Reactive Spin-locks: A Self-tuning Approach

Phuong Hoai Ha, Marina Papatriantafilou, Philippas Tsigas
Department of Computer Science and Engineering,
Chalmers University of Technology
S-412 96 Göteborg, Sweden
{phuong,ptrianta,tsigas}@cs.chalmers.se

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 built on 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 applications with different characteristics achieve good 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 the MCS queue-lock are good for high-load environments, whereas others such as the *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, which implies 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.

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

Figure 1. The structure for parallel 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-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 can be found in [5].)

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.
- 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 [13] 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 new spin-lock uses synchronization primitives *fetch-and-add (FAA)* and *compare-and-swap (CAS)*, which are available in most recent systems either in hardware like Intel and Sun machines or in software like SGI machines.

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 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 5 concludes this paper. The correctness proof of the new spin-lock is in [5] due to space constraints.

2. Problem and model

At a high abstraction level, parallel applications in our research are typically described as a set of threads that run the software 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. 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, ac-

cording 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,

$$base_l \leq delay_i \leq P \cdot base_l \quad (1)$$

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 [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

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

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 [4]. 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 2 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 2, 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 some point b_j before the load reaches point B. That means the

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

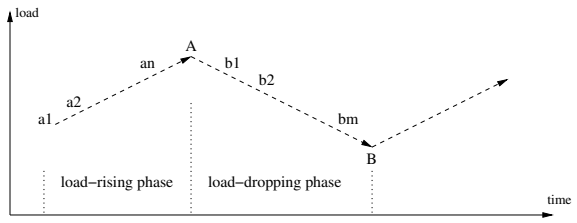

Figure 2. The transaction phases of contention variations on the lock.

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 p_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 assets 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 [4]. 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 3).

The online algorithm is presented via pseudocode in Figure 3. 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 3. 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 an additional bottleneck beside the lock. Therefore, we used a single-word variable to contain both the lock and the counter (cf. $LockType$ in Figure 3).

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).

```
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 3. The Acquire and Release procedures

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).

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 [4] 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 in [5] due to space constraints. □

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. In [5], we describe 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. The estimation is left out due to space constraints.

4. 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*. For non-arbitrating spin-locks, we chose as the representative the $TTSE$ with backoff parameters tuned for both the benchmarks and the evaluation systems.

Choosing application benchmarks: In order to compare performance among different spin-lock algorithms, the application benchmarks chosen should have highly contended locks, which will noticeably promote efficient lock algorithms (cf. Performance Goals for Locks in [3]). 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 [13]: Volrend, which uses one lock, instead of an array of locks $QLock$, to protect a global queue, and 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. The system ran IRIX 6.5 and it was used exclusively. We also used as an evaluation platform a popular workstation with two Intel Xeon 3GHz CPUs. The workstation ran Linux kernel 2.6.8.

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

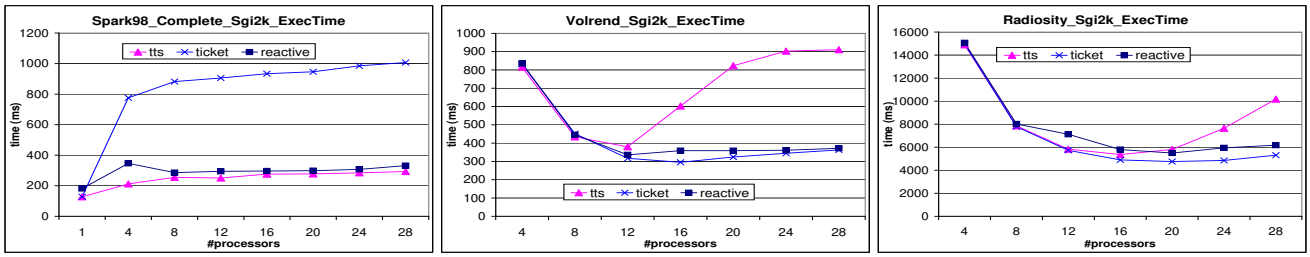

Figure 4. The execution time of Spark98, Volrend and Radiosity applications on the SGI Origin2000.

4.1. Results

The new reactive spin-lock in Figure 3 involved in all locks with high contention. 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 4 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. Experiments on the Intel platform showed a similar result: the new spin-lock performed as well as the best representative.

Further discussions on the evaluation as well as on the results can be found in [5].

5. 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 achieved good 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.

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

[4] 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.

[5] P. H. Ha, M. Papatriantafilou, and P. Tsigas. Reactive spin-locks: A self-tuning approach. *Tech. Report 2005:16, Computing Science, Chalmers Univ.*, 2005.

[6] A. Kägi and D. B. J. R. Goodman. Efficient synchronization: Let them eat QOLB. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, pages 170–180, 1997.

[7] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proc. of the 13th ACM Symp. 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 *Proc. of the Intl. Conf. on Measurement and Modeling of Computing Systems*, pages 23–34, 1999.

[9] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, 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. *Tech. Report CMU-CS-97-178,Comp. Science, Carnegie Mellon Univ.*, 1997.

[13] 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 Intl. Symp. on Computer Architecture*, pages 24–36, 1995.