Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1077–1090
The Journal of Systems and Software
www.elsevier.com/locate/jss

Efficient self-tuning spin-locks using competitive analysis ☆

Phuong Hoai Ha *, Marina Papatriantafilou, Philippas Tsigas

Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden

Received 28 February 2006; received in revised form 31 October 2006; accepted 22 November 2006
Available online 11 January 2007

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reactive spin-locks the reaction relies on (i) some *fixed* experimentally tuned thresholds, which may get frequently outdated in dynamic environments like multiprogramming/multiprocessor systems, or (ii) known probability distributions of inputs.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 both synchronization structures of applications and online algorithmic techniques. Our experiments, which use the Spark98 kernels and the SPLASH-2 applications as application benchmarks, on a multiprocessor machine SGI Origin2000 and an Intel Xeon workstation have showed that the new *self-tuned* spin-lock performs as well as the best of *hand-tuned* spin-lock representatives in a wide range of contention levels.

Keywords: Reactive synchronization; Spin-locks; Shared memory multiprocessors; Concurrent; Performance; Online algorithms

1. Introduction

Multiprocessor systems aim at supporting parallel computing environments, where processes are running concurrently. In such parallel processing environments the interference among processes is inevitable. Many concurrent processes may cause high traffic on the system bus (or network), high contention on memory modules and high loads on processors; all these slow down process executions. The interference generates a variable and unpredictable environment to each process. Such a variable environment consequently affects interprocess-synchronization methods like spin-locks. Some sophisticated spin-locks such as the MCS queue-lock (Mellor-Crummey and Scott, 1991) are good for high-load environments, whereas others such as the *test-and-test-and-set* lock (Agarwal and Cherian, 1989; Anderson, 1990) are good for low-load environments (Lim, 1995). This fact raises a question on constructing reactive spin-locks that can adapt to load variation in their environment so as to achieve good performance in all conditions.

There exist reactive spin-lock algorithms in the literature (Agarwal and Cherian, 1989; Anderson, 1990; Lim and Agarwal, 1994; Lim, 1995). Spin-lock using the *test-and-test-and-set* operation with exponential backoff (*TTSE*) (Agarwal and Cherian, 1989; Anderson, 1990) 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 sophisticated local-spin queue-lock when the contention is considered high was suggested in (Lim and Agarwal, 1994; Lim, 1995).

However, these reactive spin-locks suffer some drawbacks. First of all, their reactive schemes rely on either

☆ Expanded version of a preliminary result published in the Proceedings of the 8th IEEE International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'05), December 2005, pp. 33–39, IEEE press.

* Corresponding author. Tel.: +46 31 772 1024; fax: +46 31 772 3663.
E-mail address: phuong@cs.chalmers.se (P.H. Ha).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.11.016

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 like the time consumed in the critical section have a large impact on choosing appropriate spin-locks. Lim's reactive spin-lock (Lim, 1995), which switches to *TTSE* when contention is low and to the MCS queue-lock when contention is high, was showed inefficient to some real applications (Kumar et al., 1999). A good reactive spin-lock should not only react to the contention variation on the lock, but also adapt to a variety of applications with different characteristics.

These issues have 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.

1.1. Contributions

We have designed and implemented a new reactive spin-lock that 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 contention on the lock as well as characteristics of applications. The scheme is built as a competitive online algorithm. What it needs from the system is only the ratio of the latency of remote memory references to the latency of level 1 cache references, which is available in documents about the system architecture. The ratio represents the overhead for a processor to yield its cached variable such as a lock to another processor. 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 cache affinity (Karatza, 2000; Squillante and Lazowska, 1993; Torrellas et al., 1993; Vaswani and Zahorjan, 1991; Wang et al., 1997). Combining a *loose* arbitration with a suitable reactive backoff scheme helps the new spin-lock algorithm achieve the advantages of both cache affinity and low contention on the lock.

In addition to proving the correctness of the new spin-lock, in order to test its feasibility we ran experiments using the Spark98 kernels (O'hallaron, 1997) and the SPLASH-2 applications (Woo et al., 1995) as application benchmarks on an SGI Origin2000, a well-known commercial ccNUMA system, and a popular workstation with two Intel Xeon processors. These experiments showed that in a wide range of contention levels, the new reactive *self-tuned* spin-lock performed as well as the best of *hand-tuned* spin-lock representatives, which were manually tuned for each benchmark on each system.

The rest of this paper is organized as follows. Section 2 describes our problem analysis, which led and motivated this work. Section 3 models the spin-lock problem as an online problem. Section 4 presents a new competitive algorithm for reactive spin-locks. Section 5 presents correctness proofs of the new spin-lock. Section 6 presents a heuristic for the new reactive spin-lock to adapt to synchronization characteristics of the applications. Section 7 presents the performance evaluation of the new reactive spin-lock and compares the spin-lock with representatives of arbitrating and non-arbitrating spin-locks using the application benchmarks. Finally, Section 8 concludes this paper.

2. Problem analysis

2.1. Spin-lock categories

We classify spin-locks into two categories: *arbitrating locks* such as ticket-locks (Lamport, 1974) and queue-locks (Craig, 1993; Graunke and Thakkar, 1990; Magnussen et al., 1994; Mellor-Crummey and Scott, 1991) and *non-arbitrating locks* such as *TAS* locks (Agarwal and Cherian, 1989; Anderson, 1990). *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 (Anderson, 1990; Anderson et al., 2003; Kägi and Goodman, 1997; Kumar et al., 1999; Mellor-Crummey and Scott, 1991). 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) avoidance of lock-convoy[1] in the lock-competing phase, the *Entry section* in Fig. 1, and (ii) ability of exploiting *cache affinity* (Karatza, 2000; Squillante and Lazowska, 1993; Torrellas et al., 1993; Vaswani and Zahorjan, 1991; Wang et al., 1997). The lock holder, together with its neighbors in the same module, can re-acquire the lock and re-use the exclusive shared data many times before the lock is acquired by remote processors in another module, saving time used for transferring the lock and the shared data from one module to another.

From experiments we observe that when arbitrating/non-arbitrating spin-locks are well-tuned for each experimental setting, the contention on the lock is kept minimum and thus it does not significantly contribute to the perfor-

[1] Lock-convoy is the situation in which a slow processor prevents other fast processors from progress due to blocking.

```
while true do
        Noncritical section;
        Entry section;
        Critical section;
        Exit section;
od
```

Fig. 1. The structure of parallel applications accessing a critical section.

mance difference between arbitrating and non-arbitrating spin-locks. What makes the difference is the benefit resulting from parallel executing non-critical section, in the case of arbitrating spin-locks, and from locally accessing the lock and the shared data by processors connected to the same memory module in non-uniform memory access (NUMA) systems, in the case of non-arbitrating spin-locks. Therefore, applications with large critical section like Spark98 kernels (O'hallaron, 1997) are favored by non-arbitrating spin-locks due to cache affinity. Using non-arbitrating spin-locks, processors connected to the same memory module in NUMA systems can in turn acquire the lock and locally access the shared data, which consequently keeps the lock and the shared data in the local module. The benefit of parallel executing the non-critical section by moving the lock and the shared data around is less than the benefit of locally accessing the lock and the shared data. On the other hand, applications with large non-critical section like Volrend and Radiosity in the SPLASH-2 suites (Woo et al., 1995) are favored by arbitrating spin-locks due to parallel executing the non-critical section. This implies that characteristics of a specific application can decide which kind of locks helps the application achieve better performance.

2.2. 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. Furthermore, reactive spin-locks may also need to take care of properties of applications such as time intervals spent inside/outside the critical section when choosing locking protocols.

Regarding the algorithm-system interplay, there is also the issue of arbitrating versus non-arbitrating locks, which implies different benefits, as explained in the introduction. 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. In contrast, in non-arbitrating locks the processors closest to the current lock owner, for instance processors in the same module in NUMA systems, have higher probability to acquire the lock because they will realize the lock available sooner. Moreover, when there are many requests for 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.

Although arbitrating locks, such as ticket-locks and queue-locks, are considered as fair locks in the literature, their fairness may still depend 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 execute the iteration in Fig. 1 so fast that they continuously and repeatedly get new tickets before the ticket variable can be accessed by processors on other nodes, the ticket lock becomes unfair. Similarly for the queue-lock, if processors in the same node of a NUMA system execute the iteration in Fig. 1 so fast that they continuously enter the waiting queue before processors on other nodes have a chance to do so, the queue-lock may become unfair.

2.3. Experimental studies

To see whether the above concerns have a sound basis, we conducted an experimental study. We used the Spark98 shared memory program *lmv* (O'hallaron, 1997) on an SGI Origin3800. The system has 31 500 MHz 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 *sf*5.1.*pack* (O'hallaron, 1997) as input.

The left and the right charts in Fig. 2 show the execution times and the fairness of the Spark98 kernel using the MCS queue-lock (*mcs*), the ticket lock with proportional backoff tuned for the SGI Origin3800 (*ticket*), TTSE with backoff parameters tuned for the SGI Origin3800 (*tss*), TTSE with backoff parameters mentioned in (Scott and Scherer, 2001) (*tts*0) and the RH lock (Radovic and Hagersten, 2002) with backoff parameters tuned for the SGI Origin3800 (*rh*). The contention level on the lock is adjusted by changing the number of processors accessing it, for instance the case of 31 processors generates the highest contention level on the lock. The source codes for *TTSE* and *MCS* are from Scott and Scherer (2001). The implementation of *ticket* is similar to Fig. 2 in Mellor-Crummey and Scott (1991).

From the left chart in Fig. 2, we can see that the non-arbitrating locks such as *TTSE* and *RH* both with tuned parameters outperform the arbitrating locks such as the MCS queue-lock and the 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.

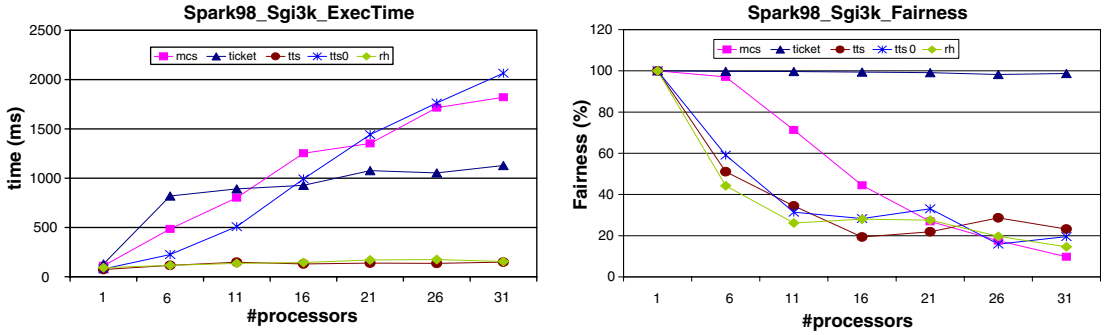

Fig. 2. The execution time and the lock fairness of the Spark98 benchmark on an SGI Origin3800.

The left chart also shows the problem of existing reactive spin-locks such as *TTSE*: their performance strongly 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 Scott and Scherer (2001) (*tts*0). 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 Fig. 2 shows the fairness of these spin-locks. Fairness is introduced to evaluate unbalanced situations where a processor may successfully acquire the lock many times more than the others may (cf. Definition 7 for the fairness formula used in the experiment). Here, the processor first finishing its own task 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 observation is the fairness of the MCS queue-lock, which is normally considered as a fair lock. Beyond a certain number of processors, from the 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.

All these factors are considered in the design of our new reactive spin-lock.

3. Modeling the problem

In this section we model the spin-lock problem as an online problem. The theoretical model of parallel applications in our research is typically described as a set of threads with the structure shown in Fig. 1 (Anderson et al., 2003). 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. Fig. 1), i.e. there is no context-switching cost in the spin-lock overhead (Karlin et al., 1991).

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/processor. In order to obtain a high probability of spinning on 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 $base_l$, $base_l \leqslant delay_i$. On the other hand, according to Anderson (1990) the upper bound of 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 $delay_i \leqslant P \cdot base_l$, where P is the maximum number of processors concurrently interested in the lock (i.e. the maximum contention on the lock). In conclusion,

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

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 (Anderson, 1990), 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 those of spin-locks. In networks the cost to a collision is equal and independent of the number of processes whereas in spin-locks the cost depends on the number of participating processes (Anderson, 1990). Therefore, the backoff scheme in the *TTSE* spin-lock is not competitive and its performance strongly depends on how well its base/limit values are chosen.

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

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

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

Definition 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.[2] A load-rising phase ends when a decrease in contention is observed. At that point, a load-dropping phase begins.

Our goal at this moment 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 competing measures: (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 the spin-lock using *TTS* like high network traffic and 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 be 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 (El-Yaniv et al., 2001). 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.

Fig. 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 $\cdots \leqslant a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n < A$ 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.

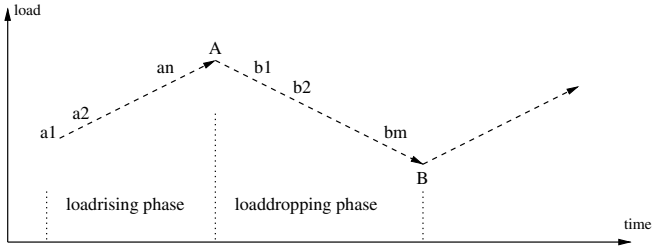

Fig. 3. The transaction phases of contention variations on the lock.

The goal of online/offline algorithms is to maximize the following objective function:

$$\mathscr{P} = \sum_{t \in T_j} \Delta surplus_{i,t} \cdot l_t \quad (3)$$

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 reasonably put a longer delay at a higher contention level. For the game in Fig. 3, the adversary achieves the best value $\mathscr{P}$ 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 $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_m > B$. 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 player decreases his delay too soon, causing high network traffic and high contention on the lock.

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

In summary, the spin-lock problem can be described as the following online game. With known upper/lower bounds of loads l_t on the lock,$1 \leqslant l_t \leqslant P$, the player (a process i_i) needs to spend his initial delay surplus (e.g. $(P-1) \cdot base_l$) at l_t efficiently. Loads 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.

[2] The contention level on a lock is measured by the number of processes that are competing for the lock, cf. Section 4.

4. The algorithm for computing backoff delay

In order to play against the malicious adversary, the player needs a *competitive* online algorithm (Borodin and El-Yaniv, 1998) for computing his backoff delay. When the load on the lock increases, the player has to reduce his delay surplus, *surplus*, by exchanging it with another asset called *savings*. When the load 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 (El-Yaniv et al., 2001). The online algorithm for computing the delay can be described by the following rules:

- The delay is increased only when the load is the highest so far in the present transaction phase.
- When increasing the 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 and stays there until the end of the transaction phase.

The amount of time by which the delay should increase is

$$\Delta delay = \Delta surplus = initSurplus \cdot \frac{1}{c} \cdot \frac{load - load^{-}}{load - 1} \tag{4}$$

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 Fig. 5).

In the rest of the section, we briefly describe the intuition behind the formulas to make the paper self-contained.

Consider a load-rising transaction phase. For the moment, assume that there is a (deterministic) online algorithm with the optimal competitive ratio c^* for computing the backoff delay and that c^* is known to the online player. At the end of this section, we present how the online player computes c^* based on P, the maximum contention on the lock. At the first load l_1 observed in the transaction phase, the player must increase the delay by time s_1 such that the competitive ratio c^* is guaranteed. Note that $c^* = \frac{\mathscr{P}_{Adversary}}{\mathscr{P}_{Player}}$, where $\mathscr{P}_{Adversary}$ and $\mathscr{P}_{Player}$ are the objective functions $\mathscr{P}$ (cf. Eq. (3)) of the adversary and the player, respectively. Since c^* is attainable by a deterministic online algorithm, there always exists such amount $s_1 \in (0, initSurplus]$ (formal analysis is presented below). If s_1 is always chosen as large as the whole delay "budget" of the player, i.e. *initSurplus*, the player will be unable to increase the delay further when the load continues increasing in the next step, which makes the competitive ratio c^* unattainable. Therefore, s_1 must be the *minimum* that guarantees the competitive ratio c^*. This results in the second rule mentioned above:

Rule 2: When increasing the delay, increase just enough to keep the competitive ratio c^*, even if the load drops to the minimum in the next observation and stays there until the end of the transaction phase.

Besides, since s_1 is chosen so that the competitive ratio c^* is guaranteed even if the load drops to the minimum in the next step and stays there until the end of the transaction phase, in the rest of the transaction phase the player can ignore all loads less than or equal to l_1 while still guaranteeing the competitive ratio c^*. This results in the first rule:

Rule 1: The delay is increased only when the load is the highest so far in the present transaction phase.

Similar argument can be used to inductively justify the amount s_i for the remaining steps of the transaction phase.

In summary, the algorithm for computing the backoff delay obeys the following rules

- The delay is increased only when the load is the highest so far in the present transaction phase.
- When increasing the delay, increase *just enough* to keep the competitive ratio c^*, even if the load drops to the minimum in the next observation and stays there until the end of the transaction phase.

The following is formal analysis to compute the competitive ratio c^* based on P, the maximum contention on the lock (or the maximum number of processors concurrently competing for the lock).

Since the load that is not a global maximum at the time it is observed is ignored (cf. Rule 1), we consider only the load sequence of successive maxima $1 \leqslant l_1 < l_2 < \cdots \leqslant P$. Let U_i and A_i be the amount of remaining *surplus* and the amount of accumulated *savings*, respectively, right after the ith step, $i = 1, 2, \ldots$, where initial savings $A_0 = 0$. Let $s_i = U_{i-1} - U_i$ be the amount of *surplus* exchanged to *savings* at step i. Following the second rule, we have

$$\frac{U_0 \cdot l_i}{U_i \cdot 1 + A_i} = \frac{U_0 \cdot l_i}{(U_{i-1} - s_i) + (A_{i-1} + s_i \cdot l_i)} \leqslant c^* \tag{5}$$

where the denominator $(U_i \cdot 1 + A_i)$ represents the player's profit in the case that the load drops to the minimum in the next step and stays there until the end of the current transaction phase. At the end, the remaining *surplus* U_i is computed as it is exchanged at the minimum load, which is 1; the numerator $U_0 \cdot l_i$ represents the adversary's profit in this case: she exchanges the whole *surplus* budget U_0 at the highest l_i.

Since function $\mathscr{F}(s_i) = \frac{U_0 \cdot l_i}{(U_{i-1} - s_i) + (A_{i-1} + s_i \cdot l_i)}$ is decreasing with $s_i \geqslant 0$ (due to $l_i \geqslant 1$), the minimum value for s_i is found when $\mathscr{F}(s_i) = c^*$, i.e.

$$\frac{U_0 \cdot l_i}{U_i \cdot 1 + A_i} = \frac{U_0 \cdot l_i}{(U_{i-1} - s_i) + (A_{i-1} + s_i \cdot l_i)} = c^* \tag{6}$$

It follows that:

$$s_i = \frac{1}{l_i - 1}\left(\frac{U_0 \cdot l_i}{c^*} - (U_{i-1} + A_{i-1})\right) \tag{7}$$

For the case $i = 1$,

$$s_1 = \frac{1}{l_1 - 1}\left(\frac{U_0 \cdot l_1}{c^*} - (U_0 + A_0)\right) = \frac{U_0}{c^*} \cdot \frac{l_1 - c^*}{l_1 - 1} \tag{8}$$

For the case $i \geqslant 2$, $U_{i-1} + A_{i-1} = \dfrac{U_0 \cdot l_{i-1}}{c^*}$ (Eq. (6)). We have

$$s_i = \frac{1}{l_i - 1}\left(\frac{U_0 \cdot l_i}{c^*} - \frac{U_0 \cdot l_{i-1}}{c^*}\right) = \frac{U_0}{c^*} \cdot \frac{l_i - l_{i-1}}{l_i - 1} \tag{9}$$

Since the length of the load sequence of successive maxima in the transaction phase is unknown to the player, she must consider an arbitrary long load sequence, i.e. a load sequence with length $k \to \infty$. From the definition of s_i, we have

$$\lim_{k\to\infty} \sum_{i=1}^{k} s_i = \lim_{k\to\infty} (U_0 - U_k) \leqslant U_0 \tag{10}$$

The optimal solution for the player should leave no remaining *surplus* at the end of the transaction phase (i.e. $U_k = 0$), which must be exchanged to *savings* at the minimum exchange rate. This implies the following property

$$\lim_{k\to\infty} \sum_{i=1}^{k} s_i = U_0 \tag{11}$$

$$\iff \frac{U_0}{c^*}\left(\frac{l_1 - c^*}{l_1 - 1} + \lim_{k\to\infty} \sum_{i=2}^{k} \frac{l_i - l_{i-1}}{l_i - 1}\right) = U_0 \tag{12}$$

$$\iff c^* = 1 + \frac{l_1 - 1}{l_1} \lim_{k\to\infty} \sum_{i=2}^{k} \frac{l_i - l_{i-1}}{l_i - 1} \tag{13}$$

Since the adversary has full control on generating the load sequence of successive maxima l_i, she will choose the worst sequence to make c^* as large as possible. That is,

$$c^* = 1 + \max_{1 \leqslant l_1 < l_2 < \cdots \leqslant P} \left(\frac{l_1 - 1}{l_1} \lim_{k\to\infty} \sum_{i=2}^{k} \frac{l_i - l_{i-1}}{l_i - 1}\right) \tag{14}$$

We can prove that $c^* \leqslant P - \frac{P-1}{P^{1/(P-1)}}$, which implies that the algorithm for computing the backoff delay that obeys the aforementioned rules with $c = P - \frac{P-1}{P^{1/(P-1)}}$ achieves the competitive ratio c. The proof is tedious and the reader who is interested in the proof is referred to El-Yaniv et al. (2001).

Symmetrically, in a load-dropping transaction phase the player uses a similar algorithm to decrease the backoff delay by exchanging *savings* back to *surplus*.

The analysis results in the following lemma.

Lemma 2. *In each load-rising/load-dropping phase, the new algorithm for computing backoff delay is competitive with a competitive ratio* $c = P - \frac{P-1}{P^{1/(P-1)}} = \Theta(\log P)$, *where* P *is the maximum contention on the lock (i.e. the maximum number of processors concurrently interested in the lock).*

4.1. Implementation

Synchronization primitives: Our algorithm uses the synchronization primitives *fetch-and-add* (*FAA*) and *compare-and-swap* (*CAS*), which are available in recent systems either in hardware like Intel and SPARC processors or in software like SGI systems. The definitions of the primitives are described in Fig. 4.

The online algorithm is presented via pseudo-code in Fig. 5. 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, the load 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 for the first time (line *A*1). Each process chooses an initial *surplus* with respect to its own ticket/counter (line *A*2)

$$initSurplus = (P - counter) \cdot base_l \tag{15}$$

This chosen initial *surplus* 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 the pseudo-code in Fig. 5. 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 use a single-word variable to contain both the lock and the counter (cf. *LockType* in Fig. 5).

A process p_i calls procedure *Acquire*(L) when it wants to acquire lock L. First, p_i increases both values $\{lock, counter\}$ by 1 (line *A*1). The lock L has been occupied if $L.lock \neq 0$. When spinning the lock locally (line *A*5), 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 *A*7). It will successfully acquire the lock if no other processes have acquired the lock in this interval, i.e. $cond.lock = 0$ (line *A*8).

Process p_i calls procedure *Release*( ) when releasing the lock. The procedure has to do two tasks atomically: (i)

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**$(false)$; }

Fig. 4. Synchronization primitives, where x is a variable and v, *old*, *new* are values.

```
type LockType = record lock, counter : [1..MaxProcs]; end; /*stored in one word*/
     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); /*backoff delay variable*/
      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, pL.base); 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; /*variable to indicate the first exchange in a phase*/
    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*/
```

Fig. 5. The Acquire and Release procedures.

reset the *lock* field and (ii) decrease the *counter* field by 1. The *CAS* primitive can do these tasks atomically (line *R*2).

5. Correctness

The correctness of the new algorithm follows almost straightforward from its description. □

Lemma 3. *The number of processors currently competing for the lock L.counter is counted correctly.*

Proof. We prove that the counter for the number of contending processors *L.counter* is increased/decreased correctly. Processors increase the counter by one when trying to acquire the lock (line *A*1 in Fig. 5) and decrease the counter by one when releasing the lock (line *R*2). Although processors change the counter concurrently, the atomicity property of *FAA* and *CAS* helps the algorithm avoid all possible races. In particular, if a processor p' changes the counter between steps R_1 and R_2 executed by another processor p, the *CAS* primitive at the R_2 step will detect that the current value of the counter *pL*. *L* is different from the value *L* read at R_1 and return *False*. Therefore, the processor p must repeatedly read the counter (line R_1) and try to decrease the counter by one (line R_2) until success (i.e. the *CAS* primitive returns *True*). This results in that p decreases the counter by one precisely as expected. □

The precise counting supports an accurate estimation on the lock contention, helping the algorithm react correctly.

Lemma 4. *The space need for the lock field of LockType is* $\log(P)$ *for systems with P processors.*

Proof. Let Δt denote an interval from the time the *lock* field of a lock L is increased to 1 at line $A1$ or $A7$ to the time it is reset to 0 at line $R2$. In Δt, each processor p_i can increase the *lock* field 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. □

This lemma shows that the space complexity of the new reactive spin-lock is as small as that of the simple ticket lock.

Lemma 5. *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 the *FAA* primitive at line $A1$ or $A7$ is 0. Due to the atomicity property of *FAA*, 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 6. *The new spin-lock algorithm guarantees mutual exclusion and non-livelock. Its space complexity is* $\Theta(\log P)$ *for systems with P processors.*

6. Estimating the delay base

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 9).

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 4 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 the terms used in this section.

Fairness: In order to evaluate the fairness of spin-locks, we consider them in the context of 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 tasks. 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. Fig. 1).

However, fairness is just a factor that contributes to overall performance. There are other performance factors like cache affinity. Unfortunately, factors like fairness and cache affinity are opposite to each other. In order to increase the fairness factor we need to decrease the cache affinity factor to give remote processors an equal chance for acquiring the lock. Therefore, although the new reactive spin-lock takes fairness into account, its main target is performance. Depending on applications, the new reactive spin-lock may adaptively gain performance at the cost of fairness.

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

Definition 7. Let n_i be 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:

$$fairness_{\Delta t} = \frac{\sum_i n_i}{\max_i n_i \cdot N} \tag{16}$$

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

This fairness formula satisfies all the desired properties for fairness measures (Jain et al., 1984) consisting of population size independence, scale and metric independence, boundedness and continuity. Moreover, this chosen formula is computationally simpler than those suggested in Jain et al. (1984), which makes it more appropriate for online evaluation.

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 8. The *overhead* of yielding a cached variable such as a lock to another processor in order to achieve good fairness is

$$overhead = \frac{latency\ of\ remote\ memory\ reference}{latency\ of\ L1\ cache\ reference} \tag{17}$$

Definition 9. The *delay outside a critical section* (*DoCS*) is the time 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. Fig. 1). The *delay inside a critical section* (*DiCS*) is the time 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. If the backoff delay is chosen inaccurately, the application performance degrades significantly (cf. *tts* (*TTSE* with tuned thresholds) and *tts*0 (*TTSE* without tuned thresholds) in Fig. 2).

We have found a method to estimate the delay base by *DoCS*:

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:

$$base_l = \frac{a \cdot DoCS + b}{DoCS^2} \tag{18}$$

where a and b are constants.

To see why this works, note that 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 infinity, a processor after finishing its current iteration (cf. Fig. 1) should immediately yield the lock to other processors so that other processors can use the lock, i.e. $base_l \approx 0$.[3] 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:

$$y = g(x) = \frac{f_n(x)}{f_{n+k}(x)} \tag{19}$$

where $k \geqslant 1$ 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 8). Therefore, $overhead \leqslant base_l$. If all processors keep the lock in the minimum time $base_l = overhead$ to minimize Δt in Definition 7, the time for the lock to visit all $P-1$ other processors and then come back to p_i is

$$(base_l + transmission\ delay) \cdot P = (overhead + overhead) \cdot P \tag{20}$$

$$= 2 \cdot overhead \cdot P \tag{21}$$

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 the lock, 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 the 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 (Fig. 5) can be accessed by $P-1$ other processors before the current lock holder gets another ticket, i.e.

$$base_l = transmissiondelay \cdot (P-1) = overhead \cdot (P-1) \tag{22}$$

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

$$y = g(x) = \frac{a \cdot x + b}{x^2} \tag{23}$$

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[4]: 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 that 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 Fig. 5.

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

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

7. 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:

- the *TicketP* performs better than the MCS queue-lock[5] when using application benchmarks Spark98 (cf. Fig. 2) and SPLASH-2 (cf. Kumar et al., 1999), and
- from the fairness point of view, the *TicketP* is better than the queue lock (cf. Fig. 2).

Although the ticket lock is considered not as scalable as the MCS queue-lock, since in 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 Mellor-Crummey and Scott (1991), 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 Radovic and Hagersten (2002) 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 references to the other node, which is not the case in our experiments. The source code for *TTSE* was from Scott and Scherer (2001).

Choosing application benchmarks: In order to compare performance among different spin-lock algorithms, the application benchmarks should have a highly contended lock, which will noticeably promote efficient lock algorithms (cf. Performance Goals for Locks in Culler et al. (1999)). Therefore, we chose as our application benchmarks the shared memory program *lmv* from the Spark98 kernel (O'hallaron, 1997) and the following applications from the SPLASH-2 suite (Woo et al., 1995): Volrend, which uses one lock, instead of an array of locks *QLock*, to protect a global queue, and Radiosity. The Spark98 kernel computes sparse matrix vector products coming from a family of three-dimensional finite element earthquake applications (O'hallaron, 1997). The Volrend application renders a three-dimensional volume using a ray casting technique (Woo et al., 1995). The Radiosity application computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method (Woo et al., 1995). Both Volrend and Radiosity have highly unstructured access patterns to irregular data structures (Woo et al., 1995). 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 sophisticated spin-lock algorithms like the new reactive spin-lock. The input data for the benchmarks were *sf*5.1.*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 250 MHz 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 Laudon and Lenoski (1997).

We also used as an evaluation platform a popular workstation with two Intel Xeon 3 GHz 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 Besedin (2005).

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 Fig. 6. Contention on the lock was varied by changing the number of participating processors/threads. The execution times of the application benchmarks were measured.

7.1. Results

The new reactive spin-lock in Fig. 5 is used in all locks generating high contention. Such locks play a significant role in application execution time and promote efficient spin-lock algorithms. Working on such high contention locks, processes always have to delay between two consecutive attempts to acquire the lock. The new reactive spin-lock utilizes the delay interval to compute a reasonable value for the next delay. This is the reason why the new reactive spin-lock is actually efficient even though it appears quite heavy compared with the non-arbitrating/arbitrating representatives.

Fig. 7 shows the 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

[5] The MCS queue-lock performance is comparable with those of other queue-locks (Mellor-Crummey and Scott, 1991; Scott and Scherer, 2001).

	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	$b_p = 60$	$b_p = 30$	$b_p = 90$

Fig. 6. 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.

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 $b_e = 50{,}000$ and $l_e = 650{,}000$ (cf. Fig. 6), 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 the competitive online algorithm that increased/decreased the backoff delay *just enough* to alleviate contention on the lock. That means the algorithm tried to keep 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 6). 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. Fig. 6), 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 Anderson (1990), Anderson et al. (2003), Kägi and Goodman (1997), Kumar et al. (1999) and Mellor-Crummey and Scott (1991). 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 the 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 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.

When the number of processors increases from 4 to 12, the contention is still low enough for the three application benchmarks to gain performance on the increase of participating processors (as behavior of real parallel applications). At these contention levels, the *hand-tuned TTSE* is the best for all the applications and the new *self-tuning*

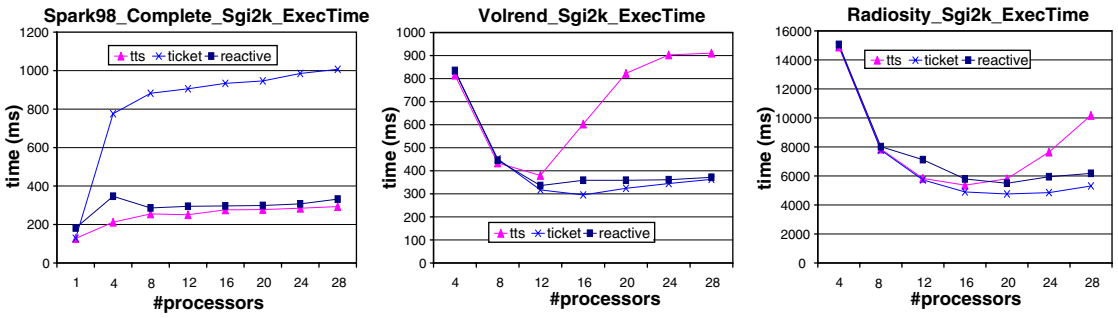

Fig. 7. The execution time of Spark98, Volrend and Radiosity applications on the SGI Origin2000.

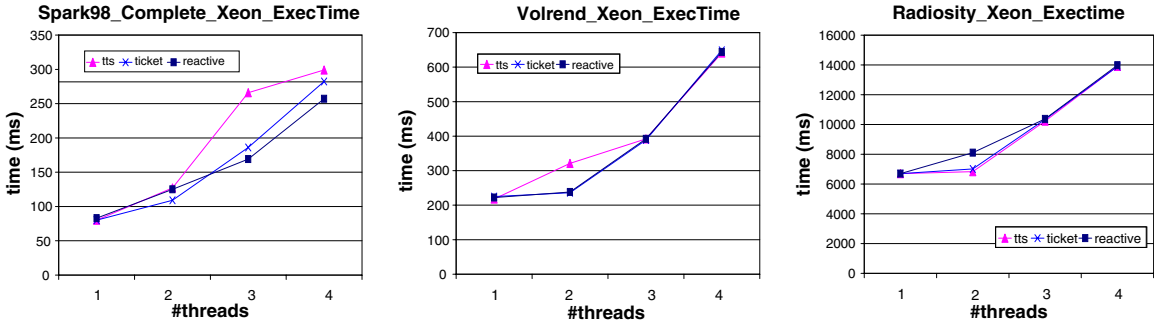

Fig. 8. The execution time of Spark98, Volrend and Radiosity applications on a workstation with 2 Intel Xeon processors.

spin-lock approaches the best. Note that in order to be the best, *TTSE* must be manually tuned for each application and each platform (cf. Fig. 6) whereas the new spin-lock approaches the best without the need of hand-tuned parameters. Without the hand-tuned parameters, the performance of *TTSE* significantly degrades: it is even two times worse than the performance of *TicketP* in the Spark98 application (cf. *tts*0 in the left chart of Fig. 2).

Experiments on the Intel platform showed a similar result: the new spin-lock performed as well as the best representative (cf. Fig. 8). 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 due to their highly contended locks, 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, reacted 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* and *TicketP*, the spin-locks that were *manually* tuned for both each application and each platform.

8. 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 the backoff delay adaptively 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 have showed that the new *self-tuned* spin-lock achieves as good performance as the best of *hand-tuned* spin-lock representatives in the literature does, on various applications and various platforms.

Acknowledgements

The authors wish to thank the anonymous reviewers for their helpful and thorough comments on the earlier version of this paper.

References

Agarwal, A., Cherian, M., 1989. Adaptive backoff synchronization techniques. In: Proc. of the Annual Intl. Symp. on Computer Architecture, pp. 396–406.

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

Anderson, J.H., Kim, Y.-J., Herman, T., 2003. Shared-memory mutual exclusion: major research trends since 1986. Distrib. Comput. 16 (2–3), 75–110.

Besedin, D., 2005. Detailed platform analysis in rightmark memory analyzer. Part 6 – intel xeon. <http://www.digit-life.com/articles2/rmma/rmma-nocona.html>.

Borodin, A., El-Yaniv, R., 1998. Online Computation and Competitive Analysis. Cambridge University Press.

Craig, T.S., 1993. Building FIFO and priority-queuing spin locks from atomic swap. Tech. Rep. TR-93-02-02, Department of Computer Science, University of Washington.

Culler, D.E., Singh, J.P., Gupta, A., 1999. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publisher.

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

Graunke, G., Thakkar, S., 1990. Synchronization algorithms for shared-memory multiprocessors. IEEE Computer 23 (6), 60–69.

Jain, R.K., Chiu, D., Hawe, W.R., 1984. A quantitative measure of fairness and discrimination for resource allocation in shared computer system. Tech. Rep. DEC-TR-301, Digital Equipment Corporation, Eastern Research Lab.

Kägi, A., Goodman, D.B.J.R., 1997. Efficient synchronization: let them eat QOLB. In: Proc. of the Annual Intl. Symp. on Computer Architecture (ISCA-97). Computer Architecture News, pp. 170–180.

Karatza, H.D., 2000. Cache affinity and resequencing in a shared-memory multiprocessing system. J. Syst. Software 51 (1), 7–18.

Karlin, A.R., Li, K., Manasse, M.S., Owicki, S., 1991. Empirical studies of competitive spinning for a shared-memory multiprocessor. In: Proc. of the ACM Symp. on Operating Systems Principles, pp. 41–55.

Kumar, S., Jiang, D., Singh, J.P., Chandra, R., 1999. Evaluating synchronization on shared address space multiprocessors: methodology and performance. In: Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computing Systems (SIGMETRICS-99), pp. 23–34.

Lamport, L., 1974. A new solution of Dijktra's concurrent programming problem. Commun. ACM 17 (8), 453–455.

Laudon, J., Lenoski, D., 1997. The SGI origin: a ccNUMA highly scalable server. In: Proc. of the Annual Intl. Symp. on Computer Architecture (ISCA-97), pp. 241–251.

Lim, B., 1995. Reactive synchronization algorithms for multiprocessors. Ph.D. thesis, MIT-LCS-TR-664, Massachusetts Institute of Technology.

Lim, B.-H., Agarwal, A., 1994. Reactive synchronization algorithms for multiprocessors. In: Proc. of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems, pp. 25–35.

Magnussen, P., Landin, A., Hagersten, E., 1994. Queue locks on cache coherent multiprocessors. In: Proc. of the Intl. Parallel Processing Symp. pp. 165–171.

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

O'hallaron, D.R., 1997. Spark98: Sparse matrix kernels for shared memory and message passing systems. Tech. Rep. CMU-CS-97-178, Computing Science, Carnegie Mellon University.

Radovic, Z., Hagersten, E., 2002. Efficient synchronization for nonuniform communication architectures. In: Proc. of the IEEE/ACM SC2002 Conf., p. 13.

Scott, M.L., Scherer, W.N., 2001. Scalable queue-based spin locks with timeout. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming, pp. 44–52, source code is available from: <ftp://ftp.cs.rochester.edu/pub/packages/scala7ble_synch/PPoPP_01_trylocks.tar.gz>.

Squillante, M.S., Lazowska, E.D., 1993. Using processor-cache affinity information in shared-memory multiprocessor scheduling. IEEE Trans. Parallel Distrib. Syst. 4 (2), 131–143.

Torrellas, J., Tucker, A., Gupta, A., 1993. Benefits of cache-affinity scheduling in shared-memory multiprocessors a summary. In: Proc. of the 1993 ACM Sigmetrics Conf., pp. 272–274.

Vaswani, R., Zahorjan, J., 1991. The implications of cache affinity on processor scheduling for multiprogrammed shared memory multiprocessors. In: Proc. of the ACM Symp. on Operating System Principles, pp. 26–40.

Wang, Y.M., Wang, H.H., Chang, R.C., 1997. Clustered affinity scheduling on large-scale NUMA multiprocessors. J. Syst. Software 36 (1), 61–70.

Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A., 1995. The SPLASH-2 programs: characterization and methodological considerations. In: Proc. of the Annual Intl. Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News, pp. 24–36.

Phuong Hoai Ha received the BEng degree from the Department of Information Technology, HCMC University of Technology, Vietnam and the Ph.D. degree from the Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. His research interests are online algorithms, parallel/distributed computing and systems, including reactive synchronization, fault-tolerant coordination and concurrent data structures (www.cs.chalmers.se/~phuong).

Marina Papatriantafilou is an associate professor at the Department of Computing Science, Chalmers University of Technology, Sweden. She received the BSc and Ph.D. degrees from the Department of Computer Engineering and Informatics, University of Patras, Greece. She has also worked at the National Research Institute for Mathematics and Computer Science in the Netherlands (CWI), Amsterdam and at the Max-Planck Institute for Computer Science, (MPII) Saarbruecken, Germany. She is interested in research on distributed and multiprocessor computing, including synchronization, communication/coordination, networking, scalability, real-time and fault-tolerance aspects.

Philippas Tsigas' research interests include concurrent data structures for multiprocessor systems, communication and coordination in parallel systems, fault-tolerant computing, mobile computing. He received a BSc in Mathematics from the University of Patras, Greece and a Ph.D. in Computer Engineering and Informatics from the same University. Philippas was at the National Research Institute for Mathematics and Computer Science, Amsterdam, the Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken, Germany, before. At present he is an associate professor at the Department of Computing Science at Chalmers University of Technology, Sweden (www.cs.chalmers.se/~tsigas).