Robust Contention Management in Software Transactional Memory

Rachid Guerraoui
School of Computer and Communication Sciences, EPFL
rachid.guerraoui@epfl.ch

Maurice Herlihy
Microsoft Research Cambridge and Brown University
mph@cs.brown.edu

Michał Kapałka
School of Computer and Communication Sciences, EPFL
michal.kapalka@epfl.ch

Bastian Pochon
School of Computer and Communication Sciences, EPFL
bastian.pochon@epfl.ch

ABSTRACT

Software transactional memory (STM) systems use lightweight, in-memory software transactions to address concurrency in multi-threaded applications, ensuring safety at all times. A contention manager is responsible for the system as a whole to make progress (liveness).

In this paper, we study the impact of transaction failures on contention management in the context of STM systems. The failures we consider include page faults as well as actual process or thread crashes. We observe that, even with a small number of failures, many of the previously defined contention managers do not behave well, in the average case, and none provides worst case guarantees.

We introduce FTGreedy, a new contention manager that is able to cope with faulty transactions. In short, FTGreedy (a) compares well with previous contention managers when no failures occur, (b) has good performance in the face of failures, and (c) provable worst case properties even if transactions can fail, as long as the system features some synchrony assumptions (which need only be weak and eventual).

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*

General Terms

Algorithms, Theory, Performance

Keywords

Concurrency, software transactional memory, contention management

1. INTRODUCTION

In modern multi-core architectures, multi-threaded applications are the norm. *Software transactional memory* (STM) systems provide an appealing approach to address the concurrency raised by multi-threading. The basic idea consists in accessing shared memory through lightweight, in-memory transactions. A transaction is a basic unit of computation, which can either *abort* (the effects of the transaction are rollbacked and do not appear to other transactions) or *commit* (the effects of the transaction appear to take place atomically to other transactions).

In STM systems, safety is ensured at all times despite concurrency conflicts among transactions hosted by distinct threads and processes. Progress guarantees (liveness) are delegated to a *contention manager*, a separate, possibly external, user-provided module (we consider here only obstruction-free STM systems; for lock-free ones it is the STM that provides progress guarantees, not a contention manager). The approach adopted by STM systems thus completely decouple safety from liveness, enabling the programmer to test various liveness strategies, under the form of distinct contention managers, while always guaranteeing safety. Roughly speaking, when two transactions encounter a conflict by accessing the same memory location, the contention manager will decide to abort or delay one of them, and later restart or resume it.

Many contention managers have been proposed in the literature. These include simple ad-hoc contention managers (e.g., Backoff) [8], contention managers using more sophisticated priority schemes among transactions [9, 4], and randomized contention managers [7]. One of these contention managers, which we introduced earlier and called Greedy, actually ensures an upper bound on the time to commit transactions [4].

As we show in this note through various benchmarks, previous contention managers may not be able to cope well with transaction failures. Although some contention managers, like Karma, Polka [9] or Aggressive [5], seem to be quite fault-tolerant, none of them provides any worst case guarantees. On the other hand, Greedy [4], a contention manager with provable properties, may tolerate only small number of failures.

In practice, there are indeed reasons for transactions to fail. Clearly, a thread might encounter an illegal instruction (for instance, when it dereferences a null pointer), producing a segmentation fault. The thread silently dies, and by the same occasion, the transaction.

In addition, most if not all traditional operating systems offer a memory paging mechanism, i.e., a mechanism for moving pages of main memory into secondary storage, when these pages are not supposed to be accessed by any thread in a very near future. In presence of paging, a thread may experience a *page fault* when the page containing the desired data has been swapped out of main memory, and needs to be moved back into memory before it can be accessed. In the case of a page fault, a thread that executes in a transaction needs to wait for the page to be available. Waiting for a transaction that experiences a page fault is a waste of time, as the time for a page to be moved back into memory is several orders of magnitude higher than the time between two successive instructions for a transaction that executes normally.[1] In that sense, the transaction experiencing a page fault is considered to have failed.

In this note, we extend the concept of Greedy [4], and we introduce a new contention manager, called FTGreedy. This new contention manager is able to cope with faulty transactions using an adaptive timeout mechanism when aborting a transaction. Basically, a slow transaction that is aborted because it is considered faulty, is given more time the next time a conflict is encountered. As the timeout grows exponentially, a slow transaction is eventually not aborted: it eventually commits.

We show that the performance of FTGreedy is comparable to those of contention managers like Karma or Polka. In addition, FTGreedy retains some worst case provable properties of Greedy, even in presence of failures.

In the following sections we first compare Greedy, FTGreedy and other contention managers experimentally to highlight the main issues and show the practicality of our results. Next, we prove that FTGreedy guarantees global progress, even when failures occur, under the assumption that the system is eventually synchronous. Then we focus on executions without failures and prove upper bounds on the time to complete n concurrent transactions (throughput) and on the time to commit every transaction despite the conflicts it might encounter (latency). We do so under the assumptions that the system is synchronous from the beginning and transactions are not preemptive.

Our benchmarks were all performed on two machines: a 4-processor Intel Xeon and a uniprocessor Pentium IV, both with HyperThreading turned on (this gives 8 and 2 virtual processors, respectively). We used SXM [6], an STM system provided as a C# library. We considered the .NET C# platform, with an update ratio of 20% for transactions, and a varying number of failures. We measured the number of committed transactions per second as a function of the number of threads (either a total number or the maximum number of threads that could crash inside a transaction at some point at the beginning of the experiment[2]). In cases when performance was measured as a function of the maximum number of failures, the total number of threads was constant and equal to 32.

[1]We discuss in Section 4 how we may optimize contention management to address this situation when the operating system is able to tell whether a transaction experiences a page fault [1] or not.

2. FAULT-TOLERANT GREEDY

If a transaction T accesses an object that is already used by another transaction T', it has two choices: it can either wait for some time and retry later or abort T'. A contention manager is a module which tells T what to do and all thransactions behave accordingly, some progress properties might be guaranteed. The simplest contention manager one can imagine is Aggressive. It always tells T to abort the conflicting transaction. Obviously, it is fault-tolerant as no transaction ever blocks on waiting for a faulty one. However, not only it does not provide any liveness guarantees, but also in practice it often leads to livelocks, in which case no transaction makes any progress.

A more complicated scheme is used by the Backoff contention manager. Here, a transaction T can abort a conflicting transaction T' only after waiting for some random time, the maximum of which grows exponentially with each abort of T'. Backoff does not give any deterministic guarantees and, what is more important, does not perform very well in practice (as will be shown later). However, it should be, at least in theory, quite resistant to transaction failures.

The idea behind the Karma [9] contention manager is that the transactions which have already done a lot of work and have almost finished should not be aborted by transactions that have just started or are very short. The number of objects that a transaction has opened is taken as its priority. When a transaction T encounters a conflict with a transaction T', it checks whether the number of times it attempted to open an object is higher than the difference in priorities between itself and T'. If yes, it can abort T'. Otherwise, it waits for a fixed period of time. The Polka contention manager [9] works in a similar way, but it also incorporates a randomized exponential backoff scheme instead of waiting for predefined amount of time. Both contention managers are very efficient and should be fault-tolerant; however, they do not have any provable progress properties, even in failure-free systems.

The algorithm of Greedy [4] is the following. Each transaction, upon its start, gets a timestamp. The timestamps are unique and monotonically increasing over time. A transaction T can abort a conflicting transaction T' only if the timestamp of T is lower than the one of T' or T' is waiting for another transaction. This simple scheme is clearly not resistant to failures, as a faulty transaction with a very low timestamp can force all other ones to wait indefinitely long. However, it has been proved in [4] that Greedy has the following properties:

1. Every transaction commits within a bounded time.
2. The time to complete n concurrent transactions that share s objects is within a factor of $s(s+1)+2$ of the time that would have been taken by an optimal off-line list scheduler.

[2]It is worth noting that, for some contention managers, all threads were blocked after the first crash and could not perform any steps, including a subsequent crash.

To evaluate the performance of abovementioned conention managers in presence of failures we use three benchmark applications: `List`, `RBTree` and `RandomBenchmark`. The first two model the classical data structures of a linked list and a red-black tree, respectively, both containing integer elements in the interval $[0, 255]$. In `RandomBenchmark`, transactions access randomly chosen objects. They, however, respect defined limits on the maximum number of objects they can read or update. This benchmark corresponds to applications for which the object access pattern is not regular.

In presence of failed transactions, Aggressive performs well (though it compares poorly when transactions do not fail on a multiprocessor machine where it livelocks pretty often). On the other hand, Backoff and Greedy behave rather badly. For the former one, surprisingly, the performance often grows with the number of failures (more or less exponentially). The latter one, on the contrary, performs well only when there is no or very few failures. Karma and Polka perform very well in general and seem to be quite resistant to failures. Figures 1, 2 and 3 illustrate our results for `List`, `RBTree` and `RandomBenchmark`, respectively.

Greedy was the first contention manager that combined non-trivial provable properties with good practical performance [4]. However, Greedy loses its guarantees in situations in which transactions may fail. We introduce here FTGreedy – an extended version of Greedy that can deal with failures.

A transaction T is given a timestamp ts when it starts: T retains ts over its whole lifecycle. Each timestamp is unique, and timestamps monotonically grow as transactions are created. For each transaction T_i we also define a delay δ_i initialized to δ_0. In the benchmarks that follow, we assume $\delta_0 = 1$ millisecond.

In case a conflict is encountered, FTGreedy behaves as follows. Consider that a transaction T_a is executing, and a transaction T_b tries to access the same object. Therefore, there is a conflict between T_a and T_b that has to be resolved by the contention manager. We say that T_b is the attacking transaction and T_a is the victim one. FTGreedy obeys the following two simple rules:

1. If $ts_b < ts_a$ or T_a is waiting, then T_b aborts T_a.

2. If $ts_b > ts_a$ and T_a is not waiting, then T_b starts waiting, until T_a commits, aborts, starts waiting, or a timeout of δ_a expires. Afterwards, if T_a starts waiting, see Rule 1; if timeout δ_a has expired, T_b aborts T_a and doubles δ_a.

Figures 4, 5, 6 depict the very fact that FTGreedy performs well in presence of failed transactions, with one exception. FTGreedy seems to perform poorly for `RandomBenchmark` with a high number of failed transactions, but only on the 4-processor machine (see Figure 6). This might be due to the fact that the timeouts of the transactions reached such a high level that they exceeded the length of the benchmark (30 seconds).

Interestingly, FTGreedy performs equally well as Greedy in case no transaction fails in the execution, as shown in Figures 7, 8 and 9. Its performance does not also diverge much from the performance of Karma and Polka.

3. PROPERTIES OF FAULT-TOLERANT GREEDY

3.1 Global Progress

We assume here that the system is eventually synchronous, i.e., that at some point in time (called *global stabilization time* and denoted by GST) there is an upper bound on the time it can take to entirely execute and commit any non-faulty transaction (with no conflict). The bound does not need to be known and, in practice, needs only to hold long enough for a correct (i.e., non-faulty) transaction to commit. Denote by N the number of threads or processes that can execute transactions concurrently.

THEOREM 1. FTGreedy *guarantees that at any time after GST, if there is at least one correct transaction, at least one correct transaction will eventually commit.*

PROOF. By contradiction, let us assume that after GST the correct transaction T with the lowest timestamp never commits. This means that it is always aborted by another transaction. By the algorithm of FTGreedy, T cannot wait for other transactions because it has the lowest timestamp. Therefore, it can be aborted only by a transaction T' in a situation in which T' was waiting for T and its waiting timeout expired. But each time T is aborted, the related timeout is increased (note that the timeout value is stored at T, not at T').

T has the lowest timestamp so it will abort all conflicting transactions. Therefore, if no other transaction performs steps for sufficiently long time, T will commit. Furthermore, as the system is synchronous after GST, there is an upper bound t_{max} on the time it can take to perform the transaction T and commit it. If T is aborted sufficiently many times, the related timeout δ_T will be larger than $(N-1)t_{max}$.

Let us denote by t a point in time after GST in which $\delta_T > (N-1)t_{max}$ and in which T is restarted after being aborted by a transaction T_0. T has to be aborted by a transaction T_1 at latest at time $t_1 \in (t, t + t_{max})$; otherwise, it would commit. T_1 can no longer abort T once again earlier than at time $t_1 + \delta_T > t + (N-1)t_{max}$, because it has to wait for period of δ_T before doing so. For the same reason, T_0 cannot abort T once again before time $t + \delta_T > t + (N-1)t_{max}$. However, another transaction, T_2, can abort T at time $t_2 \in (t_1, t_1 + t_{max})$. But also T_2 has then to wait for $\delta_T > (N-1)t_{max}$ before aborting T once again. We can apply analogous reasoning to transactions $T_3, \ldots, T_{N-2}$. T can be aborted by a transaction T_{N-2} at time $t_{N-2} \in (t_{N-3}, t_{N-3} + t_{max})$. As we have maximum of N concurrent transactions, after t_{N-2}, T can be aborted not earlier than at time $t' \geq t + (N-1)t_{max}$ (recall that we number transactions from 0 and T also counts for N). But $t_{N-2} < t + (N-2)t_{max}$ and T cannot be aborted between t_{N-2} and t'. Therefore, as the time between t_{N-2} and t' is at least t_{max}, T have enough time to commit – a contradiction. □

It is worth noting that after a correct transaction T commits, another one becomes the one with the lowest timestamp and will also eventually commit (unless there are no correct transactions anymore). As the timestamps are unique and monotonically increasing (older transactions have lower timestamps than new ones), the proven property implies global progress, under the stated assumption that

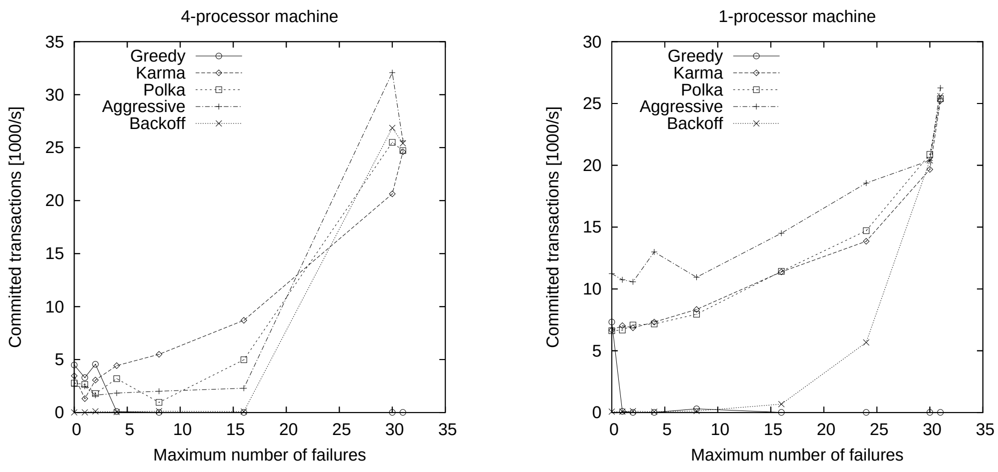

Figure 1: `List` **with failed transactions**

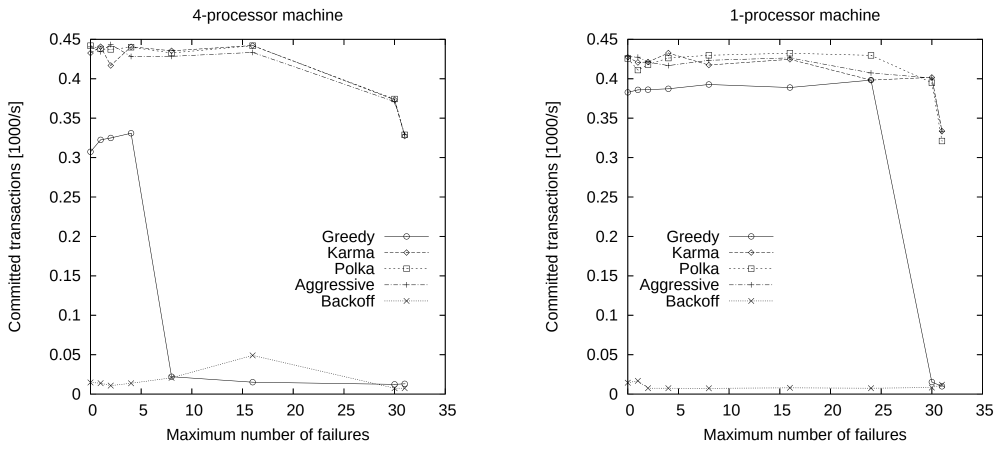

Figure 2: `RBTree` **with failed transactions**

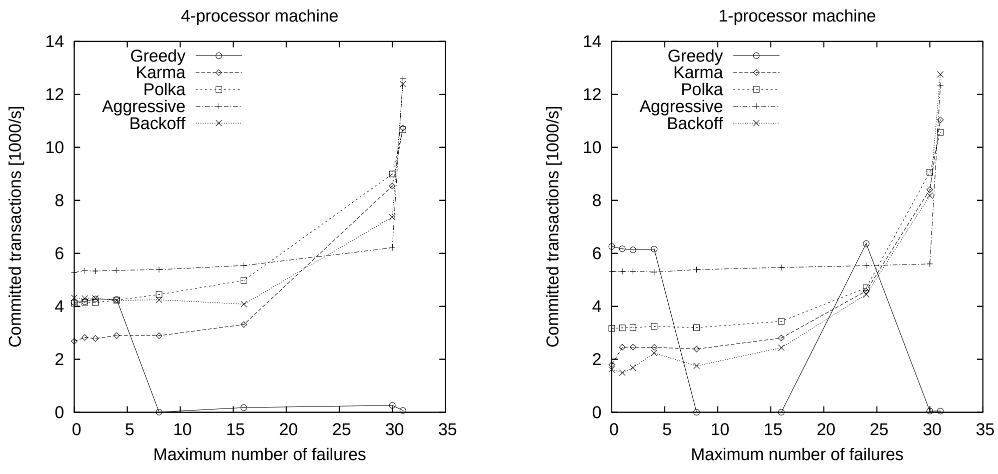

Figure 3: `RandomBenchmark` **with failed transactions**

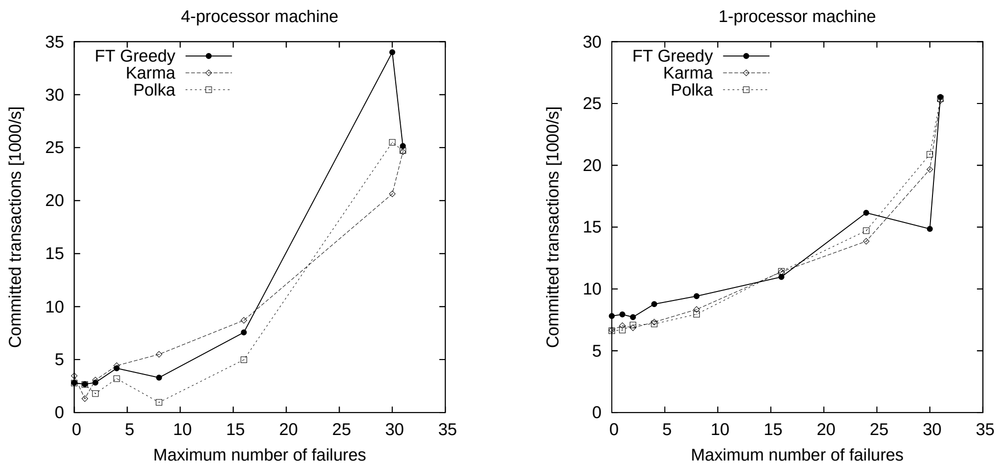

Figure 4: FTGreedy performs well against failures with `List`

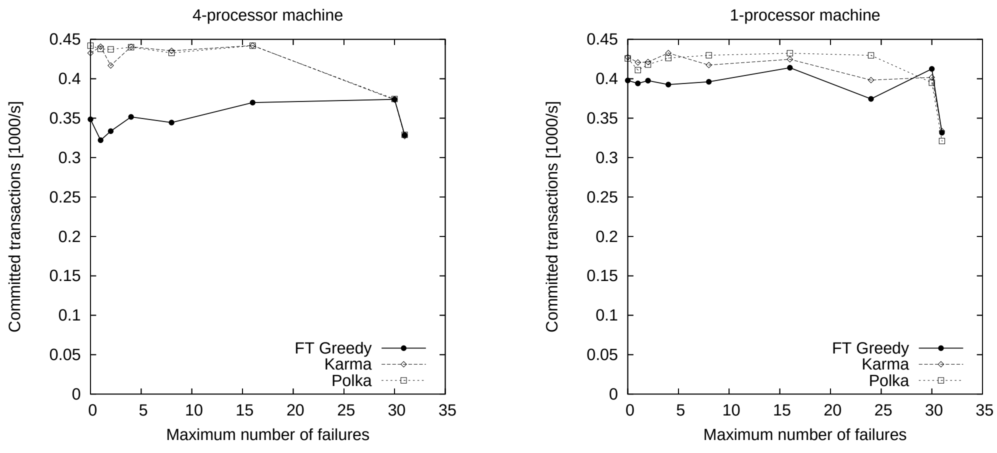

Figure 5: FTGreedy performs well against failures with `RBTree`

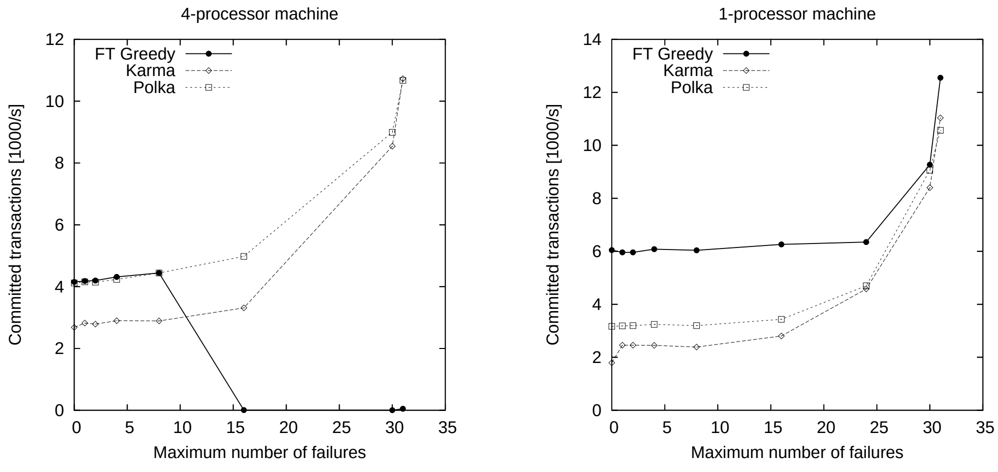

Figure 6: FTGreedy performs sometimes well against failures in `RandomBenchmark`

the system is eventually synchronous. This also means that FTGreedy guarantees *wait-freedom* in a sense that every correct transaction will eventually commit.

3.2 Worst Case Performance

In general, in the periods of asynchrony of the system or when failures occur, it is not possible for FTGreedy to guarantee anything besides the global progress proved in Section 3.1. However, one would like to know, what the contention manager can provide in the most common scenarios in which the system is synchronous from the beginning and no failure occurs. This might seem to be a very optimistic view, but in real systems such assumptions will usually hold most of the time and for sufficiently long periods of time.

In the following proofs we will assume that:

1. All transactions are correct (there are no faulty ones),
2. The system is synchronous from the beginning, i.e., there exists an upper bound t_{max} on the time it can take for a transaction to commit when it runs without conflicts,
3. Transactions are not preemptive.

In FTGreedy a transaction T_i with a timestamp ts_i might be aborted by a transaction T_j with a timestamp $ts_j > ts_i$ only when T_j, after waiting for time δ_i, has suspected T_i of having crashed. However, if T_i is not faulty, its timeout δ_i will double each time it is aborted. Therefore, at some point $\delta_i > t_{max}$. Let us denote by K the maximum number of aborts after which this condition will always hold (as the system is synchronous, K will always exist). It is worth noting that FTGreedy cannot determine that the system is synchronous and has no knowledge of K.

3.2.1 Throughput

We gave in [4] a worst case analysis of the class of contention managers that have the following property, called *pending commit*: at any time, some running transaction will run uninterrupted until it commits (of course, provided that there is at least one running transaction at that time). For this kind of contention managers the following theorem is proved:

THEOREM 2. *The time to complete n concurrent transactions that share s objects is within a factor of $s(s+1)+2$ of the time that would have been taken by an optimal off-line list scheduler.*

This holds under the following assumptions:

1. All the n transactions start at the same time and there are no other transactions in the system,
2. Transactions are not preemptive,
3. There is at least as many processors as transactions.

Although Greedy has the pending commit property, FTGreedy does not and so the proof does not directly hold for it. However, under the synchrony assumptions we make, we can define a *weak pending commit property* in the following way: at any time, some running transaction will abort at most K times and then commit. Let us first prove the following lemma:

LEMMA 3. *FTGreedy has the weak pending commit property.*

PROOF. Let us assume, by contradiction, that at some point in time t there is no transaction that will commit after at most K aborts. Let us take the transaction T_i that at time t has the lowest timestamp ts_i from all the transactions running at time t. Therefore, T_i is aborted more than K times after t. By the algorithm of FTGreedy, T_i can be aborted only by a transaction T_j after T_j waited for the timeout δ_i. However, after K aborts of T_i, by the algorithm, $\delta_i > t_{max}$ and so every transaction has to wait at least t_{max} before it can abort T_i once more. But in this time, by the synchrony assumptions for the system, T_i will commit – a contradiction.

Once T_i commits, all transactions that were waiting for it are woken up (notification mechanism) and some other transaction T_j becomes the one with the lowest timestamp (note that timestamps are unique and monotonically increasing). □

To prove Theorem 2 for contention managers that satisfy the pending commit property, we proceeded in [4] as follows. The transactions are divided into actions. The last action commits and all previous ones abort. The pending commit property guarantees that at any time there is at least one action that will commit. After introducing some additional shared objects that track indirect dependencies between transactions, the schedule obtained with a contention manager and last (commiting) actions can be mapped to a valid schedule for a list scheduler. Then, the rest of the proof becomes analogous to the results of Garey and Graham presented in [2].

Unfortunately, this schema cannot be applied directly when only weak pending commit property is satisfied. Instead, one has to consider the worst case scenario in which every transaction aborts at least K times before it commits. Then we do the mapping using not only the last action, but also the K preceding ones. The resulting *task* has the length of at most $(K+1)\tau_i$, where τ_i is the length of the last action. That is because the transaction with the lowest timestamp does not wait after it is aborted (wrongly suspected of having crashed), but immediately restarts. Using so defined tasks, we can follow the same arguments as used in [4] and prove the following theorem:

THEOREM 4. *Every contention manager that has the weak pending commit property guarantees that the time to complete n concurrent transactions that share s objects is within a factor of $(K+1)[s(s+1)+2]$ of the time that would have been taken by an optimal off-line list scheduler.*

3.2.2 Time to Commit

For many applications an important metric is a maximum time between a transaction is started and committed, even in presence of conflicts with other transactions. Let us call it a *time to commit*. Obviously, it is bounded only when the time every transaction needs to commit *without* conflicts (i.e., without being aborted) is bounded. For Greedy this a sufficient requirement [4]. However, for FTGreedy we have to assume additionally, what has already been mentioned, that we have no failures[3], the system is not preemptive and is

[3]Note that for Greedy it was implicitly assumed that there are no failures, as this contention manager is not fault-tolerant.

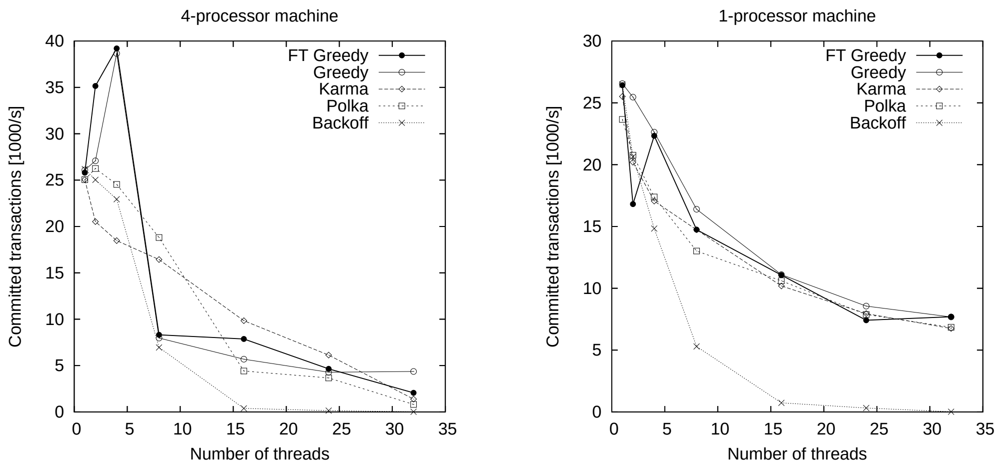

Figure 7: FTGreedy **and** Greedy **perform equally well where there is no failure (**`List` **application)**

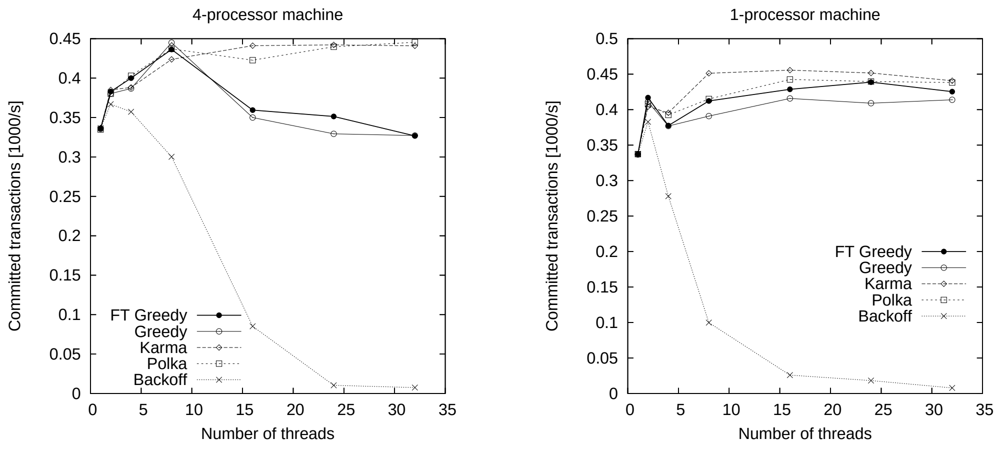

Figure 8: FTGreedy **and** Greedy **perform equally well where there is no failure (**`RBTree` **application)**

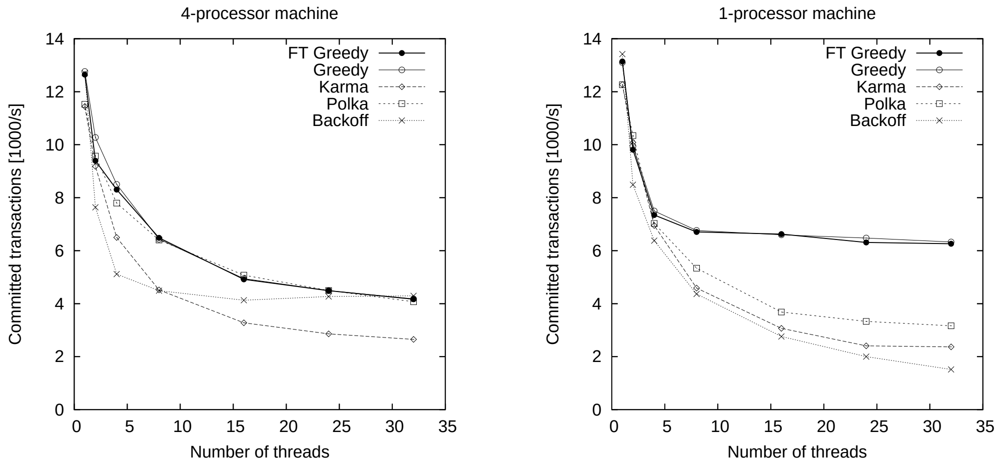

Figure 9: FTGreedy **and** Greedy **perform equally well where there is no failure (**`RandomBenchmark` **application)**

synchronous from the beginning. Under these assumptions we can prove the following theorem:

THEOREM 5. FTGreedy *guarantees that every correct transaction that starts at time t will commit at latest at time $t + M(K+1)t_{max}$, where M is the number of threads (K and t_{max} are defined in Section 3.2).*

PROOF. By Lemma 3, FTGreedy has a weak pending commit property. This means, that at each time t' there is a transaction T_j that will commit after at most K aborts from this point in time. The maximum time it can take for a transaction T_j to abort K times and commit is $(K+1)t_{max}$.

Now let us assume that at time t there is a transaction T_i with timestamp ts_i. As there are M threads, each of which can run a single transaction, there can be maximum $M-1$ running transactions with timestamps lower than ts_i at time t. Furthermore, by the algorithm, every transaction that will start after t will have a timestamp greater than ts_i.

Let us denote by $T'_1, T'_2, \ldots, T'_k$, $k < M$, the transactions that are running at time t and have the timestamps $ts'_1, ts'_2, \ldots, ts'_k$, respectively, such that $ts'_1 < ts'_2 < \ldots < ts'_k < ts_i$. By weak pending commit property, T'_1 will have to commit at latest at time $t + (K+1)t_{max}$. Then T'_2 will have the lowest timestamp from all the running transactions and will commit by the time $t + 2(K+1)t_{max}$. Analogously $T'_3, \ldots, T'_k$. The transaction T'_k will commit at latest at time $t + k(K+1)t_{max}$. Then T_i will become the transaction with the lowest timestamp and so it will commit at latest at time $t + (k+1)(K+1)t_{max} \le t + M(K+1)t_{max}$. This finishes the proof. $\square$

4. CONCLUDING REMARKS

One might consider a simpler variant of FTGreedy that only uses a single timeout δ for all transactions performed by a single thread and in which the attacker's timeout (not the victim's one) is taken when the attacker transaction has to wait. This variant of FTGreedy, which is simpler from an implementation perspective, degrades, however, significantly with respect to the implementation we considered.

In some operating systems, we might consider a notification mechanism to indicate that a transaction is waiting because of a page fault [1]. Clearly, such a notification might directly lead to aborting such a transaction, transforming a system with paging into one without it (from the perspective of contention management).

To conclude, it is important to notice that we do not claim FTGreedy to be the universal contention manager. Even if it does perform well in many cases we considered, it does not, for instance, in the case of the RandomBenchmark (with a high number of failed transactions), and does not deal with transactions that might execute forever (e.g., those with infinite loops). Different applications might require different contention managers and the same application might even sometimes benefit from mixing contention managers [3]. Identifying a manager like FTGreedy helps, however, better understand the space of contention management, in particular in our quest of combining worst case guarantees with good average case performance. Such understanding will help choosing the right contention manager for a specific situation and come up with new managers.

5. ACKNOWLEDGEMENTS

We are very grateful to Marc Moir, Victor Luchangco, Nir Shavit, and Bill Scherer for useful discussions during PODC 2005 in Las Vegas.

6. REFERENCES

[1] B. Bershad, D. Redell, and J. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS'92: Proceedings of the Fifth Architectural Support for Programming Languages and Operating Systems*, pages 223–233, October 1992.

[2] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.*, 4(2):187–200, 1975.

[3] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *DISC'05: Proceedings of the nineteenth International Symposium on Distributed Computing*, September 2005.

[4] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC'05: Proceedings of the twenty-fourth ACM Annual Symposium on Principles of Distributed Computing*, July 2005.

[5] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC'03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.

[6] M. Research. *C# software transactional memory.* Available at: `http://research.microsoft.com/research/downloads/default.aspx`.

[7] W. Scherer and M. Scott. *Private Communication.*

[8] W. Scherer and M. Scott. Contention management in dynamic software transactional memory. In *Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[9] W. Scherer and M. Scott. Advanced contention management for dynamic software transactional memory. In *PODC'05: Proceedings of the twenty-fourth Symposium on Principles of Distributed Computing*, July 2005.