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.

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 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

* A full version including all proofs, is available as TIK Report 310 at http://www.tik.ee.ethz.ch

```
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.

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.

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

[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

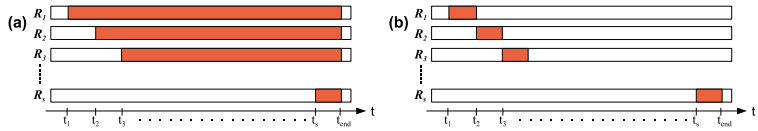

Fig. 2. Transactional allocation of ring nodes **(a)** by `incRingCounters` and **(b)** by `incRingCountersGP`.

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.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 analytically that most CMs proposed in the literature so far lack such an incentive compatibility. 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 [3].

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. 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, 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 aborts the other competing transaction. 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.[3] We say a job J_i *is running* if its

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

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 CM. 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 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

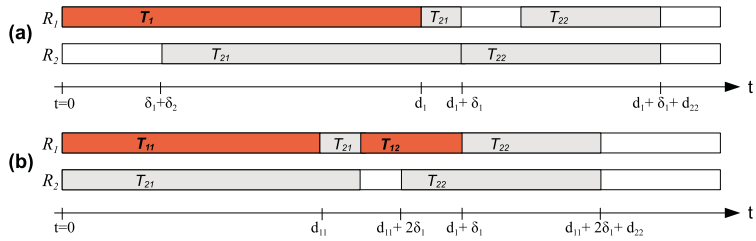

Fig. 3. Partitioning example. The picture depicts the optimal allocation of two resources R_1 and R_2 over time in two situations **(a)** and **(b)**. In **(a)**, the programmer of job J_1 does not partition T_1. In **(b)**, she partitions T_1 into T_{11} and T_{12}. The overall execution time is shorter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

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}. □

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 CM 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 (cf. Figure 3). If we assume developers act selfish then also a system managed by an optimal offline scheduler suffers a performance loss and a 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, commonly known as *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.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 CMs 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 $T_{ix} \in J_i$ and $T_{jy} \in J_j$ 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_{ix} wins otherwise T_{iy} 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 $\tilde{\boldsymbol{\omega}}_{\boldsymbol{i}}$ is defined by $\tilde{\boldsymbol{\omega}}_{\boldsymbol{i}}[k] = \boldsymbol{\omega_i}[k] - \min_{i=1\ldots n} \boldsymbol{\omega}_i[k]$, $\forall k = 1 \ldots s$. If the CM uses scalar priorities, J_i's relative priority $\tilde{\omega}_i$ is obtained by subtracting $\min_{i=1\ldots n} \omega_i$ 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: A time step ($\mathcal{T}$); T_{ij} wins a conflict ($\mathcal{W}$); T_{ij} loses a conflict and is aborted ($\mathcal{A}$); T_{ij} successfully allocates a resource R_k ($\mathcal{R}_k$); T_{ij} 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. 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 $\omega_i(t) = \int_{t_{ij}}^{t} \alpha dt = \alpha(t - t_{ij})$. Timestamp is quasi-priority-accumulating since a job's priority always increases and never decreases over time except it is reset when a contained transaction commits.

Waiting Lemma. We argue in this paragraph that delaying the execution of a job[4] is not rational with the assumption that the execution environment $\mathcal{E}_{-i}$ is arbitrary. 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 CM's priority modification mechanism: (I.) An increase (or decrease) of ω_i never depends on ω_i's current value[5] 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 (Sketch). If a programmer delays a transaction by Δ, the adversary can preserve the environment and thus increase its execution time by Δ. □

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

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

[5] 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.

adapt our model of a risk-averse agent in that we let her suppose that a worst case environment yields a finite execution time. 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.

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-acc. CMs restricted by (I.–II.) are not GPI compatible.*

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.*

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 (Sketch). 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. □

For instance, 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, priority based CMs are generally dangerous in the sense that they bear a potential for programmers to cheat, i.e., to boost their job's priority at the expense of other jobs. 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-acc. 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.*
(iv) *$\mathcal{M}$ restarts aborted transactions immediately and increases relative priorities on $\mathcal{A}$-events (aborting).*

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 works similarly to the proof of Lemma 2. Note that Lemma 1 does not apply here as Randomized is not priority-accumulating. However, to show that waiting is irrational also with Randomized is easy. 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 TM constitutes an inalienable convenience to programmers in concurrent environments, it does not automatically defuse the danger that selfish programmers might exploit a multicore system. GPI compatibility has to be addressed when designing a TM system. 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 conjecture that by combining randomized conflict resolving with a time-based priority mechanism, chances of finding an efficient, GPI compatible CM are high.

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: Proc. of 37th annual symposium on Theory of computing*, pages 67–73, 2005.
3. R. Eidenbenz and R. Wattenhofer. Brief announcement: Selfishness in transactional memory. In *SPAA '09: Proc. of the 21st annual symposium on Parallelism in Algorithms and Architectures*, pages 41–42, 2009.
4. R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proc. 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: Proc. 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 *ISAAC '09: Proc. of the 20th International Symposium on Algorithms and Computation*, 2009.