Speculative Prefetching of Optional Locks in Distributed Systems

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

ABSTRACT
We present a family of methods for speeding up distributed locks by exploiting the uneven distribution of both temporal and spatial locality of access behaviour of many applications. In the worst case, some of our methods will not produce higher network latencies than equivalent conventional distributed locking methods. In best case, the total number of messages can be constantly bounded, approximating the impression that no network latencies exist at all.

Measurements and simulations based on variants of TPC database benchmarks show that hit rates enabled by speculative prefetching of optional locks appear to be similar to hit rates of conventional data caches, typically in the range from 90% to 99%. Thus overall speedup factors of 10 or more for the average latencies of distributed locks are possible. Compared to purely temporal prefetching, adding exploitation of spatial locality may significantly improve performance, typically by factors of 2 or more.

We discuss some implications for the construction of distributed systems. For the class of programs well-suited for distributed systems, network latencies will nearly vanish, blurring performance differences between local and distributed systems. For program classes exposing poor locality, there is likely no help independent from distributed computing paradigms. We explain how the communication paradigm can be efficiently implemented on top of distributed shared memory (DSM) using region locks. Thus we believe that the DSM paradigm will become more attractive than explicit communication (e.g. RPC, CORBA) for the construction of distributed applications.

KEY WORDS
Resource Allocation, Distributed Locking, Distributed Algorithms, Distributed Databases, Operating Systems

1 Introduction

When multiple processes operate concurrently on shared data, some synchronization discipline is needed [1, 2, 3]. Most of the past research has focused on the synchronization problem on a single resource, treating mechanisms on large sets of fine-grained resources as straightforward extension to n resources where each resource is regarded as independent from each other (although application level dependencies have been observed [4, 5]). There have also been generalizations from exclusive resources to resource classes where all resources in a class C_i are interchangeable [6], but the principle of treating two resource classes C_i and C_j with $i \neq j$ as independent from each other is also obeyed at the class level.

There are some exceptions, notably hierarchical or intentional locking [7] and opportunistic locking [8]. The latter may be viewed as a variant of hierarchical locking with an additional lock retraction mechanism. Hierarchical locking may be characterized as introducing additional lock objects which each *replace* or *subsume* a set of other lock objects. 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.

In contrast, the method proposed in this paper (prior publication in [9]) does not assume any order which must be explicitly obeyed by the lock partners (although generalizations of our method to hierarchical or intentional locks are possible), but rather assumes that application behaviour bears some *implicit* order, also called *spatial locality of access*, which may be approximately *determined* by a (distributed) lock manager and exploited for better performance in distributed systems. We use an order on lock objects which an application does not necessarily need to be aware of. We may take a *natural order*, such as the memory addresses of lock representation objects. Alternatively, address orders may be explicitly used by applications, for example `flock()` operations or `fcntl()` locking on Unix files [10] where start addresses and length of *lock regions* for possibly *overlapping* locks are specified by the caller. Applications are free to give that ”addressing order” *any* internal semantics they like, or they may just ignore that order for non-overlapping locks. Even when programmers are non-conscious of spatial properties of their locks, programs often exhibit spatial locality.

The intuition behind our approach is that memory objects and their locks are often organized in spatial ”clusters”, ”compact regions”, ”segments” or similar aggregations in an address space. Spatial locality has been exploited by data prefetching strategies in data caches [11]. Temporal prefetching of locks has appeared in the literature, e.g. [12] augmenting applications with specialized lock prefetch instructions similar to hardware data prefetch instructions [11], and [13] automatically predicting future lock operations from past temporal behaviour. In contrast, our method exploits both spatial and temporal locality.

We give an example how the *working set* theory of Denning [14, 15] which models temporal behaviour could be extended to a notion of *working region* in order to additionally model spatial behaviour. However, there seem to exist numerous ways for modeling spatial locality and neighbourhood relations. A full work-out and comparison of many possible lock prefetching policies derived from each of those many possible models will require much work beyond the limits of this paper. Thus we concentrate on an informal presentation of the mechanism and some examples for prefetch policies, and we provide practical evidence that the method yields high speedups at least with some kinds of applications.

2 The Method

2.1 Informal Explanation

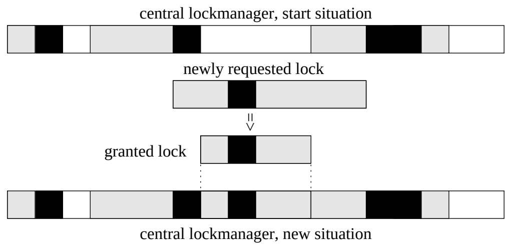
We assume the existence of one or more *requestor instances* which may *request* locks, and one or more *grantor instances* which may *grant* any requested locks. For simplicity, we describe only the case of one grantor instance.

The central idea of this paper is to discriminate between two kinds of lock requests: *obligatory* lock requests, and *optional* lock requests[1]. Obligatory lock requests must be granted to an application sometime, otherwise the application cannot continue with its work (deadlock). In contrast, optional locks need not be granted, or need only be granted *partially*. The grantor of an optional lock will tell the requestor whether and how far an optional lock has been granted to him. In a nutshell, an optional lock may be *partially denied.*

Conventional applications will normally issue only obligatory lock requests; it is the task of *local lock managers* in a distributed system to automatically add appropriate optional lock requests for overall minimization of network traffic. Whenever an optional lock has already been granted to a local site in the network, it may be (partially) converted to one or many obligatory locks at any time without causing any network traffic. Local lock managers should therefore try to *speculate* as well as possible to obtain the ”right” optional locks in advance. A ”good” speculation is characterized by minimizing network traffic, i.e. obtaining those optional locks which are most likely required in the future, but not obtaining non-required optional locks when other sites may require them, because a purely optional lock must be transferred back to the other site on demand (*retraction* of optional locks) in order to retain deadlock-freedom for the obligatory locks of the other site.

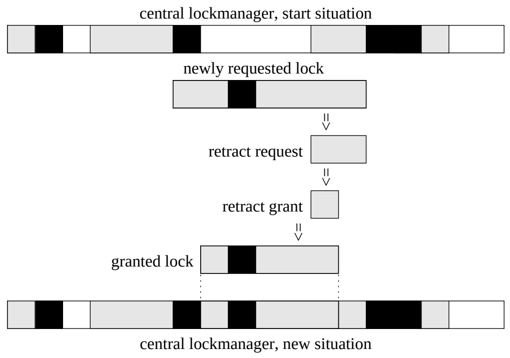
Locks are issued on (possibly overlapping) *regions* of an address space similar to Unix `flock()` or `fcntl()` locking; unlocking may be done in other granularity than locking. Simple non-overlapping locks are regarded as a trivial special case (e.g. by constantly using a region length of 1). A region may be viewed as a contiguous subset of the set of addresses occurring in the address space, thus a region may be used to replace a set of contiguous conventional single-sized locks. In the following example picture, obligatory lock regions are depicted as black areas, optional lock regions as grey areas, and free areas are left white:

[1] These are not to be confused with classification of lock types into *mandatory* versus *advisory* locking [10], which is orthogonal to our lock kinds (by classification of applications whether they are *explicitly* issuing obligatory lock requests or issuing them *implicitly*).


central lockmanager, start situation
newly requested lock
granted lock
central lockmanager, new situation

The upper long stripe shows the initial situation at the central lock manager, where some regions have been optionally locked, and some subsets of them have been obligatorily locked. Then a new lock request arrives, consisting of an obligatory lock surrounded by a larger optional lock region. Since optional lock regions can only be exclusively granted to a single requestor, the optional part must be shortened in order to fulfill the request at once. The granted optional lock[2] is thus a smaller subset of the requested one.

Now we look at a variant: 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 some optional part from that third party:


central lockmanager, start situation
newly requested lock
retract request
retract grant
granted lock
central lockmanager, new situation

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 the optional part. That current holder responds with a *retract grant*[3], which may be a subset of the retract request. There may be cases where the retract request cannot be granted fully or partially, e.g. when another conflicting obligatory lock has been granted locally in the meantime which the global lock manager is not (yet[4]) aware of; in extreme case, an empty region may be granted back. In the example, the local lock manager policy has decided to bisect the requested retract area and

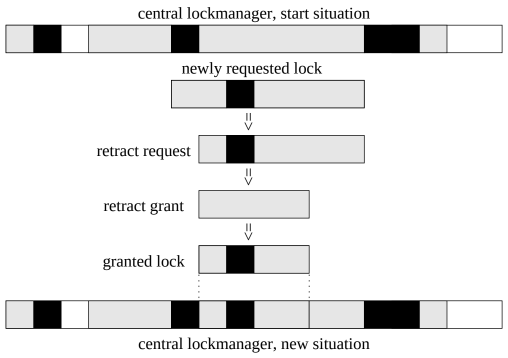
[2] Note that in general optional locks may be granted by the central lock manager asynchronously at any time, even without request. Similarly, a local lock manager may issue optional lock requests asynchronously at any time even if no local process demands an obligatory lock.

[3] In general, both retract requests and retract grants may be sent asynchronously at any time without requests, e.g. for spontaneous reorganization of optional areas.

[4] Local grants of obligatory requests may be forwarded to the central lock manager asynchronously (preferably having such messages coalesced). Although the central lock manager will get only a delayed image of local scenarios, this can help in making better policy decisions.

keeping the one half for itself, speculating that it will 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.

Now we look at at third example where the retraction mechanism is necessary for avoiding deadlock. Here the initial situation is different from the first two examples:


central lockmanager, start situation
newly requested lock
retract request
retract grant
granted lock
central lockmanager, new situation

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[5] would ever "force" the holder of that optional region to return it. Thus the retract request contains the obligatory part, which indicates to the current owner of the optional part that he *must* give back at least that part in order to avoid 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 that 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.

2.2 Preliminary Analysis

Clearly, the benefit of speculative prefetching of optional locks will depend on the quality of the speculation, i.e. how "well" it anticipates the future lock demands at a site. In this subsection, we roughly investigate only the bisecting strategy as explained in the examples from section 2.1. Other heuristics and strategies are subject to future research; there are good chances for finding better strategies at least in the following cases:

1) when knowledge on future demands of applications is explicitly available; e.g. sequential scans of data or traversal strategies on B-trees in databases; cf [17].

2) when some known access patterns (e.g. sequential scan, stack-like behaviour) can be *inferred* by *observation* of implicit application behaviour with good reliability.

2.2.1 From Working Sets to Working Regions

First, the concept of working set as defined by Denning [14] is slightly generalized in the following way: let $W_i(t, D)$ be the set of obligatory locks held at network site $i \in I$ during the time[6] interval $[t - D, t]$. We call D the *duration* of the measurement of the working set. In difference to Denning [14] working sets are related to network nodes and measured on acquired locks instead of memory pages and are also measured on *real time* instead of process time, besides minor notational differences. The reason for using real time is manifold: while Denning can safely assume independence of program runs from each other, in general cooperating processes synchronizing with each other by means of distributed locks are dependent from real time due to network load, scheduling at local sites, and many other factors. While Denning can assume that a program as a whole is suspended during the I/O service time for a page fault, network sites running multiple processes locally are not suspended as a whole while waiting for some lock. We suspect that the use of real time will likely complicate models of network site behaviour and thus something like a "local process time" would be preferable, but currently we see no way for achieving that in a simple and intuitive way.

We denote $W_i(t, 0)$ as a shorthand for $\lim_{D\to 0} W_i(t, D)$ informally yielding the current set of locks at a point in time, similarly $W_i(t, \infty)$ for $\lim_{D\to\infty} W_i(t, D)$ as a shorthand for all locks obtained during a long-term run. The *foreign working set* at site i is the union of the working sets of all other sites, denoted $\overline{W}_i(t, D) := \cup_{j\in I\setminus i} W_j(t, D)$.

Definitions for working regions face a fundamental problem: even if we knew the future access behaviour of a site in advance, there are many artificial ways for defining regions and their borders such that they form compact supersets of sparsely locked areas in a large address space. The problem is similar to inferring general rules from samples. Working regions could e.g. be defined as follows:

1. we could measure the *density* of locks in all memory intervals $[a, b]$ and select those intervals exceeding some given threshold.

2. local versus global models: the access behaviour of foreign sites leading to "disruptions" of working regions may be optionally taken into account, since nonlinear dependencies due to synchronization will exist anyway.

3. instead of defining working regions with binary borders, we could use *probablility density functions* on both spatial and temporal dimensions which either model past access behaviour in both temporal and spatial dimensions and/or predict future behaviour. Their characteristics could be compared with those from foreign sites to determine borders for optional locks dynamically.

4. probability density functions could be defined in

[5]Optional regions may also be granted for a limited time similar to leases [16], such that after timeout a retract occurs automatically and implicitly, possibly leading to aborts of corresponding obligatory locks.

[6]We do not need a totally ordered global time, but only synchronized local clocks obeying causality, e.g. physical Lamport clocks [18]. The parameter t is always measured in the corresponding local time at site i.

various ways, e.g. simply based on reciprocal distance of addresses to locks, or on reciprocal squares of distances etc.. They could be weighted by the size of locks, or by the number of lock occurrences during some time interval D, or by the average time distance between occurrences, or by the acquisition costs (deviations in network latencies), and in various other ways.

This list is surely not exhaustive, and we could probably build even more models for program behaviour on top of each of these definition variants. From each of those models, we could probably derive a variety of global and local lock manager policies. However, there is a danger of choosing inappropriate models as Denning has described in [19]. For example, an analogy to temporal phase transitions of program behaviour [20] may also exist in the spatial density behaviour of programs, or even a non-orthogonal corrolation between temporal and spatial behaviour of phase transitions. Recently there were advances in the field of modeling of both temporal and spatial locality of data access patterns [21] without assuming phase transitions, but there is neither a consensus on a commonly acceptable model nor is it clear how well results from that area may apply to our problem of lock locality.

Thus we don't discuss models of program behaviour in this paper. Only for the sake of providing some explanation why the bisecting strategy works in practice, we present a specific definition of working regions as an example. It is a simplistic global model.

The *set of working regions* $R_i(t, D, t', D')$ at site i is defined as $R_i(t, D, t', D') := \{[s, e] \mid s, e \in W_i(t, D)$ and no $x \in \overline{W}_i(t', D')$ exists with $x \in [s, e]\}$. Intuitively, a working region is a contiguous area where obligatory locks have been acquired during time interval D but have not been disrupted by foreign locks at (another) time t' and during interval D'. In case of $t = t'$, we write $R_i(t, D, D')$ in place of $R_i(t, D, t', D')$. When additionally $D = D'$, we simply write $R_i(t, D)$.

The *reduced set of working regions* $RR_i(t, D, t', D')$ is defined as $RR_i(t, D, t', D') := R_i(t, D, t', D') \setminus \{[s, e] \mid [s, e] \subsetneq [s', e'] \in R_i(t, D, t', D')\}$. Intuitively, the reduced set of working regions contains exactly the longest possible regions which are not disrupted by foreign locks. The short notations $RR_i(t, D, D')$ and $RR_i(t, D)$ are used analogously to above. In the sequel, we will deal exclusively with reduced sets of working regions.

2.2.2 Best Case

In best case the working sets at different sites are *always* disjoint. This means, all $W_i(t, \infty)$ are pairwise disjoint for any i at any point t in time, or equivalently $W_i(t, D) \cap \overline{W}_i(t, D) = \emptyset$ for all sites i and $\lim_{D \to \infty}$. Note that due to exclusiveness of obligatory locks, this condition would always hold for $\lim_{D \to 0}$, but may *in general* be violated for "sufficiently large" durations D. The best case means informally that all sites are doing long-term work which is completely unrelated to each other.

It is easy to see that in best case all working regions $RR_i(t, \infty)$ are also disjoint with each other. Even when no two consecutive locks belong to the same site, any working region will cover only one obligatory lock. However, we expect that in many practical applications the number of different working regions is often rather low.

(subcase 1) We assume that $|RR_i(t, \infty)|$ for any $i \in I$ is bounded by some constant k which is independent from $|I|$. We expect this to be true for a large class of practical applications. Then the total number of working regions in the system is bounded by $k \cdot |I|$. The system will become *stable*, i.e. it will cease sending messages if all optional regions in the systems have become supersets of their respective working region and are not intersecting with any foreign working region. Any initially wrong speculation at a single border between two working regions (either obtaining a too large or too small optional region with respect to the "correct" working size) will be corrected by the bisecting strategy in at most $\log |s|$ retract cycles where $|s|$ is the length of the total address space $s \in RR(\infty, \infty, 0, 0)$ where $RR(\infty, \infty, 0, 0)$ contains exactly the total working region occupied by all sites $i \in I$. Thus the total message effort for the system is bounded by $O(k \cdot |I| \cdot \log |s|)$, which is a constant not depending on the time t.

(subcase 2) When assuming that $|RR_i(t, \infty)| = O(|s|)$ at any time t, we analogously get a total effort $O(\log^2 |s|)$, which is also a constant independent from time.

Further improvements of these raw bounds are possible, but not subject to this paper.

2.2.3 Worst Case

We assume that transferring two region informations for an obligatory and another optional lock in a single request or grant message will not increase network bandwidth and overhead in a significant way[7]. We compare relative performance with respect to the increase in number of messages and in network latency when adding optional locking to a known distributed locking method.

In worst case, no optional lock region will ever be valuable for executing obligatory locks locally, i.e. there will never be "hits" to speculatively prefetched regions. Such a behaviour could e.g. be provoked by "intelligent clients" which always know not only the local optional lock regions, but also know which optional regions have been granted to another client, and are thus constantly requesting locks outside of locally reserved regions and always in foreign regions. Of course, such an application behaviour is not very realistic.

For a centralized lock manager as described in section 2.1, the message overhead will increase in worst case from 2 to 4 messages, since a retract request and a retract grant are added to the protocol. This factor of 2 in message

[7]Usually the overhead for packet handling is independent from packet size and often an order of magnitude larger than the overhead for creating and transferring slightly increased network packets.

overhead may be reduced to 1.5 if the retract grant is not sent to the server, but directly to the originally requesting client such that retract grant and lock grant are fused together. The network latencies will also increase by a factor of 1.5 in worst case when compared to conventional central lock managers (assuming equal distribution of network latencies).

A further reduction of latencies is possible, but at the expense of additional messages; such protocol modifications may be advantageous because current trends in networking show up exponential growth of bandwidth, but latencies remain constantly bounded due to the physical speed of light. If atomic broadcast is available where each broadcast message is counted as a single message, even that overhead may be reduced to a factor near 1. The method works as follows:

Each site $i \in I$ is always keeping track of the reservation state at all other sites $I \setminus \{i\}$. This may be accomplished by always sending all grant and unlock messages to all participants, so each of them can keep the state of each other. Lock requests for regions currently held by other clients are directed to and interpreted by the corresponding client directly. This means in essence that each client becomes the lock manager for those optional locks it is currently holding. The task of the central lock manager is reduced to managing only empty regions, which may be characterized as optional locks currently belonging to ”nobody” or to the central manager. In the last view, the role of the central manager becomes symmetric, such that he may be eliminated if he never requests locks for itself and if the whole address space is preallocated to someone else upon initialization of the distributed system.

Other distributed locking methods, such as voting methods [22], may also be enhanced by speculative prefetching of optional locks, increasing overhead in a similar way. Details are outside the scope of this paper.

2.2.4 ”Average” Cases

For each of the many possible models of program behaviour, another average case would hold. Since these models are not yet stated and validated, we pose some working hypotheses of how an average case could look like and give some qualitative arguments on their performance.

As a working hypothesis, we assume that phases of slow change of working regions occur as well as transition phases where some working regions are interchanged [19]. We also assume that working regions at different sites will intersect only with low probability.

At the phase change events, optional regions acquired at a site will rather soon approximate the shapes of working regions due to the bisecting strategy, analogously to the argumentation in section 2.2.2. This forms a difference to conventional data caching where phase transitions usually result in bad performance due to heavy data reloading. At the slow change hypothesis, optional regions will only occasionally mismatch the shapes of working regions such that in average good hit rates for obligatory locks will result. The main problem of our method are intersections of working regions at different sites. The larger the percentage of intersections and the more activity occurs in those intersections, the worse the preformance in general. We call it the ”hot spot problem”.

Note that the hot spot problem is caused by the *behaviour* of processes operating at different sites, e.g. when incidently operating on common data or when communicating with each other via distributed shared memory (DSM). When processes are behaving this way, they use DSM for *transfer of information* to each other. When transfer of information is *required* for the problem solution, nothing can help. Any distributed paradigm will have to send packets over the network to satisfy these communication demands.

The effect of hot spots can be described by the notion of *thrashing* [15]. Thrashing has been observed in local caches long time ago, but may be transferred to distributed systems to describe the communication behaviour in presence of hot spots. The notion of thrashing is from practical observation of real OS behaviour when serious *degradation* of system performance occurs (usually an order of magnitude or more). In standalone systems, performance is *limited* by an I/O *bottleneck*. Transferred to distributed systems, thrashing means that communication is *limited* due to *network latencies*.

In contrast to traditional theory on thrashing, our lock regions are potentially unlimited virtual resources spanning almost arbitrarily large regions of (virtual) memory spaces. Thus there is no problem analogously to cache overrun. While retractions in standalone data caches are traditionally caused by the cache replacement strategy, retractions in distributed systems are caused by other sites in the network[8].

A typical ”average” case may consist of a mixture of the following phenomena:

(1) some parts of the working sets form totally independent local working regions.

(2) some local working regions are showing rather slow changes in their shape or size, but are mostly indepedent from other sites. Slow means that miss rates for local optional regions are rather low in comparison to hit rates. Mostly means that intersections of working regions at different sites play no significant role for performance.

(3) some local working regions are intersecting with those of other sites to a significant part, such that miss rates become a significant factor for slowdown.

(4) some local working regions are moving so fast that miss rates also become a significant performance problem. Typical examples are ”random” access behaviour, e.g. caused by hashing methods, or other ”irregular” or ”unpredictable” behaviour of access patterns.

When behaviours (1) and (2) are dominant, a distributed system based on transparent speculative prefetch-

[8]When adding data transmission and data caching to our lock protocols, we probably would also have to introduce replacement strategies.

ing of optional locks will perform nearly like a local system. Note that this need not be true in conventional distributed systems (e.g. based on the message passing paradigm), because the spatial locality of access regions is usually not exploited, at least not for locks. When behaviours (3) and (4) are dominant, the distributed application will show bad performance both when speculative optional locking is or is not available.

Thus we conclude that speculative prefetching of locks is one of the best methods available for speeding up distributed locking. With behaviour (1) and (2), the bisecting strategy is already close to a theoretical optimum, and further improvements by better lock manager policies are likely to occur. For behaviour (3) and (4), other strategies may be better. Achieving optimality would likely require to know *in advance* the future access behaviour of such an application; often such information is not available.

3 Experiments

We measured locking patterns of variants of TPC benchmarks and simulated what would have happened if those accesses would have been distributed to n sites in a network using various strategies for speculative prefetching of optional locks.

The experiments were carried out on a dual processor AMD Athlon MP 1900+ workstation with 2 gigabyte of RAM, running Ret Hat Linux 9 with kernel 2.4.20-20.9custom and PostgreSQL version 7.3.3. We chose that system because of easy access to source code. We instrumented the lock manager of PostgreSQL (file `src/backend/storage/lmgr/lock.c`) with `printf()` statements in order to write a log of all granted and released locks to a file. Each entry in the log file contained the type of operation (lock / unlock), a timestamp, the lock type (read / write and many more used internally by PostgreSQL), the process ID of the PostgreSQL server process executing that lock, the internal transaction ID, the internal database ID $dbID$, the internal relation ID $relID$, and the internal object ID $objID$ of the lock. The latter three values identify a lock uniquely when taken together.

The log file was later analyzed by a small program written in Perl which simulated prefetching of optional locks with various strategies as if the measured locking patterns had been executed on n sites, with n being a choosable parameter. Only locks stemming from different PostgreSQL server processes were spread to the virtual network, i.e. we simulated a scenario where the PostgreSQL server processes would have been distributed nearly unmodified. The internal IDs identifying a lock uniquely were mapped to a hypothetical single address space by the formula $addr = objID + (max\_objID - min\_objID + 1) * (relID + (max\_relID - min\_relID + 1) * dbID)$. All PostgreSQL locks were treated as obligatory locks with a length of 1. As a result, we determined the number of hits which would have been caused by prefetching of optional locks.

The first benchmark was DBT-3 version 1.0 issued by the Open Source Development Lab OSDL [23] which should be similar to the well-known TPC-H. We treated all PostgreSQL lock types as equal, i.e. we treated read locks as write locks. This may possibly lead to a distortion of results, since PostgreSQL may grant read locks in parallel to other locks. However, the error is likely in favor of our method, because allowing parallelism between read locks would most probably lead to even better results due to less frequent retractions.

The DBT-3 benchmark has several parameters, in particular the scale factor and the number of concurrent processes operating on the database. We present results only for 8 processes (the results for 2 and 4 processes are similar; 16 processes and more led to extreme running durations). A scale factor of 1.0 yields a database size of approximately 1 gigabyte data, leading to a real database size of about 4 gigabytes including internal overhead. Since databases of larger size than RAM for the Linux buffer cache led to heavy thrashing, we measured only with scale factors of 0.2 (resulting in 106972 lock operations in total), 0.1 (75286 locks), 0.05 (58802 locks), and 0.025 (51938 locks). For each scale factor we ran the following strategies:

(a) upon local miss of an obligatory lock, a maximally sized optional lock was requested from the central manager. Retracts were issued causing maximally sized retract grants, such that the final granted optional lock was the full surrounding area not held by any other obligatory lock at that moment.

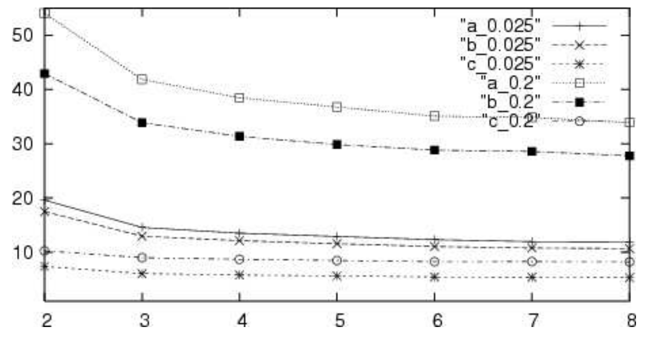
(b) in contrast to (a), the currently free surrounding area was always bisected at both sides such that the respective foreign site could retain the half of that area.

(c) to evaluate the impact of spatial locality, this variant requested and granted optional locks only with the same size and location as the new obligatory lock, such that spatial neighbourhood was completely ignored and only temporal bursts of the same lock could lead to hits.

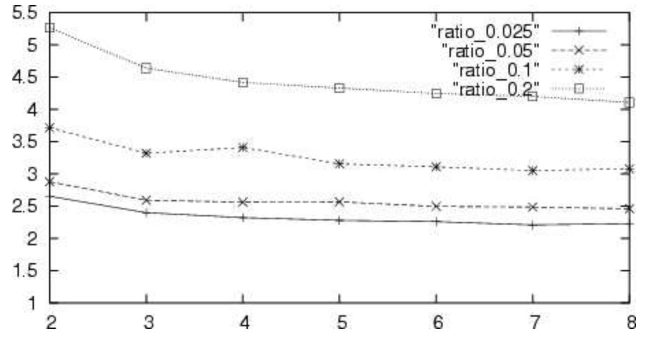
The following table contains the hit rates in percent for simulated spreading to $n \in \{1 \dots 8\}$ sites. Each box contains the results for strategies (a) through (c) from top to bottom:

scale		1 site	2 sites	3 sites	4 sites	5 sites	6 sites	7 sites	8 sites
	(a)	100.00	94.90	93.14	92.60	92.26	91.88	91.65	91.59
0.025	(b)	100.00	94.29	92.29	91.76	91.35	90.96	90.74	90.65
	(c)	97.83	86.47	83.55	82.82	82.36	81.65	81.55	81.24
	(a)	100.00	95.62	94.42	93.88	93.90	93.44	93.43	93.26
0.05	(b)	100.00	94.95	93.56	92.98	92.96	92.41	92.45	92.27
	(c)	98.04	87.39	85.53	84.32	84.34	83.63	83.67	83.42
	(a)	100.00	97.05	96.19	96.24	95.60	95.49	95.33	95.37
0.1	(b)	100.00	96.60	95.65	95.71	94.94	94.87	94.67	94.71
	(c)	98.34	89.04	87.35	87.18	86.11	85.98	85.75	85.74
	(a)	100.00	98.15	97.61	97.40	97.28	97.15	97.13	97.05
0.2	(b)	100.00	97.67	97.05	96.81	96.65	96.53	96.50	96.40
	(c)	98.65	90.26	88.91	88.51	88.22	87.90	87.94	87.88

The next table shows the maximum possible speedups of strategies (a) through (c) for the extreme scale factors 0.025 and 0.2 (other scale factors are omitted to keep the graphics readable), calculated by the formula $speedup := 100/missrate$ where $missrate := 100 - hitrate$. Our formula is motivated by the observation that processor speeds are many orders of magnitude greater than network latencies, so we drop processor speeds.



The following table shows the relative speedup of strategy (a) with respect to stategy (c), i.e. $speedup_a/speedup_c$:



As expected, the number of network sites has some negative influence on the hit rates, because wider distribution increases the probability that an optional region is currently held at a foreign site. The influence of the scale factor is stronger as one might expect. It seems to be a property of the DBT-3 benchmark; possibly a larger database decreases the probability for a constant number of processes that different working regions are intersecting during a fixed period in time. Note that DBT-3 executes an extremely high amount of heavy-weighted read-only transactions which can be handled very efficiently by the multiversion concurrency protocol of PostgreSQL, possibly introducing some distortions to our measurements. Also notice that the update processes started by version 1.0 of DBT-3 execute only `insert` commands but no `delete` commands as required by the original TPC-H; for some further differences between these benchmarks see [23].

Another experiment was carried out using DBT-2 [23] which should be similar to the well-knwn TPC-C benchmark. It models short online transaction processing (OLTP) from warehouse applications. Although the beta version 0.16 of DBT-2 contained some minor bugs occasionally leading to some SQL errors, we present the results here because they indicate that spatial prefetching of optional locks may drastically improve performance even in cases where purely temporal prefetching leads to worse performance. DBT-2 runs a high percentage of short read-write transactions on equally distributed random search keys. This is expected to produce worse performance in distributed systems. Here are the hit rates for 1, 2, and 4 warehouses (corresponding to scale factors) and 8 client processes (each working as a "terminal concentrator" for 10 simulated terminals) running for a measurement period of 1 hour:

scale	1 site	2 sites	3 sites	4 sites	5 sites	6 sites	7 sites	8 sites
(a)	100.00	90.62	88.06	86.09	85.61	85.18	84.47	83.91
1 (b)	100.00	90.34	87.70	85.67	85.19	84.76	84.01	83.45
(c)	87.49	68.89	63.25	59.52	58.55	57.58	56.09	54.99
(a)	100.00	91.08	88.65	87.06	86.73	86.11	85.61	85.15
2 (b)	100.00	90.98	88.58	87.00	86.70	86.02	85.53	85.06
(c)	93.79	79.61	75.40	72.86	72.38	71.14	70.33	69.53
(a)	100.00	95.18	93.75	92.95	92.81	92.38	92.13	91.82
4 (b)	100.00	94.91	93.38	92.57	92.42	91.97	91.71	91.39
(c)	93.50	82.15	78.66	76.52	75.97	75.18	74.47	73.76

The random access behaviour of DBT-2 seems to produce a high number of uncorrelated accesses. Thus purely temporal prefetching strategy (c) is worse even for distribution to only 1 site, because each distinct lock has to be fetched at least once. However, there seems to exist some intra-transactional spatial locality which would explain the significantly better behaviour of our strategies (a) and (b).

In summary, our measurements and simulations provide some evidence that our method can achieve hit rates similar to conventional data caches, and sigificantly better miss rates (i.e. difference to 100%) than purely temporal prefetching strategy (c), at least with some kinds of applications and application behaviours. For stronger evidence, measurements should be repeated by different people and on different applications exposing different working region behaviour, preferably with real-world loads instead of synthetic benchmarks. The variety in our measured miss rates is dependent on scale factor, number of processes, and distribution to network sites. Before modeling application behaviour, further possible influence factors should be experimentally determined. Our observations indicate that the working region behaviour of applications may have a strong influence on performance, which is predicted by our qualitative analysis and which has also been observed in the area of data caching.

4 Discussion of Implications

Our measurements and simulations indicate that speculative prefetching of optional locks, possibly combined with prefetching of the corresponding data to local data caches, can speed up at least some applications running on top of distributed shared memory (DSM) [24] when compared to running on local shared memory (LSM).

An obstacle to DSM has been its poor performance when implementing the message passing paradigm on top of it. Classical LSM solutions use semaphores, monitors or similar concepts for the implementation of communication on top of LSM, e.g. for synchronization on ring buffers. There is not much work on implementation of distributed semaphores [25]. Transferring LSM solutions for message passing to DSM has led to poor performance due to increase of network messages and latencies for simulation of semaphore operations. In this section, we show how message passing can be implemented very efficiently on DSM employing speculative prefetching of region locks.

We assume the existence of an infinite buffer[9] in the address range $[0, \infty]$ and one single writer and one reader.

[9]Generalizations to two-way-locking on finite buffers (e.g. circular ring buffers) such that the writer is blocked on full buffers are left to the reader as an exercise.

Synchronization can be achieved in the following way: upon initialization of the system, the writer obtains a lock for the whole region $[0, \infty]$. The writer starts writing at address 0, keeping a write pointer in its local memory. A write of n bytes is performed by the following steps: first the n bytes are written to the address pointed to by the write pointer, afterwards the write pointer is incremented by n, and finally the written region is unlocked such that the rest of the buffer remains locked. The reader is working in a mirror symmetric way: first obtaining a lock at the read pointer position for a length of n bytes, afterwards reading data and incrementing the read pointer accordingly. Other solutions exist which are also based on dynamic changes of lock regions[10]. The common headline may be characterized as applying the principle of *direct manipulation* to locks, i.e. always locking exactly the necessary memory regions instead of using substitute locking objects like semaphores.

The worst-case communication overhead for this type of synchronization will be no worse than with conventional message passing and message confirmation if the data transfer is integrated with the lock messages at least for small sizes of n. Interestingly, speculative prefetching of locks will *automatically* improve communication overhead in best case and also in many average cases: if the writer issues a burst of writes, the locally unlocked obligatory regions will *accumulate* to a larger unlocked obligatory region, but the corresponding optional region will remain intact until the reader starts to issue a read operation. Using an appropriate prefetching strategy for linear sweeps, the reader can then obtain a much larger optional region as originally requested by the local obligatory lock, such that access permissions for larger portions of the buffer are transferred in a single step. This effect is very similar to *message buffering* and *message coalescing* in conventional message-passing systems.

This effect becomes important when multiple *anonymous* readers or writers are entering the scene. Note that conventional message passing is based on explicit naming of sender and receiver, while DSM *allows* anonymous senders and receivers as well. In our model, synchronization among multiple readers or writers can be achieved by mutex locking of the write pointer and read pointer respectively; in case of only a single reader or writer no network messages besides initialization will be generated for synchronization on those pointers. In case of multiple readers or writers, speculative locking leads to automatic coalescing of messages without forcing the consumers to really consume all of the data (similar to lookahead on data messages). Note also that we provide a solution for *disjoint distribution* of messages such that messages are guaranteed to never arrive twice at different sites, which is very different from broadcasting of messages. By introduction of read locks versus write locks, even broadcasts can be efficiently simulated by DSM. To increase the degree of parallelism between readers and writers, more sophisticated models like multiversion DSM (MVDSM) in analogy to multiversion databases can also be derived.

Communication on top of DSM has also advantages for fault tolerance due to better symmetry properties. An obligatory lock is only an access right independent from replication of the corresponding data. In case of emergency (e.g. site failure), even obligatory locks may be retracted. However, for achieving consistency on complex data structures we probably need some transactional concepts; integration of speculative prefetching of optional locks with transactions may become an interesting research area. Other problems of distributed systems like network partitions [26] should be independent from the paradigms DSM versus explicit message passing.

A further well-known argument for DSM is its ability to perform *reference semantics* very naturally, while message passing implementations like RPC or CORBA provide only value semantics as basic mechanisms. Simulation of reference semantics at application level by means of value semantics may become extremely complicated and expensive. Probably there are a lot of distributed applications which would profit from cheap and easy reference semantics.

We conclude that there hardly remains a reason to build distributed systems on message passing as an application-level basic communication mechanism instead of on DSM. Since application-level format conversions, marshalling, object brokers, and other consequences of message passing implementations like RPC, CORBA etc. are either not necessary with DSM or can be simulated and added easily, we expect that building applications on top of DSM can significantly *simplify* the architecture of future distributed applications. In addition, speculative prefetching strategies can *automate* tasks such as message coalescing which are currently often done by hand at application level, and can improve the *performance* where the effort for implementation of sophisticated message coalescing has not been spent in all places.

There is an interesting historic analogy: before demand paging in large virtual address spaces was widely available, progammers used *overlays* and explicit swapping. Explicit message passing at application level may be automated in sufficiently homogeneous environments in a similar way as demand paging did for modular applications running on the same machine.

Using DSM as base mechanism for distributed systems will not render middleware superfluous; on the contrary, the heterogeneity of current and probably of future application architectures remains a serious challenge even if all performance problems of distributed systems were completely solved. Improving the lock locality properties of DSM applications may also become an attractive research field.

[10]It suffi ces when the writer always keeps a lock of a size of at least 1 at any point in time. Before entering n bytes at position i, a lock must be present at i with length 1. Before actually writing the n bytes, this lock is extended to the interval $[i, i+n]$ and after writing it is cut to $[i+n, i+n]$ leading to the impression of a ”moving” lock which moves in parallel to the write pointer.

5 Conclusions

We have presented a family of methods for speeding up distributed locks when applications show locking patterns with locality properties well suited for distributed systems. When applications show bad temporal and spatial locality, we have argued that there is likely no help for any distributed locking method. For applications showing behaviour in the middle, there remains some room for future improvements, better heuristics etc.

Our experiments provide some evidence that there exist applications showing both good spatial and temporal locality, such that more than 90% of all lock requests can be handled locally without network latencies; exploiting both spatial and temporal locality is often significantly better than exploiting only temporal locality. Since local processors are typically many orders of magnitudes faster than network latencies, overall speedup factors of 10 or more for average locking latencies are possible. Speedup is expected with similar properties and similar results as data caching has speeded up data access. More research is needed for modeling application behaviour with respect to locking, for revealing detailed commonalities and differences to data access, for finding better global and local lock manager policies, and for integration with distributed data caching.

We have explained how message passing can be implemented on top of DSM very efficiently when our prefetching methods are used. We expect that a combination of our methods with distributed data caching will have a similar impact on distributed systems as local data caching had for standalone systems. In particular, the DSM model should become more attractive in favour of explicit message passing, such that many applications need no longer be written for specialized APIs for distributed systems.

References

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

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

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

[4] J. Jürgens, "Synchronisation paralleler prozesse anhand von zuständen," Ph.D. dissertation, Technische Universität München, 1973.

[5] L. Lamport, "A new solution of dijkstra's concurrent programming problem," *CACM*, vol. 17, no. 8, pp. 453–455, 1974.

[6] A. N. Habermann, "Prevention of system deadlocks," *CACM*, vol. 12, no. 7, pp. 373–385, 1969.

[7] 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.

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

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

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

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

[12] 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

[13] 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.

[14] P. J. Denning, "The working set model for program behavior," *CACM*, vol. 11, no. 5, pp. 323–333, 1968.

[15] ——, "Virtual memory," *Computing Surveys*, vol. 2, no. 3, pp. 153–189, 1971.

[16] C. G. Gray and D. R. Cheriton, "Leases: An effi cient fault-tolerant mechanism for distributed fi le cache consistency," *Symposium on Operating System Principles*, pp. 202–210, 1989.

[17] T. Härder and E. Rahm, *Datenbanksysteme – Konzepte und Techniken der Implementierung*, 2nd ed. Springer-Verlag, 2001.

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

[19] P. J. Denning, "Working sets past and present," *IEEE Transactions of Software Engineering*, vol. 6, no. 1, pp. 64–84, 1980.

[20] A. W. Madison and A. P. Batson, "Characteristics of program localities," *CACM*, vol. 19, no. 5, pp. 285–294, 1976.

[21] M. Wang, A. Ailamaki, and C. Faloutsos, "Capturing the spatio-temporal behavior of real traffi c data," Preprint of a paper submitted to Elsevier Science, citeseer.nj.nec.com/592012.html or http://www.db.cs.cmu.edu/Pubs/Lib/perfeval02wang/pqrs.ps.gz, Performance Evaluation 2002, Rome, Italy, 2002.

[22] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM transactions on database systems*, vol. 4, no. 2, pp. 180–209, 1979.

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

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

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

[26] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in partitioned networks," *Computing Surveys*, vol. 17, no. 3, pp. 341–370, 1985.