GENERALIZED OPTIONAL LOCKING IN DISTRIBUTED SYSTEMS

Thomas Schöbel-Theuer, Universität Stuttgart, Germany, email: schoebel@informatik.uni-stuttgart.de

ABSTRACT
We present optional locking as a method for significantly speeding up distributed locks, and we generalize it to multiple lock types obeying a conflict relation. The generalized version can simulate the message passing paradigm on top of Distributed Shared Memory (DSM) with no more messages than explicit message passing would need. Thus we argue that message passing can be viewed as a true special case of optional locking. As a consequence, the attractiveness of DSM programming models should increase significantly due to well-recognized advantages such as simplicity and reduced software engineering cost; mixtures of both message passing and shared data access patterns can be treated *uniformly*. Measurements and simulations based on database benchmarks indicate a substantial performance improvement of optional locking over conventional locking even in presence of multiple lock types.
KEY WORDS
Distributed Locking, Distributed Shared Memory, DSM, Distributed Algorithms

1 Introduction

1.1 Problem

The *mutual exclusion problem* [1] occurs whenever multiple processes have to cooperate on shared data. While efficient solutions for concurrently scheduled processes [2] and for multiprocessors [3] have been known for a long time, solutions for distributed systems [4] suffer significantly from network latencies and bottlenecks. There is less work on distributed semaphores [5], and common belief is that their performance is *so extremely poor* that distributed systems require totally different programming concepts and paradigms from standalone systems.

Due to massive performance degradation caused by the *distributed mutual exclusion problem*, most distributed systems are built on top of explicit message passing. Even distributed shared memory (DSM) systems [6] try to avoid that problem; instead weak memory consistency models are advocated for enhancing concurrency and decreasing contention. However, weak consistency models complicate programming, disturbing the well-recognized cost-effectiveness of software engineering under DSM.

The DSM model promises to reduce software engineering cost significantly in comparison to explicit message passing. In practice, explicit message passing is nevertheless preferred over DSM in many cases, although passing of references and simulation of shared (composite) objects at application level may become extremely cumbersome. Since DSM has not met all expectations, some people even regard research on (software) DSM as a dead topic. In our opinion, one of the reasons is that message passing on top of DSM is extremely inefficient due to the poor performance of the necessary additional distributed locking overhead. Since many distributed problems *require* some kind of message passing behaviour, the conceptual elegance of DSM and its full potential for reducing software development cost cannot be exploited in many cases.

1.2 Main Results / Conclusions

This paper presents a method called *generalized optional locking* for speeding up distributed locks by exploitation of both spatial and temporal locality of access behaviour. Experimental results indicate that high speedups of one or two orders of magnitude and more are possible when compared to conventional distributed locking, and even significant speedup factors when compared to some known temporal lock caching strategies.

Moreover, we point out that message passing on top of DSM can be implemented with no additional overhead in the number of messages and network latencies, provided that DSM is combined with a variant of generalized optional locking. Thus the message passing paradigm *itself* may be viewed as a special case of optional locking.

In consequence, the main obstacle of DSM can be overcome: there is no longer a need for a separate message passing API when application behaviour is dominated by message passing or by mutual exclusion access patterns. DSM combined with generalized optional locking can do it all, can do it efficiently, and can do it within a *uniform* programming model.

1.3 Significance

First, the distributed mutual exclusion problem can be treated much more efficiently by exploitation of both spatial and temporal behaviour of applications. Our experiments indicate possible speedup factors from around 10 to more than 100 in some cases.

Second, DSM should become more attractive for a larger application area than is now. Overhead for marshalling and creation of additional interfaces like CORBA objects can be omitted in many cases, leading to substantial reductions of software development cost in large (homogenous) systems. The problems of heterogenous systems should be solvable by future middleware concepts

based on DSM. In essence, the programming model for distributed systems should become as easy and simple as the programming model for standalone systems, without *additionally* sacrificing performance by that model. This is contrary to some common belief. We open the door to substantial reductions of application software complexity and development cost.

1.4 Key Ideas

In place of using substitute objects like semaphores, we propose to issue lock requests directly on the *memory regions* occupied by shared objects (principle of *direct manipulation* applied to locked objects). Region locks may overlap and conflict in multiple different ways; unlocking may be done in a different granularity from locking. This alleviates not only fine-grained control over locking granularity (e.g. different lock sizing strategies for aggregated objects) at the application level, but also allows for *automated prefetching* of *speculatively enlarged* lock regions at the system level.

We introduce two different kinds of locks: *obligatory* locks and *optional* locks. While an obligatory lock *must* be granted at some time (otherwise the requesting process cannot continue, e.g. deadlock or demanding a transactional rollback), optional locks can be granted with *smaller size* than requested, or probably need not be granted at all. The *size* of optional locks can be dynamically adjusted to the *working region behaviour* of distributed processes. Whenever an optional lock has been granted to some network site, it can be locally converted to (multiple) (smaller) obligatory locks at any time without causing any network traffic or latency.

In order to shield applications from using optional locks directly, we present a general algorithm for prefetching of optional locks and local conversion to obligatory locks, which can be executed by *local lock managers* in a distributed system.

Emulation of message passing on top of DSM is a simple special case of optional locking, where the size of locked / unlocked regions is dynamically shrunk / enlarged according to the filling level of a shared communication buffer.

1.5 Related Work

Optional locking is different from conventional optimistic lock prefetching, in particular because of *granularity* matters: optional locks usually have coarser granularity than obligatory locks. Due to dynamic size adjustment of optional locks, the choice of granularity is *automated*.

Almost all current locking schemas employ non-overlapping substitute objects like semaphores, monitors, database locks (e.g. in PostgreSQL [7]), and the like. They can be characterized as having a *unique identity*, which either does not overlap with another object instance, or overlaps identically (e.g. in case of a conflict).

Hierarchical / intentional locks [4] and opportunistic locks [8] also belong to the class of uniquely identifiable locks. The latter may be viewed as a variant of hierarchical locks with an additional lock retraction mechanism. Hierarchical locks may be characterized as introducing additional lock objects each *replacing* or *subsuming* a set of other locks. This introduces a tree-structured order on the set of objects, such that the order has to be *explicitly* used and obeyed by the synchronisation partners.

Temporal prefetching of non-overlapping locks has appeared in the literature, e.g. [9] augmenting applications with specialized lock prefetch instructions similar to hardware data prefetch instructions [10], and [11] automatically predicting future lock operations from past temporal behaviour.

Region locks with possible fine-grained overlapping have been implemented in Unix [12]. They can be characterized as having no unique identity during their lifetime, since locking may merge regions, and unlocking may split regions. We found only very few serious open source applications actually using them for fine-grained locking beyond simple whole-file locks; closed-source applications are hard to check. There is a special non-default configuration option in MySQL enabling them for cooperation of multiple server process instances, but in our attempt with a newer version of MySQL it did not compile.

We don't know of any publication of the idea of optional locking prior to [13] and [14].

2 Optional Locking

This section is a condensed informal introduction to the basic ideas of optional locking on a mutually exclusive lock type. We assume that locks are always issued on (possibly overlapping) *regions* of an address space similar to Unix `lockf()` or `fcntl()` locking [12] applied to an `mmap()`ed area; unlocking may be done in different granularity from locking. In this section, we assume only a single lock type (exclusive locks) which cannot be granted to several sites in parallel.

The central idea of optional locking is to discriminate between two kinds of lock requests: *obligatory* lock requests, and *optional* lock requests. Obligatory lock requests must be granted to an application at some time, otherwise the application cannot continue with its work (deadlock). In contrast, optional locks need not be granted, or need only be granted *partially*. In particular, a granted optional lock may be *smaller* than requested.

Conventional applications will normally issue only obligatory lock requests; it will be the task of *local lock managers* in a distributed system to automatically add appropriate optional lock requests for overall minimization of network traffic and latencies. Whenever an optional lock has already been granted to a local site in the network, it may be (partially) converted to one or many (smaller) obligatory locks at any time without causing any network

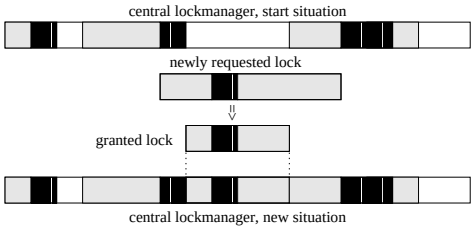

Figure 1. Example

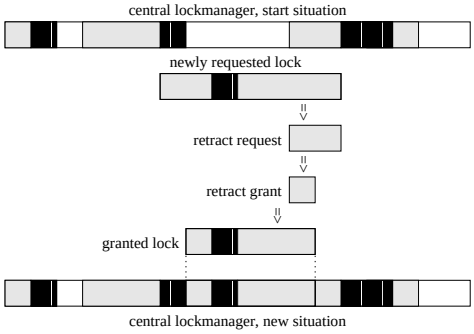

Figure 2. Possible Retract

traffic. When applications show high spatial locality of access behaviour for their obligatory locks, speculatively enlarged optional lock regions may transfer exclusive access rights for many smaller obligatory locks over the network bottleneck in a single transfer, leading to substantial performance improvements.

In the example figure 1, obligatory lock regions are depicted as black areas, optional lock regions as grey areas, and free areas are left white. The upper long stripe shows the initial situation at the central lock manager. Some regions have been optionally locked, and some subsets of them have been obligatorily locked. Then a new lock request arrives from some local lock manager, consisting of an obligatory lock surrounded by a larger optional lock region which has been speculatively enhanced from the obligatory region. Since optional lock regions can only be exclusively granted to a single requestor, the optional part must be shortened in order to immediately fulfill the request. The granted optional lock is thus a smaller subset of the requested one.

Figure 2 shows a variant of the same initial situation: at the right side, the requested optional part and the already granted optional part of another site are overlapping. Instead of leaving that optional part to the current holder (aka ”first come, first serve”), we could try to get back some optional part from that third party:

Upon detection of an overlap of an optionally requested part with an already granted optional part, the central lock manager policy may decide to issue a *retract request* to the current holder of that optional part. That current holder responds with a *retract grant*, which may be a subset of the retract request. Of course, only optional parts which have not yet been converted to obligatory parts can be retracted without waiting. When a conflicting obligatory lock has been granted locally in the meantime, the retract grant will be smaller than the retract request. In our example, the local lock manager policy has decided to bisect the transferable retract area and to keep the remaining half for itself, speculating that it could be needed locally in the future. As a final result, the original requestor can be granted a larger optional lock than compared with the first example.

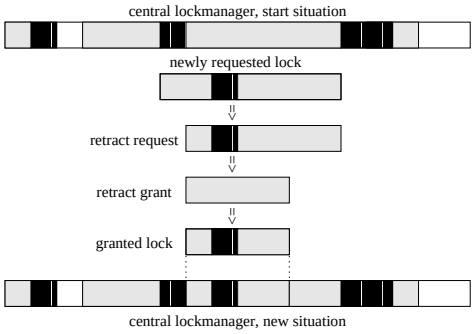

Figure 3. Mandatory Retract

In figure 3 the retraction mechanism is necessary for avoiding deadlock. Here the initial situation is different from the first two examples: the requested obligatory part conflicts with an already granted optional part. If the central lock manager would not decide to send a retract request, deadlock would occur if nothing would ever ”force” the holder of that optional region to return it. Thus the retract request is enriched with the obligatory part in order to tell the current owner of the optional part that he *must* give back at least that part for avoiding potentially unnecessary deadlock. In the example, the optional part between both obligatory locks is bisected by the local lock manager policy. In general, there is only one reason for denying retraction of an obligatory part: when another conflicting obligatory lock is already locally present, we must wait until it has been released by the application. If the application will never release it, deadlock will occur, but it will also occur if we omit any optional locking completely. We just have to ensure that we don’t introduce *additional* deadlocks by optional locking.

3 Message Passing and Multicast

Conventional DSMs are often implemented on top of software message passing, but not always. In case of hardware DSM implementations, the granularity of messages, the communication protocols, and properties like atomicity may be different from software communication needs. Thus it makes sense to look at the converse: simulation of message passing on top of DSM. Another moti-

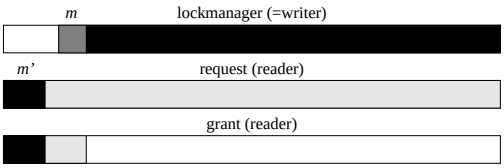

Figure 4. Simulation of Message Passing by Optional Locking

vation comes from the well-known observation that message passing on top of conventional software DSMs is extremely slow due to additional latencies caused by distributed semaphore operations or the like. This is an obstacle for the DSM programming model, although it otherwise has a well-recognized high potential for simplifying programming of distributed systems; overhead like marshalling and creation of additional interfaces like in CORBA can be saved, leading to lower software development cost. We believe that bad message passing performance over DSM could have had a high impact at the wide-spread practical preference of explicit message passing over DSM. We present a method for overcoming this obstacle.

The following picture illustrates the basic idea for simulation of message passing on top of DSM combined with optional locking and an appropriate prefetching strategy for linear scans. To simplify the presentation and to omit discussion of some unrelated problems, we assume the existence of an infinite message buffer. Generalizations to finite circular ring buffers etc. are left to the reader. Furthermore, we assume that data transfer is integrated with lock / unlock messages and thus does not cause extra messages.

In the first model, we assume exactly one writer and one reader. Both are assumed to have a local read-pointer resp. write-pointer indexing the buffer at the current read or write position. Furthermore, we assume that the writer is identical with the central lock manager.

Figure 4, first strip: the writer had locked the whole buffer upon initialization of the system using an exclusive write-lock, and currently has already released some part of that lock at the left. Writing m bytes works as follows: at the write pointer position (coinciding with the start of the locked area), m bytes are written (depicted dark grey). Afterwards that area is unlocked (leaving the black rest) und the write pointer is incremented by m. Next strip: the reader then tries to aquire an obligatory non-exclusive read-lock (also depicted black) with requested size m' at its current read pointer position. An optional read-lock request for the whole buffer (depicted light grey) is added by the local lock manager. The last strip shows the granted locks. In general, m' bytes can only be atomically read at the read pointer position and the read pointer be incremented by m' after the obligatory part has been fully granted.

When this game is repeated n times at both sides in an interleaved way with $m = m'$, the reader generates n lock requests and the writer responds with n grants, leading to a total of $2n$ messages in worst case. Retracts are not necessary due to the assumption of identity of writer and manager[1]. These numbers of messages correspond to message passing with explicit acknowledge of receipt of each individual message.

However, when the reader gets late with respect to the writer, the number of grants issued by the writer may be smaller than n, transferring multiple blocks of size m with a single message. This corresponds to *message coalescing* in communication systems, but performed *automatically* by the lock manager strategy.

In a second model, the total number of messages can be decreased from $2n$ to $n + 1$ if the algorithm is modified as described informally: optional lock requests are buffered at the central manager for a longer time, until they have been either completely fulfilled or until cancelled by the original requestor. Each time a new part of the optional region becomes available, the corresponding optional part is sent to the requestor *incrementally*. With that modification, only the first request of the reader need to specify the optional read lock. The first request issued by the reader corresponds to channel setup in conventional message passing systems. In this model, network latencies for transfer of data will be comparable to conventional "push"-strategies of message passing systems. By requesting optional regions of some certain size w, mixed strategies between both models can be created which roughly correspond to windowing in conventional message passing systems (e.g. TCP windows in the Internet Protocol).

It is easy to see that all mentioned model variants can be directly used to support multiple readers, thanks to usage of read-locks versus write-locks. Since each reader has its own local read pointer, arbitrary delays will be handled correctly. Similar to notes in [14], specializations to finite circular ring buffers or other communication structures are possible and left to the reader.

Finally, note that this usage of optional locking does not exploit the full power of it. In some sense, locks are "misused" for *one-way* synchronization between writer and reader. In general, locks may be used for achieving arbitrary *permutations* of access synchronization (*mutual* exclusion), which is not used here. Thus we conclude that usage of optional locking this way is a *true special case* of general optional locking, and that the synchronization behind the message passing paradigm can be viewed as a *true special case* of optional locking.

4 Generalized Algorithm

In the following semi-formal description of a generalized algorithm, nondeterministic choices are indicated by the

[1] The same number of messages besides initialization will occur if the reader instead of the writer is made identical to the manager; the difference is only that retract requests and grants are exchanged instead of direct requests and grants.

wording "choose some ...". We don't discuss strategies and heuristics for nondeterministic choices in this paper.

Let A be a totally ordered set of addresses. Let T be an alphabet, called *lock types alphabet*. Let $C \subseteq T \times T$ be a symmetric relation, called *lock conflict table*. See figure 6 as an example. In contrast to [14], there may exist lock types which don't conflict among each other, i.e. they may be granted to many network sites in parallel.

A lock type t_1 is said to be *weaker than* t_2, denoted $t_1 \leq t_2$ iff for all $t' \in T$: $(t', t_1) \in C \Longrightarrow (t', t_2) \in C$. Intuitively, this means that t_2 is "at least as strong" as t_1 with respect to conflicts with other locks. For example, classical read locks are weaker than write locks. In general the lock weakness relation is a partial order. As an example, figure 7 shows the lock weaknesses of PostgreSQL as derived from the lock conflict table from figure 6.

A *lock* l is a pair $l = (I, t)$ consisting of a *locking interval* $I = [a, b]$ with $a, b \in A$, $a \leq b$ and a lock type $t \in T$. Two locks $l_1 = ([a_1, b_1], t)$ and $l_2 = ([a_2, b_2], t)$ with same lock type t are said to be *mergeable to* $l_3 = ([a_3, b_3], t)$, iff $[a_1, b_1] \cup [a_2, b_2]$ is an interval (without a hole) and $[a_3, b_3] = [a_1, b_1] \cup [a_2, b_2]$. A set of locks L is said to be *reduced*, iff all pairs of mergeable locks have been merged. In the sequel, we assume that all sets of locks are always kept in reduced state.

Two locks $l_1 = ([a_1, b_1], t_1)$ and $l_2 = ([a_2, b_2], t_2)$ are said to *conflict*, denoted $l_1 \dagger l_2$, iff $[a_1, b_1] \cap [a_2, b_2] \neq \emptyset$ and $(t_1, t_2) \in C$. A lock l is said to *conflict with a set of locks* L, denoted $l \dagger L$, iff there exists an $l' \in L$ with $l \dagger l'$. Similarly, two sets of locks L_1 and L_2 are said to *conflict with each other*, iff there exists $l_1 \in L_1$ and $l_2 \in L_2$ with $l_1 \dagger l_2$.

A lock $l_1 = ([a_1, b_1], t_1)$ is said to be *present in* lock $l_2 = ([a_2, b_2], t_2)$, denoted $l_1 \leq l_2$, iff $[a_1, b_1] \subseteq [a_2, b_2]$ and $t_1 \leq t_2$. When additionally $t_1 = t_2$ holds, we say that l_1 is *strongly present* in l_2, denoted $l_1 \leqq l_2$. A lock l is *present* in set L iff there exists $l' \in L$ with $l \leq l'$. Similary, a set L_1 is *present* in L_2 iff all $l \in L_1$ are present in L_2. The same notions are analogously defined for *strong* presence by denoting $\leqq$ instead of $\leq$.

The *subtracted set* $L' := L - l$ with L being a set of locks and l a lock, is defined as the largest $L' \leq L$ such that l does not conflict with L'.

Let S be a finite set of *network sites*, with $\overline{s} \in S$ being a special site called *central server*. We use the following sets of locks in our algorithm as illustrated in figure 5:

L_s is the set of locally aquired *obligatory locks* at each site $s \in S$, and R_s the set of *reserved optional locks*. Consequently, $L_{\overline{s}}$ and $R_{\overline{s}}$ are the sets of *obligatory* resp. *reserved optional* locks as known by the central lock manager. $R_{\overline{s}}^{Pend}$ denotes the set of all *pending optional retract requests* known to the central manager, which are currently processed by other sites.

In order to model network messages flowing through the network, we introduce the following sets of locks: $LR_{s,\overline{s}}^{Rq}$ is the set of *(obligatory, optional) lock request pairs* currently flowing from some site s to the central lock manager $\overline{s}$. $LR_{\overline{s},s}^{Gr}$ is the set of *(obligatory, optional) lock grant pairs* currently flowing from the central lock manager $\overline{s}$ to some network site s. Local *lock aquisitions* are propagated asynchronously to the central manager via the set of locks $L_{s,\overline{s}}^{Lock}$. Lock releases are similarly propagated via $L_{s,\overline{s}}^{Unlock}$.

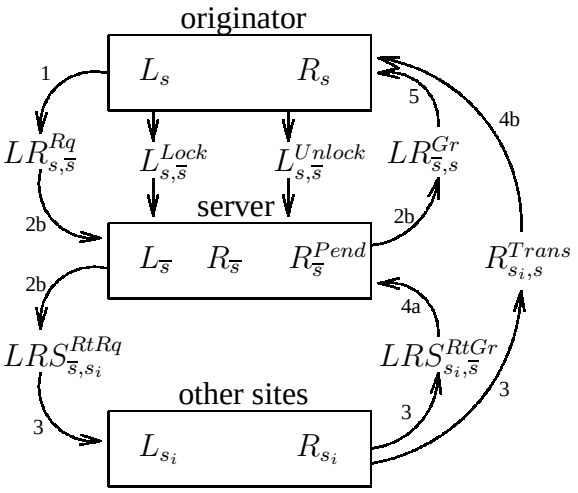

Figure 5. Data Flow

In order to describe retract requests and retract grants, we introduce the notation $LRS_{\overline{s},s_i}^{RtRq}$ for *retract request triples* (obligatory, optional, originating site) flowing from the central manager to site s_i, and $LRS_{s_i,\overline{s}}^{RtGr}$ for *retract grants* flowing in the opposite direction. We add the ability to transfer an optional lock directly from a site s to another site s' by the notation $R_{s,s'}^{Trans}$, denoting *transfer optional locks*. All these sets are assumed to be empty at startup of the distributed system.

Each step of the following algorithms is assumed to run atomically, even if different threads are running in parallel. All steps except step 1 are assumed to be executed by some (server) thread in an endless loop.

Algorithm A (aquisition of obligatory lock $l = ([a, b], t)$ at local site s):

Step 1 (at local site s): if $l \leq R_s$, then if $l \dagger L_s$ then block the calling thread, else add l to L_s and to $L_{s,\overline{s}}^{Lock}$ and end the algorithm. Else, choose some optional lock r with $l \leq r$ (for example, take the largest r with $l \leqq r$ such that $\neg(r \dagger R_s)$). Add (l, r) to $LR_{s,\overline{s}}^{Rq}$ (i.e. send it as request pair to the central manger) and block the calling thread.

Step 2a (at central manager $\overline{s}$, executed by some server thread): upon receipt of request $l \in L_{s,\overline{s}}^{Lock}$, remove it from $L_{s,\overline{s}}^{Lock}$ and add it to $L_{\overline{s}}$.

Step 2b (at central manager $\overline{s}$): upon receipt of request pair $(l, r) \in LR_{s,\overline{s}}^{Rq}$, remove it from $LR_{s,\overline{s}}^{Rq}$. Choose some r' with $l \leq r' \leq r$ (e.g. such that there is no conflict $r' \dagger L_{\overline{s}}$). Wait until r' does not conflict with $R_{\overline{s}}^{Pend}$. If r' does not conflict with $R_{\overline{s}}$, then add (l, r') to both $(L_{\overline{s}}, R_{\overline{s}})$ and to $LR_{\overline{s},s}^{Gr}$. Else (when r' conflicts with $R_{\overline{s}}$), add it to $R_{\overline{s}}^{Pend}$

and add (l, r', s) to $LRS^{RtRq}_{\overline{s},s_i}$ for each site $s_i \notin \{s, \overline{s}\}$. [2]

Step 3 (at each local site s_i): upon receipt of retract request $(l, r', s) \in LRS^{RtRq}_{\overline{s},s_i}$, remove it from $LRS^{RtRq}_{\overline{s},s_i}$. Wait until there is no conflict $l \dagger L_{s_i}$. Choose some r_i with $l \leq r_i \leqq r'$ [3] such that there is no conflict $r_i \dagger L_{s_i}$. Subtract r_i from R_{s_i}. Add (l, r_i, s) to $\mathrm{L}RS^{RtGr}_{s_i,\overline{s}}$ and add r_i to $R^{Trans}_{s_i,s}$.

Step 4a (at central manager $\overline{s}$): wait until all $(l, r_i, s) \in LRS^{RtGr}_{s_i,\overline{s}}$ with $r_i = ([a_i, b_i], t_i)$ have arrived from all sites $s_i \notin \{s, \overline{s}\}$. Compute $r'' := (\bigcap [a_i, b_i], \min\{t_i\})$. Remove each (l, r_i, s) from each $LRS^{RtGr}_{s_i,\overline{s}}$. Remove the old r' from $R^{Pend}_{\overline{s}}$. Add (l, r'') to $(L_{\overline{s}}, R_{\overline{s}})$.

Step 4b (at local site s): wait until all $r_i = ([a_i, b_i], t_i) \in R^{Trans}_{s_i,s}$ have arrived from all sites $s_i \notin \{s, \overline{s}\}$. Compute $r'' := (\bigcap [a_i, b_i], \min\{t_i\})$. Remove each r_i from each $R^{Trans}_{s_i,s}$. Add r'' to R_s. Unblock all threads waiting for lock l and let them re-execute step 1.

Step 5 (at local site s): upon receipt of grant (l, r'), remove it from $LR^{Gr}_{\overline{s},s}$. Add r' to R_s. Unblock all threads waiting for lock l and let them re-execute step 1.

Algorithm B (release of a lock $l = ([a, b], t)$ at local site s):

Step 1 (at local site s): subtract l from L_s. Add l to $L^{Unlock}_{s,\overline{s}}$. Unblock any threads waiting for lock l (if any) .

Step 2 (at central server $\overline{s}$, executed by some server thread): upon receipt of release request $l \in L^{Unlock}_{s,\overline{s}}$, remove it from $L^{Unlock}_{s,\overline{s}}$ and subtract it from $L_{\overline{s}}$.

Note that this version of the algorithms will never release an optional lock without explicit demand by a retract request; thus $R_{\overline{s}}$ will always grow and never shrink. Improved versions should use some LRU-like aging strategy for returning optional regions which have not been accessed for a longer time; this will decrease the probability for retracts becoming necessary.

In the following correctness proof, we consider all possible data flow paths through algorithm A: path0 is $1 \rightsquigarrow 2a$, path1 is $1 \rightsquigarrow 2b \rightsquigarrow 5$, and path2 is $1 \rightsquigarrow 2b \rightsquigarrow 3 \rightsquigarrow \left\{ \begin{array}{l} \rightsquigarrow 4a \\ \rightsquigarrow 4b \end{array} \right.$.

Lemma 1: $L_s \leq R_s$ always holds.

Proof: at startup, it holds trivially. The only place where L_s is increased is step 1 of algorithm A, and it occurs only after the added lock is ensured to be present in R_s. The only place where some R_s is shrunk is step 3, and it occurs only when the removed part r_{s_i} does not conflict with L_{s_i}. Thus the condition can never be violated. □

Lemma 2: $R_s \leq R_{\overline{s}} \cup R^{Pend}_{\overline{s}}$ for any site s.

Proof: startup is trivial. Path0 does not change anything relevant for the condition. Whenever path1 adds some r' to R_s at step 5, it has been added to $R_{\overline{s}}$ previously at step 2b. When path2 adds some r'' to R_s at step 4b, an $r' \geq r''$ had beed added to $R^{Pend}_{\overline{s}}$ at step 2b before. When r' is removed from $R^{Pend}_{\overline{s}}$ at step 4a, the same r'' as computed in step 4b is added to $R_{\overline{s}}$ instead, not violating the original condition even if step 4a is executed after step 4b (conversely, it will hold anyway). $R^{Pend}_{\overline{s}}$ is only shrunk at step 4a without causing harm, and $R_{\overline{s}}$ is never shrunk. Thus the condition can never be violated. □

Lemma 3: All R_s are always pairwise non-conflicting, i.e. $\neg(R_{s_i} \dagger R_{s_j})$ for $i \neq j$.

Proof: startup is trivial. Path0 does not change any R_s. When path1 adds some r' to R_s at step 5, the $r \geq r'$ from the previous step 2b did not conflict with $R_{\overline{s}} \cup R^{Pend}_{\overline{s}}$; due to lemma 2 it also cannot conflict with any other R_{s_j}. When path2 adds some r'' to R_s in step 4b, an $r_i \geq r''$ has been subtracted from R_{s_i} at all other sites s_i, making them non-conflicting. Thus the condition can never be violated. □

Theorem 1: All L_s are always pairwise non-conflicting.

Proof: follows directly from lemma 3 and lemma 1. □

Lemma 4: whenever R_s is increased by some r''' on behalf of some l, that l is present in l'''.

Proof: path0 does not apply. When path1 adds $r' =: r'''$ to R_s at step 5, $l \leq r'$ has been ensured by the previous step 2b. When path2 adds $r'' =: r''' = (\bigcap [a_i, b_i], \min\{t_i\})$ to R_s at step 4b, all other sites s_i have obeyed $l \leq r_i = ([a_i, b_i], t_i)$ at their previous step 3, implying $r'' \geq l$. □

Theorem 2: provided that communication is reliable and that arbitrary permutations of mutually conflicting obligatory locks l can never lead to a deadlock with purely obligatory locking, algorithm A has no deadlock, too.

Proof: When there are no infinite waits inside of steps 2b, 3, 4a or 4b, any of the paths will finally complete due to the assumption of reliable communication, and due to lemma 4 ensuring that each thread delayed at step 1 can eventually continue. It remains to show that none of the waits (if any) is infinite. We can totally order all starts of executions of any steps via a physical Lamport clock [15]. If there are some infinite waits, one of them must have started first according to that ordering. The waits at steps 4a and 4b cannot occur as the first infinite one, because they depend on the waits at step 3 one of them must then have occurred as the first infinite one. The wait at step 3 is always finite due to the assumption of deadlock-freedom at the obligatory level which guarantees that any local obligatory lock will be released after finite time. The wait at step 2b depends on release of $R^{Pend}_{\overline{s}}$ which must occur after finite time (otherwise it would not be the first infinite wait, leading to a contradiction). Thus there is no first infinite wait at all. □

[2] The algorithm can certainly be improved in various ways. In particular, step 2b needs to send retract requests only to those sites actually possessing a (potentially) retractable lock; doing so would complicate step 4b somewhat. We have left out bookkeeping of ownership of locks to simplify matters for easier understanding of the basic principle.

[3] When $\leq$ on T is a total order (at least in the relevant part), we can use $\leq$ instead of $\leqq$ at this condition. Otherwise $\min\{t_i\}$ could not be unique later at steps 4a and 4b.

5 Experiments

Many applications use different lock types such as read-locks versus write-locks, in order to reduce the probability of lock contention. Database systems often use even more lock types, such as intentional or hierarchical locks [16]. PostgreSQL [7], for example, uses 8 different lock types with a compatibility matrix as depicted in figure 6. Combining optional locking with different lock types is expected to improve the performance of distributed locking.

We implemented the algorithm from section 4, simularing a client-server model consisting of a central server and multiple clients. The experiment is tailored to mutual exclusion, not message passing. Other variants like replicated servers or arbitrarily distributed replicas among clients can be derived from it; for some general ideas see [14].

We measured the runtime locking behaviour of three different database benchmarks, called DBT1 version 1.3, DBT2 version 0.21 (beta stage), and DBT3 version 1.3 [17], which should be very similar to the well-known TPC-W, TPC-C, and TPC-H database benchmarks issued by the Transaction Processing Council, respectively. For some differences between these benchmarks see [17]. Synthetic database benchmarks can tell us some *trends* about the mutual exclusion behaviour of *some* kinds of applications, and about the *speedup potential* of optional locking achievable with *some* distributed applications.

Our experiments were carried out on a dual processor 1.8 GHz Opteron machine with 6 GB main memory, running under a 64-bit version of SuSe Linux version 9 and kernel 2.6.0. We installed PostgreSQL version 7.3.4. We chose that system because of easy access to source code.

We modified PostgreSQL by a patch of the internal lock manager (file `src/backend/storage/lmgr/lock.c`) with `printf()` statements. It writes the type of operation (lock / unlock), the realtime timestamp, the process id of the backend process, lock mode, transaction id, the database id $dbID$, internal relation id $relID$, and the internal object id $objID$ of PostgreSQL backend processes to a log file. The latter three values identify a lock uniquely when taken together. PostgreSQL was designed with uniquely identifiable locks in mind, not with region locks. In order to approximate our kind of locking, we remapped the database id, relation id and object id to a single hypothetical linear address space by the formula $addr = objID + (max\_objID - min\_objID + 1) * (relID + (max\_relID - min\_relID + 1) * dbID)$, such that each PostgreSQL lock was treated as an obligatory lock of length 1.

We wrote a simulation program executing the algorithm from section 4 on the measured locking patterns, as if they had been executed on n sites of a network, with n being a choosable parameter. Only locks stemming from different PostgreSQL backend server processes were spread to the simulated network, i.e. we simulated distribution of nearly unmodified PostgreSQL server processes.

We compare 5 different versions of possible prefetching strategies:

a) at step 1 of algorithm A, choose always the full address space and the same lock type for requesting optional locks. At steps 2b and 3, always choose the maximum region satisfying the respective condition.

b) like strategy a), but at step 1 request the largest area at both sides of the requested lock up to the nearest globally known obligatory lock.

c) like strategy a), but at step 3 choose only the *half* of the available area at both sides of the obligatory part for retraction (bisecting strategy).

d) like c), but at step 1 use strategy b).

e) at step 1), choose the *smallest* possible optional request, which is always the same as the obligatory lock. This results in no spatial prefetching at all, but retains temporal caching of locks (cf. [14]).

Each benchmark was run with different scale factors (depending on the benchmark type [17]). The tables in the appendix show the hit rates in percent (i.e. percentage of times a lock request can be granted without any network communication), as if the backend processes had been spread to $n \in \{1, 2, 4, 8, 16, 32\}$ simulated network sites. The last column shows absolute and relative speedup factors for $n = 32$. The absolute speedup factor is (100 / miss rate), where miss rate is (100 - hit rate); when there are no hits, the speedup factor will be 1. This is motivated by the observation that network latencies are usually many orders of magnitude slower than processor cycles, so we approximately neglect local operations and count only the number of network transmission cycles which would occur when caching of locks is either enabled or disabled. The relative speedup factor relates the miss rates of strategy a) to d) to strategy e), showing the benefit of spatial prefetching in isolation.

We ran two simulations on each benchmark: the first simulated generalized optional locking obeying the PostgreSQL lock types. The second treated all lock types equally as exclusive locks (cf. results in [14]), which may lead to some distortions, but roughly indicates the influence of lock types when compared to the first simulation.

Although the experiments may contain some distortions as more deeply discussed in [14] and although they are probably not representative for all possible kinds of applications, we cautiously try to identify some trends. Treating all lock types as equal appears to be an order of magnitude worse than using generalized optional locking. Different applications may perform very differently: while dbt1 and dbt3 appear well-suited for generalized optional locking, the relative speedup of dbt2 does not take off in the generalized version as the others. This behaviour is expected, because TPC-C executes many short-time transactions. A look at the log file reveals that only a small number of locks is issued per transaction, and a great deal of locks is allocated with linearly incremented lock numbers serving as pseudo-locks for transaction IDs. When adjacent members of the linear scan are allocated by different network

sites, spatial prefetching in that area is near its worst case (thrashing) and will not yield any advantage over purely temporal prefetching.

When interpreting the simulation results, we should be aware that PostgreSQL has neither been designed with region locks or spatial behaviour of locks in mind, nor has it been designed for running distributedly. Thus there are high chances that there may exist other applications which will profit from optional locking even more than indicated by our benchmarks. On the other hand, we found a case very close to the worst case. In summary, our results suggest that much more in-depth exploration of generalized optional locking is necessary.

6 Future Research

Generalized optional locking may become a good candidate as a base mechanism for future distributed systems when combined with DSM. Contrary to some common belief, it is possible to obtain good performance of message passing on top of DSM when employing generalized optional locks. Thus it is feasible to build up distributed systems on the DSM paradigm solely as a uniform base mechanism. This would enable many advantages, in particular allow easy passing of references and remove the well-known annoying difficulties of simulating shared (composite) objects at application level by explicit message passing.

In order to become practical, certainly much more work has to be done on (generalized) optional locking and on integration with DSM.

Preferring the DSM model even for message passing will not render middleware superfluous. There remain problems like security, safety, recovery, heterogenity of hardware / software platforms and architectures, and many more. Middleware will have to focus on these issues under changing circumstances. For example, large parts of middleware could be implemented as shared libraries under DSM, provided that architecture-independent fat binaries or other relocation / indirection / translation mechanisms are available.

Our experiments and simulations indicate that generalized optional locking is also a good candidate for speeding up distributed locks, even if not used for message passing in a narrow sense, and even under control of current middleware paradigms.

There remains much work to be done, in particular on models of application behaviour, prefetching strategies, thrashing detection and prevention, optimization of the working region behaviour of applications, performance forecasts for particular application behaviours, fault tolerance, and many practical aspects of implementation and handling. This will certainly require cooperation among many researchers and research disciplines.

References

[1] P. B. Hansen, "Concurrent programming concepts," *Computing Surveys*, vol. 5, no. 4, pp. 223–245, 1973.

[2] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *CACM*, vol. 8, no. 9, p. 569, 1965.

[3] H. S. Bright, "A philco multiprocessing system," *AFIPS Conference*, vol. 26, no. 2, pp. 97–141, 1964.

[4] W. H. Kohler, "A survey of techniques for synchronization and recovery in decentralized computer systems," *Computing Surveys*, vol. 13, no. 2, pp. 159–183, 1981.

[5] M. Ramachandran and M. Singhal, "Distributed semaphores," citeseer.nj.nec.com/315992.html.

[6] M. R. Eskicioglu, "A comprehensive bibliography of distributed shared memory," *Operating Systems Review*, vol. 30, no. 1, pp. 71–96, 1996.

[7] "Postgresql home page," http://www.postgresql.org/.

[8] "Opportunistic locks," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio%/base/opportunistic_locks.asp.

[9] M. Karlsson and P. Stenstrom, "Lock prefetching in distributed virtual shared memory systems — initial results," *Newsletter of the IEEE CS Technical Committee on Computer Architecture*, pp. 41–48, 1997. [Online]. Available: citeseer.ist.psu.edu/karlsson97lock.html

[10] S. P. Vanderweil and D. J. Lija, "Data prefetch mechanisms," *Computing Surveys*, vol. 32, no. 2, pp. 174–199, 2000.

[11] C. B. Seidel, R. Bianchini, and C. L. Amorim, "Exploiting lock-related primitives in distributed shared-memory systems," Technical report ES-517/99, COPPE/UFRJ, citeseer.ist.psu.edu/408110.html, 1999.

[12] W. R. Stevens, *Advanced Programming in the Unix Environment*. Addison-Wesley, 1997.

[13] T. Schöbel-Theuer, "Verfahren zur regulierung des datenzugriffs bei einem aus mehreren einzelsystemen bestehenden system auf wenigstens eine datenspeichereinrichtung," patent application PCT / EP03 / 10794.

[14] ——, "Speculative prefetching of optional locks in distributed systems," in *PDCN 2004, Innsbruck*. IASTED conference proceedings, 2004.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM*, vol. 21, no. 7, pp. 558–565, 1978.

[16] C. J. Date, *An Introduction to Database Systems*. Addison Wesley, 1995.

[17] "Osdl database test suite," http://www.osdlab.org/projects/performance/.

A Appendix

Figure 6. Compatibility matrix used by PostgreSQL

	1	2	3	4	5	6	7	8
1	+	+	+	+	+	+	+	-
2	+	+	+	+	+	+	-	-
3	+	+	+	+	-	-	-	-
4	+	+	+	-	-	-	-	-
5	+	+	-	-	+	-	-	-
6	+	+	-	-	-	-	-	-
7	+	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-

Figure 7. Lock weakness relation of PostgreSQL

	1	2	3	4	5	6	7	8
1	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$
2		$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$
3			$\leq$	$\leq$		$\leq$	$\leq$	$\leq$
4				$\leq$		$\leq$	$\leq$	$\leq$
5					$\leq$	$\leq$	$\leq$	$\leq$
6						$\leq$	$\leq$	$\leq$
7							$\leq$	$\leq$
8								$\leq$

Figure 8. dbt1, generalized locks

scale		1 sites	2 sites	4 sites	8 sites	16 sites	32 sites	speedup
100000	a	99.99	99.37	99.11	98.95	98.84	98.72	78 / 5.41
	b	99.99	99.38	99.14	98.96	98.85	98.73	78 / 5.45
	c	99.99	99.39	99.15	98.98	98.85	98.70	76 / 5.32
	d	99.99	99.42	99.17	98.98	98.86	98.71	77 / 5.36
	e	98.83	98.47	97.97	97.18	95.77	93.08	14 / 1.00
10000	a	99.99	99.37	99.08	98.95	98.79	98.70	76 / 5.28
	b	99.99	99.41	99.13	98.98	98.81	98.71	77 / 5.32
	c	99.99	99.43	99.15	99.00	98.80	98.67	75 / 5.16
	d	99.99	99.45	99.20	99.04	98.85	98.72	78 / 5.36
	e	98.73	98.39	97.94	97.17	95.71	93.14	14 / 1.00
1000	a	99.99	99.26	98.99	98.85	98.76	98.64	73 / 4.85
	b	99.99	99.29	99.02	98.85	98.75	98.62	72 / 4.78
	c	99.99	99.30	99.05	98.89	98.79	98.66	74 / 4.93
	d	99.99	99.33	99.07	98.93	98.83	98.66	74 / 4.93
	e	98.79	98.40	97.97	97.20	95.84	93.40	15 / 1.00

Figure 9. dbt1, only exclusive locks

scale		1 sites	2 sites	4 sites	8 sites	16 sites	32 sites	speedup
100000	a	100.00	93.78	91.30	90.15	89.81	89.56	9 / 2.17
	b	100.00	93.64	91.12	89.96	89.62	89.37	9 / 2.13
	c	100.00	92.77	89.78	88.44	88.03	87.72	8 / 1.84
	d	100.00	87.19	83.41	86.67	86.42	86.11	7 / 1.63
	e	98.87	86.07	80.78	78.67	78.00	77.38	4 / 1.00
10000	a	100.00	94.19	91.47	90.65	89.92	89.68	9 / 2.26
	b	100.00	94.06	91.36	90.53	89.79	89.55	9 / 2.23
	c	100.00	93.01	89.78	88.77	87.85	87.57	8 / 1.88
	d	100.00	87.73	83.51	86.93	86.29	86.08	7 / 1.68
	e	98.77	86.49	80.82	78.93	77.23	76.65	4 / 1.00
1000	a	100.00	94.12	92.06	91.50	91.22	90.98	11 / 2.44
	b	100.00	93.93	91.87	91.31	91.02	90.78	10 / 2.39
	c	100.00	92.98	90.44	89.72	89.34	88.98	9 / 2.00
	d	100.00	87.36	83.56	87.98	87.95	87.61	8 / 1.78
	e	98.83	86.30	81.27	79.37	78.64	77.98	4 / 1.00

Figure 10. dbt2, generalized locks

scale		1 sites	2 sites	4 sites	8 sites	16 sites	32 sites	speedup
1	a	100.00	99.09	98.54	98.00	97.39	96.81	31 / 0.93
	b	100.00	99.23	98.75	98.16	97.57	96.98	33 / 0.98
	c	100.00	99.14	98.59	98.04	97.42	96.83	31 / 0.93
	d	100.00	99.24	98.77	98.29	97.73	97.17	35 / 1.05
	e	99.41	98.94	98.60	98.17	97.62	97.04	33 / 1.00
2	a	100.00	99.10	98.55	98.00	97.40	96.82	31 / 0.92
	b	100.00	99.24	98.77	98.25	97.58	97.01	33 / 0.98
	c	100.00	99.15	98.60	98.04	97.44	96.85	31 / 0.93
	d	100.00	99.24	98.76	98.28	97.75	97.23	36 / 1.06
	e	99.41	98.94	98.60	98.17	97.64	97.07	34 / 1.00
4	a	100.00	99.10	98.54	98.00	97.41	96.85	31 / 0.92
	b	100.00	99.26	98.70	98.17	97.61	97.08	34 / 1.00
	c	100.00	99.15	98.60	98.05	97.45	96.88	32 / 0.93
	d	100.00	99.25	98.76	98.29	97.76	97.24	36 / 1.05
	e	99.40	98.94	98.59	98.17	97.65	97.09	34 / 1.00
8	a	100.00	99.05	98.47	97.86	97.19	96.61	29 / 0.92
	b	100.00	99.18	98.68	98.09	97.36	96.80	31 / 0.98
	c	100.00	99.11	98.55	97.94	97.27	96.68	30 / 0.94
	d	100.00	99.19	98.68	98.13	97.53	96.99	33 / 1.04
	e	99.42	98.91	98.52	98.03	97.43	96.87	31 / 1.00

Figure 11. dbt2, only exclusive locks

scale		1 sites	2 sites	4 sites	8 sites	16 sites	32 sites	speedup
1	a	100.00	95.83	93.87	92.88	92.42	92.20	12 / 1.47
	b	100.00	95.82	93.86	92.87	92.41	92.19	12 / 1.47
	c	100.00	95.55	93.45	92.39	91.90	91.66	11 / 1.37
	d	100.00	94.00	91.33	91.93	91.68	91.47	11 / 1.34
	e	99.41	93.64	90.90	89.50	88.85	88.54	8 / 1.00
2	a	100.00	95.87	93.74	92.75	92.29	92.01	12 / 1.46
	b	100.00	95.86	93.73	92.74	92.28	92.00	12 / 1.45
	c	100.00	95.58	93.28	92.22	91.73	91.42	11 / 1.36
	d	100.00	94.06	91.17	91.84	91.49	91.21	11 / 1.32
	e	99.41	93.69	90.76	89.39	88.74	88.37	8 / 1.00
4	a	100.00	95.82	93.83	92.80	92.30	92.06	12 / 1.46
	b	100.00	95.81	93.82	92.79	92.30	92.06	12 / 1.46
	c	100.00	95.51	93.36	92.24	91.70	91.44	11 / 1.36
	d	100.00	94.02	91.23	91.75	91.45	91.22	11 / 1.32
	e	99.40	93.65	90.84	89.39	88.71	88.38	8 / 1.00
8	a	100.00	95.43	93.14	92.01	91.44	91.18	11 / 1.37
	b	100.00	95.42	93.14	92.01	91.43	91.18	11 / 1.37
	c	100.00	95.04	92.55	91.31	90.68	90.40	10 / 1.26
	d	100.00	93.82	90.87	90.86	90.36	90.11	10 / 1.22
	e	99.42	93.48	90.47	88.98	88.23	87.89	8 / 1.00

Figure 12. dbt3, generalized locks

scale		1 sites	2 sites	4 sites	8 sites	16 sites	32 sites	speedup
0.025	a	100.00	99.05	98.71	98.51	98.38	98.28	58 / 1.98
	b	100.00	99.08	98.76	98.62	98.55	98.50	66 / 2.27
	c	100.00	99.09	98.74	98.55	98.41	98.31	59 / 2.02
	d	100.00	99.03	98.77	98.63	98.51	98.44	64 / 2.19
	e	97.95	97.77	97.67	97.50	97.18	96.59	29 / 1.00
0.05	a	100.00	99.15	98.86	98.67	98.57	98.50	66 / 2.05
	b	100.00	99.20	98.94	98.81	98.74	98.70	76 / 2.37
	c	100.00	99.19	98.90	98.72	98.61	98.55	68 / 2.12
	d	100.00	99.13	98.96	98.82	98.75	98.65	74 / 2.28
	e	98.17	98.01	97.91	97.73	97.43	96.92	32 / 1.00
0.1	a	100.00	99.26	99.02	98.89	98.81	98.76	80 / 2.03
	b	100.00	99.35	99.13	99.03	98.98	98.76	80 / 2.03
	c	100.00	99.29	99.04	98.91	98.83	98.78	81 / 2.07
	d	100.00	99.31	99.13	99.05	98.98	98.77	81 / 2.05
	e	98.47	98.35	98.27	98.13	97.90	97.48	39 / 1.00
0.2	a	100.00	99.43	99.28	99.19	99.14	99.10	111 / 2.34
	b	100.00	99.46	99.33	99.25	99.21	99.17	120 / 2.54
	c	100.00	99.45	99.30	99.20	99.15	99.10	111 / 2.34
	d	100.00	99.41	99.33	99.26	99.21	99.16	119 / 2.51
	e	98.74	98.63	98.56	98.44	98.24	97.89	47 / 1.00
0.4	a	100.00	99.65	99.54	99.48	99.45	99.42	172 / 2.33
	b	100.00	99.66	99.58	99.53	99.49	99.46	185 / 2.50
	c	100.00	99.65	99.55	99.49	99.45	99.42	172 / 2.33
	d	100.00	99.63	99.58	99.53	99.50	99.45	181 / 2.45
	e	99.17	99.11	99.06	99.00	98.87	98.65	74 / 1.00

Figure 13. dbt3, only exclusive locks

scale		1 sites	2 sites	4 sites	8 sites	16 sites	32 sites	speedup
0.025	a	100.00	95.00	92.71	91.69	91.23	91.04	11 / 2.18
	b	100.00	94.95	92.65	91.62	91.16	90.96	11 / 2.16
	c	100.00	94.50	91.99	90.87	90.36	90.14	10 / 1.98
	d	100.00	87.44	84.40	89.76	89.96	89.78	9 / 1.91
	e	97.95	86.16	82.80	81.34	80.75	80.49	5 / 1.00
0.05	a	100.00	95.81	94.19	93.33	92.98	92.83	13 / 2.43
	b	100.00	95.71	94.07	93.20	92.85	92.69	13 / 2.38
	c	100.00	95.23	93.42	92.43	92.05	91.88	12 / 2.14
	d	100.00	88.48	86.17	91.55	91.47	91.34	11 / 2.01
	e	98.17	87.38	84.58	83.29	82.83	82.60	5 / 1.00
0.1	a	100.00	95.79	94.10	93.39	93.06	92.91	14 / 2.13
	b	100.00	95.67	93.95	93.23	92.90	92.76	13 / 2.09
	c	100.00	95.16	93.31	92.54	92.17	92.01	12 / 1.89
	d	100.00	90.35	87.78	91.29	91.42	91.33	11 / 1.74
	e	98.47	89.42	86.58	85.56	85.08	84.88	6 / 1.00
0.2	a	100.00	97.55	96.56	96.12	95.92	95.83	23 / 2.89
	b	100.00	97.48	96.48	96.03	95.83	95.74	23 / 2.83
	c	100.00	97.05	95.95	95.45	95.23	95.13	20 / 2.47
	d	100.00	91.92	90.18	94.37	94.72	94.69	18 / 2.27
	e	98.74	91.13	89.17	88.42	88.11	87.95	8 / 1.00
0.4	a	100.00	98.16	97.49	97.03	96.87	96.77	30 / 2.72
	b	100.00	98.11	97.44	96.97	96.81	96.71	30 / 2.67
	c	100.00	97.83	97.07	96.56	96.38	96.27	26 / 2.36
	d	100.00	94.37	93.06	95.89	95.98	95.93	24 / 2.16
	e	99.18	93.82	92.38	91.69	91.38	91.21	11 / 1.00