Transactional Memory: How to Perform Load Adaption in a Simple And Distributed Manner

David Hasenfratz, Johannes Schneider, Roger Wattenhofer
Computer Engineering and Networks Laboratory,
ETH Zurich, 8092 Zurich, Switzerland,
hdavid@ee.ethz.ch,{jschneid, wattenhofer}@tik.ee.ethz.ch

ABSTRACT

We analyze and present different strategies to adapt the load in transactional memory systems based on contention. Our experimental results show a substantial overall improvement for our best performing strategies QuickAdapter *and* AbortBackoff *on the throughput compared to the best existing contention management policies (without load adaption). Opposed to prior work our load adapting schemes are simple and fully distributed, while maintaining the same throughput rate. Our theoretical analysis gives insights into the usefulness of load adaption schemes. We show a constant expected speed-up compared to systems without load adaption in several important scenarios, but also illustrate that the worst-case behavior can result in an exponential increase in the running time.*

KEYWORDS: algorithms, scheduling, transactions, transactional memory, concurrency control, contention management

1. INTRODUCTION

The era of computers with a single central processing unit (CPU) seems to end. Practically any new notebook or desktop computer is equipped at least with a dual-core processor. Quad-core chips are widely available and oct-core chips are already being tested. The hardware development is likely to continue at this pace. In contrast, a large fraction of the available software today is not designed to make full use out of these mighty processors. To be able to fully utilize them, software development must focus more on issues related to parallel programming. Programming with multiple threads is demanding; It is considered difficult and time-consuming to write efficient and correct parallel code. Major defects such as deadlocks or race conditions arise from locking shared resources (such as data objects) wrongly.

Transactional memory promises to make parallel programming easier. Instead of specifying exactly when to lock and unlock each shared resource, a programmer only has to define a section of code as transaction. The transactional memory system should guarantee correctness and efficiency. Despite intensive research the success of transactional memory has yet to come. One of the main issues is performance. A key influence factor on the throughput of a transactional memory system is the behavior when conflicts arise. A conflict occurs when a transaction demands a resource that is in use by another transaction. The system can resolve the conflict by aborting the transaction holding (or demanding) the resource or by delaying the transaction requesting the resource. Wrong decisions have a dramatic influence on the overall performance, i.e. the system might abort a long running transaction just a split second before it commits, thereby wasting its entire work. The system might also delay a short transaction for a long time which itself uses a lot of resources and thereby blocks other transactions. Almost all transactional memory systems assign the task of resolving conflicts to a specific module called contention manager. In the past a number of policies have been proposed and evaluated, however, none of them has performed really well for all tested applications. The ones that do perform somewhat well, often lack progress guarantees and thus might suffer from starvation or livelock. Furthermore, often non-trivial tuning of the system and benchmark specific parameters are required to achieve good performance. A general idea to improve performance

for many kinds of systems is to adapt the load of the systems, since frequently throughput peaks at a specific load. Thus, it might be better to leave some of the resources, such as network bandwidth or processor cores unused, to reduce the coordination effort and prevent harmful interaction, i.e. packet collisions and conflicts of transactions, respectively. The approach of load control based on the contention level has been investigated for transactional memory systems in prior work. However, as we shall see in the related work section, the attempts so far are complex and non-distributed. Any system relying on a central scheduler will eventually face scalability issues. In particular, complexity becomes an issue, if an actual implementation is to be carried out in hardware. Our *QuickAdapter* and *AbortBackoff* algorithms are simple and fully distributed. Both improve on the throughput (the number of completed transactions per time) of existing policies without load adaption and can keep up with more complex implementations for load regulation.

Apart from our experimental evaluation we also compare different strategies through analysis. In principle any strategy chooses a subset of all available transactions to run, postponing the others. Clearly, a good strategy guesses/predicts a large set of non-conflicting transactions and ideally, only delays those that would face a conflict anyway. Thus, a strategy might assume conflicts among transactions it has not observed. In general the effectiveness of a strategy depends on the topology of the so called *conflict graph*, which has transactions as nodes and edges between nodes, where a conflict might occur. We look at conflict graphs of typical benchmarks such as linked lists and analyze the throughput of several strategies. If the load adapting strategy correctly guesses the conflicts (or equivalently the topology of the graph), the expected speed-up can be a constant. However, if, for instance, a dense conflict graph is assumed, but the true conflict graph is sparse, the worst-case behavior can be exponentially slower compared to a scheme without load adaption.

2. RELATED WORK

Transactional memory came up about 20 years ago[11, 7]. It was extended to dynamic data structures and the use of a contention manager as an independent module was suggested [6]. Since then, many systems have been proposed and large advances have been made, e.g. DSTM2 [5]. A variety of contention managers have been proposed [9, 8, 10, 3] and and evaluated theoretically [10] and experimentally in [9]. From a theoretical point of view picking random priorities seems valuable to prevent chains of waiting (or aborting) transactions. In practical systems, no single contention manager outperformed all the others, but a policy termed *Polka* yielded good overall results on many benchmarks. However, it did not do that well in our evaluation and also in [1]. Timestamping managers (e.g. [3, 9]) seem more robust and yield good results in a number of scenarios. The idea of mixing contention managers has been implemented in [4]. The evaluation was limited to red black trees and the reported results were good for high contention. However, for low contention the introduced overhead might slow down the system considerably. In [12] the idea of load balancing based on contention has been investigated. A thread approximates the current contention based on the number of previous commits and aborts of transactions it has executed. Recent aborts and commits have a larger influence. When a transaction starts, it checks whether its contention approximation is beyond a threshold and resorts to a central scheduler that maintains a queue of transactions. The first element in the queue can execute until commit and is then removed. The evaluation was done using an HTM and an STM system on similar benchmarks as in this paper, e.g. RBTree, LinkedList, LFUCache. With the load adaption scheme in [12] as well as with our system the throughput is kept high, though (still) with some decrease with increasing threads.[1] Though the system of [12] was designed to be simple, our system is simpler, entirely decentral and does not rely heavily on setting parameters correctly. In the spirit of [12], in [2] a system was proposed that also serializes transactions. Initially, a central dispatcher assigns a transaction to a core. Each core contains a queue of transactions. Suppose a transaction A running on core 1 is aborted due to a transaction running on core 2. In this case transaction A is appended to the queue of 1 and the next transaction in the queue of core 1 is executed. Unfortunately, the results cannot be directly related to our system and [12], since the evaluation was done using a different benchmark. However, not surprisingly their implementation also outperforms the compared non-load adapting system. Still, a central instance sooner or later becomes a bottleneck.

3. DISTRIBUTED LOAD ADAPTING POLICIES

A requirement for our approach is to maintain scalability. Thus, for instance, any kind of central scheduler or dispatcher is not acceptable. However, a thread may collect information about its (executed) transactions and furthermore, if two transactions of two different threads conflict, i.e. share data, then also the aggregated information of each thread might be shared (until one of the transactions commits). We investigate two different schemes from both a practical and theoretical perspective. The first scheme uses an exponential backoff based on the number of aborts, i.e.

[1] In [12], the benchmarks were only performed on a 2-way (dual core) SMP machine, whereas we tested on a 16 core machine.

before a restart a transaction must wait a timespan exponential in the number of its aborts. In the second scheme, load adaption is performed through delaying a transaction until some other (conflicting) transaction(s) have committed.

4. THEORETICAL INVESTIGATION

4.1. Model

A set of *transactions* $S_T := \{T_1, ..., T_n\}$ are executed on n *processors (or cores)* $P_1, ..., P_n$. Transaction T_i executes on processor P_i until it committed. The *duration* of transaction T is assumed to be fixed and is denoted by t_T.[2] It refers to the time T executes until commit without facing a conflict (or equivalently, without interruption). In one time unit one instruction of a transaction is executed. An instruction can be a read or write or some arbitrary computation. A value written by a transaction T takes effect for other transactions only after T commits. A transaction either successfully finishes with a commit of unsuccessfully with an abort anytime. A transaction commits after executing all instructions and acquiring all modified (written) resources exclusively. A read of transaction A of resource R is *visible*, if another transaction B accessing R is able to detect that A has already read R. If a transaction A writes to resource R it conflicts with any other transaction reading or writing R. Furthermore, if a transaction A reads a resource R it conflicts with any other transaction writing R. A resource can be read in parallel by arbitrarily many transactions. A contention manager decides how to resolve the conflict. It can make a transaction wait or abort or assist the other transaction.[3] If a transaction gets aborted due to a conflict, it restores the values of all modified resources, frees its resources, might wait for a while and restarts from scratch with its first operation. Usually *conflicts* are handled in a *lazy* or *eager* way, i.e., a transaction notices a conflict once it actually occurs or once it tries to commit. Due to the limit of space we only consider eager conflict handling and visible reads.

[2]If an adversary can modify the duration of a transaction arbitrarily during the execution of the algorithm, the competitive ratio of any online algorithm is unbounded: Assume two transactions T_0 and T_1 face a conflict and an algorithm decides to let T_0 wait (or abort). The adversary could make the opposite decision and let T_0 proceed such that it commits at time t_0. Then it sets the execution time T_0 to infinity, i.e., $t_{T_0} = \infty$ after t_0. Since in the schedule produced by the online algorithm, transaction T_0 commits after t_0 its execution time is unbounded. Therefore we assume that t_T is fixed for all transactions T. In case the running time depends on the state of the resources and therefore the duration varied by a factor of c, the guarantees for our algorithms would worsen only by the same factor c.

[3]We do not explicitly consider the third option, since it is not used in state-of-the art systems.

4.2. Load Adapting Approaches

The first approach uses an exponential backoff scheme named *AbortBackoff*. A transaction maintains an abort counter, which is 0 initially and is incremented after every abort. The counter is used as a transaction's priority. For any conflict one transaction is aborted (i.e. no waiting). In case two transactions with the same priority conflict, an arbitrary one proceeds. If a transaction is aborted, it increments its waiting exponent i and waits for a random time interval in [0,2^i]. The second approach deals with different variants of (deterministically) serializing conflicting transactions. If two transactions are serialized, then at no future point in time they will run in parallel. A transaction keeps track of a set of (possibly) conflicting transactions. The set contains all transactions with which it has ever faced a conflict or with which it assumes to have a conflict. After an abort, a transaction is only allowed to restart if none of its conflicting transactions is executing. If a transaction C, having blocked two transactions A with conflict set $\{B, C\}$ and B with set $\{A, C\}$, then either A or B might restart. We choose each with probability $\frac{1}{2}$. In general, if C has blocked x transactions, one of the waiting transactions restarts. Each has a chance of $\frac{1}{x}$ to be selected. In our first conservative policy *SerializeFacedConflicts* a transaction only assumes to conflict with a job it actually had a conflict with. In our second policy *SerializeAllHopConflicts*, a transaction A having faced a conflict with B assumes that it also conflicts with all transactions in the conflict set of B. The set of conflicting transactions is always kept up to date, if transaction A conflicted with B and later B with C, then C will also be in A's set of conflicting transactions.

We consider the above strategies as well as the naive strategy where a transaction restarts without delay. For the analysis it is crucial what type of contention management strategy is used. We employ two contention management strategies covering a wide set of available managers. In the first strategy, each transaction has a random priority in $[1, n]$ such that no transactions get the same priority. A transaction keeps the same priority until commit. In the second strategy, a transaction's priority equals the net-executing time of a transaction. The first scheme covers all contention managers, where the priority calculation is done in a way that is not (or weakly) related to the actual work performed by a transaction. The second scheme is a representative of a contention manager estimating a transaction's work.

Unfortunately, the conflict graph is dynamic and depends on many factors such as what resources are needed by a transaction and from what time on the resources are needed etc. Due to these difficulties we focus on extreme cases of conflict graphs. In the first scenario all transactions want

the same resource, i.e. the conflict graph is a clique. We model a shared counter, where we assume that a transaction is very short and all transactions attempt to access the resource concurrently. We also consider a linked list and look at the expected length of the schedule. In the second scenario, all transactions want distinct resources, i.e the conflict graph is a tree and thus sparse.

4.3. Moderate Parallelism – Conflict on Start-up

Assume that all n transactions start at the same time and want to write to the same memory cell directly after their start. Thus the conflict graph is a clique. Such a situation occurs, for instance, if multiple transactions want to concurrently increment a shared counter. Our primary concern is the expected delay due to transactions with low priority holding resources also wanted by transactions of higher priority. This happens since all transactions access the resource concurrently and have the same chance to acquire it – independent of their priority. The transactions of higher priority are delayed since they must abort the resource holding transaction. We assume that it takes 1 time unit to abort a transaction and to acquire its resource. Furthermore, if multiple transactions try to get a resource concurrently, a random one gets it.

Proposition 1. *For immediate restart the expected time span until all transactions committed is* $n \cdot t_T + \Omega(n \cdot \log n)$.

Proof. Assume that the resource is available (i.e. not accessed) and x transactions try to access it, then a random transaction T gets it. Once T got the resource, all transactions face a conflict with T. If T does not have highest priority, it gets aborted by some (random) transaction U with higher priority. Again transaction U is aborted, if its priority is not highest. The expected delay until the resource with highest priority obtained the resource can be computed through a recursive formula. The expected delay for one transaction is 0. The expected delay for two transactions is $\frac{1}{2}$, since we assumed that an unsuccessful try to acquire a resource delays a transaction by 1 time unit and the probability that the transaction with smaller priority gets the resource is $\frac{1}{2}$. Given x transactions the expected delay until the transaction with largest priority has the resource can be computed through the following recursion:

$$E[x] = (1 - \frac{1}{x}) + \frac{1}{x-1} \sum_{i=1}^{x-1} E[i]$$

The first term $1 - \frac{1}{x}$ denotes the chance that a transaction not having highest priority gets the resource given x transactions try. The second term states that any of the $x - 1$ remaining transactions (of higher priority) has the same chance to be chosen. Assume $E[x] \geq \ln x$.

$$E[x] = (1-\frac{1}{x}) + \frac{1}{x-1} \sum_{i=1}^{x-1} \ln i = (1-\frac{1}{x}) + \frac{1}{x-1} \ln((x-1)!)$$

Using Stirling's Formula we get (for large x):

$$E[x] = (1 - \frac{1}{x}) + \frac{1}{x-1} \ln((x-1)!)$$

$$= (1 - \frac{1}{x}) + \frac{1}{x-1} \ln((\frac{x-1}{e})^{x-1} \cdot c_0 \cdot \sqrt{x-1})$$

$$= 1 - \frac{1}{x} + \ln \frac{x-1}{e} + \frac{\ln(c_0 \cdot \sqrt{x-1})}{x-1} - \frac{1}{x}$$

$$= \ln(x-1) + \frac{\ln(x-1)}{4 \cdot (x-1)} \geq \ln x$$

The last step can be seen as follows. We have that $\frac{\ln(x-1)}{4 \cdot (x-1)} - \frac{1}{x} > \frac{1}{x}$ for large x and also since $\ln x$ is concave, it is bounded by a tangent at an arbitrary position. In particular, $\ln x \leq \ln(x-1) + \frac{d(\ln(x))}{dx} \cdot 1 = \ln(x-1) + \frac{1}{x}$. Assume $E[x] \geq 2 \cdot \ln x$. The derivation is analog to the previous case.

$$E[x] = (1 - \frac{1}{x}) + \frac{1}{x-1} \sum_{i=1}^{x-1} 2 \cdot \ln i$$

$$= 1 - \frac{1}{x} + 2 \cdot \ln \frac{x-1}{e} + 2 \cdot \frac{\ln(c_0 \cdot \sqrt{x-1})}{x-1} - \frac{1}{x}$$

$$= 2 \cdot \ln(x-1) - 1 + \frac{\ln(x-1)}{2 \cdot (x-1)} - \frac{1}{x} \leq 2 \cdot \ln x$$

The last step can be seen as follows. We have that $\frac{\ln(x-1)}{2 \cdot (x-1)} - \frac{1}{x} < 1$. Thus we have $\ln x \leq E[x] \leq 2 \cdot \ln x$. Initially, n transactions try to access the resource. After the first commit there are $n - 1$ left a.s.o. Therefore the total time until all transactions committed is bounded by $\sum_{i=1}^{n} \ln i$. Using Stirling's Formula (for large n) as before, we have $\sum_{i=1}^{n} \ln i = \ln n! = O(n \cdot \log n)$. □

Proposition 2. *For* AbortBackoff *the expected time span until all transactions committed is* $n \cdot t_T \cdot 2^{O(\sqrt{\log n})}$.

Proof. Assume we have n jobs left in the system and all have priority (at least) $\log(8n \cdot t_T)$, i.e. each job aborted at least $\log(8n \cdot t_T)$ times. This must be the case after time $8nt_T$, since a transaction with priority i waits at most time 2^i before it restarts and the priority i corresponds to the number of aborts, thus the total time for $\log(8n \cdot t_T)$ aborts is given by $\sum_{i=0}^{\log(8n \cdot t_T)-1} 2^i \leq 8nt_T$.
If a transaction does not face a conflict, it commits. A single transaction A takes time t_T and if another transaction B starts either while A is running or $t_T - \epsilon$ (for some $\epsilon > 0$) before A then A and B face a conflict. Thus, if

each transaction runs once in an interval of duration $4nt_T$, $n-1$ transactions together occupy at most an interval of length $2(n-1)\cdot t_T$. In other words a transaction has a chance of less than $1/2$ to face a conflict, if it starts at an arbitrary point in time during this interval. If instead of all $n-1$ transactions only a fraction a of all n transactions is active, the probability for a transaction to face a conflict becomes $a2n\cdot t_T/(4n\cdot t_T) = a/2$. More generally, after $\log(8n\cdot t_T)+x$ aborts, i.e. for a waiting interval of length $4n\cdot t_T\cdot 2^x$ and a fraction a of active nodes, the chance to abort becomes $a/2/2^x < a/2^x$. Assume that for $x=0$, i.e. up to $\log(8n\cdot t_T)$ aborts all nodes are in the system, i.e. $a(0)=1$. Then, using the above relation $a/2^x$ for an interval $[8nt2^{x-1}, 8nt2^x]$ we have $a(1)=a(0)/2$, $a(2)=a(1)/2^2$, a.s.o. remain in the system in expectation. More generally, $a(x+1)\leq a(x)/2^{x+1}$. As long as $a(x)\geq c_0\cdot\log n/n$ for arbitrary constant c_0, using a Chernoff bound the probability that $a(x+1)\leq a(x)\cdot(3/4)^{x+1}$ is at least $1-1/n^{c_1}$ for some constant $c_1(c_0)$. Thus, $a(x+1)\leq(3/4)^{\sum_{i=1}^{x} i} = 1/2^{O(x^2)}$. With $x=O(\sqrt{\log n})$ we have that $a(x) < c_0\log n/n$ less than $c_0\log n$ transactions are active, i.e. all others committed. For the remaining transactions the chance to abort within a time interval $[2^{x-1}8nt, 2^x 8nt]$ with $x\in 2^{O(\sqrt{\log n})}$ is $\frac{\log n\cdot t_T}{n\cdot t_T\cdot 2^{O(\sqrt{\log n})}}\leq 1/2^{O(\sqrt{\log n})}$. Thus when looking at $O(\sqrt{\log n})$ additional intervals the chance becomes $(1/2^{O(\sqrt{\log n})})^{O(\sqrt{\log n})} < 1/n^{c_2}$ for some arbitrary constant c_2. □

Proposition 3. *For the* SerializeAllHopConflicts *policy the expected time span until all transactions committed is* $n\cdot t_T+1$.

Proof. Initially, all n transactions try to access the available resource and a random transaction T gets it. All transactions conflict with T and also assume that they conflict with each other. Therefore from then onwards, no conflicts occur. □

Proposition 4. *For* SerializeFacedConflicts *the expected time span until all transactions committed is* $n\cdot t_T + \Theta(n)$.

Proof. Initially, all n transactions try to access the (available) resource and a random transaction T gets it. All transactions conflict with T and add T to their sets of conflicting transactions. If T runs again it does not face a conflict. Therefore after n aborts all jobs have all others in their conflict set and the delay is $O(n)$. Given n transactions try to access a resource we expect $\log n$ aborts to happen before the transaction having highest priority (and thus running until commit) obtains the resource (see proof of Proposition 5). Thus, after $\frac{n}{c}$ aborts for some constant c, we expect (still) only $\frac{n}{\log n\cdot c}$ commits and the expected delay is $\Omega(n)$. □

4.4. Moderate Parallelism – Conflict at Arbitrary Time

Consider a typical benchmark such as a linked list where transactions either insert, delete or find a value in a list or traverse the list to compute some (aggregate) value. Each transaction keeps the entire read set until it committed, i.e. a transaction A considers a (long) traversed object O as read and conflicts with any transaction B modifying O. In some cases non-written objects might be releasable before commit, but this depends on the semantics of the transaction and, generally, has to be specified by the programmer. Therefore, it would make life for complex for the software engineer and we do not consider it. We focus on operations like inserts and deletes, i.e. after an arbitrary number of read operations an object is modified. Thus, the dense conflict graph is dense, since all transactions potentially conflict with each other. Still, a potential conflict might not necessarily occur. For example, consider two transactions A and B, both performing some write operation towards the end of the list. If A has started way ahead of B, then at the time A is committing, B will still be traversing the list and will not have accessed the element modified by A. Thus, A and B will not conflict. Furthermore, opposed to the shared counter example, in such a scenario a transaction does not necessarily face the conflict directly after start-up due to the first accessed resource (i.e. the head of the list) but more likely, at some later point in time. For simplicity, let us assume that all transactions conflict within time $[\frac{t_T}{c_0}, \frac{(c_0-1)\cdot t_T}{c_0}]$. This is not much of a restriction, since clearly the vast majority of transactions (more precisely a fraction $1-\frac{1}{c_0}$) is expected to face a conflict within time $[\frac{t_T}{c_0}, \frac{(c_0-1)\cdot t_T}{c_0}]$. We assume that all transactions start randomly within time $[0, t_T]$.

Proposition 5. *For immediate restart the expected time span until all transactions committed is* $\Theta(n\cdot t_T)$.

Proof. After time t_T all transactions have started and within time $2t_T$ at least one transaction – say A – has committed. The time until the transaction B with highest probability commits is at least $\frac{t_T}{c_0}$. The same holds for the transaction C of third highest overall probability. Thus, the time until all n transactions committed adds up to $\Theta(n\cdot t_T)$. □

For *AbortBackoff* the expected time span until all transactions committed is $O(n\cdot t_T\cdot 2^{O(\sqrt{\log n})})$ using an analogous analysis as in the proof of Proposition 2. For the policies *SerializeFacedConflicts* and *SerializeAllHopConflicts* the expected time span until all transactions committed is $\Theta(n\cdot t_T)$, since all transactions execute sequentially.

4.5. Substantial Parallelism

It is not surprising that the serialization policy *SerializeAllHopConflicts* works well, when we consider a clique, since the policy assumes the graph to be a clique. Given that an adversary, maximizing the running time of the policy, can choose transactions priorities and their starting time, it is not hard to construct an example, where immediately restarting is exponentially faster than *SerializeAllHopConflicts*. Therefore, from a worst case perspective, serialization might be very bad. Due to the randomization the backoff scheme is more robust in such cases.

Proposition 6. *For the* SerializeAllHopConflicts *policy the expected time span until all transactions committed is O($n \cdot t_T$) for d-ary tree conflict graphs of logarithmic height and O($\log n \cdot t_T$) for immediate restart and* SerializeFacedConflicts*.*

Proof. For immediate restart consider an arbitrary node v_0 and look at all paths S_P with $P = (v_0, v_1, ..., v_x) \in S_P$ of nodes of increasing priority in the conflict graphs. Look at an arbitrary longest path $P \in S_P$. For any such path $P = (v_0, v_1, ..., v_x) \in S$ holds that the transaction v_x has maximum priority among all its neighbors and thus commits within time t_T. Thus any longest path reduces by 1 in length within time t_T. Since the graph is a tree of height $\log_d n$, i.e. for the number of nodes holds $d^{\log_d n} = n$, the length x of any path is bounded by $O(\log n)$ and the claim follows. For *SerializeFacedConflicts* we have that for a transaction v_0 itself or a neighbor is executing. Therefore, overall at least a fraction $1/d$ of transactions is running and within time t_T they either commit or face a conflict and abort. Thus, after time $d \cdot t_T$ any transaction is aware of all its conflicting neighbors and is not scheduled together with them again. However, any transaction always has at least one neighbor that is executing, i.e. a maximal independent set of transactions is scheduled and commits. Thus, the total time is O($d \cdot t_T$), which is O($\log n \cdot t_T$) since the tree is of logarithmic height $\log_d n$. For *SerializeAllHopConflicts* we assume that all leaves have lower priority than their parents. Furthermore, we make a node first conflicts with its children. More precisely, assume all transactions start concurrently and a transaction conflicts with up to d other transactions. Assume that the root acquires all its resources within time interval $]t_T - d, t_T]$. All children of the root acquire their resources within time interval $]t_T - 2d, t_T - d]$. In general, a node at level i of the tree gets its resources within time $]t_T - id, t_T - (i-1)d]$. Thus when the root transaction R commits, all its neighboring children $N(R)$ must have faced a conflict and have got aborted by R. In general, before a transaction $T \in N(T_P)$ conflicts with its parent T_P, it aborts all its children $N(T) \setminus T_P$. Thus, the leafs are aborted first and the children of the root at last. Therefore, within time t_T all transactions are assumed to conflict with each other and are executed sequentially, resulting in a running time of $n \cdot t_T$. □

For the *AbortBackoff* policy the expected time span until all transactions committed is O($d \cdot t_T \cdot 2^{O(\sqrt{\log n})}$) for d-ary tree conflict graphs of logarithmic height. The proof is analogous to Proposition 2.

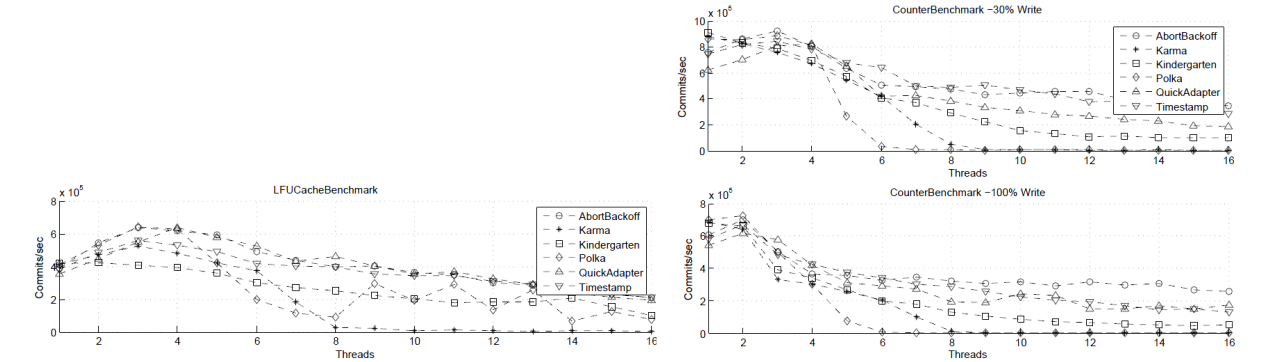
5. PRACTICAL INVESTIGATION

5.1. Contention Management Policies

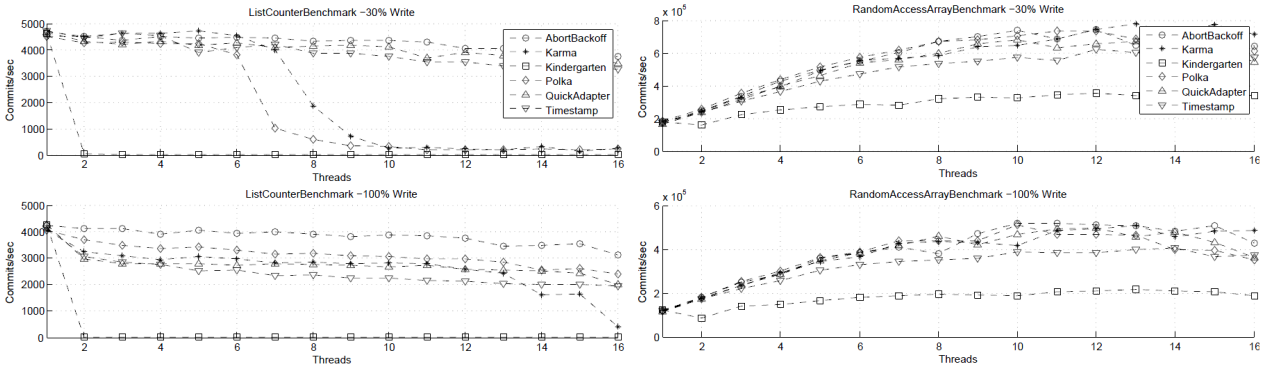
The policies *SerializeFacedConflicts* and *SerializeAllHopConflicts* ignored the overhead of keeping track of conflicts among transactions. In practice, it turned out that logging each conflict causes too much overhead in many scenarios. That is why, we derived a new serialization technique called *QuickAdapter*. It does not come with a priority calculation scheme. For the implementation we used the timestamp manager. Using time as priority results in the same theoretical properties as *SerializeAllHopConflicts*. In particular, deadlock- and livelock-freedom, since the oldest transaction runs without interruption until commit. According to [10] from a theoretical (worst case) perspective assigning random priorities yields better results. Still, in practice we found that the choice of the manager is of secondary importance. Every transaction has a flag which is set if it is not allowed to (re)start.[4] If a transaction gets aborted it sets its flag and does not restart. A committed transaction selects one of the flags and unsets it. For the implementation we chose an array (of flags), which equals the length of the maximum number of transactions. We investigated two variants. For the *QuickAdapter* each thread maintains a counter and whenever a thread commits it increments the counter and unsets the flag at the position in the array given by counter modulo array length. In case contention is very high and most transactions are aborted, any committing transaction has a high chance to restart a waiting transaction. But in such a situation it might be better to be more restrictive and rather not activate another transaction on commit. *SmartQuickAdapter* accounts for this and looks at the status of two (random) transactions. In case both are active, it selects a flag and unsets it (if it is set). Clearly, the higher contention, the more transactions are aborted and the smaller the chance for a committing transaction to reactivate an aborted transaction.

For the *AbortBackoff* manager the priority is determined by the number of aborts of a transaction. In case two transactions have the same priority, the one that runs on the thread

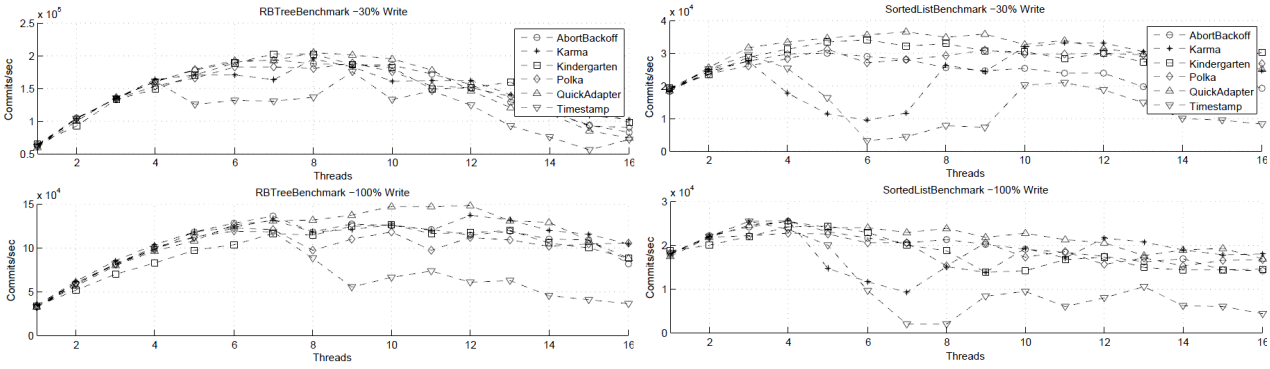
[4] If there are few commits, a single transaction having a set flag might wait for a long time until restart. Therefore one might consider adding a maximum waiting duration until a transaction restarts or check from time to time if there are any active transactions. Usually commits are frequent and, thus, this is not an issue.



(a) LFUCache (b) Counter



(c) ListCounter (d) RandomAccessArray



(e) RBTree (f) SortedList

Figure 1. Benchmarks: For 0% writes most policies behave similar since they introduce almost no overhead. For 60% writes the results lie (as expected) in between 30% and 100%.

with smaller identifier is aborted. For any conflict the transaction with smaller priority gets aborted. Before an aborted transaction can restart it has to wait a period which grows exponentially (by a factor of 4) with the number of times the transaction already got aborted. Each transaction starts with 0 aborts. We also evaluated a scheme *RememberingBackoff*, where a transaction carries over the number of aborts minus 1 of the previous transaction (executed by the thread).

5.2. Experimental Results

The benchmarks were executed on a system with four Quad-Core Opteron 8350 processors running at a speed of 2 GHz. The DSTM2.1 Java library was used, compiled with Sun's Java 1.6 HotSpot JVM. We present experimental results for the described contention managers on six different benchmarks. Five of them have been used already in prior work of Scherer at al. [9]. The sixth benchmark, RandomAccessArray, works on an array with 255 integers. The write operation chooses a random field $i \in [0, 254]$ and alternately increments or decrements the element and its eight neighbors $j \in [i + 1, i + 8]$. The read operation reads those elements. Every benchmark represents the average throughput of three runs with the same configuration for a duration of 10 seconds. Except LFUCache all the benchmarks were tested among different contention levels. Low contention was achieved through a write to read ratio of 0% and 30%, middle and high contention with ratios of 60% and 100%. The throughput was measured with 1 up to a maximum of 16 active threads.

Figure 1 shows the experimental results on the six benchmarks and several contention management policies (see [9]). In the plots we only show the *QuickAdapter* and not the *SmartQuickAdapter* scheme. Overall both perform similarly. *AbortBackoff* and *RememberingBackoff* (and other variants) all achieve similar throughput and therefore we only show *AbortBackoff* in the plots.
Generally, it can be seen that for most benchmarks the throughput declines with an increasing number of used threads, i.e. cores. This happens even in the case without writes. That is to say, even without conflicts, i.e. irrespective of the used contention management and load adaption policy, performance decreases. DSTM2 is used with visible readers. For a visible reader system the metadata of a read object is modified, which generally causes cache misses and slows down the system with an increasing number of threads.[5] Another reason for the decrease in performance when using more cores might be that for practically all benchmarks only little computation is done but (relatively) a lot of memory is accessed. In such a scenario the memory bus becomes a bottleneck and main memory accesses become slower.

The ListCounter benchmark provides the longest transactions among the six tested benchmarks. Therefore it is very prone to livelocks. Kindergarten's throughput drops to zero if more than one thread is active, same happens for Karma and Polka if more than 9 threads are running. Both *QuickAdapter* and *AbortBackoff* (and their variants) achieve consistently good results on all different benchmarks. In the majority of the cases they outperform the existing policies. If not, they do not lag behind much. The throughput of all other managers depends very much on the benchmark. Each drops off by more than 50% compared to the best manager on at least one of the benchmarks. This is not surprising, since any traditional (non-load adapting) contention manager adapts a specific heuristic for calculating the priority of a transaction, which is only efficient in some cases and fails for other cases. It is particularly interesting to compare *QuickAdapter* and *TimeStamp*, since both use time as priority. In all but one scenario *QuickAdapter* is faster: For the counter benchmark, which enforces sequential execution, for *QuickAdapter* a committing transaction A wakes up an old transaction that aborts the new transaction after A. But it would be better to assign a new timestamp whenever a transaction is woken up. This seems to be less of an issue with the *TimeStamp* manager. It is not clear how to create a final ranking. Some benchmarks might be more important than others and some application scenarios might not be covered by the benchmarks. We ranked each benchmark based on the number of committed transactions for 12 threads and 60% writes. Managers are ranked equally if the throughput differs by less than 5%. The average rank of *QuickAdapter* is 1.7, that of *AbortBackoff* is 1.8 and only then follows *TimeStamp* and *Karma* with rank 3.5. *Polka* reaches an average of 4 and *Kindergarten* was last with 4.2.

6. CONCLUSIONS AND FUTURE WORK

To this day, for transactional memory the standard method of load adaption led to a level of complexity that is not well-suited for hardware and it needed central coordination, limiting scalability sooner or later. Both our proposals *AbortBackoff* and *QuickAdapter* address these issues. Though experimental evaluation shows consistently good results, an optimal contention management strategy has yet to be found. Practice (and more experimental evaluation) has to show to what extend gathering and using (detailed) information for contention management is worth its costs. Our theoretical analysis gives insights by investigating several load adapting strategies and scenarios. Still, the principles of load adaption (and contention management), e.g. in general conflict graphs and for different models (e.g. lazy

[5] At the time of writing, the discussion whether visible or invisible readers are preferable has not come to a definite end.

conflict resolution), have to be (further) developed.

REFERENCES

[1] M. Ansari, C. Kotselidis, M. Lujn, C. Kirkham, and I. Watson. "On the Performance of Contention Managers for Complex Transactional Memory Benchmarks". In *Symp. on Parallel and Distributed Computing*, 2009.

[2] S. Dolev, D. Hendler, and A. Suissa. "CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory". In *PODC*, 2008.

[3] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. "Robust Contention Management in Software Transactional Memory". In *Workshop on Synchronization and Concurrency in Object-Oriented Languages*, 2005.

[4] R. Guerraoui, M. Herlihy, and B. Pochon. "Polymorphic contention management". *DISC*, 2005.

[5] M. Herlihy, V. Luchangco, and M. Moir. "A flexible framework for implementing software transactional memory". *SIGPLAN NOTICES: ACM SIGPLAN Notices*, 41, 2006.

[6] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. "Software transactional memory for dynamic-sized data structures". In *PODC*, 2003.

[7] M. Herlihy and J. Moss. "Transactional Memory: Architectural Support For Lock-free Data Structures". In *Symp. on Computer Architecture*, 1993.

[8] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. "MetaTM/TxLinux: transactional memory for an operating system". In *Symp. on Computer Architecture*, 2007.

[9] W. Scherer and M. Scott. "Advanced contention management for dynamic software transactional memory". In *PODC*, 2005.

[10] J. Schneider and R. Wattenhofer. "Bounds On Contention Management Algorithms". In *ISAAC*, 2009.

[11] N. Shavit and D. Touitou. "Software transactional memory". *Distributed Computing*, 10, 1997.

[12] R. M. Yoo and H. S. Lee. "Adaptive transaction scheduling for transactional memory systems". In *SPAA*, 2008.