Good Programming in Transactional Memory
Game Theory Meets Multicore Architecture

Raphael Eidenbenz and Roger Wattenhofer

Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland
`{eidenbenz,wattenhofer}@tik.ee.ethz.ch`

Abstract. In a multicore transactional memory (TM) system, concurrent execution threads interact and interfere with each other through shared memory. The less interference a program provokes the better for the system. However, as a programmer is primarily interested in optimizing her individual code's performance rather than the system's overall performance, she does not have a natural incentive to provoke as little interference as possible. Hence, a TM system must be designed compatible with good programming incentives (GPI), i.e., writing efficient code for the overall system coincides with writing code that optimizes an individual program's performance. We show that with most contention managers (CM) proposed in the literature so far, TM systems are not GPI compatible. We provide a generic framework for CMs that base their decisions on priorities and explain how to modify Timestamp-like CMs so as to feature GPI compatibility. In general, however, priority-based conflict resolution policies are prone to be exploited by selfish programmers. In contrast, a simple non-priority-based manager that resolves conflicts at random is GPI compatible.

Key words: transactional memory, game theory, multicore architecture, concurrency, contention management, mechanism design, human factors.

1 Introduction

In traditional single core architecture, the performance of a computer program is usually measured in terms of space and time requirements. In multicore architecture, things are not so simple. Concurrency adds an incredible, almost unpredictable complexity to today's computers, as concurrent execution threads interact and interfere with each other. The paradigm of Transactional Memory (TM), proclaimed and implemented by Herlihy and Moss [6] in the 1990's, has emerged as a promising approach to keep the challenge of writing concurrent code manageable. Although today, TM is most-often associated with multithreading, its realm of application is much broader. It can for instance also be used in inter process communication where multiple threads in one or more processes exchange data. Or it can be used to manage concurrent access to system resources. Basically, the idea of TM can be employed to manage any situation where several tasks may concurrently access resources representable in memory. A TM system provides the possibility for programmers to wrap critical code that performs operations on shared memory into transactions. The system then guarantees an exclusive code execution such that no other code being currently processed interferes with the critical operations. To achieve this, TM systems employ a contention management policy. In *optimistic* contention management, transactional code is executed right away and modifications on

```
incRingCounters(Node start){
  var cur = start;
  atomic{
    while(cur.next!=start){
      c = cur.count;
      cur.count = c + 1;
      cur = cur.next; }}}
```

```
incRingCountersGP(Node start){
  var cur = start;
  while(cur.next!=start){
    atomic{
      c = cur.count;
      cur.count = c + 1;}
    cur = cur.next; }}
```

Fig. 1. Two variants of updating each node in a ring.

shared resources take effect immediately. If another process, however, wants to access the same resource, a mechanism called contention manager (CM) resolves the conflict, i.e., it decides which transaction may continue and which must wait or abort. In case of an abort, all modifications done so far are undone. The aborted transaction will be restarted by the system until it is executed successfully. Thus, in multicore systems, the quality of a program must not only be judged in terms of space and (contention-free) time requirements, but also in terms of the amount of conflicts it provokes due to concurrent memory accesses.

Consider the example of a shared ring data structure. Let a ring consist of s nodes and let each node have a counter field as well as a pointer to the next node in the ring. Suppose a programmer wants to update each node in the ring. For the sake of simplicity we assume that she wants to increase each node's counter by one. Given a start node, her program accesses the current node, updates it and jumps to the next node until it ends up at the start node again. Since the ring is a shared data structure, node accesses must be wrapped into a transaction. We presume the programming language offers an **`atomic`** keyword for this purpose. The first method in Figure 1 (`incRingCounters`) is one way of implementing this task. It will have the desired effect. However, wrapping the entire while-loop into one transaction is not a very good solution, because by doing so, the update method keeps many nodes blocked although the update on these nodes is already done and the lock[1] is not needed anymore. A more desirable solution is to wrap each update in a separate transaction. This is achieved by a placement of the **`atomic`** block as in `incRingCountersGP` on the right in Figure 1. When there is no contention, i.e., no other transactions request access to any of the locked ring nodes, both `incRingCounters` and `incRingCountersGP` run equally fast[2] (cf. Figure 2). If there are interfering jobs, however, the affected transactions must compete for the resources whenever a conflict occurs. The defeated transaction then waits or aborts and hence system performance is lost. In our example, using `incRingCounters` instead of `incRingCountersGP` leads to many unnecessarily blocked resources and thereby increases the risk of conflicts with other program parts. In addition, if there is a conflict and the CM decides that the programmer's transaction must abort, then with `incRingCountersGP` only one modification needs to be undone, namely the update to the current node in the ring, whereas with `incRingCounters` all modifications back to the start node must be rolled back. In brief, employing `incRingCounters` causes an avoidable performance loss.

[1] An optimistic, direct-update TM system "locks" a resource as soon as the transaction reads or writes it and releases it when committing or aborting. This is not to be confused with an explicit lock by the programmer. In TM, explicit locks are typically not supported.

[2] if we disregard locking overhead

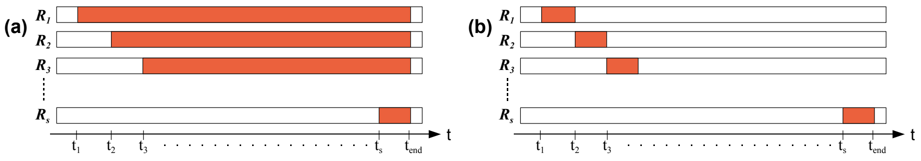

Fig. 2. Transactional allocation of ring nodes **(a)** by `incRingCounters` and **(b)** by `incRingCountersGP`.

One might think that it is in the programmer's interest to choose the placement of atomic blocks as beneficial to the TM system as possible. The reasoning would be that by doing so she does not merely improve the system performance but the efficiency of her own piece of code as well. Unfortunately, in current TM systems, it is not necessarily true that if a thread is well designed—meaning that it avoids unnecessary accesses to shared data—it will also be executed faster. On the contrary, we will show that most CMs proposed so far privilege threads that incorporate long transactions rather than short ones. This is not a severe problem if there is no competition for the shared resources among the threads. Although in minor software projects all interfering threads might be programmed by the same developer, this is not the case in large software projects, where there are typically many developers involved, and code of different programmers will interfere with each other. Furthermore, we must not assume that all conflicting parties are primarily interested in keeping the contention low on the shared objects, especially if doing so slows down their own thread. It is rather realistic to assume that in many cases, a developer will push his threads' performance at the expense of other threads or even at the expense of the entire system's performance if the system does not prevent this option.[3] In order to avoid this loss of efficiency, a multicore system must be designed such that the goal of achieving an optimal system performance is compatible with an individual programmer's goal of executing her code as fast as possible. This paper shows that, unfortunately, most CMs proposed in the literature so far lack such an incentive compatibility. In the remainder, we explain our model, explore the meaning of good programming in a TM system in Section 3, provide a framework for priority based CMs and a classification of CMs w.r.t. incentive compatibility in Section 4 and show an example of a CM not based on priority (Section 5). If a proof is only sketched or missing, you can find the complete proof in Appendix A. As a practical proof of our findings, we implemented free-riding strategies in the TM library DSTM2[5] and tested them in several scenarios. These results can be found in Appendix B.

2 Model

We use a model of a TM system with optimistic contention management, immediate conflict detection and direct update. As we do not want to restrict TM to the domain

[3] There is competition in many projects, especially within the same company. Just think of the next evaluation! If TM is to be employed in other domains such as inter process communication or managing access to system wide resources (DB, files, system variables), a competitive model is even more obtrusive.

of multithreading, we will use the notion of jobs instead of threads to denote a set of transactions belonging together. In inter process communication, e.g., a job is rather a process than a thread. A job J_i consists of a sequence of *transactions* $T_{i1}, T_{i2}, \ldots, T_{ik}$. If J_i consists of only one transaction, we sometimes write T_i instead of T_{i1}. Transactions access shared *resources* R_i. At any point in time, we denote by n the number of running transactions in the system and by s the number of resources currently accessed. For the sake of simplicity, we consider all accesses as exclusive,[4] thus, if two transactions both try to access resource R_i at the same time or if one has already locked R_i and the other desires access to R_i as well, they are in *conflict*. When a conflict occurs, a mechanism decides which transaction gains (or keeps) access of R_i and has the other competing transaction wait or abort. Such a mechanism is called *contention manager (CM)*. We assume that once a transaction has accessed a resource, it keeps the exclusive access right until it either commits or aborts. We further assume that the time needed to detect a conflict, to decide which transaction wins and the time used to commit or start a transaction are negligible. We neither restrict the number of jobs running concurrently, nor do we impose any restrictions on the structure and length of transactions.[5] We say a job J_i *is running* if its first transaction T_{i1} has started and the last T_{ik} has not committed yet. Notice that in optimistic contention management, the starting time t_i of a job J_i and therewith the starting time t_{i1} of T_{i1} is not influenced by the CM, since it only reacts once a conflict occurs. The *environment* $\mathcal{E}$ is a potentially infinite set of tuples of a job and the time it enters the system, i.e., $\mathcal{E} = \{(J_0, t_0), (J_1, t_1), \ldots\}$. We assume that the state at a time t of a TM system managed by a deterministic CM is determined by the environment $\mathcal{E}$. The execution environment of a job J_i is then $\mathcal{E}_{-i} = \mathcal{E} \setminus \{(J_i, t_i)\}$.We further assume that once a job J_i is started at time t_i, any contained transaction T_{ij} accesses the same resources in each of its executions and for any resource, the time of its first access after a (re)start of T_{ij} remains the same in each execution. Once t_i is known, this allows a description of a contained transaction by a list of all resources accessed with their relative access time. E.g., $T_{ij} = (\{(R_1, t_1), \ldots, (R_k, t_k)\}, d_{ij})$ means that transaction T_{ij} accesses R_1 after t_1 time and so forth until it hopefully commits after d_{ij} time. The *contention-free execution time* $d_{ij}^{\mathcal{E}}$ is the time the system needs to execute T_{ij} if T_{ij} encounters no conflicts. The *job execution time* $d_i^{\mathcal{M},\mathcal{E}}$ is the time J_i's execution needs in a system managed by $\mathcal{M}$ in environment $\mathcal{E}$, i.e., the period from the time T_{i1} enters the system, t_{i1}, until the time T_{ik} commits. Similarly, $d_{ij}^{\mathcal{M},\mathcal{E}}$ denotes the execution time of transaction T_{ij} and $d^{\mathcal{M},\mathcal{E}}$ is the *makespan* of all jobs in $\mathcal{E}$, i.e., the time from $\min_i t_i$ until $\max_i(t_i + d_i^{\mathcal{M},\mathcal{E}})$. We denote by $\mathcal{M}^*$ an optimal offline contention manager. We presume $\mathcal{M}^*$ to know all future transactions. It can thus schedule all transactions optimally so as to minimize the makespan. We assume that the program code of each job is written by a different selfish developer and that there is competition among those developers. *Selfish* in this context means that the programmer only cares about how fast her job terminates. A developer is considered *rational*, i.e., she always acts so as to maximize her expected utility. This is, she minimizes her job's expected execution time. Further, we assume the developers to be *risk-averse* in the sense that

[4] Invisible reads that would allow a concurrent access without conflicts are not considered.

[5] That is why we do not address the problem of recognizing dead transactions and ignore heuristics included in CMs for this purpose.

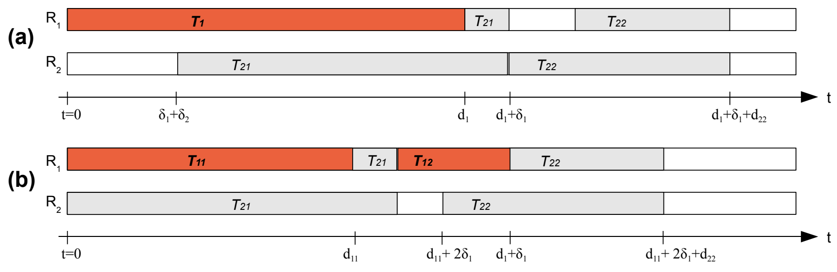

Fig. 3. Partitioning example. The picture depicts the optimal allocation of two resources R_1 and R_2 over time in two situations **(a)** and **(b)**. In **(a)**, the programmer of job J_1 does not partition T_1. In **(b)**, she partitions T_1 into T_{11} and T_{12}. The overall execution time is shorter in **(b)**, the individual execution of J_1, however, is faster in **(a)**.

they expect the worst case to happen, however they expect their job J_i to eventually terminate even if under certain environments, $\mathcal{M}$ does not terminate J_i. This assumption is inevitable since with many CMs, there exist (at least theoretically) execution environments $\mathcal{E}_{-i}$ which make J_i run forever. Thus a risk-averse developer could just as well twirl her thumbs instead of writing a piece of code without this assumption. In Lemma 1, the used notion of rationality will be further adapted in that we argue that delaying a transaction does not make sense if an arbitrary environment is assumed.

3 Good Programming Incentives (GPI)

Our main goal is to design a multicore TM system that is as efficient as possible. As we may not assume programmers to write code so as to maximize the overall system performance but rather to optimize their individual job's runtime, we must design a system such that the goal of achieving an optimal system performance is compatible with an individual programmer's goal of executing her code as fast as possible. A first step in this direction is to determine the desired behavior, that is, we have to find the meaning of *good programming* in a TM system. We want to find out how a programmer should structure her code, or in particular, how she should place atomic blocks in order to optimize the overall efficiency of a TM system.

When a job accesses shared data structures it puts a load on the system. The insight gained by studying the example in the introduction is that the more resources a job locks and the longer it keeps those locks, the more potential conflicts it provokes. If the program logic does not require these locks, an unnecessary load is put on the system.

Fact 1 *Unnecessary locking of resources provokes a potential performance loss in a TM system.*

However the question remains whether *partitioning* a transaction into smaller transactions—even if this does not reduce the resource accesses—results in a better system performance. Consider an example where the program logic of a job J_1 requires exclusive access of resource R_1 for a period of d_1. One strategy for the programmer is to simply wrap all operations on R_1 into one transaction $T_1 = (\{(R_1, 0)\}, d_1)$.

However, let the semantics also allow an execution of the code in two subsequent transactions $T_{11} = (\{(R_1, 0)\}, d_{11})$ and $T_{12} = (\{(R_1, 0)\}, d_{12})$ without losing consistency and without overhead, i.e., $d_{11} + d_{12} = d_1$. Figure 3 shows the execution of both strategic variants in a system managed by an optimal CM $\mathcal{M}^*$ in an execution environment $\mathcal{E}_{-1} = \{(J_2, 0)\}$ with only one concurrent job J_2. Both jobs J_1 and J_2 enter the system at time $t = 0$. Job J_2 consists of transactions T_{21} and T_{22} with $T_{21} = (\{(R_2, 0), (R_1, d_{21} - \delta_1)\}, d_{21})$ and $T_{22} = (\{(R_2, 0), (R_1, \delta_2)\}, d_{22})$ Furthermore, let $\delta_1 << d_1$ and $\delta_2 << d_1$. In situation **(a)**, the programmer does not partition T_1, $\mathcal{M}^*$ schedules T_1 first, at time $t = 0$, T_{21} at $t = \delta_1 + \delta_2$ and T_{22} at $t = d_1 + \delta_1$. This optimal schedule of T_1, T_{21} and T_{22} has a makespan of $d_1 + \delta_1 + d_{22}$. In situation **(b)**, the programmer partitions T_1 into T_{11} and T_{12}, an optimal CM schedules T_{11} and T_{21} concurrently at time $t = 0$, T_{12} at $t = d_{21} = d_{11} + \delta_1$ and T_{22} at $t = d_{11} + 2\delta_1$. This yields a makespan of $d_{11} + 2\delta_1 + d_{22} = d_1 + \delta_1 + d_{22} - \delta_2$. Thus, in the example of Figure 3, partitioning T_1 allows to schedule J_1 and J_2 by δ_2 faster. We can show that partitioning is beneficial in a system managed by $\mathcal{M}^*$ in general.

Theorem 1 *Let T_{ij1}, T_{ij2} be a valid partition of T_{ij}. Let J_i be a job containing T_{ij1}, T_{ij2} and J_i' the same job except it contains T_{ij} instead of T_{ij1}, T_{ij2}. A finer transaction granularity speeds up a transactional memory system managed by an optimal CM $\mathcal{M}^*$, i.e., $\forall\ \mathcal{E}_{-i}, t\ :\ d^{\mathcal{M}^*, \mathcal{E}_{-i} \cup \{(J_i, t)\}} \leq d^{\mathcal{M}^*, \mathcal{E}_{-i} \cup \{(J_i', t)\}}$ and $\exists\ \mathcal{E}_{-i}, t$ such that inequality holds.*

Proof (Sketch). Partitioning transactions only gives more freedom to $\mathcal{M}^*$. To be at least as fast with J_i as with J_i', $\mathcal{M}^*$ could execute T_{i2} right after T_{i1}. In some cases it might be even faster to schedule an intermediary transaction between T_{i1} and T_{i1}. □

Note that Theorem 1 proves partitioning to be beneficial to a system with an optimal CM. Of course, this does not hold for all CMs. As partitioning gives more freedom to the CM, though, it is highly probable that by incentivizing partitioning, a system achieves a better performance even with the additional overhead needed for incentive compatibility.

Let us reconsider the example from Figure 3. We have seen that partitioning T_1 into T_{11} and T_{12} results in a smaller makespan. But what about the individual execution time of job J_1? In the unpartitioned execution, where J_1 only consists of T_1, J_1 terminates at time $t = d_1$. In the partitioned case, however, J_1 terminates at time $t = d_1 + \delta_1$. This means that partitioning a transaction speeds up the *overall* performance of a concurrent system managed by an optimal CM, but it possibly slows down an *individual* job. Thus, from a selfish programmer's point of view, it is not rational to simply make transactions as fine granular as possible. In fact, if a finer grained partitioning of transactions might result in a slower execution of a job, why should a selfish programmer make the effort of finding a transaction granularity as fine as possible?

Avoiding unnecessary locks and partitioning transactions whenever possible is beneficial to a TM system. We say a CM $\mathcal{M}$ *rewards partitioning* of transactions if in a system managed by $\mathcal{M}$, it is rational for a programmer to always partition a transaction whenever the program logic allows her to do so. Further, $\mathcal{M}$ *punishes unnecessary locking* if in a system managed by $\mathcal{M}$, it is rational for a programmer to never lock resources unnecessarily, i.e., she only locks a resource when required by the program logic. One can expect that, from a certain level of selfishness among developers, a CM which incentivizes these two crucial aspects of good programming, performs better than the best incentive incompatible CM. In the remainder, we are mainly concerned with the question of which contention management policies fulfill the following property.

Property 1. A CM is *good programming incentive (GPI) compatible* if it rewards partitioning and punishes unnecessary locking.

As a remark, we would like to point out that the optimal CM $\mathcal{M}^*$ does not reward partitioning and hence is not GPI compatible. This is shown by the example from Figure 3. Note that the optimality of $\mathcal{M}^*$ refers to the scheduling of a given transaction set. If we assume developers act selfish then also a system managed by $\mathcal{M}^*$ suffers a performance loss and a different CM which offers incentives for good programming might be more efficient than $\mathcal{M}^*$. There is, however, an inherent loss due to the lack of collaboration. In game theory, this loss is called *price of anarchy* (cf. [2, 7]).

4 Priority-Based Contention Management (CM)

One key observation when analyzing the contention managers proposed in [1, 4, 8–10] is that most of them incorporate a mechanism that accumulates some sort of priority for a transaction. In the event of a conflict, the transaction with higher priority wins against the one with lower priority. Most often, priority is supposed to measure, in one way or another, the work already done by a transaction.[6] The intuition behind this approach is that aborting old transactions discards more work already done and thus hurts the system efficiency more than discarding newer transactions. The proposed contention managers base priority on a transaction's time in the system, the number of conflicts won, the number of aborts or the number of resources accessed. Definition 1 introduces a framework that comprises priority-based CMs. It allows us to classify priority-based CMs and to make generic statements about GPI compatibility of certain CM classes.

Definition 1 *A* priority-based *CM $\mathcal{M}$ associates with each job J_i a priority vector $\boldsymbol{\omega}_i \in \mathbb{R}^s$ where $\boldsymbol{\omega}_i[k]$ is J_i's priority on resource R_k. $\mathcal{M}$ resolves conflicts between two transactions $T_{ij} \in J_i$ and $T_{jr} \in J_j$ over resource R_k by aborting the transaction with lower priority, i.e., if $\boldsymbol{\omega}_i[k] \geq \boldsymbol{\omega}_j[k]$ then T_{ij} wins otherwise T_{ij} is aborted.*

In many CMs, all entries of the vector $\boldsymbol{\omega}_i$ are equal. In this case, we can also replace $\boldsymbol{\omega}_i$ by a scalar priority value $\omega_i \in \mathbb{R}$. We call such a CM *scalar-priority-based.* In the remainder we often use ω_i instead of $\boldsymbol{\omega}_i$, for the sake of simplicity, even if we are not talking about scalar-priority-based CMs only. Mostly, for a correct valuation of a job's competitiveness, absolute priority values are not relevant, but the relative value to other job priorities. A job J_i's *relative priority* vector $\tilde{\omega}_i$ is defined by $\tilde{\omega}_i[k] = \boldsymbol{\omega}_i[k] - \min_{i=1\ldots n} \boldsymbol{\omega}_i[k]$, $\forall k = 1 \ldots s$. If the CM uses scalar priorities, J_i's relative priority $\tilde{\omega}_i$ is obtained by subtracting $\min_{i=1\ldots n} \boldsymbol{\omega}_i$ from the absolute priority ω_i. Since optimistic CMs feature a reactive nature it is best to consider the priority-building mechanism as event-driven. We find that the following *events* may occur for a transaction $T_{ij} \in J_i$ in a transactional memory system:

[6] Timestamp and Greedy measure the priority by the time a transaction is already running. Karma takes the number of accessed objects as priority measure. Kindergarten gives priority to transactions which already backed off against the competing transaction.

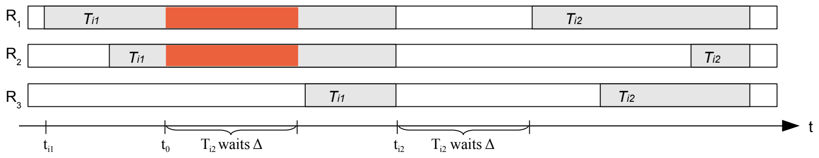

Fig. 4. Job $J_i = T_{i1}, T_{i2}$ waits at time t_0 for a period of Δ. In this period, J_i keeps locking the already locked resources R_1 and R_2. After T_{i1} commits, the system would let T_{i2} start immediately, but the programmer of J_i decided to let T_{i2} wait Δ before it accesses the first resource.

- A time step ($\mathcal{T}$),
- T_i wins a conflict ($\mathcal{W}$),
- T_i loses a conflict and is aborted ($\mathcal{A}$),
- T_i successfully allocates a resource R_k ($\mathcal{R}_k$),
- T_i commits ($\mathcal{C}$).

The following two subtypes of priority-based CMs capture most contention management policies in the literature.

Definition 2 *A priority-based CM is* priority-accumulating *iff no event decreases a job's priority and there is at least one type of event which causes the priority to increase.*

Definition 3 *A CM is* quasi-priority-accumulating *iff it is priority-accumulating w.r.t. events $\mathcal{T}$, $\mathcal{W}$, $\mathcal{A}$ and $\mathcal{R}$ but it resets J_i's priority when a transaction $T_{ij} \in J_i$ commits.*

As an example consider a Timestamp CM $\mathcal{M}_T$. $\mathcal{M}_T$ uses only events of type $\mathcal{T}$ and $\mathcal{C}$, i.e., in a time step dt after $T_{ij} \in J_i$ entered the system, ω_i is increased by $d\omega = \alpha dt$, $\alpha \in \mathbb{R}^+$ until $\mathcal{C}$ occurs, then reset to 0. J_i's scalar priority at time t, $t_{ij} < t \leq t_{ij} + d_{ij}^{\mathcal{M}_T,\mathcal{E}}$ is $\omega_i(t) = \int_{t_{ij}}^{t} \alpha dt = \alpha(t - t_{ij})$. If not for the event of a commit, where a job's priority is reset, Timestamp would be priority-accumulating, since a contained transaction's priority always increases and never decreases over time. It is, however, a quasi-priority-accumulating CM.

4.1 Waiting Lemma

We argue in this section that delaying the execution of a job[7] is not rational with the assumption that the execution environment $\mathcal{E}_{-i}$ is arbitrary. We consider cases where J_i waits before (re)starting a transaction T_{ij} as well as cases where T_{ij} is already running, has locked some resources and then waits before resuming (cf. Figure 4).

This assumption implies that at any point in time, the history of the transactions does not hold any information about their future. Furthermore, we demand two restrictions on the contention manager's priority modification mechanism:

[7] A programmer can implement waiting by executing code without allocating shared resources.

(I.) An increase (or decrease) of ω_i never depends on ω_i's current value[8] or on any other job's priority value.
(II.) In a period where no events occur except for time steps, all priorities ω_i increase by $\Delta\omega \geq 0$. Note that if $\Delta\omega$ is always 0, the priority is not based on time.

Lemma 1. *If $\mathcal{E}_{-i}$ is arbitrary, the strategy of waiting is irrational in a system managed by a priority-based CM $\mathcal{M}$ restricted by (I.–II.).*

Proof. Let $\omega_i(t)$ be ω_i at time t. Let σ denote the strategy of not waiting. Let the wait-free job employed by σ be J_i and let it enter the system at time t_i. We show that σ is the best strategy and therefore waiting is not rational. Let us assume for the sake of contradiction that there is a strategy σ' which is better than σ. Let the job employed by σ' be J'_i where J'_i is the same job as J_i except it waits in an interval $[t_0, t_0 + \Delta]$. In particular, let $T'_{ij} \in J'_i$ be the transaction which waits to be started or resumed in $[t_0, t_0 + \Delta]$ and $T_{ij} \in J_i$ its wait-free counterpart. We argue that during the interval $[t_0, t_0 + \Delta]$, an adversary can establish a situation for T'_{ij} which is at least as bad at time $t_0 + \Delta$ as the situation at t_0. The adversary simply keeps all the locks and does not start any new transactions or jobs, nor does she access new resources. Because of restriction (II.), we have $\tilde{\omega}'_j(t_0 + \Delta) = \tilde{\omega}'_j(t_0)\ \forall\, j = 1 \ldots n$, i.e., the relative priorities are conserved. Since the conflict-resolving mechanism of $\mathcal{M}$ does not depend on the priorities' absolute values, but only on their order, and further, modifications of priorities never depend on the priorities' absolute values, by resuming all work at $t_0 + \Delta$ and delaying all jobs in $\mathcal{E}_{-i}$ with starting time $> t_0$ by Δ, the adversary achieves an execution time of J'_i which is Δ larger than J_i's execution time. Thus σ is better than σ'. Contradiction. $\square$

Note that the assumption on $\mathcal{E}_{-i}$ being arbitrary naturally applies if the programmer has no information about the environment in which her program will be executed. Indeed, if the environment would be truly a worst case environment, the execution of job J_i would take forever. As with this assumption, starting a job would be completely pointless, we adapt our model of a risk-averse agent in that we let her suppose that a worst case environment yields a finite execution time. Although Lemma 1 is not counterintuitive, this supposition is needed to establish the claim. In practice, the programmer often has some information about the environment in which her programm will be deployed. Hence it might make sense to presume some structure of $\mathcal{E}_{-i}$. E.g., she could assume that lengths of locks follow a certain distribution, or that each resource has a given probability of being locked. In such cases waiting might not be irrational. In the following, we will sometimes argue that a CM is (not) GPI compatible by comparing two jobs J_i and J'_i where both are equal except for J'_i either locks a resource unnecessarily or does not partition a transaction although this would be semantically possible. We will show that in the same execution environment $\mathcal{E}_{-i}$, one job either perfoms faster or if it is slower, this is because it does not wait at a certain point in the execution. Since waiting is irrational, a developer will prefer this job even if it is not guaranteed to perform better in any environment.

[8] E.g., rules such as "if ω_i is larger than 10 add 100" or "$\omega_i = 2\omega_i$" are prohibited. "$\omega_i = \omega_i + 2$" is permitted.

4.2 Quasi-Priority-Accumulating CM

Quasi-priority-accumulating CMs increase a transaction's priority over time. Again, the intuition behind this approach is that, on one hand, aborting old transactions discards more work already done and thus hurts the system efficiency more than discarding newer transactions and on the other hand, any transaction will eventually have a priority high enough to win against all other competitors. This approach is legitimate. Although the former presupposes some structure of $\mathcal{E}$ and the latter is not automatically fulfilled, examples of quasi-priority-accumulating CMs showed to be useful in practice (cf. [9]). However, quasi-priority-accumulating CMs bear harmful potential. They incentivize programmers to not partition transactions and in some cases even to lock resources unnecessarily. Consider the case where a job has accumulated high priority on an resource R_i. It might be advisable for the job to keep locking R_i in order to maintain high priority. Although it does not need an exclusive access for the moment, maybe later on, the high priority will prevent an abort and thus save time. In fact, we can show that the entire class of quasi-priority-accumulating CMs is not GPI compatible.

Theorem 2 *Quasi-priority-accumulating CMs restricted by (I.–II.) are not GPI compatible.*

Proof. Let J_i' be a job containing T_{ij} and J_i the exact same job where T_{ij} is partitioned into T_{ij1} and T_{ij2}. We compare the performance of both J_i and J_i' under the same circumstances, i.e., the same CM $\mathcal{M}$, the same execution environment $\mathcal{E}_{-i}$ and they both enter the system at the same time t_i. Let ω_i' denote the priority associated with J_i'. Let t_0 be the time when T_{ij} is started in $\mathcal{E}_{-i} \cup \{(J_i', t_i)\}$. Thus t_0 is also the time when T_{i1} is started in $\mathcal{E}_{-i} \cup \{(J_i, t_i)\}$. We prove Theorem 2 for two subtypes of quasi-priority-accumulating CMs seperately. Case A: $\mathcal{M}$ starts T_{ij2} immediately after T_{ij1}. Assume that T_{ij} is not aborted until commit. Let $t_0 + d_{ij1}$ be the time when T_{ij1} commits. Since both jobs are executed in the same environment, T_{ij1} is not aborted until commit at time $t_0 + d_{ij1}$. Let there be at least one event that increases ω_i in $(t_0, t_0 + d_{ij1})$. Thus $\omega_i'(t_0 + d_{ij1}) = \omega_i(t_0 + d_{ij1}) > \omega_i(t_0) \geq 0$. When T_{ij1} commits, ω_i is reset to 0 and $\omega_i'(t_{ij2}) > \omega_i(t_{ij2})$. Since T_{ij2} is started immediately, it will provoke the exact same conflicts as T_{ij} at times $t \geq t_{ij2}$. Let the first event on T_{ij2} (except for time steps) be a conflict against a transaction T_k whose priority is lower than the priority of J_i', but higher than the priority of J_i, i.e., $\omega_i' > \omega_k > \omega_i$ at the time this conflict occurs. Thus, T_{ij2} is aborted and must be restarted, whereas T_{ij} wins the conflict. Let t_{ij2}^2 be the time when T_{ij2} is restarted. As the programmer could have achieved the same effect by letting T_{ij2} wait from the time of the conflict until t_{ij2}^2, Lemma 1 implies that $\mathcal{M}$ does not reward partitioning. Case B: $\mathcal{M}$ does not start T_{ij2} immediately after T_{ij1} but waits δ before starting T_{ij2}. If we assume that $\mathcal{E}_{-i}$ is s.t. both J_i and J_i' do not provoke any conflicts then J_i will be δ slower than J_i'. $\mathcal{M}$ does not reward partitioning. □

Theorem 2 reflects the intuition, that if committing decreases an advantage in priority then there are cases where it is rational for a programmer not to commit and start a new transaction but to continue instead with the same transaction. Obviously, the opposite case is possible as well, namely that by not committing, the developer causes a conflict with a high priority transaction on a resource, which could have been freed if the transaction would have committed earlier, and thus is aborted. As in our model of a risk-averse programmer, she does not suppose any structure on $\mathcal{E}_{-i}$, she does not know which case is more likely to happen either and therefore has no preference among the two cases. She would probably just choose the strategy which is easier to implement. If we assumed, e.g., that a resource R_i is locked at time t with probability p by a transaction with priority x where both, p and x follow a certain probability distribution, then there would be a clear trade-off between executing a long transaction and therewith risking more conflicts and partitioning a transaction and thus losing priority.

Note that a similar proof can be used to show that no priority-based CM rewards partitioning unless it prevents the case where, after a commit of transaction $T_{ij} \in J_i$, the subsequent transaction $T_{i(j+1)} \in J_i$ starts with a lower priority than T_{ij} had just before committing. In fact, we can show that all priority-accumulating CMs proposed by [1, 4, 8–10] are not GPI compatible.

Corollary 1 *Polite, Greedy, Karma, Eruption, Kindergarten, Timestamp and Polka are not GPI compatible.*

4.3 Priority-Accumulating CM

The inherent problem of quasi-priority-accumulating mechanisms is not the fact that they accumulate priority over time but the fact that these priorities are reset when a transaction commits. Thus, by comitting early, a job loses its priority when starting a new transaction. One possibility to overcome this problem is to not reset ω_i when a transaction of J_i commits. With this trick, neither partitioning transactions nor letting resources go whenever they are not needed anymore resets the accumulated priority. We further need to ensure that two subsequent transactions of J_i are scheduled right after each other, because otherwise partitioning would result in a longer execution even in a contention-free environment. We denote this property of a CM as *gapless transaction scheduling*. If a CM $\mathcal{M}$ only modifies priorities on a certain event type $\mathcal{X}$, we say $\mathcal{M}$ *is based only on $\mathcal{X}$-events*.

Lemma 2. *Any priority-accumulating CM $\mathcal{M}$ which schedules transactions gapless and is based only on time ($\mathcal{T}$-events) is GPI compatible.*

Proof. Outline: unnecessary locking is punished since it can cause the transaction to abort and restart. Thus restarted, the transaction might be lucky and catch a better slot for execution. However, this is the same as waiting and hence irrational. Partitioning is rewarded since committing and restarting does not decrease priority. Furthermore, if a finer-grained job loses in a conflict, it has to redo less work.

Proof: In a first step, we show that unnecessary locking is punished. Since $\mathcal{M}$ bases priority only on time, unnecessary locking does not increase priority unless it also increases the contention-free runtime. Let T'_{ij} be a transaction with an unnecessary lock l, i.e., T'_{ij} either locks a resource too early or it accesses a resource which it would not need at all. Let T_{ij} be the same transaction without l and $d^{\mathcal{E}}_{ij} = d'^{\mathcal{E}}_{ij}$, where $d'^{\mathcal{E}}_{ij}$ is the contention-free runtime of T'_{ij} in environment $\mathcal{E}$. Lock l either does not provoke a conflict or it does. In the former case, choosing T_{ij} or T'_{ij} results in the same execution time. In the latter, T'_{ij} aborts and T_{ij} continues. T'_{ij} is restarted. If T_{ij} runs until commit, playing T_{ij} yields a better execution time. Otherwise, let t_{last} be the time when

T'_{ij} is restarted for the last time before commit, i.e., T'_{ij} commits at time $t_{last} + d^{\mathcal{E}}_{ij}$. T_{ij} could also be scheduled at t_{last} and reach a commit time at least as good as T'_{ij}. This is because the resources allocated by T_{ij} would always be a subset of the resources allocated by T'_{ij} in the interval $[t_{last}, t_{last} + d^{\mathcal{E}}_{ij}]$. To do so, however, the programmer would delay T_{ij} from the abort until t_{last}. Thus, employing T'_{ij} instead of T_{ij} coincides with delaying T_{ij} and as $\mathcal{M}$ is time-based, Lemma 1 applies and implies that locking is irrational. It remains to show that if $d^{\mathcal{E}}_{ij} < d'^{\mathcal{E}}_{ij}$ then T_{ij} is still preferable. Let T'_{ij} be exactly like T_{ij} except for one lock l at time t_l which delays T'_{ij} by Δ. If l does not provoke a conflict, l has the same effect as waiting in the period of $[t_l, t_l + \Delta]$. With $\mathcal{M}$ being time-based, Lemma 1 implies that using T'_{ij} with the additional lock is irrational. In case l provokes a conflict, if T_{ij} runs until commit, playing T_{ij} yields a better execution time. Otherwise, let t_{last} be the time when T'_{ij} is restarted for the last time. T_{ij} could also be scheduled at $t_{last} + \Delta$ and reach a commit time at least as good as T'_{ij} because T_{ij} would provoke a subset of the conflicts T'_{ij} provokes and as the priority only depends on time, T_{ij} would also win all conflicts. Again, unnecessary locking coincides with waiting from the abort until $t_{last} + \Delta$ and omitting the unnecessary lock coincides with starting T_{ij} immediately. With Lemma 1 it follows that unnecessary locking is irrational.

In a second step, we show that partitioning is rewarded. Let T_{ij} be partitionable in T_{ij1} and T_{ij2}. Let J'_i be a job with a transaction T_{ij} and J_i the same job except it uses T_{ij1} and T_{ij2} instead of T_{ij}. If we compare the strategy of using J_i to the strategy of using J'_i then we can make the following observations. Since T_{ij1} starts at the same time as T_{ij}, T_{ij1} will lock the exact same resources and thus provoke the same conflicts as T_{ij}. T_{ij2} locks less or equally many resources as T_{ij} at any time $t \in [t_{ij2}, t_{ij} + d^{\mathcal{E}}_{ij}]$. Since $\mathcal{M}$ schedules transactions gapless, if T_{ij2} provokes a conflict at time $t \in [t_{ij2}, t_{ij} + d^{\mathcal{E}}_{ij}]$ then T_{ij} provokes the same conflict at time t. In the execution of J'_i there are three possible courses of an attempted execution of transaction $T_{ij} \in J_i$: (a) T_{ij} runs until commit, (b) T_{ij} is aborted in $[t_{ij}, t_{ij2}]$ or (c) T_{ij} is aborted in $[t_{ij2}, t_{ij} + d^{\mathcal{E}}_{ij}]$. Remember that T_{ij} and T_{ij1} start at the same time for the first attempt, i.e., $t_{ij} = t_{ij1}$. In case (a), T_{ij1}, T_{ij2} also run until commit because J_i provokes a subset of the conflicts that J'_i provokes. Since all those conflicts occur at the same time and the priority depends only on time, $\mathcal{M}$ also resolves the conflict in favor of J_i. Therefore, $d^{\mathcal{M},\mathcal{E}}_{ij1} + d^{\mathcal{M},\mathcal{E}}_{ij2} = d^{\mathcal{M},\mathcal{E}}_{i}$ and J'_i's runtime equals J_i's runtime. In case (b), T_{ij1} is aborted as well and restarts T_{ij1} at the same time as T_{ij} is restarted and the proof applies recursively. In case (c), if T_{ij} is aborted because of a conflict before T_{ij2} aborts or commits, T_{ij2} could have used an unnecessary lock in order to provoke the same conflict. However, this is irrational, since $\mathcal{M}$ punishes unnecessary locks. Therefore, choosing J'_i instead of J_i is irrational as well. Partitioning is rewarded. □

Lemma 2 shows that it is possible to design priority-based contention managers which are GPI compatible. As an example, by simply not resetting a job J_i's priority when a contained transaction $T_{ij} \in J_i$ commits, we can make a Timestamp contention manager GPI compatible. Nevertheless, contention managers based on priority are generally dangerous in the sense that they bear a potential for selfish programmers to cheat, i.e., to find ways of boosting their program's priority such that their program is executed faster. This typically harms the overall system performance. E.g., consider a CM like Karma [8], where priority depends on the number of resources accessed. One way to gain high priority for a job would be to quickly access an unnecessarily large number of objects and thus become overly competitive. Or if priority is based on the number of aborts or the number of conflicts, a very smart programmer might use some dummy jobs which compete with the main job in such a way that they boost its priority. In fact, we can show that a large class of priority-accumulating CMs is not GPI compatible.

Theorem 3 *A priority-accumulating CM $\mathcal{M}$ is* not GPI compatible *if one of the following holds:*

(i) *$\mathcal{M}$ increases a job's relative priority on $\mathcal{W}$-events (winning a conflict).*
(ii) *$\mathcal{M}$ increases relative priority on $\mathcal{R}$-events (having exclusive access of a resource).*
(iii) *$\mathcal{M}$ schedules transactions gapless and increases relative priorities on $\mathcal{C}$-events (committing).*
(iv) *$\mathcal{M}$ restarts aborted transactions immediately and increases relative priorities on $\mathcal{A}$-events (aborting).*

Proof. Throughout the proof, we suppose w.l.o.g. that in a CM $\mathcal{M}$ each job J_i has exactly one priority $\omega_i \in \mathbb{R}$ associated to it. Let $\omega_i(t)$ denote J_i's priority at time t. For parts *(i)*, *(ii)* and *(iv)*, let T_{ij} be a transaction which locks resource R_1 at time t_u unnecessarily and let T'_{ij} be exactly the same transaction as T_{ij} except it does not lock R_1 at time t_u.

(i). Let us assume T_{ij} provokes an unnecessary conflict on R_1 with another transaction T_k at time t_u and $\omega_i(t_u) > \omega_k(t_u)$, i.e., T_{ij} wins the competition for R_1 and $\mathcal{M}$ increases ω_i by δ. Let us further assume that at time $t_u + \epsilon$, T_{ij} provokes a conflict on a resource R_2 with a transaction T_l and $\omega_l(t_u + \epsilon) < \omega_i(t_u + \epsilon) < \omega_l(t_u + \epsilon) + \delta$. If J_i would use T'_{ij} instead of T_{ij} then $\omega_l(t_u + \epsilon) > \omega_i(t_u + \epsilon)$ for an ϵ small enough. T'_{ij} would abort and prolongate the execution time of J_i. Thus $\mathcal{M}$ does not punish unnecessary locking.

(ii). Suppose there is no conflicting transaction on R_1 during $[t_u - \epsilon, t_u + \epsilon)$ and the contribution of having exclusive access of R_1 to the priority increase in this period is δ. Further assume that at time $t_u + \epsilon$, T_{ij} has a conflict with T_l and $\omega_l(t_u + \epsilon) < \omega_i(t_u + \epsilon) < \omega_l(t_u + \epsilon) + \delta$. If J_i would use T'_{ij} instead of T_{ij} then $\omega_l(t_u + \epsilon) > \omega_i(t_u + \epsilon)$, T'_{ij} would abort and prolongate the execution time of J_i. Thus $\mathcal{M}$ does not punish unnecessary locking.

(iii). Let J_i consist of the transactions T_{i1}, T_{i2} and T_{i3}. Let J'_i consist of T_{i1} and T_{i3}. Let T_{i2} be a simple transaction which unnecessarily locks R_1 for a period of ϵ and then commits. Let T_{i3} be a transaction which only accesses R_1. Assume the following scenario: $\mathcal{M}$ executes T_{i1} and commits at time t_0. T_{i2} starts immediately, locks R_1 for a period of ϵ and commits. $\mathcal{M}$ increases ω_i by δ and immediately starts T_{i3}. T_{i3} runs conflict-free for a time period d and then provokes a conflict with T_l where $\omega_l(t_3) < \omega_i(t_3) < \omega_l(t_3) + \delta$, $t_3 = t_0 + \epsilon + d$. We can further assume that if the programmer would use J'_i instead of J_i then T_{i3} would also run from time t_0 to t_3 provoking the same conflict with T_l. However, J'_i would lack the additional priority δ which was received by J_i for committing T_{i2}, i.e., for an ϵ small enough, it holds that $\omega_l(t_3) > \omega_i(t_3)$. T_{i3} would abort and prolongate the execution time of J'_i. Thus $\mathcal{M}$ does not punish unnecessary locking.

(iv). In a first step, we show *(iv)* under the assumption A that the time d_{rb} needed for rolling back is negligibly small. Suppose T_{ij} starts at $t_u - \epsilon$ and provokes a conflict with

T_k at time t_u and $\omega_i(t) < \omega_k(t)$. $\mathcal{M}$ aborts T_{ij} and increases ω_i by δ. $\mathcal{M}$ immediately restarts T_{ij}. Assume that at time $t_u + \epsilon$, T_{ij} provokes a conflict on with T_l and $\omega_l(t_u + \epsilon) < \omega_i(t_u + \epsilon) < \omega_l(t_u + \epsilon) + \delta$, after $t_u + \epsilon$, T_i runs conflict-free until commit. If J_i would use T'_{ij} instead of T_{ij} then T'_{ij} would not abort at time t_u and $\omega_l(t_u + \epsilon) > \omega_i(t_u + \epsilon)$. T'_{ij} would thus abort at time $t_u + \epsilon$ and prolongate the execution time of J_i. $\mathcal{M}$ does not punish unnecessary locking.

In a second step, we omit assumption A and assume for the sake of contradiction, that there is a GPI compatible CM $\mathcal{M}$ which increases ω_i by δ on the event of an abort of $T_{ij} \in J_i$. $\mathcal{M}$ has to ensure that the increase in priority δ due to an abort of T'_{ij} does not exceed the priority increase on T'_{ij}'s unnecessary lock-free counterpart, T_{ij}, during the roll back period d_{rb}. If this is not ensured the same argument we used for the proof with A applies. The CM would have to assume that no event except $\mathcal{T}$-events occur for T'_{ij} in d_{rb} as the future is unknown. Let $\Delta\omega_{rb}$ be the part of the priority increase on T'_{ij} in d_{rb} which is only due to $\mathcal{T}$-events. $\mathcal{M}$ cannot let T_{ij}'s priority grow more than $\Delta\omega_{rb}$ in d_{rb}. Since the increase in priority due to time steps is added anyway, $\mathcal{M}$ can only choose $\delta = 0$. $\mathcal{M}$ does not increase priority on $\mathcal{A}$-events. Contradiction. □

5 Non-Priority Based CM

One example of a CM which is not priority-based is Randomized (cf. [8]). To resolve conflicts, Randomized simply flips a coin in order to decide which competing transaction to abort. The advantage of this simple approach is that it bases decisions neither on information about a transaction's history nor on predictions about the future. This leaves programmers little possibility to boost their competitiveness.

Lemma 3. *Randomized is GPI compatible.*

Proof (Sketch). The proof is very similar to the proof of Lemma 2. In order to show that Randomized incentivizes partitioning and necessary locking, we compare J_i with J'_i under all possible environments. The environment includes the CM's randomized decisions. In some cases, the same argument as in Lemma 1 is used, namely that starting immediately is the better strategy than waiting, although Randomized is not a priority-accumulating CM. In fact, to show this for Randomized is much easier. An adversary can provoke the same conflicts for a transaction, if it is started immediately or if it is delayed for some time Δ. Since in any conflict, the probability of winning is the same, the expected runtime increases by Δ when the transaction is delayed. □

Employing such a simple Randomized CM is not a good solution although it rewards good programming. The probability $p_{success}$ that a transaction runs until commit decreases exponentially with the number of conflicts, i.e., $p_{success} \sim p^{|C|}$ where p is the probability of winning an individual conflict and C the set of conflicts. However, we see great potential for further developement of CMs based on randomization.

6 Conclusion and Future Work

While Transactional Memory constitutes an inalienable convenience to programmers in concurrent environments, it does not automatically defuse the danger that selfish programmers might exploit a multicore system to their own but not to the general good.

A TM system thus has to be designed strategy-proof such that programmers have an incentive to write code that maximizes the system performance. Priority-based CMs are prone to be corrupted unless they are based on time only. CMs not based on priority seem to feature incentive compatibility more naturally. We therefore conjecture that by combining randomized conflict resolving with a time-based priority mechanism, chances of finding an efficient, GPI compatible CM are high. Further potential future research includes the analysis of GPI compatibility if the programmer makes assumptions about the execution environment $\mathcal{E}_{-i}$ or if the system employs a pessimistic CM policy. Does waiting make sense in these settings? How accurate is the model of selfish, independent programmers and what is the actual efficiency loss due to GPI incompatibility in existing systems?

References

1. H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC '06: Proc. of the 25th annual ACM symposium on Principles of Distributed Computing*, pages 308–315, 2006.
2. G. Christodoulou and E. Koutsoupias. The price of anarchy of finite congestion games. In *STOC '05: Proceedings of the 37th annual ACM symposium on Theory of computing*, pages 67–73, 2005.
3. R. Eidenbenz and R. Wattenhofer. Brief announcement: Selfishness in transactional memory. In *SPAA '09: Proceedings of the 21st annual symposium on Parallelism in Algorithms and Architectures*, pages 41–42, New York, NY, USA, 2009. ACM.
4. R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the 24th annual ACM symposium on Principles of Distributed Computing*, pages 258–264, 2005.
5. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.
6. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.
7. T. Roughgarden. *Selfish Routing and the Price of Anarchy.* MIT Press, 2005.
8. W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP), St. John's, NL, Canada*, July 2004.
9. W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the 24th annual ACM symposium on Principles of Distributed Computing*, pages 240–248, 2005.
10. J. Schneider and R. Wattenhofer. Bounds On Contention Management Algorithms. In *Proc. of the 20th International Symposium on Algorithms and Computation (ISAAC)*, 2009.

A Appendix: Proofs

A.1 Proof of Theorem 1

First notice that we can assume w.l.o.g. that under $\mathcal{M}^*$ there are no conflicts: Any transaction T_{ij} will finally run from start to commit. If in an optimal schedule, T_{ij} is started and aborted already before the final run, then the same schedule where T_{ij} is not started until the final run is also optimal. Hence, we may assume that $\mathcal{M}^*$ schedules each transaction such that it runs completely until commit. This reflects the fact that an optimal CM is able to "look into the future". Hence it can avoid mistakes.

It is easy to see that $\mathcal{M}^*$ performs at least as good with J_i as with J_i'. $\mathcal{M}^*$ sets $t_{ij1} = t_{ij}$ and starts T_{ij2} immediately after T_{ij1} commits, i.e., $t_{ij2} = t_{ij1} + d_{ij1}^{\mathcal{M}^*,\mathcal{E}}$. J_i accesses the same resources as J_i' while T_{i1} is running. Since the time needed for committing and starting is negligibly small, T_{ij2} accesses the same resources at the same time as T_{ij}. T_{ij2} starts with no resources locked, hence J_i accesses a subset of the resources that are accessed by J_i' while T_{ij2} is running. As J_i' does not provoke a conflict, J_i neither does so and T_{ij2} will commit at the same time as T_{ij}. Thus, the system has the same performance with J_i as with J_i'.

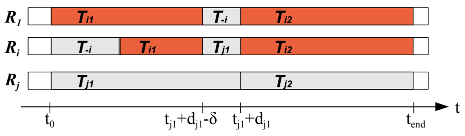

Fig. 5. Example illustrating the proof of Theorem 1.

Now we show how to choose an execution environment $\mathcal{E}_{-i}$ that makes J_i favorable to J_i' (cf. Figure 5). Let $R(T_{ij})$ be the set of all resources which T_{ij} accesses during execution.Let t_0 be the time when J_i and J_i' enter the system. Let $\mathcal{E}_{-i}$ contain a job $J_j = T_{j1}, T_{j2}$ that enters the system at time t_0 as well. Let $T_{j1} = (\{(R_j, 0), (R_i, d_{j1} - \delta)\}, d_{j1})$ and $T_{j2} = (\{(R_j, 0)\}, d_{j2})$ where $R_j \notin R(T_{ij})$ and $R_i \in R(T_{ij1})$. Furthermore, let $d_{j1} = d_{ij}^{\mathcal{E}} + \delta$ and $d_{j2} = d_{ij2}^{\mathcal{E}}$. Let $\mathcal{E}_{-i}$ be s.t. $\mathcal{M}^*$ schedules T_{ij1} at time t_0 and T_{ij2} as well as T_{j2} at time $t_{j1} + d_{j1}$ when the programmer uses J_i. In addition, assume that all resources $R(T_{ij})$ are locked by other jobs in the period $[t_{j1}, t_{j2} + d_{j2}]$ unless T_{j1}, T_{ij1} or T_{ij2} needs them. $\mathcal{M}^*$ completes all jobs at $t_{end} = t_{j2} + d_{j2}$. If the developer decides to not partition T_{ij} and thus she uses J_i' then $\mathcal{M}^*$ can not schedule T_{ij} at time t_0 since both T_{ij} and T_{j1} need R_i in the period $[t_{j1}', t_{j1}' + \delta]$. Note that the only modification to the scheduling of J_j that would not delay the execution is swapping T_{j1} and T_{j2}. However, this is not allowed by J_j's logic. Thus, in order to prevent conflicts, $\mathcal{M}^*$ must either schedule T_{ij} s.t. it commits before T_{j1} accesses R_i or s.t. it accesses R_i after $t_{j1} + d_{j1}$. The former is impossible since J_i' is not available to $\mathcal{M}^*$ before t_0. The latter would force $\mathcal{M}^*$ to delay the execution by at least δ. □

A.2 Proof of Corollary 1

For Timestamp, Karma, Eruption and Polka, the claim follows immediately, since they are quasi-priority-accumulating. For the other CMs we can show that they do not prevent the case where, after a commit of transaction $T_{ij} \in J_i$, the subsequent transaction $T_{i(j+1)} \in J_i$ starts with a lower priority than T_{ij} had at commit time. In the following, we provide counter-examples showing the claim for Timestamp- and Karma-like CMs as well as for Polite and Kindergarten. In addition to proving Corollary 1, this section also illustrates how one can use the framework introduced in Definition 1 to describe priority-based CMs. If we write $\mathcal{X} \rightsquigarrow f$ this means that the described CM reacts to an event $\mathcal{X}$ with the modifications f. Usually, a few such modification rules suffice to define a CM.

Timestamp, Greedy A Timestamp-like manager assigns priorities to transactions according to their time in the system. Older transactions win against newer ones. We can describe Timestamp with the following two modification rules.

$\mathcal{T} \rightsquigarrow \omega_i = \omega_i + d\omega.$
$\mathcal{C} \rightsquigarrow \omega_i = 0.$

Timestamp-like CMs are not GPI compatible since they reset a job's priority when a contained transaction commits. By partitioning a transaction, a job loses its high priority when committing the transaction's first part. Figure 6 depicts Timestamp's partitioning incentive incompatibility.

Karma, Polka Karma and Polka take the number of accessed resources by a transaction as an estimate for the work done. A process keeps its priority after an abort, but not after commit. The difference between the two managers is that Polka uses exponential backoff when transactions encounter contention.

$\mathcal{R}_k \rightsquigarrow \omega_i = \omega_i + \epsilon$ where $\epsilon > 0$.
$\mathcal{C} \rightsquigarrow \omega_i = 0.$

Apart from the fact that a job can cheat, e.g., by creating and accessing many resources and thus gaining high priority, the situation in Figure 7 shows that also partitioning may lead to a longer execution time.

Eruption Eruption is a variant of Karma which adds the losing transaction's priority to the winner's priority.

$\mathcal{R}_k \rightsquigarrow \omega_i = \omega_i + \epsilon$ where $\epsilon > 0$.
$\mathcal{W}$ against $T_j \rightsquigarrow \omega_i = \omega_i + \omega_j.$
$\mathcal{C} \rightsquigarrow \omega_i = 0.$

In addition to accessing dummy resources, a programmer can also exploit Eruption by having several dummy jobs lose against a main job and thus boost the latter's priority (cf. Figure 8).

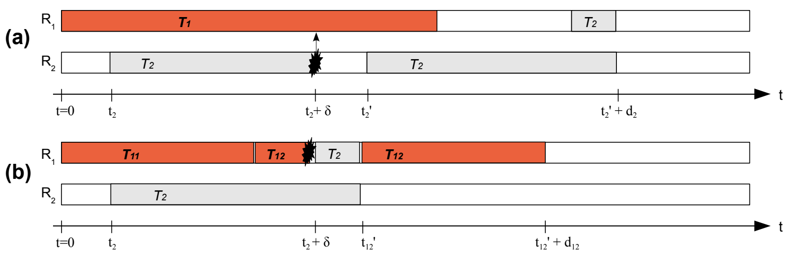

Fig. 6. Timestamp Manager. **(a)** T_1 is not partitioned. **(b)** T_1 is partitioned into T_{11} and T_{12}. A competing transaction T_2 starting at t_2 tries to access R_1 at time $t_2 + \delta$. In **(a)** T_1 wins because it was started before T_2. In **(b)** T_{12} loses because it was started after T_2. Partitioning T_1 is not rewarded.

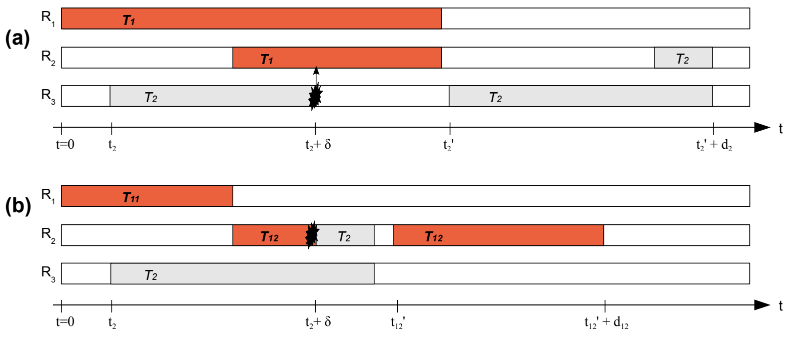

Fig. 7. Karma Manager. **(a)** T_1 is not partitioned. **(b)** T_1 is partitioned into T_{11} and T_{12}. A competing transaction T_2 starting at t_2 tries to access R_1 at time $t_2 + \delta$. In **(a)** T_1 wins because it locks more resources than T_2 at time $t_2 + \delta$. In **(b)** T_{12} loses since it does not lock more resources than T_2 anymore. Partitioning T_1 is not rewarded. Necessary locking is not punished.

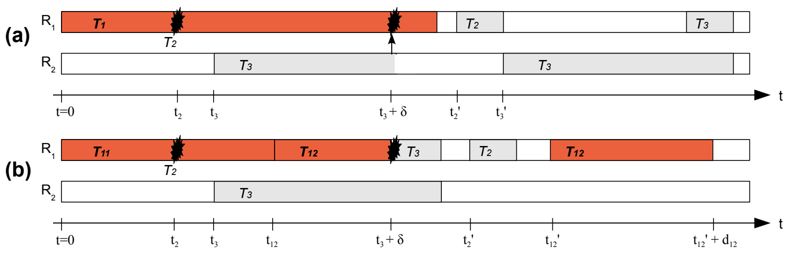

Fig. 8. Eruption Manager. **(a)** T_1 is not partitioned. **(b)** T_1 is partitioned into T_{11} and T_{12}. A competing transaction T_2 starting at t_2 tries to access R_1 at time t_2 and a transaction T_3 at time $t_3 + \delta$. In both **(a)** and **(b)**, T_1, resp. T_{11}, wins against T_2 because it has accessed more resources than T_2. In **(a)**, T_1 also wins against T_3. In **(b)**, however, T_{12} lacks the priority gained in the earlier victory over T_2 and hence loses against T_3. Partitioning T_1 is not rewarded.

Polite With the Polite manager, the transaction that has already been locking a resource R has higher priority than one that tries to newly gain access to R. When a transaction fails getting access of a resource it retries to gain access using exponential backoff. Note that we neglect Polite's killing rule which allows a transaction to kill a competing transaction after a certain number of unsuccessful access attempts. Polite modifies priorities on $\mathcal{R}$-, $\mathcal{A}$- and $\mathcal{C}$-events as follows:

$\mathcal{R}_k \rightsquigarrow \boldsymbol{\omega}_i[k] = \epsilon$ where $\epsilon > 0$.
$\mathcal{A} \rightsquigarrow \boldsymbol{\omega}_i = \mathbf{0}$.
$\mathcal{C} \rightsquigarrow \boldsymbol{\omega}_i = \mathbf{0}$, where $\mathbf{0}$ is a vector of zeros.

The scenario in Figure 9 illustrates that polite does not reward partitioning and it is therefore not GPI compatible.

Kindergarten Kindergarten is the only CM presented here that uses priorities specific to competing jobs. Every job maintains a so called *hit list* of the jobs against which it has previously lost a conflict. At conflict time, a job may abort its competitor if it is in its hit list. This CM is not captured by our framework since we only allow resource specific priorities. One could, however, model Kindergarten with job specific priorities. Let $\boldsymbol{\nu}_i[j]$ denote J_i's job specific priority against J_j. If J_i has a conflict on resource R_k with J_j then it wins if $\boldsymbol{\nu}_i[j] + \boldsymbol{\omega}_i[k] > \boldsymbol{\nu}_j[i] + \boldsymbol{\omega}_j[k]$. In this extended framework, Kindergarten is modeled as follows:

$\mathcal{R}_k \rightsquigarrow \boldsymbol{\omega}_i[k] = \epsilon$ where $0 < \epsilon < 1$.
$\mathcal{A}$ against $T_j \rightsquigarrow \boldsymbol{\nu}_i[j] = 1$ and $\boldsymbol{\omega}_i = \mathbf{0}$.
$\mathcal{C} \rightsquigarrow \boldsymbol{\nu}_i = \boldsymbol{\omega}_i = \mathbf{0}$.

Kindergarten does not seem to be exploited easily by selfish programmers. Nevertheless, it suffers the same problem as any quasi-priority-accumulating CM, namely, that a job loses its built up priority on commit, i.e., the hitlist is cleared. Thus, partitioning is not rewarded. □

A.3 Proof of Lemma 3

In a first step, we show that Randomized, denoted by $\mathcal{M}$, incentivizes partitioning. Let T_{ij} be partitionable in T_{ij1} and T_{ij2}. Let J_i' be a job with T_{ij} and J_i the same process except it uses T_{ij1} and T_{ij2} instead of T_{ij}. We compare J_i with J_i' under all possible environments $\mathcal{E}_{-i}$. Let $\mathcal{E}_{-i}$ include the CM's randomized decisions.

In case (A) T_{ij} runs until commit, T_{ij1} and T_{ij2} run until commit in the same time, since we assume that the time needed for committing and restarting is negligible. J_i and J_i' provoke the exact same conflicts and thus $\mathcal{M}$ always decides in favor of J_i, or J_i' respectively. In case (B) T_{ij} is aborted in the period $[t_{ij2}, t_{ij} + d_{ij}^{\mathcal{E}}]$, T_{ij1} runs until commit, T_{ij2} is started immediately and is aborted in the same conflict as T_{ij}. T_{ij} and T_{ij2} are restarted. Let t_a be the time when T_{ij} or T_{ij2} respectively are aborted. Let t_{ij2}' be the time when J_i' has successfully completed all operations corresponding to T_{ij1} after the restart. Employing J_i' instead of J_i coincides with delaying T_{ij} from t_a until t_{ij2}'. In order to show that Randomized rewards partitioning we can use the

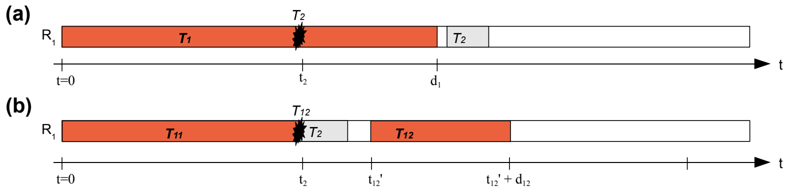

Fig. 9. Polite Manager. **(a)** T_1 is not partitioned. **(b)** T_1 is partitioned into T_{11} and T_{12}. A competing transaction T_2 tries to access R_1 at time t_2. In **(a)** T_1 wins because it has already locked R_1. In **(b)** T_{12} and T_2 try to access R_1 at the same time. J_1 does not have higher priority on R_1 than J_2 and possibly loses. Thus, partitioning T_1 is not rational.

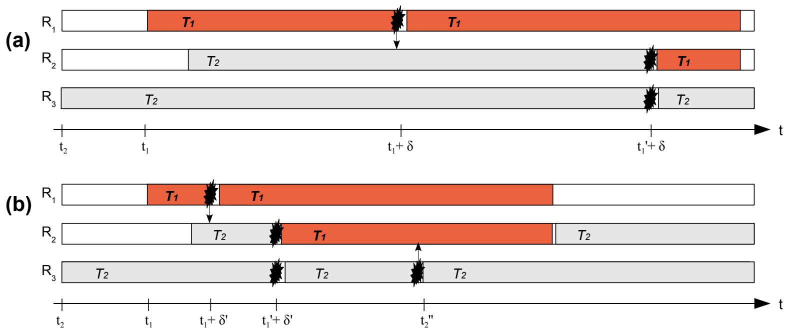

Fig. 10. Kindergarten Manager. **(a)** T_1 is partitioned optimally. **(b)** T_1 accesses R_2 earlier than the program logic requires, i.e., after a period of δ' instead of δ where $\delta > \delta'$. In **(a)** T_1 aborts after $t_1 + \delta$ and adds T_2 to its hitlist. After a restart and another period of δ, T_2 is still locking R_2, but this time T_1 may abort T_2. In **(b)**, T_1 aborts and restarts much quicker and thus wins against T_2 earlier, i.e., at time $t_1 + 2\delta'$. The programmer of T_1 can achieve a better execution time with the unnecessary lock. Therefore Kindergarten does not punish unnecessary locking.

same argument from Lemma 1, namely that starting immediately is the better strategy than waiting, although Randomized is not a priority-accumulating CM. To show this for Randomized is much easier. An adversary can provoke the same conflicts for a transaction, if it is started immediately or if it is delayed for some time Δ. Since in any conflict, the probability of winning is the same, the expected runtime increases by Δ when the transaction is delayed. In case (C) T_{ij} is aborted before $t_{ij} + d^{\mathcal{E}}_{ij1}$, T_{ij1} has the same conflict and is aborted as well. They are both restarted at the same time and we can recurse the argument until case (A) or (B) occur.

In a second step, we show that $\mathcal{M}$ punishes unnecessary locking. Let T'_{ij} be a transaction with an unnecessary lock l. Let T_i be the same transaction without l and $d^{\mathcal{E}}_{ij} = d'^{\mathcal{E}}_{ij}$ where $d'^{\mathcal{E}}_{ij}$ is the contention-free runtime of T'_{ij} in the environment $\mathcal{E}$. l either does not provoke a conflict or it does. If it does not, choosing T_{ij} or T'_{ij} results in the same execution time. If l provokes a conflict and $\mathcal{M}$ decides in favor of T'_{ij} then l does not change the course of T_{ij}'s execution either. If $\mathcal{M}$, however, decides against T'_{ij} it is aborted and T_{ij} continues. T'_{ij} is restarted immediately. If T_{ij} runs until commit, playing T_{ij} yields a better execution time. Otherwise, let t^{last} be the time when T'_{ij} is restarted for the last time, i.e., T'_{ij} commits at time $t^{last} + d'^{\mathcal{E}}_{ij}$. T_{ij} could also be scheduled at t^{last} and reach a commit time at least as good as T'_{ij}. But this is the same as if the programmer would let T_{ij} wait from the abort of T'_i until t^{last}. Thus, employing T'_{ij} instead of T_{ij} coincides with delaying and it follows that unnecessarily locking is irrational.

It remains to show that if $d^{\mathcal{E}}_{ij} < d'^{\mathcal{E}}_{ij}$ then T_{ij} is still preferable. Let T'_{ij} be exactly like T_{ij} except for one lock l at time t_l which prolongates $d'^{\mathcal{E}}_{ij}$ by Δ. If l does not provoke a conflict, l has the same effect as waiting in the period of $[t_l, t_l + \Delta]$ which we showed to be irrational. In case l provokes a conflict and $\mathcal{M}$ decides for T_{ij}, this again coincides with waiting. If T'_{ij} is aborted, though, and T_{ij} runs until commit, playing T_{ij} yields a better execution time. If T_{ij} does not run until commit in its first execution, let t^{last} be the time when T'_{ij} is restarted for the last time, i.e., T'_{ij} commits at time $t^{last} + d'^{\mathcal{E}}_{ij}$. T_{ij} could also be scheduled at $t^{last} + d'^{\mathcal{E}}_{ij} - d^{\mathcal{E}}_{ij}$ and reach a commit time at least as good as T'_{ij} because T_{ij} would provoke a subset of the conflicts that T'_{ij} provokes and as $\mathcal{M}$ would make the same decisions, T_{ij} would also win all conflicts. Unnecessary locking coincides again with delaying T_{ij} from the abort until $t^{last} + d'^{\mathcal{E}}_{ij} - d^{\mathcal{E}}_{ij}$ and not locking coincides with starting T_{ij} immediately after abort. Hence unnecessary locking is irrational. □

B Appendix: Simulations

To verify our theoretical insights, we implemented free-riders in DSTM2 [5], a software transactional memory system in Java, and let them compete with the threads originally provided by the authors of the included benchmark.

B.1 Setup

In particular, we added a subclass `TestThreadFree` to `dstm2.benchmark.IntSetBenchmark` that uses coarse transaction granularities, i.e., instead of just updating one resource, a free-rider updates several resources per transaction at once. See Figure 11 for the code executed by the free-rider and

```
while (true) {
 thread.doIt(new Callable<void>() {

   @Override
   public void call()

      // access dummy resource <priority> times
     Factory<INode> factory = Thread.makeFactory(INode.class);
     INode nd = factory.create();
     for(int k=0; k < priority; k++){
       nd.setValue(k);<

     // access shared resource <granularity> times
     Random random = new Random(System.currentTimeMillis());
     for(int i=0; i < granularity; i++){
       intSet.update(random.nextInt(TRANSACTION_RANGE))
     }
   }
 });
}
```

Fig. 11. Free-riding thread. The call() method is executed as a transaction by the STM.

```
while (true) {
 value = random.nextInt(TRANSACTION_RANGE);
   thread.doIt(new Callable<void>() {

   @Override
   public void call() {
     intSet.update(value);
    }
   });
}
```

Fig. 12. "Good" thread. The call() method consists of only one update call.

Figure 12 for the collaborative threads' code. The latter is what we call "good code", as it only performs one action per transaction and thus avoids unnecessary locking. We added a mechanism to the free-rider that attempts to build up priority before accessing the shared resource. To this end, it simply creates a dummy resource and updates it a number of times. When the system is managed by Timestamp- or Karma-like contention managers this could be an advantage as priority is built up in a conflict-safe environment and once it accesses the truly shared resources, it has higher priority than most of its competitiors. Hence the free-rider can vary two parameters, the transaction granularity γ and the priority π it tries to build up before actually starting its work.

We tested and compared the performance of free-riding threads with collaborative threads with two benchmarks. In both, there is a total number of 16 threads which start using a shared data structure for 10 seconds, before they are all stopped. In the first benchmark, the threads all work on one shared ordered list data structure, in the second, they work on a red-black tree data structure. All operations are update operations, i.e., a thread either adds or removes an element. We ran various configurations of the scenario in both benchmarks managed by the Polite, Karma, Polka, Timestamp or the Randomized contention manager. The variable parameters were the number of free-riders (0, 1, 8, 16) among the 16 threads, their transaction granularity $\gamma \in \{1, 20, 50, 100, 500, 1k, 5k, 10k, 50k, 100k, 500k, 1M\}$ and the number of initial dummy accesses $\pi \in \{0, 200, 500, 2000\}$ performed by the free-riders. The benchmarks were executed on a machine with 16 cores, namely 4 Quad-Core Opteron 8350 processors running at a speed of 2 GHz. The DSTM2.1 Java library was compiled with Sun's Java 1.6 HotSpot JVM. To get accurate results every benchmark was run five times with the same configuration. The presented results are averaged across the five runs.

B.2 Results

The results confirm the theoretical predictions that a free-rider can outperform and sometimes almost entirely deprive the collaborative threads of access to the shared resources if the TM system is managed by the Polite, Karma, Polka or the Timestamp CM. With the Randomized manager on the other hand, the collaborative threads are much better off than the "free-riders"(cf. Figure 13). In all of our tests, if the system was managed by Polite, the free-riders were always better off. Under Karma, they were better off in 92% of all cases and if they used granularities γ of at least 20 operations per transaction, they always performed better. With Polka, the free-rider success rate was 70% over all runs and 100% for $\gamma \in \{20, 50, 100\}$. Of all tests run with the Timestamp manager, free-riding paid off in 92% of the cases and in 100% if the granularity γ was at least 20. Under Randomized, free-riders had a larger throughput in only 7% of all cases.

Further, our simulations suggest that the mechanism included to boost priority π before actually accessing the shared data does not influence the free-rider's relative performance significantly. The transaction granularity however has a huge impact. Figure 14 shows the average throughput of both a free-rider and a collaborative thread. In our experiments, a free-riding thread's throughput was practically always higher than the collaborators' under the Karma, Polka and the Timestamp manager if it used a granularity of at least twenty update operations per transaction. This may in part be because a coarser transaction needs less overhead than a transaction with granularity $\gamma = 1$, however, with the Randomized contention manager, we see that even a transaction with a granularity of only twenty updates is unlikely to succeed. To a larger extent, this higher performance of the free-riders derives from the fact that—except for the first update—they have higher priority than the collaborative threads. At first, it might be surprising that the average throughput, i.e., the system efficiency, does not decrease when introducing more free-riders[9]. However, with large granularities, there will usually be one transaction with very high priority. The latter is not endangered of being aborted by any other transaction and hence runs to commit untouched. It seems that in our setting

[9] except for the Randomized manager, where free-riders have a very low chance of success, independent of whether the competitors are collaborators or free-riders themselves.

with high contention, one fast free-rider locking the entire datastructure is still quite efficient. With lower contention, this would obviously not hold. Note also the break in the throughput increase between $\gamma = 1000$ and $\gamma = 5000$ under the Polka manager. This is probably caused by the mechanism included in Polka which allows a transaction trying to access a locked resource to abort the competitor after a certain number of unsuccessful access attempts. This seems to happen much more often if the free-riders use granularities higher than 1000.

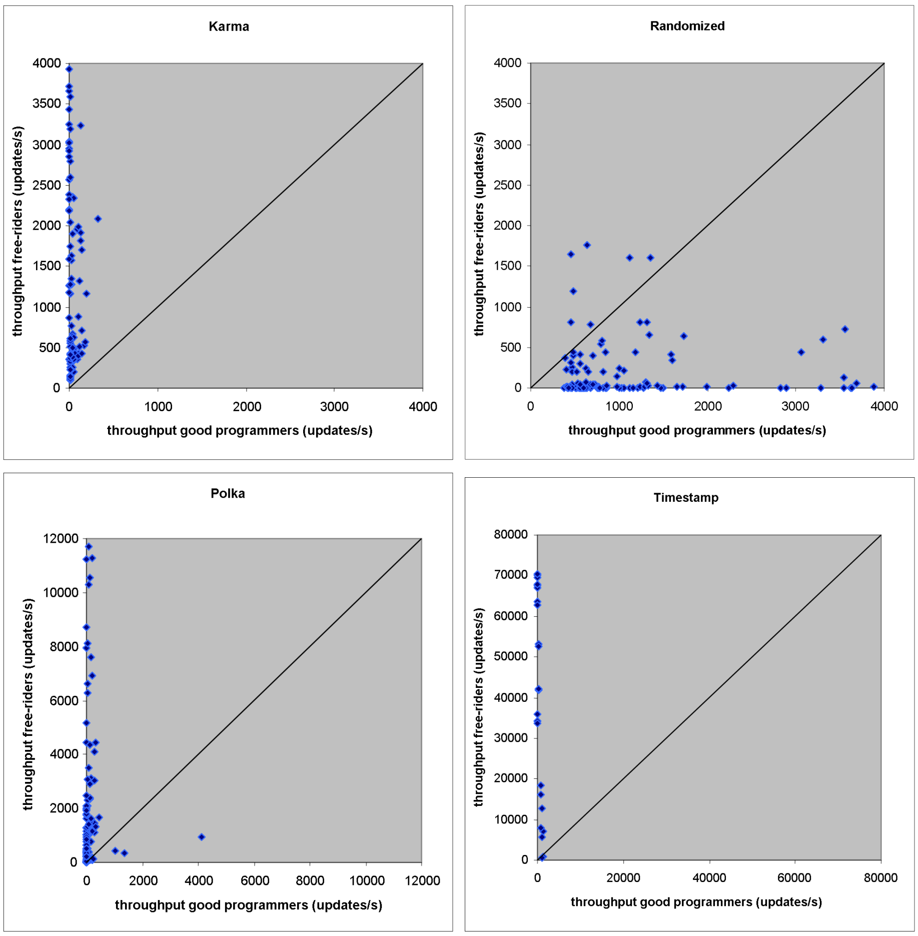

Fig. 13. Plot of all cases simulated under a Karma, a Randomized, a Polka and a Timestamp CM. If a point is above the diagonal line this indicates that in the corresponding test run, a free-rider had a larger throuhput than a good programmer who only employs transactions of granularity 1. For Karma, the cases where $\gamma = 1$ are omitted.

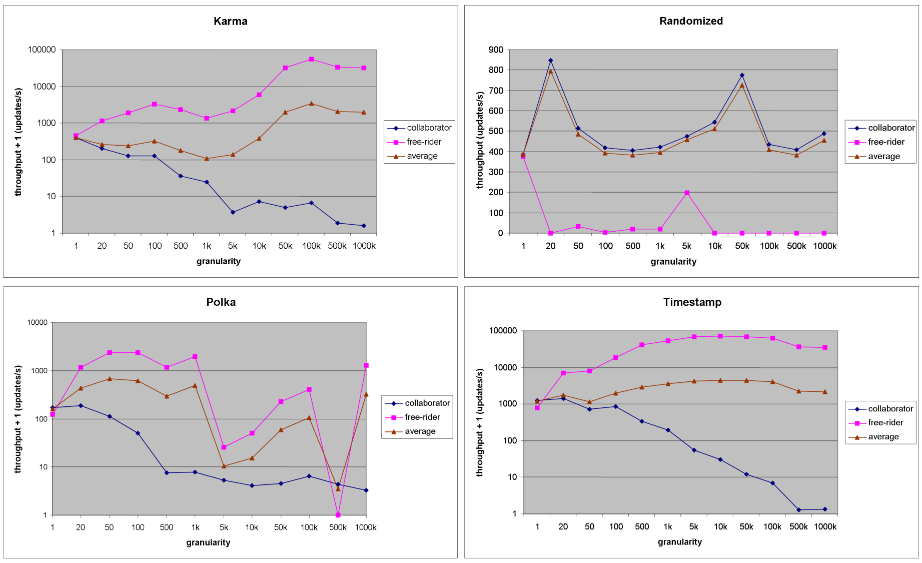

Fig. 14. Average throughput of a free-riding and a collaborative thread in the red-black tree benchmark with 15 collaborators and one free-rider. The free-rider does not employ a priority boosting mechanism ($\pi = 0$). In addition to the collaborators' and the free-rider's throughput, the average throughput of all 16 concurrent threads is depicted. Except for Randomized, we added 1 to the actual throughput and used a logarithmic scale.