Efficient Multi-Word Locking Using Randomization

Phuong Hoai Ha
Department of Computer Science
Chalmers University of Technology
Sweden
phuong@cs.chalmers.se

Philippas Tsigas
Department of Computer Science
Chalmers University of Technology
Sweden
tsigas@cs.chalmers.se

Mirjam Wattenhofer
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
mirjam.wattenhofer@inf.ethz.ch

Roger Wattenhofer
Computer Engineering and Networks Laboratory
ETH Zurich
8092 Zurich, Switzerland
wattenhofer@tik.ee.ethz.ch

ABSTRACT

In this paper we examine the general multi-word lock problem, where processes are allowed to multilock arbitrary registers. Aiming for a highly efficient solution we propose a randomized algorithm which successfully breaks long dependency chains, the crucial factor for slowing down an execution. In the analysis we focus on the 2-word lock problem and show that in this special case an execution of our algorithm takes with high probability at most time $O(\Delta^3 \log n / \log\log n)$, where n is the number of registers and Δ the maximal number of processes interested in the same register (the contention). Furthermore, we implemented our algorithm for the general multi-word lock problem on an SGI Origin2000 machine, demonstrating that our algorithm is not only of theoretical interest.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming;
D.4.1 [**Operating Systems**]: Process Management—*Multiprocessing/multiprogramming/multitasking*;
F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures* ;
G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*;
G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms

Algorithms, Theory, Performance, Experimentation

PODC'05, July 17–20, 2005, Las Vegas, Nevada, USA.

Keywords

shared memory, dining philosophers, multi-word locking, randomization

1. INTRODUCTION

Edsger Dijkstra's dining philosophers problem is widely recognized as a prototypical resource allocation instance. We are given n philosophers, sitting at a round table. Each philosopher is an asynchronous process who cycles through the three states thinking, hungry, and eating. Between each neighboring pair of philosophers, there is a fork. When becoming hungry, a philosopher tries to grab her left and right fork. After having acquired both forks, the philosopher eats. When finished eating, the philosopher returns her forks and goes back to thinking mode.

We can represent the classic dining philosophers problem in a shared memory multi-processor system by having n shared registers (the forks) and n processes (the philosophers). The two registers ("forks") which are of interest to process p_i (with $i = 1, 2, \ldots, n$) are registers i and $i + 1$ (with the notable exception that the "right fork" of processor p_n is register 1, and not $n + 1$, to achieve the desired ring topology). In shared memory dining philosophers, each process repeatedly and asynchronously tries to lock its two registers (hungry), then performs some atomic operation on these two registers, such as multi-word compare-and-swap (eating), and then continues with other operations (thinking).

The dining philosophers problem perfectly illustrates typical multi-process synchronization difficulties. If all philosophers become hungry at the same time, and pick up their left fork simultaneously, we have a *deadlock*, since no philosopher can grab her right fork as well. Similarly, if a process crashes (or behaves awfully slow) after locking its registers, the two neighbor processes cannot make progress; as a remedy the research community has proposed *non-blocking* protocols, such as recursive helping schemes, or transactional memory.

In this paper we focus on a third fundamental multi-process synchronization issue, *efficiency*. In dining philosophers, "even" philosophers (processes with even process id) do not have a conflict of interest among themselves. An

efficient implementation striving for maximum concurrency would therefore always let even and odd processes eat in turns, thus maximizing the available resources.

In this paper we examine the general multi-word lock problem, where processes are allowed to multi-lock arbitrary registers. The remainder of the paper is organized as follows: In Section 2 we set our paper into context of prior art. The model is then formally introduced in Section 3. In Section 4 we present our algorithm and analyze it; in particular we show that a process has to wait at most $O(\Delta^3 \log n / \log\log n)$ time until it can eat, where n is the number of registers and Δ the maximal number of processes interested in the same register (the contention). In Section 5 we present extensive results from our implementation on an SGI Origin2000 machine, proving that our idea is not only of theoretical interest. Finally, in Section 6 we conclude the paper.

2. RELATED WORK

Since processes without a conflict can proceed concurrently it seems promising to first compute a minimum coloring of the conflict graph. Yet solving the multi-lock problem using coloring remained theory.

In a generalized variant of dining philosophers, a process shares d forks and can only eat if it has obtained all d forks. For this generalized problem [9] gave a solution with waiting chains of length $O(c)$, assuming that an oracle has colored the conflict graph[1] with c colors. The waiting chain length was reduced to $O(\log c)$ in [13]. Assuming that a vertex coloring with $d+1$ colors is known, in [4] this length was further reduced to 3. For a simplified version of dining philosophers [11] manage to have waiting chains of length at most 4 in constant time.

Unfortunately, even in a powerful message passing model, coloring is a tough problem. It was proven in [8] that such colorings cannot be found in constant time. In fact, even simpler problems (such as independent sets) have logarithmic lower bounds [8]. More severely, the conflict graph is not available straightforwardly. To compute the conflict graph, processes need some form of synchronization. We believe that this synchronization is as hard to achieve as the original multi-lock problem.

In this paper we present an efficient but blocking algorithm for the general multi-lock problem, consequently without making a detour through coloring. A blocking algorithm for the multi-lock problem can be turned into a lock-free algorithm if it is combined with a *helping technique*. The basic idea of the so called cooperative technique [3], a form of helping technique, was improved and is still developing in a series of nifty research papers [7, 12, 1, 10, 6, 5].

For readability we do not integrate our algorithm with a helping scheme; however, it can be added to our algorithm: Each process, before locking registers, somewhere notes what it wanted to do with that register. In case the process crashes while holding the lock, others can help it finish by following its steps. The implementation of our algorithm used in Section 5 of this paper includes a helping scheme, rendering our implementation lock-free.

[1]Here the conflict graph is a graph where each node represents a process and each edge represents a resource which is shared by the two endpoint processes. Our conflict graph is different, see also Section 3.

In distributed graph algorithms (message passing), randomization techniques are widely used. With a few exceptions [2], the shared memory community does generally not apply randomization, presumably because its alleged overhead. Our experiments show that the overhead due to randomization is less than 1% of the total execution time.

3. PROBLEM AND MODEL

In this section we recapitulate the problem we consider and formally define the model used in the next sections.

We study the *multi-lock problem*, which is a generalization of dining philosophers. In the multi-lock problem each participating process needs to lock multiple registers in order to do some operation on the locked registers, like an N-word compare-and-swap (CASN).

Typically, in a multi-lock implementation a process tries to lock all its registers one by one. To avoid deadlocks, the registers are totally ordered, conventionally by their identifiers (id). When executing a k-lock, a process p locks its registers $r_1, \ldots, r_k$ according to their total order.

As discussed in Section 2, there exist several schemes which can be employed once a process is blocked by other processes from locking its registers. In the analysis we assume that processes simply wait (spin-lock) until the block is resolved, yet for the implementation (Section 5) we include a helping scheme.

We consider m *asynchronous* processes which can access n shared registers. In the analysis we concentrate on 2-locks. Each process only executes a single 2-lock and then goes to sleep.

The dependencies between the processes are modelled by a directed acyclic *conflict graph* $G = (V, E)$. In G each node represents a register and each edge represents a process. In the following, we will use the terms node/register and edge/process interchangeably. There is a directed edge p from node r_1 to node r_2 iff process p tries to lock register r_1 first and after being successful tries to lock register r_2, meaning that r_1 comes before r_2 in the total order of registers. Since all directed edges point from nodes with lower id to nodes with higher id the resulting graph G is acyclic.

Following the conventions for asynchronous processes, in the analysis we assume that each atomic operation, like reading, writing, or locking a register, incurs a delay of *at most* one time unit. An operation on multiple locked registers (e.g. *CAS2*) incurs a delay. For convenience let c be the longest time which elapses from the moment a process has locked its last register until it releases the lock on all its registers.

In the remainder of this section, to illustrate our model, we quickly analyze a classical implementation of dining philosophers. We show that it is a factor $\Omega(n)$ less efficient than an optimal implementation.

The classical implementation proceeds as follows: the processes try to lock their registers one by one, each starting with the register with smaller identifier. Consider the following execution: First, each process p_i, $i < n$, locks register r_i, whereas p_n fails to lock r_1 (due to p_1). Then, each process tries to lock register r_{i+1}, yet only process p_{n-1} succeeds. The second register of all other processes is locked by another process. Thus, process p_i has to wait until process p_{i+1} releases its lock on r_{i+1}. By induction, after having waited $\Theta(cn)$ time units p_1 releases its lock on r_1 and p_n may lock both its registers. Thus, the execution time is $\Omega(cn)$. An

optimal implementation needs only $O(c)$ time and hence the classical algorithm is a factor $\Omega(n)$ less efficient than an optimal algorithm.

4. RANDOMIZED REGISTERS

In this section we present a more efficient algorithm for multi-lock and analyze it according to two standard criteria for the special case of 2-lock.

4.1 The Algorithm

Alerted by the poor execution time of the classical algorithm for dining philosophers due to its long dependency chain, we aim at breaking dependency chains. A promising yet simple (allowing for an efficient implementation) approach is randomization. Specifically, we suggest to randomly permute the order of the registers. Let Π be a permutation on the registers, chosen uniformly at random. The permutation represents the new total ordering of the registers. For details on how the randomization can be implemented we refer to Section 5.

In short we henceforth write $p = (r_i, r_j)$ meaning that process p wants to acquire register r_i and r_j and that $\Pi(\mathrm{id}(r_i)) < \Pi(\mathrm{id}(r_j))$. Thus, r_i is p's *first* register and r_j is p's *second* register.

In the next sections we analyze the efficiency of the suggested 2-lock algorithm which uses a randomized total ordering of registers. As in [13] we evaluate two related properties: the maximum length of a waiting chain and the longest time a process needs until it successfully performs a 2-lock. Towards this goal, we first prove some basic properties of the conflict graph[2]. Thereafter, we analyze the length of waiting chains and finally show that with high probability after $O(c\Delta^3 \log n/\log\log n)$ time the execution is finished.

4.2 Length of Directed Paths

Henceforth, we denote by G the conflict graph as obtained by the random permutation of the registers. Let the maximum degree in G be Δ. In this section we analyze the length of a directed path in G. The following facts are used for the analysis, the proofs of which can be found in standard mathematical textbooks.

FACT 4.1 (STIRLING).

$$k! \geq 2\frac{\sqrt{k}k^k}{e^k}.$$

FACT 4.2 (MARKOV).

$$\mathbb{P}(X \geq t) \leq \mathbb{E}[X]/t.$$

Throughout the paper $\log n$ denotes the logarithm with base two.

To estimate the length of a directed path in G, we first upper bound the number of distinct *undirected* paths of length k in G: To obtain an undirected path of length k one can choose one out of n nodes in G as start node. In any node there are at most Δ neighbor nodes to continue the path. Therefore:

OBSERVATION 4.3. *There are at most $n\cdot\Delta^k$ distinct undirected paths of length k in G.*

[2]Note, that the conflict graph is only needed for analysis purposes. The processes do not know the conflict graph.

As a next step we give the probability that a given path of length k in G is directed.

OBSERVATION 4.4. *The probability that a given path of length k in G is directed is $\frac{2}{(k+1)!}$.*

PROOF. In a path of length k there are $k+1$ nodes $u_1, \ldots, u_{k+1}$. For the path to be directed, it must hold that either $\Pi(\mathrm{id}(u_1)) < \Pi(\mathrm{id}(u_2)) < \ldots < \Pi(\mathrm{id}(u_{k+1}))$ or $\Pi(\mathrm{id}(u_1)) > \Pi(\mathrm{id}(u_2)) > \ldots > \Pi(\mathrm{id}(u_{k+1}))$. Hence, there are exactly two good out of $(k+1)!$ possible choices. □

Thus, the probability that a path is directed decreases exponentially with increasing path-length. Combining both Observation 4.3 and Observation 4.4 gives an upper bound on the number of directed paths of length k.

LEMMA 4.5. *Let C be the number of directed paths of length k. Then, $\mathbb{E}[C] < \frac{1}{n^\Delta}$, for $k \geq 3\Delta\frac{\log n}{\log\log n}$.*

PROOF. Let p_i denote a path of length k and let X_{p_i} be defined as follows

$$X_{p_i} = \begin{cases} 1, & \text{if } p_i \text{ directed} \\ 0, & \text{otherwise.} \end{cases}$$

Then by linearity of expectation,

$$\begin{aligned} \mathbb{E}[C] &= \mathbb{E}[\sum_{\forall p_i} X_{p_i}] \\ &= \sum_{\forall p_i} \mathbb{E}[X_{p_i}] \\ &\leq n\Delta^k \frac{2}{(k+1)!}, \end{aligned}$$

by Observation 4.3 and Observation 4.4. Applying Stirling's formula (Fact 4.1) and substituting $3\Delta\log n/\log\log n$ for k yields the following inequalities

$$\begin{aligned} \mathbb{E}[C] &< n\Delta^k \frac{e^k}{\sqrt{k}k^k} \\ &\leq n\Delta^k \frac{e^k}{(3\Delta\log n/\log\log n)^k} \\ &< n\frac{1}{(\log n/\log\log n)^k} \\ &= n\frac{1}{2^{3\Delta\log n/\log\log n(\log\log n - \log\log\log n)}} \\ &= n\frac{1}{n^{3\Delta(1-\log\log\log n/\log\log n)}} \\ &\leq \frac{1}{n^\Delta}. \end{aligned}$$

□

Finally, we bound the probability that there exists a directed path in G by applying Markov's inequality.

COROLLARY 4.6. *With probability at most $1/n^\Delta$ there exists a directed path of length at least $3\Delta\log n/\log\log n$.*

PROOF. By Fact 4.2 and Lemma 4.5

$$\mathbb{P}(C \geq 1) \leq \mathbb{E}[\mathbb{C}] \leq 1/n^\Delta.$$

□

4.3 Length of Waiting Chains

Following the notation of [13] we define a *waiting chain* as a series of processes such that each process in the chain is waiting for some action by the next process in the chain. Avoiding long waiting chains is important since it implies a long wait for the last process in the chain.

A process p is delayed by another process q if q can, by slowing down or stopping, cause p to have a longer total waiting time than if q stayed in its remainder section. The maximum length of a waiting chain is the maximum distance between two processes such that one process can delay the other.

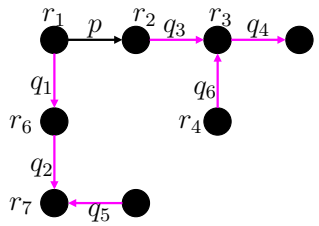

Figure 1: Delation of p.

We want to intuitively depict the concept of delaying with the example of Figure 1. Process p can be delayed by all processes in the figure: E.g. process q_2 can delay p if q_1 locks r_1 before p and q_2 locks r_6 before q_1. Thus, q_1 has to wait for q_2 until it can acquire (and release) both its locks and consequently p (which waits for q_1 to release the lock on r_1) also has to wait for q_2. Process q_6 can also delay p: q_6 locks r_3, q_3 locks r_2 and p locks r_1. Thus, p is waiting for q_3 to release r_2 which itself is waiting for q_6 to release r_3. On the other hand, process p could not be delayed by a (not shown) process $q_7 = (r_4, x)$, where x is some register not depicted in Figure 1. This is because process q_7 may acquire register r_4 before q_6 and thus q_6 has to wait until q_7 releases this register again before it can proceed. Yet, this does in no way delay p since it does not impose a longer waiting time on p. In general:

LEMMA 4.7. *Process $q = (R_1, R_2)$ can delay process $p = (r_1, r_2)$ if and only if R_2 lies on a directed path starting in either r_1 or r_2.*

PROOF. By definition, if process q delays process p the waiting time of p must be longer if q slows down or stops than if q stayed in its remainder section. If processes q's second register R_2 lies on a directed path starting in r_i, $i \in \{1, 2\}$, then there exists a directed path of processes $q_1 = (r_i, r_j), \ldots, q_k = (r_l, R_2)$, with possibly $q_k = q$. In the case that each process in this path locked its first register– and given that $q_k \neq q$ q also locked its second register–none of the processes $q_1, \ldots, q_k$ makes any progress as long as q does not make any progress. Thus, process p will not be able to lock its register r_i and hence its waiting time is longer than if q stayed in its remainder section, showing that q delays p.

In order to show the other direction of the lemma we let Q be the set of processes which do not lie with their second register on a directed path from r_i, $i \in \{1, 2\}$. Between any arbitrary process q in Q and any directed path $\mathcal{P}_i$ from r_i there is at least one process $\bar{q}$ which breaks this directed path, that is $\bar{q}$'s second register lies on $\mathcal{P}_i$ whereas its first register does not. By slowing down or stopping its execution q may hinder $\bar{q}$ in acquiring its first register, yet it does not hinder $\bar{q}$ in acquiring its second register, otherwise one of q's registers would also lie on a directed path from r_i, a contradiction to the assumption that q is in Q. Thus, by slowing down or stopping q either does not affect $\bar{q}$ or $\bar{q}$ cannot participate in the execution at all as long as q does not make any progress. Hence, process q cannot delay p via $\bar{q}$ and consequently it cannot delay p via any process on $\mathcal{P}_i$. Furthermore, q is not incident to p and thus it cannot delay p directly. Since q cannot delay p indirectly via a process on a directed path nor directly, p is not affected if q slows down or stops which concludes the proof. □

COROLLARY 4.8. *The maximum length of a waiting chain in the randomized registers algorithm is with probability at least $1 - 1/n^{\Delta}$ at most $3\Delta \log n / \log \log n + 1$.*

PROOF. The maximal number of edges between a process p and a process q which delays p is at most the length of the longest directed path plus one, since by Lemma 4.7 a process which delays p must be incident with its second register to a directed path. Hence, we can directly apply Corollary 4.6. □

4.4 Execution Time

Though the length of a waiting chain is an indicator of the efficiency of an algorithm, it is only a lower bound for the execution time. For the execution time we must bound two values: First, we need to bound the time until a process is able to lock its first register, then we need to bound the time until it can lock its second register. Towards this goal we introduce some helpful definitions.

The execution starts at time zero. A process $p = (r_1, r_2)$ locks its first register at time $t_1(p)$ and its second register at time $t_2(p)$. Using the definition of Section 3 process p releases both its locks at time $t_3(p) \leq t_2(p) + c$. (See also Figure 2.)

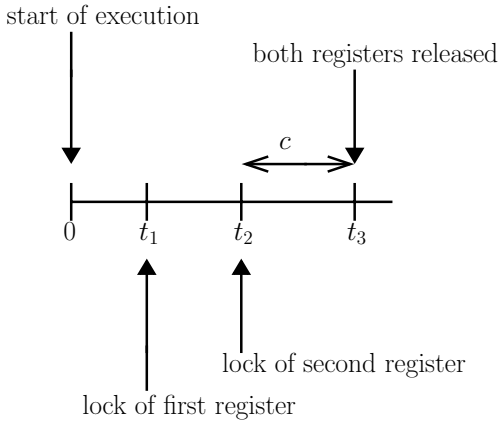

Figure 2: t_1, t_2 and t_3 for a process.

DEFINITION 4.9 (DELAY GRAPH). *Let p be a process with $p = (r_1, r_2)$. Then p's* delay graph*, denoted by $D(p)$ contains all processes with $q = (R_1, R_2)$ where R_1 lies on a directed path starting in r_1. The depth of process p depth(p) is then defined as the length of the longest directed path (number of processes in the path) in $D(p)$.*

In the example of Figure 3 processes p, q_1, q_2, q_3, q_4 are in p's delay graph, whereas process q_5, q_6 are not, since there is no directed path from r_1 to q_6's first register r_4, respectively q_5's first register r_8. Intuitively, processes which are incident to a directed path from p merely by their second register, do not delay p much, since those processes release their lock on the crucial register quickly after acquiring it. Note, that the depth of a process is at least one since at least the process itself lies in its delay graph.

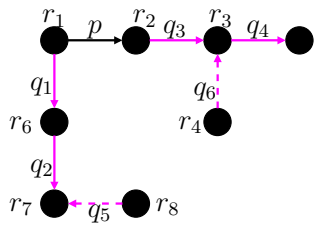

Figure 3: Process p, q_1, q_2, q_3, q_4 are in p's delay graph. The depth of p is 3, q_1's depth is also 3.

We now bound the maximal depth of any process by directly applying Corollary 4.6:

COROLLARY 4.10. *The maximum depth k^* of any process is at most $3\Delta \log n / \log\log n$ with probability at least $1 - 1/n^{\Delta}$.*

The next lemma reveals a key property of the delay graph.

LEMMA 4.11. *Let $D(p)$ be processes $p = (r_1, r_2)$ delay graph and let $q = (R_1, R_2)$ be a process in $D(p)$. Then,*

$$\mathit{depth}(q) \leq \mathit{depth}(p).$$

Furthermore, if $R_1 \neq r_1$ then

$$\mathit{depth}(q) < \mathit{depth}(p).$$

PROOF. Assume without loss of generality that $\mathcal{P}_{R_j u} = (R_j, u_1, \ldots, u_k, u)$, $j \in \{1, 2\}$, is a longest directed path in q's delay graph. Then, $\text{depth}(q) = |\mathcal{P}_{R_j u}| = |\{R_j, u_1, \ldots, u_k, u\}| - 1$. Since $q \in D(p)$ there is a directed path $\mathcal{P}_{r_1 R_1} = (r_1, v_1, \ldots, v_l, R_1)$ from r_1 to R_1, where $|\mathcal{P}_{r_1 R_1}| = |\{r_1, v_1, \ldots, v_l, R_1\}| - 1$ is the length of this path. Consequently, there exists a directed path $\mathcal{P}_{r_1 u} = (r_1, \ldots, R_1, R_j, \ldots, u)$ from r_1 to u. Thus,

$$\begin{aligned}
\text{depth}(p) &\geq |\mathcal{P}_{r_1 u}| \\
&= |\{r_1, \ldots, v_l, R_1, R_j, \ldots, u\}| - 1 \\
&\geq |\{R_j, \ldots, u\}| - 1 \\
&= \text{depth}(q).
\end{aligned}$$

If furthermore, $r_1 \neq R_1$ then

$$\begin{aligned}
\text{depth}(p) &= |\{r_1, \ldots, v_l, R_1, R_j, \ldots, u\}| - 1 \\
&\geq |\{r_1, \ldots, v_l\}| + |\{R_1, R_j, \ldots, u\}| - 1 \\
&\geq 1 + |\mathcal{P}_{R_j u}| \\
&> \text{depth}(q).
\end{aligned}$$

□

The following corollary shows that along a directed path the depth of the processes is strictly decreasing.

COROLLARY 4.12. *Let $\mathcal{P} = (r_1, r_2, \ldots, r_{k+1})$ be a directed path and let $p_i = (r_i, r_{i+1})$, $1 \leq i \leq k$, be the processes on this path. Then, $\mathit{depth}(p_i) > \mathit{depth}(p_{i+1})$, $1 \leq i \leq k-1$.*

PROOF. By the definition of a delay graph, a process $q = (R_1, R_2)$ lies in the delay graph $D(p)$ of process $p = (r_1, r_2)$ iff there is a directed path between r_1 and R_1. Thus, process p_{i+1} lies in the delay graph of process p_i since by the assumption r_i and r_{i+1} lie on a directed path. We hence may apply Lemma 4.11 which states that the depth of a process q which lies in the delay graph $D(p)$ of process p is strictly smaller than p's depth if p's first register is not equal to q's first register. Since in our case the first register of p_{i+1} is r_{i+1} and the first register of p_i is r_i this condition holds and thus the depth of p_{i+1} is strictly smaller than p_i's depth. □

COROLLARY 4.13. *There is a process with depth one in any conflict graph G.*

PROOF. Let $p_1 = (r_1, r_2)$ be a process in G with depth k. Then, there exists a directed path $\mathcal{P} = (r_1, r_2, \ldots, r_{k+1})$ from r_1 to some node r_{k+1} of length (number of processes in $\mathcal{P}$) k. By Corollary 4.12 the depth of the processes $p_i = (r_i, r_{i+1})$, $1 \leq i \leq k$, in this path is strictly decreasing. Thus, $\text{depth}(p_{i+1}) \leq \text{depth}(p_i) - 1$, $1 \leq i \leq k-1$, and since $\text{depth}(p_1) = k$ we have $\text{depth}(p_k) \leq 1$. The depth of any process is at least one and consequently $\text{depth}(p_k) = 1$. □

In the next lemma we upper bound $t_3(p)$ for a process p with depth one.

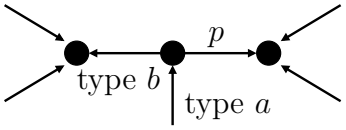

Figure 4: Depth(p)=1.

LEMMA 4.14. *For a process $p = (r_1, r_2)$ with depth(p)=1 we have $t_3(p) \leq 4c\Delta^2$.*

PROOF. A process p has depth one iff the following two conditions hold: A process q_j incident to p's *first* register r_1 is either incoming in r_1 (type a), that is $q_j = (x, r_1)$, x an arbitrary register, or $q_j = (r_1, x)$ and all processes incident to q_j's second register x are incoming in x (type b). A process q_i incident to p's *second* register r_2 is incoming in r_2, that is $q_i = (x, r_2)$, x an arbitrary register. (See also Figure 4.)

We first concentrate on type a processes: Processes of type a releases their lock on r_1 at most c time units after acquiring it. The next process acquires the lock on r_1 at most one time unit later. Thus each process of type a adjacent to r_1 delays p for at most $c + 1 \leq 2c$ time units.

A process $q_j = (r_1, x)$ of type b must wait at the utmost for all processes incident to its second register x until it can acquire the lock on x and thereafter release r_1. A process incident to x releases its lock on x at most c time units after acquiring it and the next process acquires it at most one time unit later. Thus, each process incident to x delays q_j for at most $c + 1 \leq 2c$ time units. Besides q_j there are at most $\Delta - 1$ processes incident to x. We thus immediately get that q_j acquires its lock on x after at most $2c(\Delta - 1) + 1$

time units and releases its locks after at most c more time units. Thus each process of type b adjacent to r_1 delays p for at most $2c(\Delta-1)+1+c \leq 2c\Delta$ time units.

Besides p there are at most $\Delta-1$ processes incident to r_1, each of which releases its lock on r_1 at most $2c\Delta$ time units after acquiring it. Therefore, we immediately get

$$t_1(p) \leq 2c\Delta(\Delta-1)+1.$$

The time until p can lock r_2 is by the same argument as the argument for type a processes at most $2c(\Delta-1)+1$ and hence

$$\begin{aligned} t_3(p) &\leq t_2(p)+c \\ &\leq 2c\Delta(\Delta-1)+1+2c(\Delta-1)+1+c \\ &\leq 4c\Delta^2. \end{aligned}$$

□

LEMMA 4.15. *For a process p with depth(p)=k we have*

$$t_3(p) \leq 4c\Delta^2 k.$$

PROOF. We prove the theorem by induction on the depth of a process $p=(r_1,r_2)$. By Corollary 4.13 there always exists a process of depth one in the conflict graph G and thus we may base the induction in this case.

Base Case: In case that depth(p)= 1 $t_3(p) \leq 4c\Delta^2$ by Lemma 4.14.

Induction: We henceforth assume that for a process q with depth(q)$\leq (k-1)$ it holds that $t_3(q) \leq 4c\Delta^2(k-1)$ and consider process p with depth k. By Lemma 4.11 all processes in p's dependency graph $D(p)$ which do not have r_1 as their first register have depth less than k and thus –by the induction hypothesis– finished their operations at time $4c\Delta^2(k-1)$ at latest. Thus, the only processes in $D(p)$ which are still active are those which have r_1 as their first register and consequently at time $4c\Delta^2(k-1)$ p's depth is at most one. We then apply Lemma 4.14 and get

$$t_3(p) \leq 4c\Delta^2(k-1)+4c\Delta^2 = 4c\Delta^2 k.$$

□

THEOREM 4.16. *A process p finishes its operations after time $O(c\Delta^3 \log n/\log\log n)$ with probability at least $1-1/n^{\Delta}$.*

PROOF. By Corollary 4.10 the depth of any process is at most $3\Delta\log n/\log\log n$ with probability at least $1-1/n^{\Delta}$. Thus, using Lemma 4.15,

$$\begin{aligned} t_3(p) &\leq 4c\Delta^2 \cdot 3\Delta\log n/\log\log n \\ &\in O(c\Delta^3 \log n/\log\log n). \end{aligned}$$

□

In a model where an operation takes exactly time c clearly also an optimal algorithm needs at least the congestion times c time units. Therefore:

COROLLARY 4.17. *With probability at least $1-1/n^{\Delta}$ the randomized registers algorithm is $O(\Delta^2 \log n/\log\log n)$ competitive.*

5. EVALUATION

We have proposed a *multi-lock* algorithm, where the operation performed after all registers are locked can be defined arbitrarily by the programmer. To evaluate the algorithm, we chose the operation specifically to be a single-word compare-and-swap on each register. With this choice, our algorithm became a multi-word compare-and-swap algorithm.

The multi-word compare-and-swap operations (CASN) extend the single-word compare-and-swap operations from one word to many. A single-word compare-and-swap operation (CAS) takes as input three parameters: the address, an *old* value and a *new* value of a word, and atomically updates the contents of the word if its current value is the same as the *old* value (cf. Figure 5). Similarly, an N-word compare-and-swap operation takes the addresses, *old* values and *new* values of N words, and if the current contents of these N words all are the same as the respective *old* values, the CASN will write the new values to the respective words atomically. Otherwise, we say that the CAS/CASN fails, leaving the variable values unchanged.

```
CAS(x, old, new)
atomically {
    if (x = old) {x ← new; return (true)};
    else return (false);
    }
```

Figure 5: The single-word compare-and-swap primitive

The multi-word compare-and-swap operations are powerful constructs, which make the design of concurrent data structures more effective and easier. As expected, they attracted the attention of many researchers, consequently many CASN implementations appear in the literature [7, 12, 1, 10, 6, 5]. One approach suggested to construct CASN operations is *cooperative technique*, which allows processes to concurrently access the shared data as long as they write down what they are doing. Before changing a portion of the shared data that was locked by another process p_j, a process p_i must help p_j complete its task first. The technique was first theoretically suggested by Barnes [3] and then was transformed into a more applicable one by Israeli et al. [7], which was used to implement a lock-free multi-word compare-and-swap operation. This implementation was later improved by Harris et al. [6] to reduce the per-word space overhead. A wait-free multi-word compare-and-swap was developed by Anderson based on this technique [1]. However, this *cooperative technique* uses a recursive helping policy, where a process has to help many other processes before completing its own task. The helping chains, where process p_i helps p_{i+1}, may be very long. All processes related to a chain may do the same task, the task of the last process in the chain, which reduces parallelism and creates high collision levels on the shared data needed by the common task.

In order to evaluate the performance of our algorithm (randomized CAS, in short RaCASN) and also check its feasibility in a real setting we implemented it and ran it on a ccNUMA SGI Origin2000 multiprocessor that was equipped with 30 CPUs. As discussed in Section 2 we equipped our randomized registers algorithm with a helping policy [7]. In order to see in practice the performance benefits of the ran-

domization we also implemented the deterministic recursive helping policy (DeCASN) presented in [7]. The implementation of RaCASN was similar to that of DeCASN except that the order of registers/words[3] chosen to be locked was random in RaCASN. In other words, both algorithms are lock-free. For the tests we used a micro-benchmark and a small application. The micro-benchmark was designed to generate an execution environment with high contention on the shared registers. The application was a parallel-prefix application with continuous input feed and space constraints.

5.1 The micro-benchmark

The micro-benchmark aims at generating an environment with high contention on shared registers. In the micro-benchmark, a set of $N+k$ virtual registers v_i, $1 \leq i \leq N+k$, are mapped on k system registers r_1, r_2, $\cdots$, r_k, where N is the number of registers to be updated atomically by the CASN operations. The mapping used is

$$v_i = \begin{cases} r_i & \text{if } 1 \leq i \leq k \\ r_{i-k} & \text{if } k+1 \leq i \leq k+N. \end{cases}$$

The virtual registers are accessed by k N-word compare-and-swap operations $CASN_1$, $CASN_2$, $\cdots$, $CASN_k$. During the execution of this benchmark, each $CASN_i$ operation tries to update virtual registers v_i, v_{i+1}, $\cdots$, v_{i+N-1} atomically. Note that two consecutive CASNs $CASN_i$ and $CASN_{i+1}$ have $N-1$ system registers in common and thus the micro-benchmark can generate helping chains of length up to k, where a helping chain is a chain of CASN operations that a thread has to help before completing its own CASN.

In our experiment, we ran the micro-benchmark with DeCASN and RaCASN. In the RaCASN implementation, k random numbers corresponding to the k system registers were precomputed and stored in a shared array. For each execution, we measured the longest helping chain and then computed the distribution of the chain lengths over one million executions. We also measured the average execution time of the micro-benchmark using DeCASN and RaCASN. Our experiment ran the micro-benchmark with 28 threads on 28 processors of the SGI Origin2000 machine. We tested the benchmark with $N = 2, 4, 6$ and 8, i.e. $CAS2$, $CAS4$, $CAS6$ and $CAS8$. The results are presented in Figure 6 and Figure 7.

5.1.0.1 *Results:.*

Figure 6 shows that RaCASN breaks the helping chains much better than DeCASN, thus making themselves faster. Long helping chains degrade the efficiency of the whole system since all processors related to a chain try to lock the same registers of the last CASN in the chain, which generates high collision levels on these registers.

In the case of CAS2 in Figure 6, RaCASN exhibits executions with the longest helping chain of length 4 in 61% of the total number of executions, of length 3 in 21% of the total number of executions and of length 5 in 16% of the total number of executions. The RaCASN longest helping chain over one million executions has length 7 in 0.2% of the total number of executions. Regarding DeCASN, it exhibits executions with longest helping chains of length 20 in 32% of the total number of executions, of length 21 in 16% of the total number of executions and of length 19 in 15% of the total number of executions. The DeCASN exhibits executions with longest helping chain of length 28, the maximal number of CASN operations, in 4% of the total number of executions.

[3]Terms *register* and *word* can be used interchangeably.

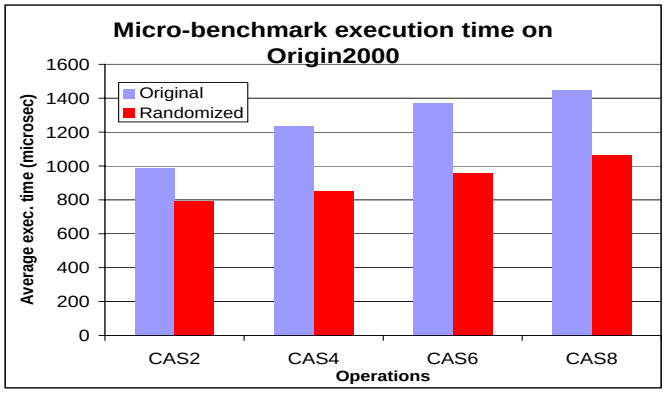

Figure 7: The micro-benchmark execution times on the SGI Origin2000.

When the number of registers to be updated increases, the distribution of RaCASN longest chain lengths shifts to the right slowly but is still much better than that of DeCASN as shown in the charts of CAS4, CAS6 and CAS8 (cf. Figure 6). Note that the probability that one CASN must help another grows with N. However, the length of the longest helping chain may not increase since a successful CASN can reduce this length by at least N. We can observe this effect in Figure 6 where the highest bar in the DeCASN longest length distribution shifts to the left slowly when N increases from 2 to 8.

Since RaCASN helps the micro-benchmark break long helping chains, which by itself reduces collision on memory and increases parallelism, RaCASN achieves better performance on the benchmark as shown in Figure 7. The RaCASN is from 20% to 31% faster than DeCASN. The overhead of computing k random numbers in RaCASN implementation is not significant, which consumed only 0.07 percent of the execution time.

5.2 The application

As we have experienced, an algorithm that gains good performance on a micro-benchmark may not keep such performance on a real application. This motivated us to do another comparison between RaCASN and DeCASN on an application.

The application comes from the following problem:

5.2.0.2 *The problem:.*

There are n registers r_1, r_2, $\cdots$, r_n, each of which belongs to one of n agents a_1, a_2, $\cdots$, a_n. The agents communicate with the underlying computational system via these registers: agent a_k reads a result in register r_k written by the system before writing there a new input i_k for the system. Input values i_k are put in register r_k randomly and independently of other agents. The input values change all the time dynamically. (We can think that they are inputs from sensors.)

The computational system computes an output/result o_k for agent a_k from the prefix i_1, i_2, $\cdots$, i_k. For simplicity,

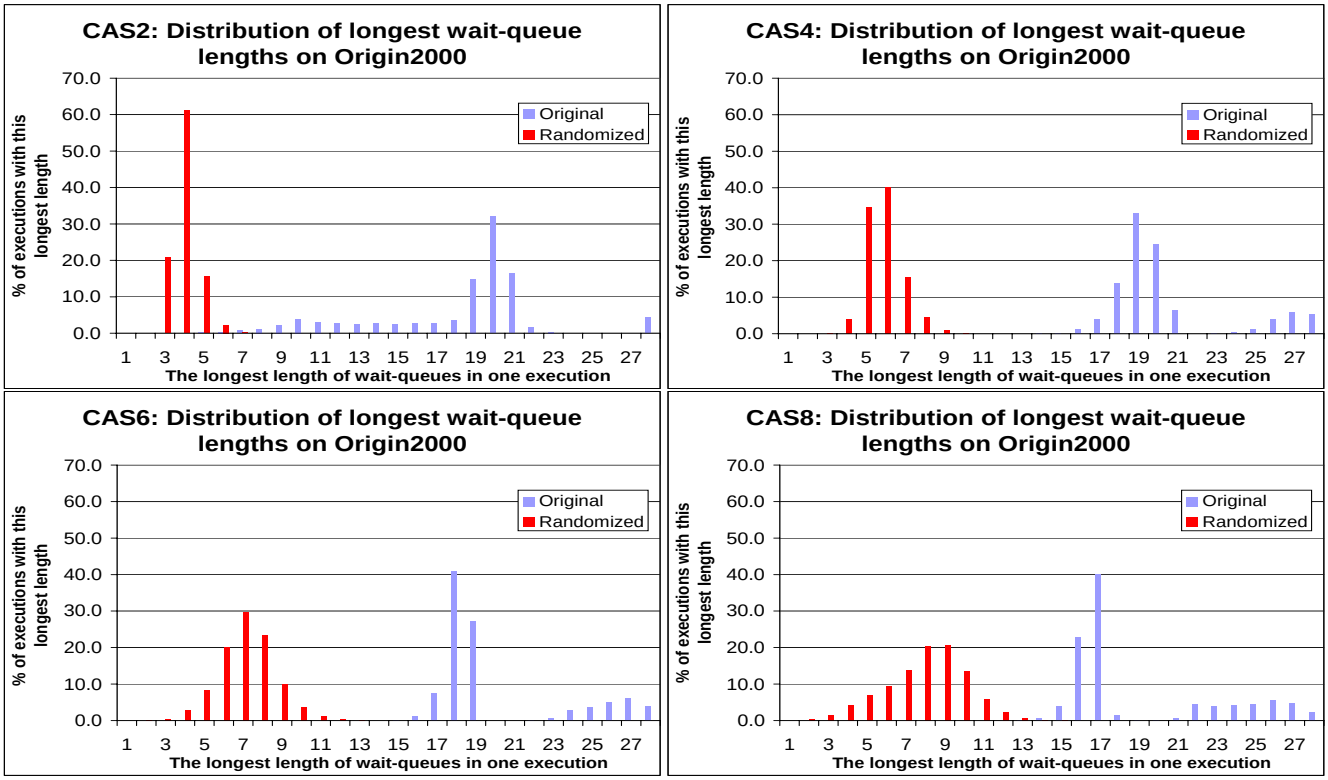

Figure 6: The distributions of the longest wait-queue lengths in the micro-benchmark on the SGI Origin2000.

we assume that it computes a prefix-sum

$$o_k = i_1 + i_2 + \cdots + i_k = o_{k-1} + i_k$$

for all k in $[2, n]$. The system writes the result o_k back to register r_k only if the values used to compute o_k have not changed yet. That means:

- either all registers r_1, r_2, $\cdots$, r_k have not changed yet if o_k is computed from i_1, i_2, $\cdots$, i_k, or
- registers r_{k-1} and r_k have not changed yet if o_k is computed from o_{k-1} and i_k, where o_{k-1} had been written successfully to register r_{k-1} and no new input i_{k-1} has been put in this register since o_{k-1} was written back.

The efficiency of the computational system is evaluated by the number of results written successfully. The more results are written successfully, the better the system is.

5.2.0.3 *A simple algorithm solving the problem:.*

The following two observations can be made:

- The results must be computed as fast as possible in order to write them back to the registers before new inputs are put in them.
- Using o_{k-1} and i_k to compute o_k has higher probability of success than using i_1, i_2, $\cdots$, i_k.

Therefore, we use n threads t_1, t_2, $\cdots$, t_n, where the main task of thread t_k is to compute o_k fast. The algorithm is illustrated in Figure 8.

In our experiment, the CASN operation in the algorithm was in turn replaced by RaCASN and DeCASN and then

---

while $True$ **do**
  Read registers from k to k', where register r'_k is the first reg. that is observed containing an output/result. Note $o_1 = i_1$.
  Compute o_k: $o_{k'+1} := o_{k'} + i_{k'+1}$; $\cdots$; $o_k := o_{k-1} + i_k$;
  if $CASN(\langle r_{k'}, r_{k'+1}, \cdots, r_k\rangle, \langle o_{k'}, i_{k'+1}, ..., i_k\rangle,$
    $\langle o_{k'}, o_{k'+1}, ..., o_k\rangle) = Success$) **then break**;
done

Figure 8: The algorithm for a thread t_k in computing one result/output

the average execution times of the application were measured over one million executions. The number of registers or threads n was varied from 4 to 28. The experiment with higher n generates higher collision level on the registers due to the helping policy. In the experiment, each thread ran exclusively on one processor of the Origin2000 machine. The result is presented in Figure 9.

5.2.0.4 *Results:.*

The experimental result shows that RaCASN helps the application run faster compared to DeCASN. It is up to 40% faster in the case of 28 registers or 28 threads. Figure 9 shows that with more threads the DeCASN/ReCASN speed-up relation grows. This implies that the randomization in RaCASN plays a significant role in reducing collisions on the shared registers, thus helping the application achieve better performance. The overhead of pre-computing n random numbers corresponding to n registers is not significant: it takes at most 0.7% of the execution time. (This worst case is measured in the case where the number of register is 4.)

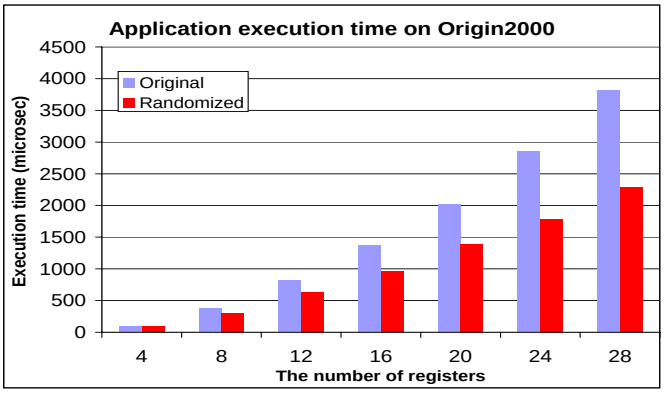

Figure 9: The application execution times on the SGI Origin2000.

6. CONCLUSIONS

In this paper we advocated randomization for implementing multi-locking such as CASN efficiently. We showed that our approach is efficient, in theory as well as in practice.

In the past, multi-lock algorithms were usually evaluated by random simulations. That is, in an evaluation/simulation of an algorithm it was assumed that *randomly* chosen registers were accessed by the processes. We believe that this is a conceptual *faux pas*. In fact, shared memory processes operate on shared data structures (e.g. search trees, linked lists) which are accessed anything but randomly. In reality, as in dining philosophers, access is not random but well-structured. For example, in a shared ordered linked list a process needs to multi-lock the two *neighbor* records in order to insert a new record.

By shifting the randomization from the simulation to the actual implementation our system is efficient in any application, as worst-case as it may be.

7. REFERENCES

[1] J. H. Anderson and M. Moir. Universal constructions for large objects. *Proc. of the International Workshop on Distributed Algorithms*, pages 168–182, 1995.

[2] H. Attiya, F. Kuhn, M. Wattenhofer, and R. Wattenhofer. Efficient adaptive collect using randomization. *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 159–173, 2004.

[3] G. Barnes. A method for implementing lock-free shared-data structures. *Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 261–270, 1993.

[4] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. *Proc. of ACM Symp. on Theory of Computing (STOC)*, pages 593–602, 1992.

[5] P. H. Ha and P. Tsigas. Reactive multi-word synchronization for multiprocessors. *The Journal of Instruction-Level Parallelism*, page http://www.jilp.org/vol6/v6paper3.pdf, 2004.

[6] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 265–279, 2002.

[7] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 151–160, 1994.

[8] N. Linial. Locality in distributed graph algorithms. *SIAM Journal on Computation*, 21(1):193–201, 1992.

[9] N. A. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal for Computer and System Sciences*, 23(2):254–278, 1981.

[10] M. Moir. Transparent support for wait-free transactions. *Proc. of the Intl. Workshop on Distributed Algorithms*, pages 305–319, 1997.

[11] M. Naor and L. Stockmeyer. What can be computed locally? *SIAM Journal on Computation*, 24(6):1259–1277, 1995.

[12] N. Shavit and D. Touitou. Software transactional memory. *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[13] E. Styer and G. L. Peterson. Improved algorithms for distributed resource allocation. *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 615–628, 1988.