ii) decrease the $counter$ field by 1. The CAS primitive can do these tasks atomically (line R2).

```
type LockType = record lock, counter : [1..MaxProcs]; end;
     LockStruct = record L : LockType; base : int; end;
     InfoType = record load⁻ : [1..MaxProcs];
                       phase : {Rising, Dropping};
                       surplus, initSurplus : int;
                       savings, initSavings : int; end;
private variables info : InfoType;
ACQUIRE(LockStruct pL)
A1  L := FAA(&pL.L, ⟨1, 1⟩); //increase counter,try to take lock
    if L.lock then //lock is occupied
A2    info.initSurplus := info.surplus :=
      (P − L.counter) · pL.base; //initialize variables
      info.initSavings := info.savings :=
      (L.counter · pL.base) · L.counter;
A3    delay := ComputeDelay(info, L.counter, pL.base);
      cond := ⟨1, 0⟩; //conditional variable for while loop
      do
A4      sleep(delay);
A5      L = pL.L; //read lock again
A6      if L.lock then //lock is still occupied
          delay := ComputeDelay(info, L.counter);
          continue;
A7      cond = FAA(&pL.L, ⟨1, 0⟩); //try to take lock
A8    while cond.lock;

int COMPUTEDELAY (InfoType I, int load, int base)
    FirstInPhase := False;
    if I.phase = Rising and load < I.load⁻ then
C1    I.phase := Dropping; I.initSavings := I.savings;
      FirstInPhase := True;
    else if I.phase = Dropping and load > I.load⁻ then
C2    I.phase := Rising; I.initSurplus := I.surplus;
      FirstInPhase := True;
C3  if I.phase = Rising then
      Surplus2Savings(I, load, FirstInPhase);
C4  else Savings2Surplus(I, 1/load, FirstInPhase);
C5  I.load⁻ := load;
C6  return (P · base − I.surplus);

SURPLUS2SAVINGS (InfoType I, int load, bool FirstInPhase)
    X := I.surplus; initX := I.initSurplus; Y := I.savings;
    rXY := load; rXY⁻ := I.load⁻;
    if FirstInPhase then
      if rXY > mXY · C then //mXY: lower bound of rXY
S1      ΔX := initX · 1/C · (rXY − mXY · C)/(rXY − mXY); //C: comp. ratio
    else
S2    ΔX := initX · 1/C · (rXY − rXY⁻)/(rXY − mXY);
S3  I.surplus := I.surplus − ΔX;
    I.savings := I.savings + ΔX · rXY;

SAVINGS2SURPLUS (InfoType I, 1/load, bool FirstInPhase)
/* Symmetric to procedure Surplus2Savings with:
    X := I.savings; initX := I.initSavings;
    Y := I.surplus; rXY := 1/load; rXY⁻ := 1/I.load⁻; */

RELEASE (LockType pL)
R1  do L := pL.L;
R2  while not CAS(&pL.L, L, ⟨0, L.counter − 1⟩);
    //release lock & decrease counter
```

Figure 4. The Acquire and Release procedures

Lemma 3.1. *In each load-rising/load-dropping phase, the new deterministic spin-lock algorithm is competitive with competitive ratio* $c = P - \frac{P-1}{P^{1/(P-1)}} = \Theta(\log P)$*, where* P *is the number of processes potentially interested in the lock.*

Proof. The proof is similar to that of the threat-based policy in [5] and is let out due to space constraints. □

Theorem 3.1. *The new spin-lock algorithm guarantees mutual exclusion and non-livelock. Its space complexity is* $\Theta(\log P)$ *for systems with* P *processors.*

Proof. The proof can be found Section 4. □

4 Correctness

The correctness of the new algorithm follows almost straightforward from its description. In particular, due to the atomicity properties of FAA and CAS, we have that:

Lemma 4.1. *The number of processors currently waiting for the lock* $L.counter$ *is counted correctly.*

Lemma 4.2. *The space need for the lock field of* $LockType$ *is* $\log(P)$ *for systems with* P *processors.*

Proof. Let Δt denote an interval since the field $lock$ of a lock L is increased to 1 at line $A1$ or $A7$ until it is reset to 0 at line $R2$. In Δt, a processor p_i can increase the field $lock$ by at most one. Indeed, if p_i increases $lock$ by 1 at line $A1$ or $A7$, it no longer increases $lock$ at line $A7$ because line $A7$ is executed only if $lock = 0$ (line $A6$); it cannot also increase $lock$ at line $A1$ because each processor only executes $A1$ once at the beginning of procedure $Acquire$.

Therefore, in Δt the $lock$ field is increased by at most P. That means the value of the lock field is never greater than P, the number of processors. □

Lemma 4.3. *The new reactive spin-lock allows only one processor to enter the critical section at a point of time.*

Proof. A processor p_i can enter the critical section only if the lock field of the value that p_i gets from FAA primitive at line $A1$ or $A7$ is 0. Due to the atomicity properties of FAA primitive, at one point of time at most one processor can observe that the lock field is 0, and can become the lock holder. Only when the lock holder exits the critical section, the field is reset to 0 (line $R2$), allowing another processor to enter the critical section. □

These lemmas imply the following theorem:

Theorem 4.1. *The new spin-lock algorithm guarantees mutual exclusion and non-livelock. Its space complexity is* $\Theta(\log P)$ *for systems with* P *processors.*

5 Estimating the delay bases

So far we have assumed that the basic interval $base_l$ in which a process p_i keeps the lock l locally before yielding it to other processes is known. This section describes how the new spin-lock estimates the $base_l$ based on characteristics of each parallel application such as delays outside/inside the corresponding critical section (cf. Definitions 5.3).

Like the *delay base* in the $TTSE$ spin-lock, the $base_l$ is just a basic value at the beginning from which the online algorithm in Section 3 starts to adjust the backoff delay according to contention variation. Instead of forcing programmers to tune the value manually, the new spin-lock estimates the value automatically.

First, we define terms used in this section.

Fairness: In order to evaluate fairness of spin-locks, we consider them on applications whose threads do the same task but on different data. Fairness is introduced to evaluate unbalanced situations where a thread may successfully acquire the lock many times more than the others may. Fairness is an interesting aspect of spin-lock algorithms, which may help the application gain performance in multiprocessor systems by utilizing all processors concurrently in high-load cases. Since threads cannot make any progress when waiting for the lock, only the lock holder utilizes one of system processors. If the lock holder continuously and successfully re-acquires the lock, only one processor will be used for useful task. In contrast, if the spin-lock is so fair that each thread can acquire the lock in turn, all threads will concurrently utilize system processors to execute their non-critical section task in parallel (cf. Figure 1).

Assume that in a period Δt there are N processors concurrently executing the code with structure as in Figure 1. These processors start and end outside Δt. That means we are only interested in the fairness for periods Δt in which all N processors *are concurrently and continuously competing* for the lock.

Definition 5.1. *Call n_i the number of times each of N processors p_i has successfully acquired a lock in a period Δt.* Fairness *of the lock in the period can be computed using the following formula:*

$$fairness_{\Delta t} = \frac{\sum_i n_i}{\max_i n_i \cdot N} \tag{5}$$

The lock that can keep its fairness in a shorter Δt is the better.

Overhead and delay: In most systems, the latencies of memory references vary with memory levels. Let the latency of accessing L1 cache be a time unit, we have the following definition:

Definition 5.2. Overhead *of yielding a cached variable such as a lock to another processor in order to achieve good fairness is:*

$$overhead = \frac{latency\ of\ remote\ memory\ reference}{latency\ of\ L1\ cache\ reference} \tag{6}$$

Definition 5.3. Delay outside a critical section (DoCS) *is the interval since the lock holder releases the lock in the Exit section until the first attempt to re-acquire it in the Entry section (cf. Figure 1).* Delay inside a critical section (DiCS) *is the interval when the lock holder is in the Critical section.*

In the new spin-lock, the delays outside/inside a critical section ($DoCS$/$DiCS$) are estimated by an individual process when needed. In fact, $DoCS$ and $DiCS$ influence the backoff delay strongly. Figure 8 shows that application performance degrades significantly if the backoff delay is chosen inaccurately (cf. tts ($TTSE$ with tuned thresholds) and $tts0$ ($TTSE$ without tuned thresholds) in Figure 8).

We have found a reasonable heuristic to estimate the delay base by $DoCS$.

The heuristic: *The delay base for a lock l, $base_l$, can be estimated by the delay outside the corresponding critical section, $DoCS$, using the following formula:*

$$base_l = \frac{a \cdot DoCS + b}{DoCS^2} \tag{7}$$

where a and b are constants.

Indeed, if the DoCS approaches 0, i.e. the whole execution time of the application is inside the critical section, the application should be executed by only one processor to reduce the cost of transferring data among processors, i.e. $base_l \to \infty$. In this case, the profit of concurrently executing non-critical sections on processors is too small compared to the cost of transferring critical data from one processor to another. On the other hand, if the DoCS approaches infinite, a processor after finishing its current iteration (cf. Figure 1) should immediately yield the lock to other processors so that other processors can use the lock, i.e. $base_l \approx 0$ [2]. In this case, without knowledge of interference pattern among processes/processors the lock holder should immediately yield the lock to others since he will almost not acquire the lock again due to $DoCS \to \infty$. Therefore, the function $g(x)$ to compute the delay base $base_l$ from $DoCS$ has the following form

$$y = g(x) = \frac{f_n(x)}{f_{n+k}(x)} \tag{8}$$

[2] Accurately, $base_l = DiCS$. Nevertheless, because $DoCS \to \infty$, i.e $DiCS$ is too small compared with $DoCS$, we ignore $DiCS$, i.e $base_l \approx 0$

where $k \geq 1$ is an integer and $f_n(x) = a_n x^n + \cdots + a_1 x^1 + a_0$

On the other hand, the benefit of successfully acquiring the lock, i.e. the period of using the lock locally, should not be smaller than the overhead of yielding the lock to another processor in order to support the fairness (cf. Definition 5.2). Therefore, $overhead \leq base_l$. If all processors keep the lock in the minimum time $base_l = overhead$ to minimize Δt in Definition 5.1, the time for the lock to visit all $P-1$ other processors and then come back to p_i is

$$\begin{aligned}&(base_l + transmission\ delay) \cdot P \\ &= (overhead + overhead) \cdot P = 2 \cdot overhead \cdot P\end{aligned}$$

If $DoCS = 2 \cdot overhead \cdot P$, this is an optimal situation. This is because each processor p_i always successfully acquires the lock when it needs, i.e. the lock comes back to p_i after $DoCS$, and all other processors can exploit p_i's interval $DoCS$ to successfully acquire the lock. Therefore, the chart of function $g(x)$ must contain a point $M = (2 \cdot overhead \cdot P, overhead)$.

Moreover, when $DoCS = overhead$, which is small, in order to support the fairness the $base_l$ should be long enough so that the ticket/counter in the new algorithm (Figure 4) can be accessed by $P-1$ other processors before the current lock holder gets another ticket, i.e.

$$\begin{aligned}base_l &= transmission\ delay \cdot (P-1) \\ &= overhead \cdot (P-1)\end{aligned}$$

Therefore, the chart of function $g(x)$ must contain a point $N = (overhead, overhead \cdot (P-1))$.

Since the chart of $g(x)$ must contain both points M and N, the simplest form of $g(x)$ that can satisfy this requirement is

$$y = g(x) = \frac{a \cdot x + b}{x^2} \tag{9}$$

where a, b are constants and can be found via points M and N. Each lock l in a parallel application has its own base $base_l$, which is estimated once at the beginning via the delay outside the corresponding critical section $DoCS$.

Applications using many small locks [3]: Timing functions are costly and thus the new spin-lock should estimate $base_l$ only for locks l with significant impact on the application performance, i.e. those are accessed many times during application execution. Moreover, in order to avoid oscillation at the beginning of application, which may make $base_l$ be estimated inaccurately, the new spin-lock starts to estimate $base_l$ after an interval that is long enough for all processes/processors to be able to acquire the lock l once. As discussed above, the benefit of acquiring the lock should be greater than the overhead of transferring the lock, so each processor should keep the lock in a period not smaller than $overhead$. Therefore, the new spin-lock starts to estimate $base_l$ after an interval of $2 \cdot overhead \cdot P$ since the beginning of the execution. In this initial interval, the new spin-lock uses the ticket lock with proportional backoff and $base_l$ is initialized to $overhead$. After the $base_l$ is estimated, the new spin-lock uses the reactive spin-lock in Figure 4.

[3]The *small* locks are locks that are used very few times and on which contention level is low.

6 Evaluation

Choosing non-arbitrating/arbitrating representatives : To keep graphs uncluttered we chose an efficient representative for each category (i.e. *arbitrating* and *non-arbitrating*).

We chose the ticket lock with proportional backoff ($TicketP$) as the representative for the *arbitrating lock* since: i) the $TicketP$ performs well for application benchmarks Spark98 (cf. Figure 8) and SPLASH-2 (cf. [8]) on the main platform used in our study and ii) from the fairness point of view, the $TicketP$ is not worse than the queue lock (cf. Figure 8). Although the ticket lock is considered not as scalable as MCS queue-lock since the former processes spin on centralized variables, this is not a performance issue for the ticket lock on recent machines with cache-coherent support as long as the backoff delay of the ticket lock is tuned well. Moreover, the ticket lock gains further performance due to its simplicity and fairness. The implementation of $ticketP$ was similar to Fig. 2 in [11], where the time unit was experimentally tuned for both the benchmarks and the evaluation systems to achieve the best performance.

For non-arbitrating spin-locks, we chose as the representative the $TTSE$ with backoff parameters tuned for both the benchmarks and the evaluation systems. The RH lock in [13] shows its advantages compared to the $TTSE$ lock only if the system has two nodes and the latency of local memory references within a node is much smaller than the latency of remote memory reference to the other node. Therefore, we think TTSE is a good representative. The source code for $TTSE$ was from [14].

Choosing application benchmarks In order to compare performance among different spin-lock algorithms, the application benchmark chosen should have highly contended lock, which will noticeably promote efficient lock algorithms (cf. Performance Goals for Locks in [4]). Therefore, we chose as our application benchmarks the shared memory program using locks lmv from the Spark98 kernel [12] and the applications from the SPLASH-2 suite [15]: Volrend, which uses one lock, instead of an array of locks $QLock$, to protect a global queue, and Radiosity. Both Volrend and Radiosity have highly unstructured access patterns to irregular data structures [15]. The Radiosity application has a special feature different from the Spark98 and the Volrend: it has too many *small* locks besides some high contention

locks. Therefore, the Radiosity is a "malicious" benchmark for complex spin-lock algorithms like the new reactive spin-lock. The input data for the benchmarks were $sf5.1.pack$ for the Spark98, $head.den$ for the Volrend and *-room* option for the Radiosity, which are the largest data sets available for the Spark98 and the Volrend, and the recommended data set for the Radiosity.

Platforms used in the evaluation: The main system used for our experiments was a ccNUMA SGI Origin2000 with twenty eight 250MHz MIPS R10000 CPUs with 4MB L2 cache each. The system ran IRIX 6.5 and it was used exclusively. In the system, each thread ran exclusively on one processor. The system latencies of memory references are available in [9].

We also used as an evaluation platform a popular workstation with two Intel Xeon 3GHz CPUs with 1MB L2 cache each. The workstation ran Linux kernel 2.6.8. Since each Xeon processor with hyper-threading technology can concurrently execute two threads, the workstation can concurrently execute four threads without preemption. The system latencies of memory references are available in [3].

We compared our new reactive spin-lock with $TTSE$ and $TicketP$, both of which were *manually tuned* for each application benchmark on each platform. The tuned parameters for both are presented in Figure 5. Contention on the lock was varied by changing the number of participating processors/threads. The execution times of the application benchmarks were measured.

6.1 Results

The new reactive spin-lock in Figure 4 involved in all locks with high contention[4]. Such locks play significant roles in application execution time and promote efficient spin-lock algorithms. Working on such high contention locks, processes always have to delay between two consecutive accesses. The new reactive spin-lock utilizes the delay interval to compute a reasonable value for the next delay. This is reason why even though the new reactive spin-lock appears quite heavy compared with the non-arbitrating/arbitrating representatives, it is actually efficient.

Figure 6 shows average execution times of applications Spark98, Volrend and Radiosity using $TTSE$ (tts), $TicketP$ ($ticket$) and the new reactive spin-lock ($reactive$) on the SGI platform. All the three charts show that the new reactive spin-lock approaches the best performances, which are the tts performance in the case of Spark98 and the $ticket$ performance in the cases of Volrend and Radiosity. Note that the new reactive algorithm *without tuning* performed similarly to the better of two representatives *with manual tuning* of non-arbitrating and arbitrating categories.

In the left chart on the Spark98 execution times, the reactive spin-lock approaches the best one, the $TTSE$. The reason why on the Spark98 $TTSE$ is better than $TicketP$ is as follows. In the Spark98, the $DoCS$ is not large and thus the Spark98 benchmark favors the spin-lock that exploits the *locality*, i.e. non-arbitrating spin-locks. With the large values $b_e = 50000$ and $l_e = 650000$ (cf. Figure 5), contention on the lock was kept low and the lock holder could re-acquire the lock and re-use the shared resource many times before the other processors re-tried to acquire the lock. This saved the time for transferring the lock as well as the shared data to another processor, the time for reading and writing data and the time for re-acquiring the lock because everything was cached locally. For the arbitrating spin-lock such as $TicketP$, all processors were in a waiting queue. Regardless of whether the distance between two consecutive processors in the waiting queue was too far, the lock and the shared data were transferred back and forth on the interconnect network, degrading the performance of $TicketP$ on the Spark98.

In the new spin-lock, the necessary backoff delay was computed reasonably by a competitive online algorithm that increased/decreased the backoff delay *just enough* to alleviate contention on the lock. The algorithm tried to kept changes as small as possible compared with the initial value. The initial value was large due to the small delay outside the critical section (cf. Section 5). Since the new reactive spin-lock is a non-arbitrating spin-lock, it got benefit from exploiting the locality like $TTSE$.

In the middle chart on the Volrend execution times, the reactive spin-lock still approaches the best one, the $TicketP$. The reason why on the Volrend $TicketP$ is better than $TTSE$ is as follows. Since the high contention lock in the Volrend has large $DoCS$ and small $DiCS$, $TTSE$'s backoff delay had to be small to minimize the interval from the last lock release to the next lock acquisition. Therefore, the Volrend had $b_e = 400$ and $l_e = 1400$ (cf. Figure 5), which are too small compared with those in the Spark98. $TTSE$ spinning the lock with such a high frequency generated high contention on the lock, degrading performance of the whole system as mentioned in [2, 1, 6, 8, 11]. Therefore, the Volrend benchmark favored arbitrating locks such as $TicketP$, which reduced overhead due to the arbitration among processors and thus reduced contention on the lock.

However, the Volrend did not degrade the new reactive spin-lock performance, a non-arbitrating spin-lock. This is because the reactive spin-lock automatically and reasonably adjusted backoff delay $delay_i$ for each processor p_i according to contention on the lock, keeping contention on the lock low. On the other hand, the fact that the initial delay for each processor p_i was proportional to the $ticket$ that p_i obtained

[4]The new reactive spin-lock algorithm does not involve in locks with low contention (cf. the last paragraph in subsection 5)

	Spark98	Volrend	Radiosity
$TTSE$/Origin2k	$b_e = 50000, l_e = 650000$	$b_e = 400, l_e = 1400$	$b_e = 200, l_e = 1200$
$TicketP$/Origin2k	$b_p = 100$	$b_p = 50$	$b_p = 130$
$TTSE$/Xeon	$b_e = 80, l_e = 700$	$b_e = 50, l_e = 350$	$b_e = 120, l_e = 1100$
$TicketP$/Xeon	$b_p = 60$	$b_p = 30$	$b_p = 90$

Figure 5. The table of manually tuned parameters for $TTSE$ and $TicketP$ in Spark98, Volrend and Radiosity applications on the SGI Origin2000 and the Intel Xeon workstation, where b_e, l_e are respectively $TTSE$'s *delay base* and *delay upper limit* for exponential backoff, and b_p is $TicketP$'s *delay base* for proportional backoff delays. The b_e, l_e and b_p are measured by the number of null-loops.

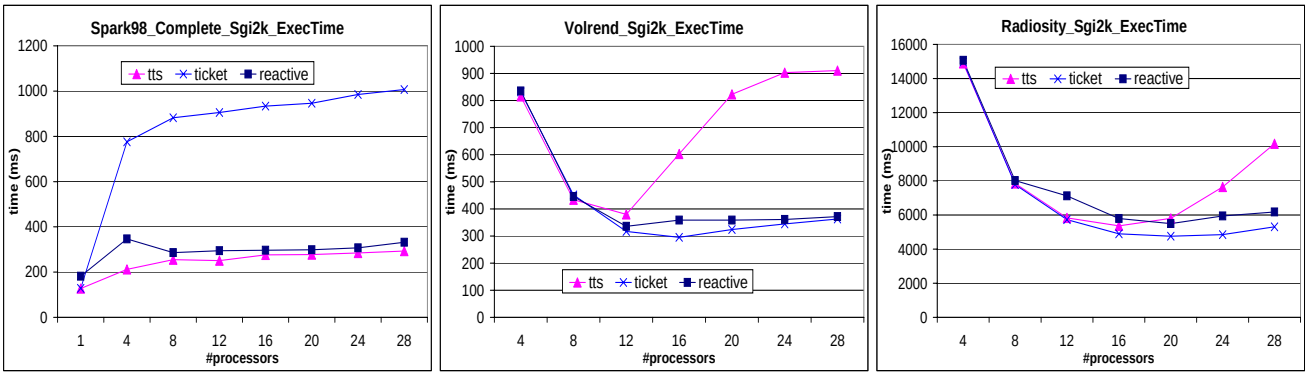


Figure 6. The execution time of Spark98, Volrend and Radiosity applications on the SGI Origin2000.

prevented partly processors from concurrently observing a free lock. These helped the new reactive spin-lock solve problems caused by high contention situation on the lock, which degraded the $TTSE$ performance.

Similar to the Volrend, the Radiosity benchmark shows that even applications with many *small* locks as Radiosity could not stop the reactive spin-lock algorithm from approaching the best performance, the $TicketP$ performance.

Experiments on the Intel platform showed a similar result: the new spin-lock performed as well as the best representative (cf. Figure 7). On this platform, performances of well-tuned $TTSE$ and $TicketP$ were similar for Volrend and Radiosity and were slightly different for Spark98. In the Spark98 benchmark, the new spin-lock still performed as the best. Although the benchmarks did not scale on the Intel platform, the result is still interesting since it shows how well the new spin-lock automatically tuned itself on different architectures compared with manually-tuned spin-locks.

In summary, the experiments on different platforms showed that the new reactive spin-lock without need of manually tuned parameters reacts well to contention variation as well as to a variety of applications. This helped the applications using the new reactive spin-lock approach the best performance gained by $TTSE$ or $TicketP$, the spin-locks that were *manually* tuned for both each application and each platform.

7 Conclusions

We have presented a new reactive spin-lock that is completely self-tuning, namely neither experimentally tuned thresholds nor probability distributions of inputs are required. The new spin-lock combines advantages of both arbitrating and non-arbitrating spin-locks. These features are achieved by a competitive algorithm for adjusting backoff delay reasonably to contention on the lock. Moreover, the new spin-lock also adapts itself to synchronization characteristics of applications to keep its good performance on different applications. Experimental results showed that the new spin-lock almost achieved the best performance on different platforms.

References

[1] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

[2] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

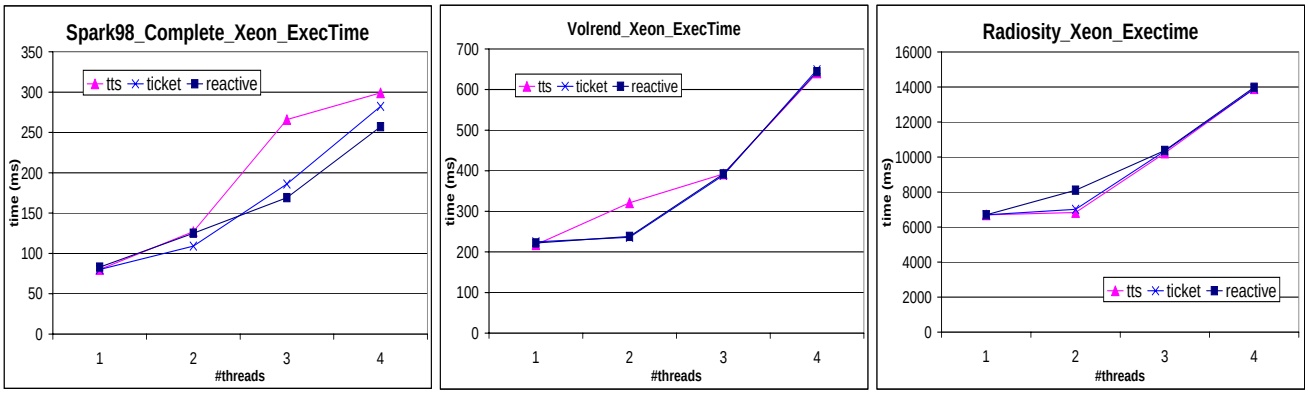


Figure 7. The execution time of Spark98, Volrend and Radiosity applications on a workstation with 2 Intel Xeon processors.

[3] D. Besedin. Detailed platform analysis in rightmark memory analyzer. part 6 - intel xeon. In *http://www.digit-life.com/articles2/rmma/rmma-nocona.html*, 2005.

[4] D. E. Culler, J. P. Singh, and A. Gupta. Parallel computer architecture: A hardware/software approach. *Morgan Kaufmann Publisher*, 1999.

[5] R. El-Yaniv, A. Fiat, R. M. Karp, and G. Turpin. Optimal search and one-way trading online algorithms. *Algorithmica*, 30(1):101–139, 2001.

[6] A. Kägi and D. B. J. R. Goodman. Efficient synchronization: Let them eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 170–180, June 2–4 1997.

[7] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 41–55, 1991.

[8] S. Kumar, D. Jiang, J. P. Singh, and R. Chandra. Evaluating synchronization on shared address space multiprocessors: Methodology and performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27,1, pages 23–34, May 1–4 1999.

[9] J. Laudon and D. Lenoski. The sgi origin: A ccnuma highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 241–251, 1997.

[10] B. Lim. Reactive synchronization algorithms for multiprocessors. *PhD. Thesis*, 1995.

[11] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[12] D. R. O'hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, Computing Science, Carnegie Mellon University, October 1997.

[13] Z. Radovic and E. Hagersten. Efficient synchronization for nonuniform communication architectures. In *Proceedings of the Proceedings of the IEEE/ACM SC2002 Conference*, page 13, 2002.

[14] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 44–52, 2001. Source code is available at `ftp://ftp.cs.rochester.edu/pub/packages/scalable_synch/PPoPP_01_trylocks.tar.gz`.

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Annual International Symposium on Computer Architecture (22nd ISCA'95), ACM SIGARCH Computer Architecture News*, pages 24–36, June 1995.

A Problem analysis

During the course of studying the behavior of spin-locks and using the Spark98 kernels [12] to evaluate them, we observed two other reasons beside network traffic that affect the performance of these locks. These are discussed below.

A.1 Tuning parameters and system characteristics

In general, besides the cost for experimentally tuning the parameters, the reactive spin-locks using tuned parameters cannot always achieve good performance, because the parameters depend on the system utilization, which in turn is affected by other applications running concurrently. Thus, tuned parameters at some point of time may become obsolete at a later point of time when they are used. Further, reactive spin-locks may also need to take care of properties of applications such as delays inside/outside the critical section when choosing locking protocols.

Regarding the algorithm-system interplay, there is also the issue of arbitrating vs. non-arbitrating locks, which imply different benefits, as explained in the introduction. RH lock presented in [13] tries to exploit the locality but the algorithm is limited to a system with two nodes [5]. In arbitrating locks, the lock and the data used in the critical section must be transferred from one processor to another according to their order in the waiting queue, regardless of how far the distance between these two processors is in the system. This generates high transmission cost. To the contrary, in non-arbitrating locks, the processors closest to the current lock owner, for instance processors in the same node in NUMA systems, have higher probability to acquire the lock because they will realize the lock available sooner. Moreover, when there are many requests on the lock from processors in the same node, the system may move the memory page containing the lock to the local memory of that node, giving these processors higher probabilities to acquire the lock the next time. Unlike the arbitrating locks, the non-arbitrating locks also have the ability to tolerate faults in the Entry section (cf. Figure 1). In the Entry section, the non-arbitrating locks prevent slow or crashed processors from blocking other fast processors.

Although arbitrating locks such as the ticket lock and queue-lock are considered fair locks in the literature, their fairness may still depends on the applications using the locks as well as on the architecture of the system on which the applications are running. Regarding the ticket lock, if processors in the same node of a NUMA system, whose local memory is storing the ticket variable, execute the iteration in Figure 1 so fast that they continuously get new tickets again before the ticket variable can be accessed by processors on other nodes, the ticket lock becomes unfair. Similar for the queue-lock, if processors in the same node of a NUMA system, whose local memory is storing the pointer used to enqueue the waiting queue, execute the iteration in Figure 1 so fast that they continuously enter the waiting queue before processors on other nodes have a chance to so, the queue lock may become unfair.

[5]The RH lock geometrically divides a group of processors into two subgroups in order to exploit the *locality* within a subgroup using $TTSE$.

A.2 Trade-offs

To see whether the above concerns have a sound basis, we conducted an experimental study. We used the Spark98 shared memory program lmv [12] on an SGI Origin3800. The system has 31 500MHz MIPS R14000 CPUs with 8MB L2 cache each. These experiments confirmed our observations. In order to compare performance among spin-lock algorithms, we need benchmarks where the contention level on the lock is high. Therefore, we used only one lock to synchronize updates of the result array in the Spark98 kernel. We used the largest pack file $sf5.1.pack$ [12] as input.

The left and the right charts in Figure 8 show the execution times and the fairness of the Spark98 kernel using MCS queue-lock (mcs), ticket lock with proportional backoff tuned for SGI Origin3800 ($ticket$), TTSE with backoff parameters tuned for SGI Origin3800 (tss), TTSE with backoff parameters mentioned in [14] ($tts0$) and the RH lock [13] with backoff parameters tuned for SGI Origin3800 (rh). The contention level on the lock is adjusted by changing the number of processors accessing it. For instance, in the case of 31 processors, there is the highest contention level on the lock. The source codes for $TTSE$ and MCS are from [14]. The implementation of $ticket$ is similar to Fig. 2 in [11].

From the left chart in Figure 8, we can see that the non-arbitrating locks such as $TTSE$ and RH both with tuned parameters outperform the arbitrating locks such as MCS queue-lock and ticket lock when the contention level increases. That is because the $TTSE$ and RH exploit the locality/caching among processors within the same node. Moreover, they do not suffer the *lock convoy* problem in the entry section.

The left chart also shows the problem of existing reactive spin-locks such as $TTSE$: their performance too much depends on the experimentally tuned parameters. Inaccurately chosen parameters will lead to bad performance as depicted in the left chart between the TTSE with parameters tuned for SGI Origin3800 (tss) and the TTSE with parameters mentioned in [14] ($tts0$). The latter is about 14 times slower than the former in the case of 31 processors, which is a big difference on application performance.

The right chart in Figure 8 shows the fairness of these spin-locks. Here, the processor first finishing its own task

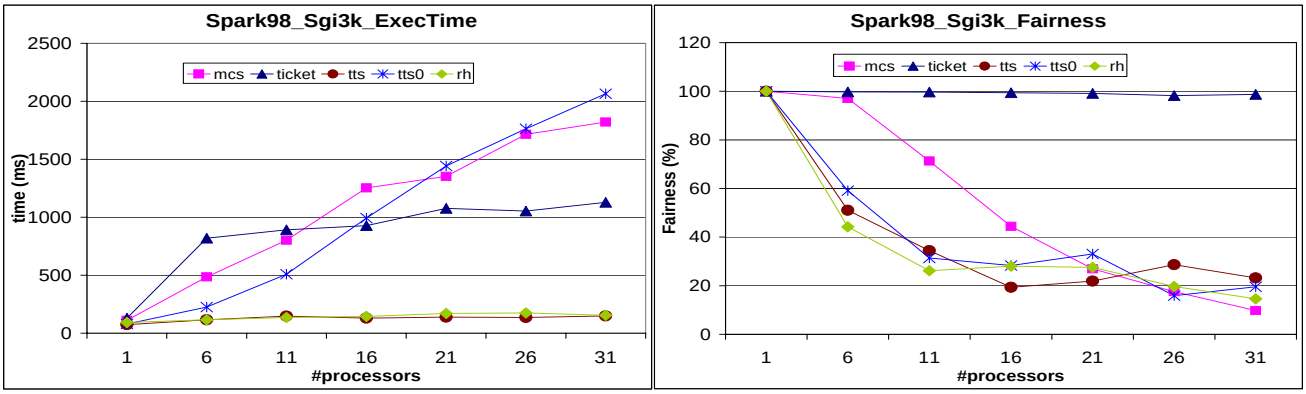


Figure 8. The execution time and the lock fairness of Spark98 benchmark on SGI Origin3800.

will send a signal to all other processors to stop and to count the number of times each processor has successfully acquired the lock. In this chart, the most interesting is the fairness of MCS queue-lock, which is normally considered fair lock. Beyond a certain number of processors, from fairness point of view, the MCS queue-lock does not seem better than other non-arbitrating locks such as $TTSE$ and RH. From the log file of the experiment in the case of 31 processors, we saw that a group of 16 processors connected together via the same router, had a number of lock accesses much greater than those of other processors connected via another router. That means that the group of 16 processors connected via the same router executed their own tasks so fast that they continuously successfully updated the pointer to enqueue before the pointer could be updated by other processors of another router. That means that even fair arbitrating spin locks cannot always ensure fairness for arbitrary applications running on arbitrary systems.