Good Programming in Transactional Memory[☆]
Game Theory Meets Multicore Architecture

Raphael Eidenbenz, Roger Wattenhofer

Computer Engineering and Networks Laboratory (TIK), ETH Zurich, CH-8092 Zürich, Switzerland

Abstract

In a multicore transactional memory (TM) system, concurrent execution threads interact and interfere with each other through shared memory. The less interference a thread 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 should coincide with writing code that optimizes an individual thread'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.

Keywords: 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), introduced by Lomet [1] in the 1970's and implemented by Herlihy and Moss [2] 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 shared resources take effect immediately. If another thread, 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

[☆] An earlier version appeared in the Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC), Honolulu, Hawaii, USA, Springer LNCS 5878, pp. 503–513, December 2009.

Email addresses: eidenbenz@tik.ee.ethz.ch (Raphael Eidenbenz), wattenhofer@tik.ee.ethz.ch (Roger Wattenhofer)

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

```
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; }}
```

Figure 1: Two variants of updating each node in a ring.

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.

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. On the contrary, 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

[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

[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.

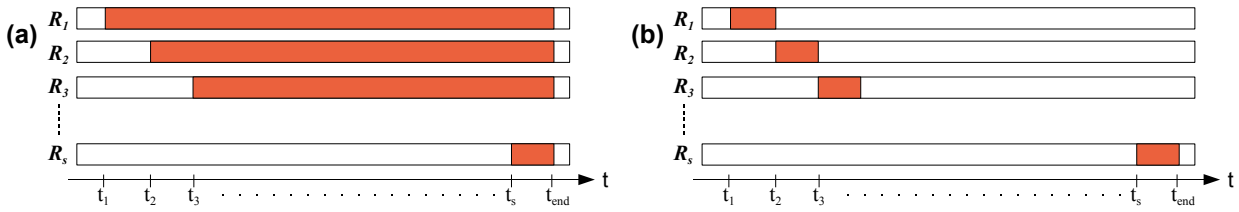

Figure 2: Transactional allocation of ring nodes **(a)** by `incRingCounters` and **(b)** by `incRingCountersGP`.

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). As a practical proof of our findings, we implemented selfish strategies in the TM library DSTM2[3] and tested them in several scenarios. The results are presented in Section 6.

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 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.

The *environment* $\mathcal{E}$ is a set of n tuples of a job and the time it enters the system, i.e., $\mathcal{E} = \{(J_0, t_0), (J_1, t_1), \ldots, (J_n, t_n)\}$. We assume that there are $m \geq n$ machines, and each job is executed exclusively on one machine. The *execution environment* of a job J_i is given by $\mathcal{E}_{-i} = \mathcal{E} \setminus \{(J_i, t_i)\}$. Each job J_i consists of a sequence of *transactions* $T_{i1}, T_{i2}, \ldots, T_{i|J_i|}$, where $|J_i|$ is the number of transactions contained in J_i. Transactions may access shared *resources* $\mathcal{R}$. For the sake of simplicity, we consider all accesses as exclusive,[4] thus, if two transactions both try to access resource $R \in \mathcal{R}$ at the same time, or if one has already locked R and the other desires access to R 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 $T_{i|J_i|}$ has not committed yet. Notice that in optimistic contention management, the starting time t_i of a job J_i is not influenced by the CM, since it only reacts once a conflict occurs. We assume that any transaction T_{ij} contained in job J_i accesses the same subset of resources $\mathcal{R}_{ij} \subseteq \mathcal{R}$ in each of its runs independently of J_i's starting time t_i, and for any resource the time of its first access after a (re)start of T_{ij} remains the same in each run.[6] This allows a description of a contained transaction by a 3-tuple $T_{ij} = (\mathcal{R}_{ij}, \tau_{ij}, d_{ij})$ where $\mathcal{R}_{ij} \subseteq \mathcal{R}$ are the resources accessed by T_{ij}, $\tau_{ij} : \mathcal{R}_{ij} \mapsto \mathbb{R}^+$ is a function that maps a resource to its relative access time, and d_{ij} is the *contention-free duration* of T_{ij}, i.e., the time needed from start to commit provided that T_{ij} encounters no conflicts. For instance $T_{ij} = (\{R_1, R_4\}, \{R_1 \mapsto 3, R_4 \mapsto 0\}, 4)$ describes a transaction that tries to gain immediate access of R_4, access of R_1 after 3 time units, and commits after 4 time units unless it was aborted before. Note that $d_{ij} > \tau_{ij}(R)\ \forall\ R \in \mathcal{R}_{ij}$. Let d_i denote the contention free duration of job J_i, i.e. the time needed from t_i to the commit time of $T_{i|J_i|}$ in an empty execution environment.

[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.

[6] Note that this is a major simplification of a real shared memory system, where datastructures change dynamically. However, as we assume code developers to consider worst case environments, only the starting time relative to competing jobs, but not the absolute starting time t_i is relevant. All other jobs could just be shifted accordingly. Thus our assumption relaxes to the assumption that resource accesses remain constant after a restart.

If the CM $\mathcal{M}$ used in a TM system is deterministic we assume that the state of the system at a certain time is determined by $\mathcal{E}$ and $\mathcal{M}$. If $\mathcal{M}$ takes randomized decisions then $\mathcal{E}$ and $\mathcal{M}$ determine a system state probability distribution at any given time. Thus, given $\mathcal{M}$ and $\mathcal{E}$, the execution of $\mathcal{E}$ is thoroughly described. In the following definitions, we presume $\mathcal{M}$ to be deterministic. Corresponding definitions for randomized $\mathcal{M}$ are straightforward by incorporating probability distributions, and we omit explicit definitions for randomized CMs. By $d^{\mathcal{M},\mathcal{E}}$ we denote the function that maps jobs and transactions in $\mathcal{E}$ to their *execution time*, i.e., $d^{\mathcal{M},\mathcal{E}}(T_{ij})$ is the time from the first start of transaction T_{ij} to its eventual commit in an execution of $\mathcal{E}$ by a TM system managed by $\mathcal{M}$, $d^{\mathcal{M},\mathcal{E}}(J_i)$ is the time the same TM system takes executing job J_i where $(J_i, t_i) \in \mathcal{E}$, i.e., the time between t_i and the commit time of the last transaction in J_i. The makespan $d^{\mathcal{M},\mathcal{E}}$ of an environment $\mathcal{E}$ in a system managed by $\mathcal{M}$ is the time from $\min_i t_i$ until $\max_i\{t_i + d^{\mathcal{M},\mathcal{E}}(T_i)\}$. Let $t^{\mathcal{M},\mathcal{E}}$ denote the function that maps transactions in $\mathcal{E}$ to their start time, i.e., $t^{\mathcal{M},\mathcal{E}}(T_{ij})$ is the time when T_{ij} is started in the execution of $\mathcal{E}$ by a TM system with CM $\mathcal{M}$. We denote by $\mathcal{L}^{\mathcal{M},\mathcal{E}}(t)$ the set of locked resources at time t in a TM system managed by $\mathcal{M}$ when executing environment $\mathcal{E}$. Similar to $\mathcal{R}_{ij}$, we denote by $\mathcal{R}_i$ the set of resources accessed by job J_i, i.e., $\mathcal{R}_i = \bigcup_{j=1}^{|J_i|} \mathcal{R}_{ij}$. We define the *concatenation* $T_{ij||ij+1}$ of two consecutive transactions T_{ij} and T_{ij+1} as $T_{ij||ij+1} = \left(\mathcal{R}_{ij} \cup \mathcal{R}_{ij+1}, \tau_{ij||ij+1}, d_{ij} + d_{ij+1}\right)$, where $\tau_{ij||ij+1}$ is the function that maps a resources $R \in \mathcal{R}_{ij} \cup \mathcal{R}_{ij+1}$ to $\tau_{ij}(R)$ if $R \in \mathcal{R}_{ij}$, and to $d_{ij} + \tau_{ij+1}(R)$ otherwise. For a job J_i, and an integer $k \in [1, |J_i| - 1]$ we define $Combine(J_i, k)$ to be the job that results when the two transactions T_{ik}, and T_{ik+1} contained in J_i are concatenated to $T_{ik||ik+1}$, i.e. $Combine(J_i, k) = T_{i1}, \ldots, T_{ik-1}, T_{ik||ik+1}, T_{ik+2}, \ldots, T_{i|J_i|}$. In our discussions, we sometimes compare a job J_i to a similar job J_i'. In such comparisions, we add a dash to notations associated with jobs to indicate the corresponding properties of J_i' rather than of J_i. E.g., $\mathcal{R}_i'$ denotes the resources accessed by J_i'.

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, the author of job J_i minimizes J_i's expected execution time. We presume programmers have no information on the runtime execution environment $\mathcal{E}_{-i}$, and are thus generally uncertain about the performance of their job. As to deal with this uncertainty, we assume developers act *risk-averse* in the sense that they expect $\mathcal{E}_{-i}$ to be such that J_i's execution time is maximal among all possible finite executions, i.e., the expected running time of job J_i in a TM system managed by $\mathcal{M}$ is $\tilde{d}^{\mathcal{M}}(J_i) = \max_{\{\mathcal{E}_{-i} | d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}}(J_i) \text{ is finite}\}} d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}}(J_i)$. Note that with many CMs the "true" worst case execution time of J_i is infinite even for finite environments $\mathcal{E}_{-i}$. If, however, a risk-averse developer would expect her job to run forever she could just as well twirl her thumbs instead of writing a piece of code. Hence, the assumption that a job eventually terminates is an inevitable feature of our programmer model. Furthermore, we say a job J_i *dominates* J_i' *under* $\mathcal{M}$ if and only if it holds for any $\mathcal{E}_{-i}$ that $d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}}(J_i) \leq d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i',t_i)\}}(J_i)$ and $\exists\ \mathcal{E}_{-i}$ such that $d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}}(J_i) < d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i',t_i)\}}(J_i')$. When implementing a task, a programmer can typically choose among a variety of jobs that all implement the desired logic. We assume that out of these available choices the programmer opts for any non-dominated job J_i that has minimal expected running time $\tilde{d}^{\mathcal{M}}(J_i) = \min_{J_i'} \tilde{d}^{\mathcal{M}}(J_i')$. We call these minimal jobs the *solution set*.[7]

3. Good Programming Incentives (GPI)

A first step towards incentive compatible transactional memory is to determine what programmer behavior is desirable for a TM system. For that matter we investigate 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 the load thereby put on the system is unnecessary.

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

[7] Note that this solution concept is idealized in the sense that it is probably infeasible for many tasks to find a job with minimal expected running time. We therefore typically derive statements that preclude certain types of jobs from the solution set rather than statements about what jobs are in the solution set. For instance, we show in Lemma 8 that jobs that contain artificial delays are not in the solution set given that the contention manager is priority-based.

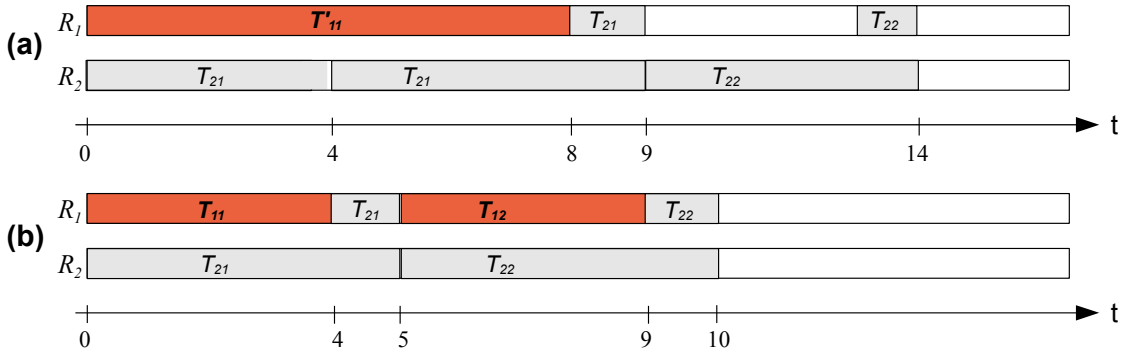

Figure 3: Partitioning example. The picture depicts the optimal allocation of two resources R_1 and R_2 over time in two situations **(a)** and **(b)**. In **(a)**, the programmer of job J_1 does not partition T'_{11}. In **(b)**, she partitions T'_{11} into T_{11} and T_{12}. The makespan is shorter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

However, the question remains whether *partitioning* a transaction into smaller transactions—even if doing so 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 8 time units. One strategy for the programmer is to wrap all operations on R_1 into one transaction $T'_{11} = (\{R_1\}, \{R_1 \mapsto 0\}, 8)$. However, let the semantics also allow an execution of the code in two subsequent transactions T_{11} and T_{12} where $T_{11} = T_{12} = (\{R_1\}, \{R_1 \mapsto 0\}, 4)$ without losing consistency. Figure 3 shows the optimal execution of both strategic variants in an environment $\mathcal{E} = \{(J_1, 0), (J_2, 0)\}$ where $J_2 = T_{21}, T_{22}$ and $T_{21} = T_{22} = (\{R_2, R_1\}, \{R_1 \mapsto 4 + \epsilon, R_2 \mapsto 0\}, 5)$ where $\epsilon \in \mathbb{R}^+$ is arbitrarily small, i.e. one clock cycle. In situation **(a)**, the programmer does not partition T'_{11}. Both jobs J_1 and J_2 start at time $t = 0$, after 4 time units there is a conflict since transaction T_{21} tries to gain access of resource R_1 that is locked by T'_{11}. To achieve an optimal allocation the contention manager $\mathcal{M}$ aborts T_{21}. T_{21} is restarted. No more conflicts occur, and a makespan of $d^{\mathcal{M},\mathcal{E}} = 14$ is achieved. Convince yourself that this is minimal for $\mathcal{E}$. In situation **(b)**, the programmer uses the partitioned version of J_1. Both jobs start at $t = 0$. T_{11} commits after 4 time units. In the period $(4, 5]$, T_{12} and T_{21} continuously compete for resource R_1. The optimal CM lets T_{21} run to commit. Transactions T_{12} and T_{22} both start at $t = 5 + \epsilon$, and run to commit without conflicts. This yields a makespan of 10.

Thus, in the example of Figure 3, partitioning T'_{11} allows to execute J_1 and J_2 four time units faster. We can show that partitioning is beneficial to a TM system in that it provides more flexibility to the allocation schedule. To make this fact clear we consider a TM system that is managed by an optimal offline CM $\mathcal{M}^*$. In contrast to the CMs in a TM system, $\mathcal{M}^*$ is assumed to know the entire environment, including the jobs that arrive in the future, and can thus precompute what runtime decisions lead to a minimal makespan. Hence, $\mathcal{M}^*$ always makes the right decision when resolving a conflict, furthermore, we allow it to postpone the beginning of a transaction T_{ij} to any optimal time t given that $t \geq t_i$ and all T_{ik} with $k < j$ have committed.

Theorem 2. *A finer transaction granularity speeds up a transactional memory system managed by an optimal CM $\mathcal{M}^*$, i.e., for any two jobs J_i, J'_i where $\exists\, k \in \{1, \ldots, |J_i| - 1\}$ such that $J'_i = Combine(J_i, k)$ it holds that $\forall\, \mathcal{E}_{-i}$: $d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J_i,t_i)\}} \leq d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J'_i,t_i)\}}$ and $\exists\, \mathcal{E}_{-i}$ such that inequality holds.*

Proof. First notice that we may assume w.l.o.g. that under $\mathcal{M}^*$ there are no conflicts: Any transaction T_{ij} will finally run from start to commit in an optimal execution of an environment $\mathcal{E}$. Let $t_{ij}^{\mathcal{M},\mathcal{E}}$ denote the time when transaction T_{ij} is started for its successful run in the execution of $\mathcal{E}$ under CM $\mathcal{M}$. If CM $\mathcal{M}^{**}$ manages $\mathcal{E}$ optimally then the CM $\mathcal{M}^*$ that works like $\mathcal{M}^{**}$, except it postpones the start of each transaction T_{ij} until $t_{ij}^{\mathcal{M}^{**},\mathcal{E}}$, manages $\mathcal{E}$ optimally as well. Moreover, since a thus $\mathcal{M}^*$ starts any transaction only when it will run until commit the produced allocation schedule has no conflicts.[8]

We proceed by showing the existence of a CM $\mathcal{B}$ that achieves $d^{\mathcal{B},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}} = d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J'_i,t_i)\}}$ for any given $\mathcal{E}_{-i}$. For convenience, let $E := \mathcal{E}_{-i} \cup \{(J_i, t_i)\}$, and $E' := \mathcal{E}_{-i} \cup \{(J'_i, t_i)\}$. $\mathcal{B}$ sets $t^{\mathcal{B},E}(T_{ik}) = t^{\mathcal{M}^*,E'}(T'_{ik})$ where $t'_{ik} = T_{ik\|ik+1}$

[8] This reflects the fact that an offline CM is able to "look into the future", and thus, it can avoid mistakes.

and starts T_{ik+1} immediately after T_{ik} commits, i.e., $t^{\mathcal{B},E}(T_{ik+1}) = t^{\mathcal{B},E}(T_{ik}) + d^{\mathcal{B},E}(T_{ik})$. At any time t, $t^{\mathcal{B},E}(T_{ik}) \leq t \leq t^{\mathcal{B},E}(T_{ik}) + d^{\mathcal{B},E}(T_{ik})$, J_i accesses the same resources as J'_i, i.e., $\mathcal{L}^{\mathcal{B},E}(t) = \mathcal{L}^{\mathcal{M}^*,E'}(t)$. Since the time needed for committing and starting is negligibly small, T_{ik+1} accesses the same resources as T'_{ik} at the same time. Furthermore, when T_{ik+1} starts it has no resources locked. Hence the resources locked by T_{ik+1} are always a subset of the resources accessed by T'_{ik}, i.e., $\mathcal{L}^{\mathcal{B},E}(t) \subseteq \mathcal{L}^{\mathcal{M}^*,E'}(t)$ where $t^{\mathcal{B},E}(T_{ik+1}) \leq t \leq t^{\mathcal{B},E}(T_{ik+1}) + d^{\mathcal{B},E}(T_{ik+1})$. Note that T_{ij} might have some resources locked from earlier accesses at time $t^{\mathcal{B},E}(T_{ik+1})$. As J'_i does not provoke a conflict J_i neither does so, and T_{ik+1} will commit at the same time as T'_{ik}. $\mathcal{B}$ executes any other transaction T_{ij} with $j \notin \{k, k+1\}$ just like $\mathcal{M}^*$, and the claim about $\mathcal{B}$'s performance follows. Since $\mathcal{M}^*$ is optimal we have $d^{\mathcal{M}^*,E} \leq d^{\mathcal{B},E} = d^{\mathcal{M}^*,E'}$.

Now we describe an execution environment $\mathcal{E}_{-i}$ with the property that $d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J_i,t_i)\}} < d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J'_i,t_i)\}}$. Let $\mathcal{E}_{-i} = \{(J_v, t_v)\}$ with $J_v = T_{v1}, T_{v2}$ and $t_v = t_i$. Let $T_{v1} = (\{R_v, R\}, \{R_v \mapsto 0, R \mapsto t^{\mathcal{M}^*,\{(J_i,0)\}}(T_{ik+1})\}, t^{\mathcal{M}^*,\{(J_i,0)\}}(T_{ik+1}) + d_{ik+1})$, and $T_{v2} = (\{R_v\}, \{R_v \mapsto 0\}, d_i - t^{\mathcal{M}^*,\{(J_i,0)\}}(T_{ik+1}) - d_{ik+1} + \delta)$ where $R_v \notin \mathcal{R}_i$, R is any resource in $\mathcal{R}_{ik}$, and δ is the amount of time that R is locked in a successful run of T_{ik+1}. $\mathcal{M}^*$ achieves an optimal execution of $\mathcal{E}_{-i} \cup \{(J_i, t_i)\}$ by starting both jobs J_i and J_v at t_i, and delaying the start of T_{ik+1} by δ after T_{ik} commits. Thus, all transactions run conflict free to commit yielding a makespan of $d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J_i,t_i)\}} = t_i + d_i + \delta$. Note that δ may equal 0, namely if $R \notin \mathcal{R}_{ik+1}$.

If the developer uses J'_i instead of J_i, in order not to provoke a conflict on resource R, $\mathcal{M}^*$ has to either postpone T_{v1} by d_{ik+1}, or postpone T'_{ik} by $d_{ik+1} + \delta'$ where $\delta' > 0$ is the period of time that R is locked in a successful run of T_{ik}. The former yields a makespan of

$$d^{\mathcal{M}^*,E'} = t_i + d_v + d_{ik+1} = t_i + d_i + \delta + d_{ik+1} > d^{\mathcal{M}^*,E},$$

the latter yields

$$d^{\mathcal{M}^*,E'} = t_i + d_v + \delta' = t_i + d_i + \delta + \delta' > d^{\mathcal{M}^*,E}.$$

For both inequalities we used the fact that $d_v = d_i + \delta$, which holds due to the construction of $\mathcal{E}_{-i}$. □

Theorem 2 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 in a selfish environment even with the additional overhead needed for incentive compatibility.

Our investigations show that both, avoiding unnecessary locks, and partitioning transactions whenever possible, are behavioral patterns that are beneficial to a TM system. In the following, we define the properties of a CM that incentivize code developers to adopt this behavior. We say a CM rewards partitioning iff it is rational for a programmer to always partition a transaction when the program logic allows her to do so, and it punishes unnecessary locking iff it is rational for a programmer to never lock resources unnecessarily.

Definition 3. *A CM* $\mathcal{M}$ rewards partitioning *iff for any two jobs* J_i, J'_i *where* $\exists\, k \in \{1, \ldots, |J_i| - 1\}$ *such that* $J'_i = Combine(J_i, k)$ *it is rational for a programmer to opt for* J_i *rather than* J'_i *given that both jobs implement the desired task.*

Definition 4. *Let* J_i, J'_i *be any two jobs with the property that for any point in time* t *it holds that* $\mathcal{L}^{\mathcal{M},\{(J_i,0)\}}(t) \subseteq \mathcal{L}^{\mathcal{M},\{(J'_i,0)\}}(t)$*, and for at least one* t *it holds that* $\mathcal{L}^{\mathcal{M},\{(J_i,0)\}}(t) \subset \mathcal{L}^{\mathcal{M},\{(J'_i,0)\}}(t)$*. A CM* $\mathcal{M}$ punishes unnecessary locking *iff for any such pair* J_i, J'_i*, it is rational for a programmer to opt for* J_i *rather than* J'_i *given that both jobs implement the desired task.*

Definition 5. *A CM is* good programming incentive (GPI) compatible *iff it rewards partitioning and punishes unnecessary locking.*

Note that by our definitions we achieve that if a job J'_i can be further improved in a TM system managed by a GPI compatible CM, i.e., if it can be further paritioned or shortened in terms of locks, a programmer has an incentive to choose an improved job J_i. Note that if J_i itself can be further improved then it will not be chosen by the programmer either, but the improvement to J_i, and so forth. Consequently, GPI compatibility incentivizes programmers to choose job implementations that cannot be further partitioned, or shortened in terms of locks (without losing consistency). However, there might still be faster jobs that implement the same task in a way that substantially differs from J_i or its improvements. For instance, mere GPI compatibility does not indicate whether it is faster to sort a shared list by

employing a merge sort, or a bubble sort algorithm. Generally, we cannot expect any CM to be able to tell whether a job implements the task desired by the programmer, nor whether the algorithm implemented solves a given task elegantly, nor whether the code makes sense at all. Therefore, in order to be GPI compatible a CM must typically make all J_i perform better than J'_i for any job pairs defined as in Definitions 3 and 4 regardless of the semantics. In that sense, GPI compatibility describes a monotonicity property, namely that a job J_i that is lighter, or finer grained than a job J'_i is guaranteed to perform at least as well as J'_i.

Let us reconsider the example from Figure 3 to illustrate that GPI compatibility is not a naturally given property. We have seen that partitioning T'_{11} 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'_{11}, J_1 terminates at time $t = 8$. In the partitioned case, however, J_1 terminates at time $t = 9$. 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? Clearly, selfish developers' incentives are not naturally aligned with the TM system designer's goal to maximize the system throughput. One can expect that from a certain level of selfishness among developers a CM that incentivizes 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 GPI compatibility.

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 that 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. [4, 5, 6]).

4. Priority-Based Contention Management

One key observation when analyzing the contention managers proposed in [7, 8, 9, 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. Timestamp[7] and Greedy[10, 9] measure the priority by the time a transaction is already running. Karma[7] takes the number of accessed objects as priority measure. Kindergarten[7] gives priority to transactions that already backed off against the competing transaction. The intuition behind a priority based 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 6 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. See Table 1 for some examples of how our framework can be used to describe CMs.

Definition 6. *A* priority-based *CM $\mathcal{M}$ associates with each job J_i a priority function $\vec{\omega}_i : \mathcal{R}_i \mapsto \mathbb{R}$ that can change over time. For resource $R \in \mathcal{R}_i$, $\vec{\omega}_i(R)$ is J_i's priority on resource R. $\mathcal{M}$ resolves conflicts between two transactions $T_{ij} \in J_i$ and $T_{qr} \in J_q$ over a resource $R \in \mathcal{R}_i \cap \mathcal{R}_q$ by aborting the transaction with lower priority on R, i.e., if $\vec{\omega}_i(R) \geq \vec{\omega}_q(R)$ then T_{ij} wins otherwise T_{ij} is aborted.*

In many CMs, the job priorities are not resource specific, i.e., $\vec{\omega}_i(R) = c \ \forall R \in \mathcal{R}_i$ where $c \in \mathbb{R}$. In this case we can replace $\vec{\omega}_i$ 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 $\vec{\omega}_i$ 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* $\tilde{\omega}_i : \mathcal{R}_i \mapsto \mathbb{R}$ is defined by $\tilde{\omega}_i(R) = \vec{\omega}_i(R) - \min_{j:R\in\mathcal{R}_j} \vec{\omega}_j(R)$. If the CM uses scalar priorities, J_i's relative priority $\tilde{\omega}_i \in \mathbb{R}$ is obtained by subtracting $\min_{j=1\ldots n} \omega_j$ from the absolute priority ω_i. Since optimistic CMs feature a reactive nature it is best to consider the priority-building mechanism as

Timestamp, Greedy	Karma, Polka	Eruption	Polite
$\mathcal{T} \rightsquigarrow \omega_i = \omega_i + c$ $C \rightsquigarrow \omega_i = 0$	$\mathcal{R} \rightsquigarrow \omega_i = \omega_i + c$ $C \rightsquigarrow \omega_i = 0$	$\mathcal{R} \rightsquigarrow \omega_i = \omega_i + c$ $\mathcal{W}$ against $T_j \rightsquigarrow \omega_i = \omega_i + \omega_j$ $C \rightsquigarrow \omega_i = 0$	$\mathcal{R} \rightsquigarrow \vec{\omega}_i(R) = 1$ $\mathcal{A} \rightsquigarrow \vec{\omega}_i(R) = 0 \; \forall R \in \mathcal{R}_i$ $C \rightsquigarrow \vec{\omega}_i(R) = 0 \; \forall R \in \mathcal{R}_i$

Table 1: Description of various popular priority-based CMs in terms of the framework introduced in Definition 6. Timestamp, Karma, Eruption, and Polite were proposed by Scherrer and Scott in [7], as well as Polka in [8]. Greedy was proposed by Gerraoui et al. in [9]. '$\mathcal{X} \rightsquigarrow f$' indicates that the described CM reacts to an event $\mathcal{X}$ with the modifications f. The value c is typically a small constant increment. Note that only Polite needs resource specific priorities, the other five CMs use scalar priority values.

event-driven. On each event, the CM may update the priority (functions). We find that the following *events* may occur for a transaction $T_{ij} \in J_i$ in a transactional memory system:

- $\mathcal{T}$: A time step,
- $\mathcal{W}$: T_{ij} wins a conflict,
- $\mathcal{A}$: T_{ij} loses a conflict and is aborted,
- $\mathcal{R}$: T_{ij} successfully allocates a resource,
- C: T_{ij} commits.

Event $\mathcal{T}$ occurs in every time step. We can use $\mathcal{T}$-events to model CMs like Timestamp[7], e.g., to model priorities that are a function of the transaction's time in the system. Event $\mathcal{W}$ occurs when the contention manager resolved a conflict in favor of T_{ij}. Event $\mathcal{A}$ occurs when the CM resolved a conflict in favor of one of T_{ij}'s competitors. Event $\mathcal{R}$ occurs when T_{ij} gains access of a resource. Note that this event happens regardless of whether resource R was freely available, or whether T_{ij} had to win in a conflict against other transactions to lock R. If T_{ij} wants to acquire a resource R that is currently locked by another transaction, the contention manager decides which transaction has to abort. If T_{ij} has to abort there occurs an $\mathcal{A}$-event for T_{ij}. If T_{ij} may continue, there occur both a $\mathcal{W}$ as well as an $\mathcal{R}$-event. Event C occurs when a transaction T_{ij} commits. Note that the priorities are associated with jobs rather than transactions. Thus, if a transaction T_{ij} commits this does not necessarily result in ω_i being reset to 0. The following two subtypes of priority-based CMs capture most contention management policies in the literature.

Definition 7. *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. Iff a CM 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 then we call it* quasi-priority-accumulating.

As an example, a Timestamp CM $\mathcal{M}_T$ is modelled as follows. $\mathcal{M}_T$ uses events of type $\mathcal{T}$ and 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 C occurs, then it is reset to 0. J_i's scalar priority at time t, $t^{\mathcal{M}_T,\mathcal{E}}(T_{ij}) < t \leq t^{\mathcal{M}_T,\mathcal{E}}(T_{ij}) + d^{\mathcal{M}_T,\mathcal{E}}(T_{ij})$ is $\omega_i(t) = \int_{t^{\mathcal{M}_T,\mathcal{E}}(T_{ij})}^{t} \alpha dt = \alpha(t - t^{\mathcal{M}_T,\mathcal{E}}(T_{ij}))$. 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. Thus, Timestamp is quasi-priority-accumulating.

4.1. Waiting Lemma

We argue in this section that delaying the execution of a job is not a rational strategy with priority-based CMs, i.e., jobs with artificial delays are not in the solution set if programmers use the solution concept defined in Section 2. Note that a programmer can make a job wait by introducing unnecessary code that does not allocate shared resources. 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). For our proof to work, we need to make two restrictions on the contention manager's priority modification mechanism:

I. The extent to which ω_i is increased (or decreased) on a certain event never depends on ω_i's current value

II. In a period where no events occur except for time steps, all priorities ω_i increase by $\Delta\omega \geq 0$.

Restriction I. implies that rules such as "if ω_i is larger than 10 add 100", or "$\omega_i = 2\omega_i$" are prohibited. A rule like "$\omega_i = \omega_i + 2$" on the other hand is permitted. Intuitively, it seems that a rule that, e.g., doubles the current priority on

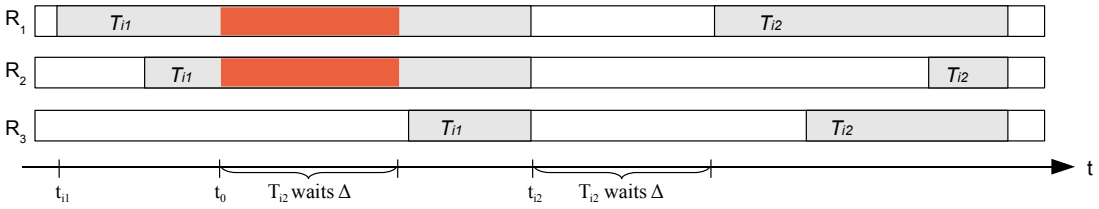

Figure 4: Job $J_i = T_{i1}, T_{i2}$ waits at time t_0 for a period of Δ. In this period, J_i keeps locking the already locked resources R_1 and R_2. After T_{i1} commits, the system would let T_{i2} start immediately, but the programmer of J_i decided to let T_{i2} wait Δ before it accesses the first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

certain types of events does not seem too far-fetched. Nevertheless, we are not aware of any contention manager in the literature that employs such an update rule. Thus, Restriction I. is probably not a substantial reduction to the CM design space. Restriction II. basically excludes CMs that decrease priorities on $\mathcal{T}$-events, and CMs in which $\mathcal{T}$-events do not affect all jobs in the same manner. Again, we do not know of any contention manager that incorporates rules of this kind. As most proofs that follow Lemma 8 rely on these restrictions, investigating CMs that do not comply with Restrictions I. and II. might still be an interesting subject for future work.

Lemma 8. *It is irrational to add artificial delays to a job, given that the TM system is managed by a priority-based CM $\mathcal{M}$ that is restricted by (I.–II.).*

Proof. We show the claim by comparing a job J'_i that incorporates artificial delays with a wait-free job J_i that results when omitting all delays in J'_i. In particular we prove that the programmers expect a shorter execution time for J_i than for J'_i, i.e. $\tilde{d}^{\mathcal{M}}(J_i) < \tilde{d}^{\mathcal{M}}(J'_i)$. Let $\omega_i(t)$ be ω_i at time t. Let J_i, or J'_i respectively enter the system at time t_i. Let $\mathcal{E}_{-i}$ be an execution environment for which $d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}}(J_i) = \tilde{d}^{\mathcal{M}}(J_i)$. We can construct an execution environment $\mathcal{E}'_{-i}$ for which it holds that $\tilde{d}^{\mathcal{M}}(J_i) < d^{\mathcal{M},\mathcal{E}'_{-i}\cup\{(J'_i,t_i)\}}(J'_i) \leq \tilde{d}^{\mathcal{M}}(J'_i)$ from $\mathcal{E}_{-i}$ as follows. Let us assume that J'_i incorporates only one artificial delay in the interval $[t_0, t_0 + \Delta]$. For any run of J'_i, $\mathcal{E}'_{-i}$ lets all other jobs J_j, $j \neq i$ delay their transactions as well during the interval $[t_0, t_0 + \Delta]$. Thus we establish a situation for J'_i that is at least as bad at time $t_0 + \Delta$ as the situation at t_0. 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, we get that $d^{\mathcal{M},\mathcal{E}'_{-i}\cup\{(J'_i,t_i)\}}(J'_i) = \tilde{d}^{\mathcal{M}}(J_i) + \Delta$. If J'_i has more than one artificial delay, we can do the same for each delay interval.

We have proven that if J_i and J'_i are either both non-dominated, or both dominated, J'_i cannot be in the solution set. However, if J'_i would be non-dominated and J_i dominated we could not make this conclusion, and it would be unclear which job is to be preferred. Luckily this case cannot occur. We prove this by showing that if J_i is dominated then J'_i must be dominated as well. Let $\hat{J}_i$ be a job that dominates J_i. We construct a job $\hat{J}_i{}'$ from $\hat{J}_i$ that basically waits whenever J'_i waits. Similar arguments as before, namely that relative priorities are preserved, imply that J'_i is dominated by $\hat{J}_i{}'$. This concludes the proof. □

Note that the claim of Lemma 8 is intimately linked to the solution concept stated in the model section. Although it seems intuitive we can only establish it since we model the programmers to be unaware of any runtime conditions, and risk-averse in that they assume a "worst-case" execution environment in which their jobs still eventually finish. In practice, a programmer often has some information about the environment in which her job 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 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 it does not partition a transaction that is partitioned in J_i. We will show that in any given execution environment $\mathcal{E}_{-i}$, job J_i

- either perfoms at least as fast as J'_i, or
- if it is slower than J'_i this is because J_i does not wait at a certain point in the execution.

Since we could achieve the same performance as J_i' in the latter case by introducing artificial delays to J_i, which we showed to be irrational, we conclude that a developer prefers J_i even if it does not dominate J_i'. All that remains to show is that there is at least one $\mathcal{E}_{-i}$ in which J_i outperforms J_i'. We will use analogous reasoning to argue that a CM is *not* GPI compatible. In particular we will show that there exists an execution environment $\mathcal{E}_{-i}$ in which J_i' is faster than J_i, and in which J_i could not achieve the performance of J_i' by introducing delays.

4.2. Quasi-Priority-Accumulating Contention Management

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. [8]). 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. It might be advisable for the job to keep locking R 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 9. *Quasi-priority-accumulating CMs restricted by (I.–II.) are not GPI compatible.*

Proof. Let J_i, J_i', and k be such that $J_i' = \mathit{Combine}(J_i, k)$. Let both jobs J_i and J_i' enter the system at time t_i for comparison. We show the claim by constructing an environment $\mathcal{E}_{-i}$ in which J_i' executes faster than J_i and in which it is impossible to achieve the performance of J_i' by introducing delays to J_i. For ease of notation, let $E := \mathcal{E}_{-i} \cup \{(J_i, t_i)\}$, and $E' := \mathcal{E}_{-i} \cup \{(J_i', t_i)\}$. Furthermore, we denote by $\omega_i(t)$ the priority ω_i at time t. Let $\mathcal{E}_{-i}$ be such that T_{ik}' is not aborted until commit. Hence, T_{ik} is not aborted until commit at time $t^{\mathcal{M},E}(T_{ik}) + d_{ik}$ either. Furthermore, let $\mathcal{E}_{-i}$ be such that there is at least one event that increases ω_i in $(t^{\mathcal{M},E}(T_{ik}), t^{\mathcal{M},E}(T_{ik}) + d_{ik})$. Thus $\omega_i'(t^{\mathcal{M},E}(T_{ik}) + d_{ik}) = \omega_i(t^{\mathcal{M},E}(T_{ik}) + d_{ik}) > \omega_i(t^{\mathcal{M},E}(T_{ik})) \geq 0$. When T_{ik} commits, ω_i is reset to 0 and $\omega_i'(t^{\mathcal{M},E}(T_{ik+1})) > \omega_i(t^{\mathcal{M},E}(T_{ik+1}))$. Since T_{ik+1} is started immediately, it will provoke the exact same conflicts as T_{ik}' at times $t \geq t^{\mathcal{M},E}(T_{ik+1})$. Let the first event on T_{ik+1} (except for time steps) be a conflict against a transaction $T_v \notin J_i$ whose priority is lower than the priority of J_i', but higher than the priority of J_i, i.e., $\omega_i' > \omega_v > \omega_i$ at the time this conflict occurs. Thus, T_{ik+1} is aborted and must be restarted, whereas T_{ik}' wins the conflict and runs to commit. Let all future transactions of J_i' run without conflicts. Thus, we get that $d^{\mathcal{M},E'}(J_i') < d^{\mathcal{M},E}(J_i)$, i.e., J_i' executes faster than J_i. Moreover, as all transactions T_{ij}' with $j \geq k$ run to commit in the first attempt it is impossible to introduce any delay into T_{ik} or T_{ik+1} without exceeding the execution time of J_i'. □

Theorem 9 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 released 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 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 due to the nature of its proof, Theorem 9 extends easily to the claim 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 $T_{ij+1} \in J_i$ starts with a lower priority than T_{ij} had just before committing. In fact, we can show that all priority-accumulating CMs proposed by [10, 9, 7, 8] are not GPI compatible. For a detailed description of the mentioned contention managers, please refer to the original work [10, 9, 7, 8], or to the technical report of this article[9]. There you can also find a discussion on the following corollary.

[9] See www.dcg.ethz.ch/publications/isaac09_EWtik.pdf.

Corollary 10. *Polite, Greedy, Karma, Eruption, Kindergarten, Timestamp and Polka are not GPI compatible.*

4.3. Priority-Accumulating Contention Management

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 a succeeding transaction is started immediately after its predecessor commits, 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*. Note that in the assumed model of optimistic contention management, gapless transaction scheduling is naturally given. This is due to the fact that in optimistic CM resource modifications are visible immediately, and commit operations are very lightweight, i.e., negligible in our model. Theorems 11 and 12 only hold for CMs that schedule transactions gapless. As gapless transaction scheduling is part of our optimistic contention management model we do not repeat it in the statements. However, we need to define the following before stating the theorems. 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*.

Theorem 11. *Any priority-accumulating CM $\mathcal{M}$ that is based only on time ($\mathcal{T}$-events) punishes unnecessary locking.*

Proof. Since the priorities grow monotone over time, and $\mathcal{T}$-events happen for all jobs at the same time, there is an implicit total order among all jobs in the system. In particular, a job J_j always has a higher priority than any job that entered the system after J_j, $t_j > t_k$ implies $\omega_j > \omega$[10]. From a job J_i's perspective, this order divides the competing jobs in the system into two sets, $L_i = \{J_j \mid \omega_j < \omega_i\}$ and $H_i = \{J_j \mid \omega_j > \omega_i\}$. By transitivity it follows that a job in L_i cannot influence a job in H_i, neither directly nor indirectly by influencing other jobs in L_i. A job in H_i will win any conflicts against any job in L_i anyway. Thus, all jobs in L_i are irrelevant for the performance of the jobs in H_i, and therewith for the execution of J_i either. Let J_i, J_i' be two jobs that are completely equal in all transactions except for one, T_{ik}, or T_{ik}' respectively, where $T_{ik}' \in J_i'$ contains an unnecessary lock. Thus, for any point in time t it holds that $\mathcal{L}^{\mathcal{M},\{(J_i \setminus T_{ik},0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(J_i' \setminus T_{ik}',0)\}}(t)$.

Consider the case where $d_{ik} = d_{ik}'$ first. Comparing T_{ik} with T_{ik}' in an empty environment, there is an interval (a, b) for which it holds that for any $t \in (a, b)$, $\mathcal{L}^{\mathcal{M},\{(T_{ik},0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(T_{ik}',0)\}}(t) \setminus \{R\}$. This is, T_{ik}' locks resource R unnecessarily in the interval (a, b). Given an execution environment $\mathcal{E}_{-i}$, the unnecessary lock of R either does not provoke a conflict, or it does. If it provokes no conflicts, or only conflicts with jobs in L_i then choosing J_i or J_i' results in the same execution time. If it provokes a conflict against a job in H_i, however, T_{ik}' is aborted whereas T_{ik} continues. If all transactions after T_{ik} run conflict free J_i is strictly faster than J_i'. Otherwise, let t_{last} be the time when T_{ik}' is restarted for the last time before commit, i.e., T_{ik}' commits at time $t_{last} + d_{ik}$. In case J_i' executes faster than J_i the programmer could delay T_{ik} until t_{last} and thus reach the same execution time as with J_i'. This is because the resources allocated by T_{ik} would always be a subset of the resources allocated by T_{ik}' in the interval $[t_{last}, t_{last} + d_{ik}]$. In order to do so, however, the programmer would introduce an artificial delay to J_i. As $\mathcal{M}$ is time-based, Lemma 8 applies and implies that choosing J_i' over J_i is irrational.

It remains to show that if $d_{ik} < d_{ik}'$ then T_{ik} is still preferable. This case occurs when T_{ik}' contains an unnecessary lock that additionally delays all future resource accesses compared to T_{ik}. Let $\delta = d_{ik}' - d_{ik}$ be the delay. In terms of resource allocation this means that there is a point in time a such that $\forall t \leq a : \mathcal{L}^{\mathcal{M},\{(T_{ik}',0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(T_{ik},0)\}}(t)$, and $\forall t, a < t \leq a + \delta : \mathcal{L}^{\mathcal{M},\{(T_{ik}',0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(T_{ik}',0)\}}(a) \cup \{R\}$, and $\forall t > a + \delta : \mathcal{L}^{\mathcal{M},\{(T_{ik}',0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(T_{ik},\delta)\}}(t)$. For this case, the empty environment $\mathcal{E}_{-i} = \emptyset$ can serve as positive instance where $d^{\mathcal{M},\{\mathcal{E}_{-i} \cup (J_i,t_i)\}}(J_i) < d^{\mathcal{M},\{\mathcal{E}_{-i} \cup (J_i',t_i)\}}(J_i')$. For an environment $\mathcal{E}_{-i}$ in which T_{ik}' is in none of its runs aborted during the interval $(a, a + \delta)$ (relative to the run's start time), a programmer could achieve the same performance by introducing a delay of δ to T_{ik} at time a. If $\mathcal{E}_{-i}$ is such that T_{ik}' is aborted due to the unnecessary lock in $(a, a+\delta)$ then J_i achieves the same performance as J_i' by delaying T_{ik} until $t_{last} + \delta$, where t_{last} is defined as before. We thus proved that whenever J_i' executes faster than J_i, the programmer could achieve the same execution time by introducing delays to J_i. From Lemma 8 it follows that unnecessary locking is irrational. □

[10] If $\mathcal{M}$ always resolves ties consistently, e.g. in favor of the job with lower id then a strict order is guaranteed also if we allow concurrent starting times.

Theorem 12. *Any priority-accumulating CM $\mathcal{M}$ that is based only on time ($\mathcal{T}$-events) rewards partitioning.*

Proof. Let J_i, J_i', and k be such that $J_i' = Combine(J_i, k)$. 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_{ik} starts at the same time as T_{ik}', T_{ik} will lock the exact same resources and thus provoke the same conflicts as T_{ik}'. T_{ik+1} locks less or equally many resources as T_{ik}, i.e., at any time t it holds $\mathcal{L}^{\mathcal{M},\{(T_{ik+1},d_{ik+1})\}}(t) \subseteq \mathcal{L}^{\mathcal{M},\{(T_{ik}',0)\}}(t)$. Let t_c be the time when T_{ik} runs to commit in $\mathcal{E}_{-i}$. Until t_c both J_i and J_i' behave exactly the same in any $\mathcal{E}_{-i}$. Since $\mathcal{M}$ schedules transactions gapless, if T_{ik+1} provokes a conflict at time $t \in [t_c, t_c + d_{ik+1}]$ then T_{ik}' provokes the same conflict at time t. After t_c, T_{ik+1} is started immediately. Depending on $\mathcal{E}_{-i}$ there are two scenarios: (a) the unfinished run of T_{ik}' provokes only conflicts with jobs in L_i, or no conflicts at all, and (b) T_{ik}' provokes a conflict with a job in H_i, where L_i and H_i are defined as in the proof of Theorem 11. In case (a), J_i provokes a subset of the conflicts that J_i' provokes. J_i and J_i' have the same start time, and thus always the same priority. T_{ik+1} wins all conflicts, and runs until commit. T_{ik+1} and T_{ik}' commit at the same time. As all succeeding transactions are equivalent, J_i''s runtime equals J_i's runtime. In case (b), T_{ik}' is aborted and restarted. We have a positive instance of an environment $\mathcal{E}_{-i}$ if the resource because of which T_{ik}' is aborted is not locked by T_{ik+1} at the time of the conflict, and all succeeding transactions of J_i run until commit in the first run. Then J_i performs strictly better than J_i'. We also have a positive instance if T_{ik+1} is aborted at the same time, and runs conflict free afterwards. This is because T_{ik} has already committed and J_i does not need to redo the work done in T_{ik}. For all other instances that create scenario (b) the programmer of J_i could achieve the same performance as with J_i' by delaying T_{ik+1} until $t_{last} + d_{ik}$, where t_{last} is the time when T_{ik}' is started for its final run. Lemma 8 implies that J_i is preferable to J_i'. Partitioning is rewarded. □

By the definition of GPI compatibility, Theorems 11 and 12 immediately imply that priority-accumulating CMs that are based on time only are GPI compatible.

Corollary 13. *Any priority-accumulating CM $\mathcal{M}$ that is based only on time ($\mathcal{T}$-events) is GPI compatible.*

This promising result for priority-accumulating CMs 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 job's priority such that their code is executed faster (at the expense of the overall system performance). E.g., consider a CM like Karma [7], 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 that 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 contention managers is not GPI compatible.

Theorem 14. *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.*

(ii) *$\mathcal{M}$ increases relative priority on $\mathcal{R}$-events.*

(iii) *$\mathcal{M}$ schedules transactions gapless and increases relative priorities on $\mathcal{C}$-events.*

(iv) *$\mathcal{M}$ restarts aborted transactions immediately and increases relative priorities on $\mathcal{A}$-events.*

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 that locks resource $R \in \mathcal{R}$ unnecessarily during the interval $[t_u - \epsilon, t_u + \epsilon)$. Let T_{ij} be exactly the same transaction as T_{ij} except it does not lock R during $[t_u - \epsilon, t_u + \epsilon)$. We are going to show the claims by comparing the performance of job J_i when containing T_{ij} with its performance when containing T_{ij}' instead of T_{ij}.

(i). Let T_{ij}' provoke an unnecessary conflict on R with another transaction T_k at time t_u, and $\omega_i(t_u) > \omega_k(t_u)$. T_{ij}' wins the competition for R, and $\mathcal{M}$ increases ω_i by δ. Furthermore, let T_{ij}' provoke a conflict on a resource $Q \in \mathcal{R}$ with a transaction T_l at time $t_u + \epsilon$, 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, given there are no more conflicts, thus prolongate

the execution time of J_i. Since there is no way to introduce delays to T_{ij} without making its execution take longer than T'_{ij} it follows that $\mathcal{M}$ does not punish unnecessary locking.

(ii). Let T'_{ij} be so that it does not access R at all, or only after the unnecessary lock at $t_u + \epsilon$. Let there be no conflicting transaction on R during $[t_u - \epsilon, t_u + \epsilon)$, and let the contribution of having acquired R to the priority increase be 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. There is no way delays could make T_{ij} perform as good as T'_{ij}. 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 $R \in \mathcal{R}$ for a period of ϵ, and then commits. Let T_{i3} be a transaction which only accesses R. Assume the following scenario: $\mathcal{M}$ executes T_{i1} and commits at time t_0. T_{i2} starts immediately, locks R 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 that was granted 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. Since introducing artificial delays to J_i could not make it perform as fast as J'_i it follows that $\mathcal{M}$ does not punish unnecessary locking.

(iv). For simplicity we assume here that the time needed for rolling back an aborted transaction is negligibly small. The proof extends easily to the general case. Let T'_{ij} start at $t_u - \epsilon$, and provoke 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'_{ij} runs conflict-free until commit. If the programmer of 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. There is no way delays could make T_{ij} perform as good as T'_{ij}. $\mathcal{M}$ does not punish unnecessary locking. □

5. Non-Priority Based Contention Management

One example of a CM that is not priority-based is Randomized (cf. [7]). 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.

Theorem 15. *Randomized is GPI compatible.*

Proof. We compare a J_i with J'_i under all possible environments $\mathcal{E}_{-i}$. Let $\mathcal{E}_{-i}$ include the CM's randomized decisions, i.e., if J'_i and J_i provoke a conflict at the same time with the same competing jobs then we compare the execution times of J'_i and J_i for both coin flips seperately. For ease of notation let $E := \mathcal{E}_{-i} \cup \{(J_i, t_i)\}$, and $E' = \mathcal{E}_{-i} \cup \{(J'_i, t_i)\}$.

In a first step, we show that Randomized, denoted by $\mathcal{M}$, rewards partitioning. Let J_i, J'_i, and k be such that $J'_i = Combine(J_i, k)$. We distinguish three cases of the execution of T'_{ik}. Case (A): T'_{ik} runs until commit; Case (B): T'_{ik} is aborted before $t^{\mathcal{M},E'}(T'_{ik}) + d_{ik}$; Case (C): T'_{ik} is aborted in the period $[t^{\mathcal{M},E'}(T'_{ik}) + d_{ik}, t^{\mathcal{M},E'}(T'_{ik}) + d^{\mathcal{M},E'}(T'_{ik})]$; Case (A). Since we assume the time needed for committing is negligible, J_i always locks a subset of the resources locked by J'_i. Thus, J_i provokes a subset of the conflicts provoked by J'_i. As $\mathcal{M}$ always decides in favor of J'_i, so it does for J_i. T_{ik} and T_{ik+1} both run to commit in the first attempt. We have $d^{\mathcal{M},E}(T_{ik}) + d^{\mathcal{M},E}(T_{ik+1}) = d^{\mathcal{M},E'}(T'_{ik})$, and hence $d^{\mathcal{M},E}(J_i) = d^{\mathcal{M},E'}(J'_i)$. Case (B). T_{ik} has the same conflicts as T'_{ik} and both are aborted at the same time. They are both restarted at the same time, and we can recurse the argument until case (A) or (C) occur. Case (C). T_{ik} runs until commit, T_{ik+1} is started immediately and is aborted in the same conflict as T'_{ik}. T'_{ik} and T_{ik+1} are restarted. Let t_a be the time when T'_{ik}, or T_{ik+1} respectively are aborted. Let t'_{ik+1} be the time when J'_i has successfully completed all operations corresponding to T_{ik} after the restart. Employing J'_i instead of J_i coincides with delaying T'_{ik} from t_a until t'_{ik+1}. In order to show that Randomized rewards partitioning we can use the same argument from Lemma 8, 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 a second step, we show that $\mathcal{M}$ punishes unnecessary locking. Let J_i and J'_i be two jobs that are exactly the same except for one contained transaction T_{ij}, or T'_{ij} respectively. Let T'_{ij} have an unnecessary lock of resource $R \in \mathcal{R}$ compared to T_{ij}, and $d_{ij} = d'_{ij}$. If $\mathcal{E}_{-i}$ is such that the unnecessary lock provokes no conflict both jobs achieve the same execution time. If the unnecessary lock provokes a conflict and $\mathcal{M}$ decides in favor of T'_{ij} then the lock 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. 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 $t_{last} + d'_{ij}$. T_{ij} could also be delayed until t_{last}, and reach a commit time at least as good as T'_{ij}. This is since J_i would provoke a subset of the conflicts provoked by J'_i. As delaying is irrational, employing T'_{ij} instead of T_{ij} is irrational. It remains to show that if $d_{ij} < d'_{ij}$ then J_i is still preferable to J'_i. Let T'_{ij} be exactly like T_{ij} except for one unneeded resource access during an interval $[t_u - \epsilon, t_u + \epsilon]$ which prolongates d'_{ij} by $\delta = d'_{ij} - d_{ij}$. If the execution environment is empty, $\mathcal{E}_{-i} = \emptyset$, we get that $d^{\mathcal{M},E}(J_i) = d^{\mathcal{M},E'}(J'_i) - \delta < d^{\mathcal{M},E'}(J'_i)$. If $\mathcal{E}_{-i}$ is such that the unnecessary lock provokes a conflict in which $\mathcal{M}$ decides for T_{ij}, the same effect would be achieved by introducing a delay to T_{ij} in the interval corresponding to $[t_u - \epsilon, t_u + \epsilon]$. If T'_{ij} is aborted, though, and T_{ij} runs until commit in the first attempt, choosing J_i 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. By introducing a delay in the interval corresponding to $[t_u - \epsilon, t_u + \epsilon]$, and additionally postponing the start of T_{ij} until t_{last} the programmer of J_i could reach a commit time at least as good as T'_{ij}. This is again because T_{ij} would provoke a subset of the conflicts that T'_{ij} provokes, and since $\mathcal{M}$ would make the same decisions T_{ij} would also win all conflicts. As introducing artificial delays is irrational the claim follows. □

Note that in order for the proof to work, the Randomized CM must schedule consequent transactions gapless. Thus, Theorem 15 holds for optimistic contention management. If a non-optimistic contention manager would entail a non-negligible gap between two consecutive transactions, however, then partitioning would not be rewarded. This is easy to see since in an empty environment, a fine grained job would yield a longer execution time than a version that combines some contained transactions.

Unfortunately, in terms of practicability, it is not a good solution to employ such a simple Randomized CM, 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. Simulations

To verify our theoretical insights, we implemented selfish threads in DSTM2 [3], a software transactional memory system in Java, and let them compete with the threads originally provided by the authors of the included benchmark under several different contention managers. DSTM2 is an experimental framework that provides some basic CMs, and allows to implement custom CMs easily.

6.1. 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 selfish thread updates several resources per transaction at once. See Figure 5 for the code executed by the selfish threads and Figure 6 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 selfish threads 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 competitors. Hence a selfish programmer 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 selfish threads with collaborative threads in 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

```
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))
     }
   }
 });
}
```

Figure 5: Selfish 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);
    }
   });
}
```

Figure 6: "Good" thread. The call() method consists of only one update call.

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 selfish threads (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 selfish threads. 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.

6.2. Results

The results confirm the theoretical predictions that a selfish programmer 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 selfish threads (cf. Figure 7). In all of our tests, if the system was managed by Polite the selfish threads 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 selfish threads' success rate was 70% over all runs and 100% for $\gamma \in \{20, 50, 100\}$. Of all tests run with the Timestamp manager, selfish behavior paid off in 92% of the cases and in 100% if the granularity γ was at least 20. Under Randomized, selfish threads 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 selfish thread's relative performance significantly. The transaction granularity however has a huge impact. Figure 8 shows the average throughput of both a selfish and a collaborative thread. In our experiments, a selfish thread's throughput was practically always higher than the collaborators' under the Karma,

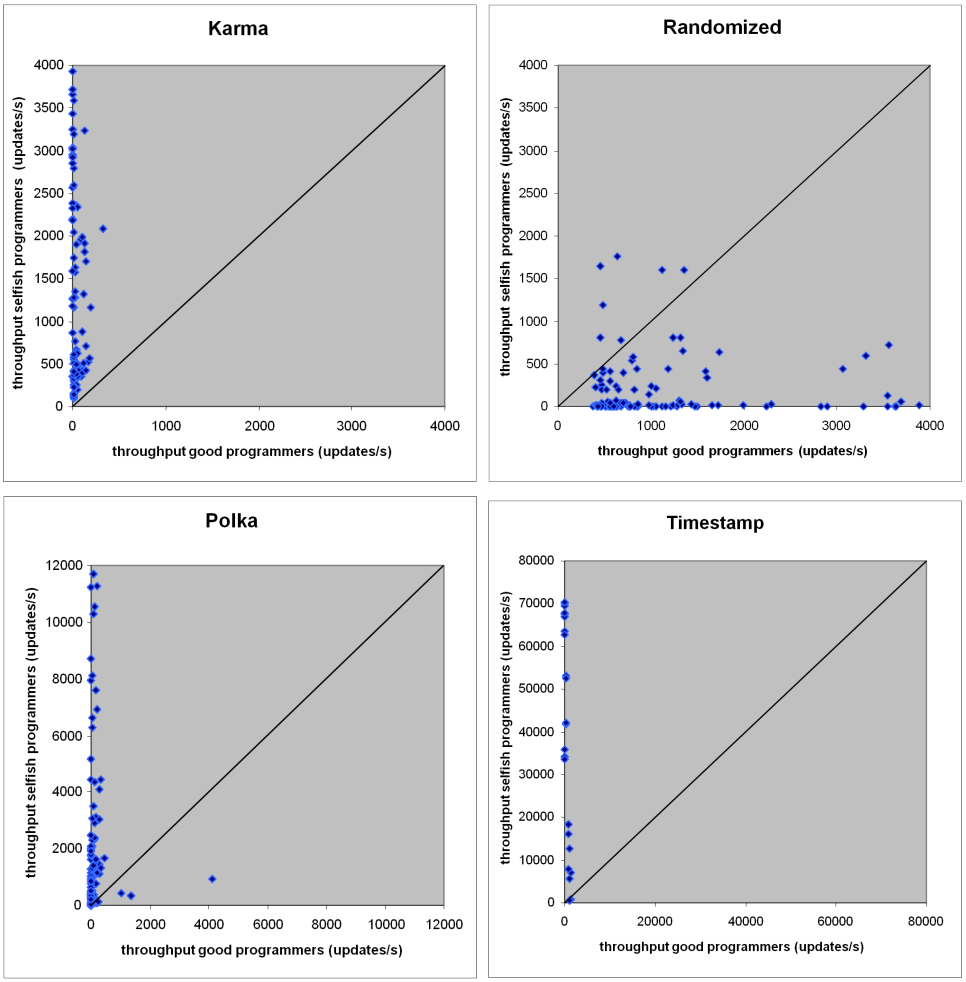

Figure 7: Plot of all cases simulated under a Karma, a Randomized, a Polka and a Timestamp CM. If a point is above the diagonal line this indicates that in the corresponding test run, the selfish thread had a larger throuhput than the good thread that only employs transactions of granularity 1. For Karma, the cases where $\gamma = 1$ are omitted.

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 selfish threads 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 selfish programmers. 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 with high contention, one fast selfish thread locking the entire datastructure is still quite efficient. More so, caching mechanisms probably speed up the system when basically only one thread is working. With the appropriate level of contention, the effect of degradation in system efficiency would possibly show. Regardless of this inability to show the system degradation explicitly, it is obviously not desirable for a multi threaded program to basically have only one thread running. Note also the break in the throughput increase between $\gamma = 1000$ and $\gamma = 5000$ with 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 selfish programmers use granularities higher than 1000.

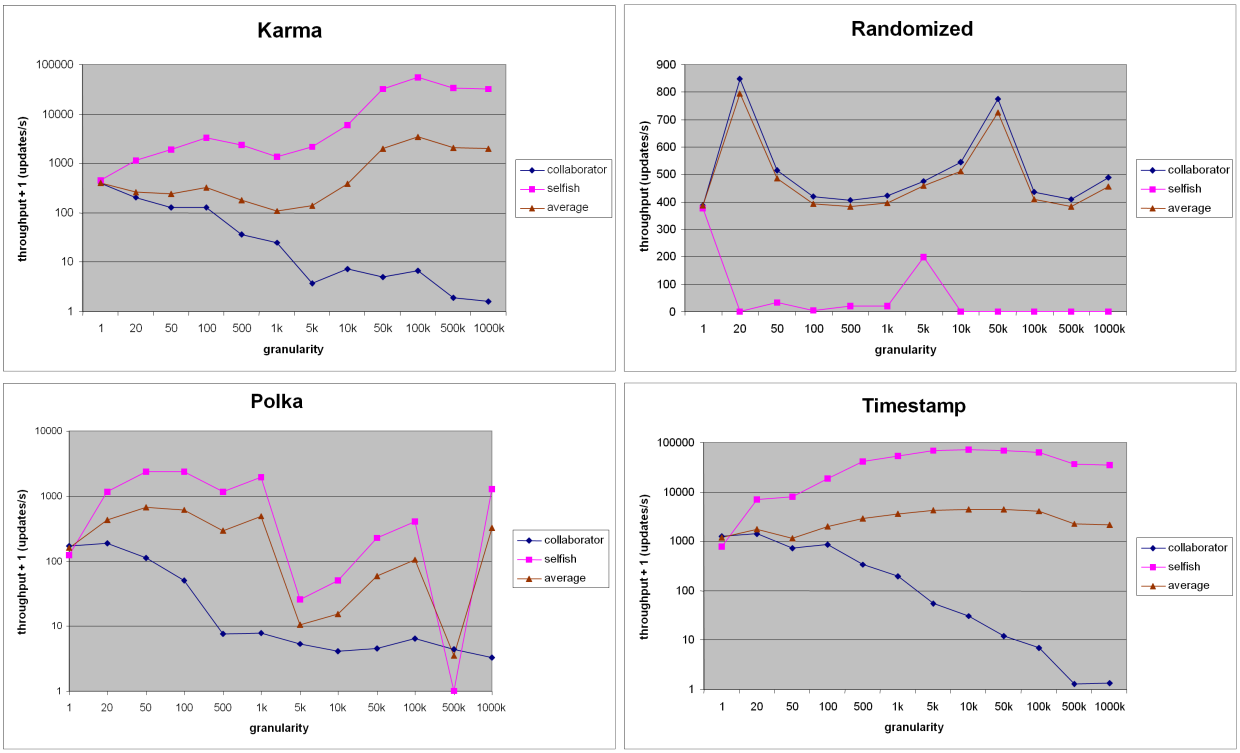

Figure 8: Average throughput of a selfish and a collaborative thread in the red-black tree benchmark with 15 collaborators and one selfish thread. The selfish thread does not employ a priority boosting mechanism ($\pi = 0$). In addition to the collaborators' and the selfish thread's throughput, the average throughput of all 16 concurrent threads is depicted. Except for Randomized, we added 1 to the actual throughput and used a logarithmic scale.

7. 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. Recent work by Schneider et al.[11] can be seen as a successful step in this direction. 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] D. B. Lomet, Process structuring, synchronization, and recovery using atomic actions, SIGOPS Operating Systems Review 11 (2) (1977) 128–137.

[2] M. Herlihy, J. E. B. Moss, Transactional memory: Architectural support for lock-free data structures, SIGARCH Computer Architecture News 21 (2) (1993) 289–300.

[3] M. Herlihy, V. Luchangco, M. Moir, A flexible framework for implementing software transactional memory, SIGPLAN Not. 41 (10) (2006) 253–262.

[4] S. Aland, D. Dumrauf, M. Gairing, B. Monien, F. Schoppmann, Exact price of anarchy for polynomial congestion games, in: Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS), 2006, pp. 218–229.

[5] G. Christodoulou, E. Koutsoupias, The price of anarchy of finite congestion games, in: Proceedings of the 37th annual ACM symposium on Theory of computing (STOC), 2005, pp. 67–73.
[6] T. Roughgarden, Selfish Routing and the Price of Anarchy, MIT Press, 2005.
[7] W. N. Scherer III, M. L. Scott, Contention management in dynamic software transactional memory, in: PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP), 2004.
[8] W. N. Scherer III, M. L. Scott, Advanced contention management for dynamic software transactional memory, in: Proceedings of the 24th annual ACM symposium on Principles of Distributed Computing (PODC), 2005, pp. 240–248.
[9] R. Guerraoui, M. Herlihy, B. Pochon, Toward a theory of transactional contention managers, in: Proceedings of the 24th annual ACM symposium on Principles of Distributed Computing (PODC), 2005, pp. 258–264.
[10] H. Attiya, L. Epstein, H. Shachnai, T. Tamir, Transactional contention management as a non-clairvoyant scheduling problem, in: Proceedings of the 25th annual ACM symposium on Principles of Distributed Computing (PODC), 2006, pp. 308–315.
[11] J. Schneider, R. Wattenhofer, Bounds on contention management algorithms, in: 20th International Symposium on Algorithms and Computation (ISAAC), 2009, pp. 441–451.