Toward a Theory of Transactional Contention Managers

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

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

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

ABSTRACT

In recent software transactional memory proposals, a *contention manager* module is responsible for ensuring that the system as a whole makes progress. A number of contention manager algorithms have been proposed and empirically evaluated.

In this paper we lay some foundations for a theory of contention management. We present the *greedy* contention manager, the first to combine non-trivial *provable* properties with good practical performance.

In a model where transaction delays are finite, the greedy manager guarantees that every transaction commits within a bounded time, and 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. No contention manager reviewed in the literature satisfies both the properties. Benchmark results convey our claim of the practicality of the greedy manager.

1. INTRODUCTION

The limitations of conventional multiprocessor synchronization techniques based on locks and condition variables are well-known. Recently, *transactional* models of synchronization have received attention as an alternative programming model.

A transaction, like a critical section, is an explicitly delimited sequence of steps to be executed atomically by a single thread. Unlike critical sections, a transaction can either *commit* (take effect), or *abort* (have no effect). If a transaction aborts, it is typically retried until it commits. Transactional APIs for multiprocessor synchronization, called *Transactional memories*, have been proposed both for hardware [6, 9, 16, 19, 20, 26] and in software [7, 8, 10, 12, 18, 25, 15]. In this paper, we are primarily concerned with the structure of software transactional memory (STM) systems.

A concurrent algorithm is *obstruction-free* [9] if it guarantees that any thread, if run by itself for long enough, will make progress. The obstruction-free property is attractive for an STM because it supports a clean separation of concerns: correctness and progress can be addressed by different modules. In recent STM proposals [9, 23], progress is the responsibility of a *contention manager* module.

Transactions read and write shared objects. Two transactions *conflict* if they access the same object and one access is a write. Transaction synchronization is *optimistic* [13]: a transaction commits only if, at the time it finishes, no other transaction has executed a conflicting access. It is not hard to see that optimistic synchronization is obstruction-free.

If transaction A discovers it is about to conflict with B, then it has a choice: it can pause, giving B a chance to finish, or it can proceed, forcing B to abort. Faced with this decision, A will ask its local contention manager module which choice to make.

The literature includes a number of contention manager proposals [10, 23], ranging from simple (exponential backoff) to elaborate (various priority-based schemes). Empirical studies have shown that the choice of a contention manager algorithm can affect transaction throughput, sometimes substantially.

While empirical evaluation will always be essential for evaluating contention manager designs, we think the time has come to develop a more systematic theory of contention management. For example, if a contention manager never allows one transaction to abort another, then deadlock can happen. If a contention manager always advises transactions to abort one another, then live-lock can happen. Between these extremes, what kinds of progress properties, if any, can one guarantee? Adaptive backoff seems to work well in practice when transactions have roughly the same size, but is less effective if long transactions must compete with shorter transactions. Can we put bounds on the worst-case behavior of a contention manager, perhaps by comparing to an optimal off-line scheduler?

In this paper we lay some foundations for a theory of contention management. We present the *greedy* contention manager, the first to combine non-trivial *provable* properties with empirically promising performance.

In a model where transaction delays are finite, the greedy manager guarantees that:

- every transaction commits within a bounded time, and
- the time to complete n concurrent transactions that share s objects is within a factor of $s(s+1)/2$ of the

PODC'05, July 17–20, 2005, Las Vegas, Nevada, USA.

time that would have been taken by an optimal off-line list scheduler.

None of the the contention managers proposed in the literature satisfies both properties.

We present experimental results showing that the greedy manager has a very reasonable worst-case performance without being outperformed in average cases by other managers experimented so far in practice. Finally, we discuss simple extensions to this manager for models in which transactions can halt undetectably and we open some problems related to the theoretical study of contention management.

The principal contribution of this paper is to demonstrate that contention management raises theoretical as well as pragmatic issues. Because effective contention management is central to making software transactional memory work in practice, this area is one where theoretical and practical analysis are intertwined.

2. SCOPING CONTENTION MANAGEMENT

Although contention management can be viewed as a task scheduling problem, contention managers differ in important ways from the schedulers considered in most of the literature (for surveys of scheduling literature, see Chaplin [2] or Gonzalez [4]). Many classical scheduling algorithms use centralized data to make decisions (for example, choosing the first runnable job from a queue). Contention managers, by contrast, are highly *decentralized*: one transaction decides whether to abort another based only on a comparison of the two transactions' states. While it might be possible to collect other information, this might be inefficient, especially if synchronization were needed, and of doubtful value. Contention managers cannot use protocols that require the active participation of multiple transactions, since one transaction may be blocked or swapped-out.

In addition, most of the scheduling algorithms in the literature are *off-line* in the sense that task durations and resource requirements are usually known in advance. By contrast, a contention manager does not know how long a transaction will take, nor any of its future resource needs. It would be unacceptably cumbersome (not to mention a violation of modularity) to require a transaction to predeclare its set of data accesses.

At first glance, a contention manager looks like a concurrency controller in a database system, e.g., [22, 13]. Both have to do with delaying or aborting transactions. There is however a fundamental difference. Whereas the goal of a concurrency controller is to ensure serializability (a safety property), a contention manager seeks only to enhance progress (liveness). The serializability of transactions in STM systems is taken for granted when devising contention managers [10, 23].

Some optimistic concurrency controllers [11] use a conflict resolution algorithm that is similar to a contention manager. In contrast with our scheme, where conflict detection occurs when a transaction executes, optimistic concurrency controllers detect conflicts when a transaction tries to commit (after it has accessed all objects).

Just like contention managers, certain *unreliable* failure detectors [1, 5] can also be viewed as progress boosters, to be used in conjunction with algorithms inherently devised to cope with safety. As their names indicate, failure detectors track the failure of processes. They depend solely on the failure pattern and not on the computation [1]. Contention managers on the other hand depend primarily on the computation, i.e., on whether two transactions conflict by accessing the same object.

3. THE GREEDY CONTENTION MANAGER

The contention manager we introduce in this paper makes use of three components of a transaction state.

- First, when a transaction begins, it is given a *timestamp* which it *retains* even if it aborts and restarts. A transaction with an earlier timestamp has *higher priority* than a transaction with a later timestamp. In the set of rules described below, timestamps are generated by atomically incrementing a shared variable. One could generate timestamps by a variety of methods, including logical clocks. The key property we need is that if a transaction takes a timestamp t, then there is henceforth a fixed bound (usually $n-1$) on the number of transactions that execute with earlier timestamps.
- Second, the transaction's *status* field indicates whether the transaction is active, committed, or aborted. A transaction commits by applying a compare-and-swap instruction to change its status field from *active* to *committed*. One transaction aborts the other by applying a compare-and-swap to change its status from *active* to *aborted*.
- Finally, each transaction has a public Boolean *waiting* field that indicates whether it is waiting for another transaction.

Our greedy contention manager has two simple rules. Suppose transaction A discovers it is about to perform an access that conflicts with transaction B.

1. If B is lower priority than A, or if B is waiting for another transaction, then A aborts B.
2. If B is higher priority than A and is not waiting, then A waits until B commits, aborts, or starts waiting. (If B starts to wait, see Rule 1.)

Naturally, these rules make sense only if transaction delays are bounded (otherwise A might wait forever for a halted B). We discuss later how to extend these rules to cope with halted transactions.

4. PROPERTIES

The number of transactions with higher priority than a given transaction is bounded. Given that the greedy manager ensures that the highest-priority transaction never waits and is never aborted by a synchronization conflict, we immediately obtain the following.

THEOREM 1. *Every transaction commits after a bounded delay.*

In the following, we prove a property (Theorem 9) on the time taken by our greedy contention manager to commit transactions. We show that 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 (one that knows transactions' resource requirements in advance).

As we discussed earlier in the paper, a contention manager is indeed a kind of scheduler. It is well-known that optimal off-line scheduling of tasks with shared resources is NP-complete. Hence, the best one can expect is to approximate an optimal schedule.

Consider an execution in which n concurrent transactions start at time 0. The *makespan* is the duration until the last of them commits.

The greedy contention manager schedules transactions while their resources appear to be available, but because it does not know a transaction's resource demands in advance, it does not always make optimal choices.

Consider transactions $T_0, \ldots, T_s$ and objects $X_1, \ldots, X_s$. Each transaction runs for one time unit. T_i has higher priority (an earlier timestamp) than T_{i-1}. T_0 accesses X_1, T_s accesses X_s, each remaining T_i accesses X_i and X_{i+1}. A list scheduler might run the even transactions and then the odd, yielding a makespan of 2 (which happens to be optimal).

By contrast, the greedy contention manager does not do as well. At time 0, each T_i, $0 \leq i < s$, accesses X_{i+1}. At time $1 - \epsilon$, T_1 accesses X_1, aborting T_0. In turn, each T_i opens X_i, aborting T_{i-1}. At time 1, only T_s commits. After repeating this scenario s more times, all transactions commit, yielding a makespan of $s + 1$.

It is known that simple off-line schedulers yield schedules within a factor of $(s + 1)$ of optimal [3]. Here, we will show that any on-line contention manager which ensures only the property that at any time, some running transaction will run uninterrupted until it commits, is within a factor of $s(s + 1)/2$ of optimal. This property, which we call the *pending commit* property, is clearly satisfied by our greedy manager. This is because the running transaction with highest priority will neither wait nor be aborted by any other transaction.

4.1 Tasks and Resources

We follow the model of Garey and Graham [3]. A *task system* is given by a set of *tasks* $\{T_1, \ldots, T_n\}$ and a set of shared *resources* $\{R_1, \ldots, R_s\}$. Each T_j has an associated *length* $\tau_j > 0$, and uses $R_i(T_j)$ units of resource R_i, where resources are normalized so that $0 \leq R_i(T_j) \leq 1$.

A *list scheduler* maintains a list of tasks, and each processor instantaneously scans the list from front to back and starts the first unstarted task whose resources are available. (Here, we are concerned only with the case in which the number of processors equals the number of tasks.) It is known that computing an optimal list scheduler is NP-complete, but that any list schedule is within a factor of $(s + 1)$ of the optimal [3]. Notice that list schedulers have the following *list scheduler property*: no task is waiting to execute if the resources it needs are available.

Informally, the system starts executing the tasks at time 0, and each T_j requires exclusive use of $R_i(T_j)$ units of resource R_i at all times during its execution. Tasks are non-preemptable: once T_j is started, it runs to completion in time τ_j. Formally, an *execution* E is a map from times to sets of tasks: $E(t)$ is the set of transactions running at time t.

We divide each time unit into m discrete "ticks", for some sufficiently large integer m, and we assume without loss of generality[1] that tasks start and stop only at these discrete ticks.

For each t, let $r_i(t)$ denote the total amount of resource R_i in use at time t:

$$r_i(t) = \sum_{T_j \in E(t)} R_i(T_j).$$

Executions satisfy the following constraint:

$$r_i(t) \leq 1 \quad \text{for all} \quad t \geq 0. \tag{1}$$

Later on, we will construct task executions in which $r_i(t)$ may exceed 1. Such executions are said to be *overloaded*.

As noted, a list scheduler guarantees that no task is waiting to be executed at a time when its resource requirements can be satisfied: for all t, if $1 - r_i(t) \geq R_i(T_j)$, then T_j has started to execute.

4.2 Optimal Transactional Executions

Transactions exhibit more complicated behavior than tasks. We are given a set of transactions $T_0, \ldots, T_n$, where T_i, if run by itself, runs for duration τ_i. Transactions share a set of *objects* $X_1, \ldots, X_s$. Each transaction T_j requires the use of $X_i(T_j)$ units of object X_i after some point in its execution. If T_j updates X_i, then $X_i(T_j) = 1$, and if it reads X_i without updating, then $X_i(T_j) = 1/n$. T_j and T_k *conflict* at X_i if $X_i(T_j) + X_i(T_k) > 1$.

We define a corresponding task system in a straightforward way. For each transaction T_j, $0 \leq j \leq n$, of duration δ_j, define a task T_j^* also of duration δ_j. For each object X_i, $1 \leq i \leq s$, define a resource X_i^* such that $X_j^*(T_i^*) = X_j(T_i)$. Let E^* be an optimal execution for this task system.

Assume without loss of generality that E^* has makespan 1. Let $r_i^*(t)$ denote the amount of resource X_i^* in use at time t.

Lemma 2. *For $1 \leq i \leq s$,*

$$\sum_{k=0}^{m-1} r_i^* \left(\frac{k}{m} \right) = \int_0^\infty r_i^*(t) dt \leq 1.$$

Proof. This function's value is constant on each interval $[k/m, (k + 1)/m)$, it does not exceed 1 in the interval $[0, 1]$, and it is 0 for values greater than 1. □

4.3 Actual Transactional Executions

Consider a contention manager that guarantees the pending commit property: at any time, some running transaction will run uninterrupted to commit.

In more detail, an execution of a transaction T_j is a sequence of *actions* where each action represents an interval during which the transaction is continuously running. The last action finishes when the transaction commits, and each previous action, if any, finishes when the transaction aborts or waits. A *transaction execution* E is a map from time to the set of actions running at that time. The pending commit property states that at any time t earlier than the execution's makespan, some action in $E(t)$ will commit.

Given a transactional execution E, we now construct a corresponding task execution E'. This task execution is (in general) *overloaded*, since it may sometimes permit more than a unit of a resource to be in use.

For each transaction T_k, $1 \leq k \leq n$, let A_k be the last (committing) action for T_k, and let δ'_k be its length. Note that $\delta'_k \leq \delta_k$. Define a task execution E' so that each task

[1] See Garey and Graham [3] Lemma 3.

runs for the same time as the corresponding transaction's last action:

$$T'_j \in E'(t) \quad \texttt{iff} \quad A_j \in E(t).$$

For each pair of objects X_i, X_j, $1 \leq i \leq j \leq s$, define a resource X'_{ij}. (There are $s(s+1)/2$ such resources.) Informally, we use X'_{ii} to track object X_i, and X'_{ij}, for distinct i and j, is a bookkeeping device to track indirect dependencies between transactions that access objects X_i and X_j. If transaction T_k updates object X_i, then task T'_k requires all of resources X'_{ij}, for $i \leq j \leq s$, and X'_{ji}, for $1 \leq j < i$, and similarly for read access. Formally,

$$X'_{ij}(T'_k) = \max(X_i(T_k), X_j(T_k)).$$

Let $r'_{ij}(t)$ be the amount of X'_{ij} in use at time t. Notice that the resulting schedule is overloaded: it is possible for $r'_{ij}(t)$ to exceed 1. The execution E' trivially satisfies the list scheduler property: if T'_i has not yet started, yet all the resources it needs are available, then no task is running, contradicting the pending commit property.

LEMMA 3. *For $1 \leq i \leq s$,*

$$\int_0^\infty r'_{ii}(t)dt \leq \int_0^\infty r^*_i(t)dt.$$

PROOF. In E^*, T^*_j contributes either δ_j (update) or δ_j/n (read). In E', T'_j contributes either δ'_j (update) or δ'_j/n (read), and $\delta'_j \leq \delta_j$. □

Combining with Lemma 2:

COROLLARY 4. *For $1 \leq i \leq s$,*

$$\int_0^\infty r'_{ii}(t)dt \leq 1.$$

LEMMA 5. *For $1 \leq i < j \leq s$,*

$$\int_0^\infty r'_{ij}(t)dt \leq \int_0^\infty r'_{ii}(t)dt + \int_0^\infty r'_{jj}(t)dt.$$

PROOF. If a task is using r units of X'_{ij} at time t, then it is using at most r units of either X'_{ii} or X'_{jj} at time t. □

Combining with Lemma 2:

COROLLARY 6. *For $1 \leq i < j \leq s$,*

$$\int_0^\infty r'_{ij}(t)dt \leq 2.$$

Let $G = (V, E)$ denote a graph with vertex set V and edge set E. A *valid labeling* of G is a function $L : V \to [0, \infty)$ which satisfies

$$\text{for all} \quad e = (u, v) \in E, L(u) + L(v) \geq 1.$$

Define the *score* of G, denoted by $S(G)$, by

$$S(G) = \inf_L \sum_{v \in V} L(v).$$

Define the graph $G(m, s)$ to have vertex set $\{0, 1, \ldots, (s+1)m - 1\}$, and an edge between vertexes a and b whenever $|a - b| \geq m$.

The following is Garey and Graham's Lemma 2:

LEMMA 7. *Suppose $G(m, s)$ is partitioned into s spanning subgraphs H_i, $1 \leq i \leq s$. Then*

$$\max_{1 \leq i \leq s} S(H_i) \geq m.$$

We will make use of the following corollary:

COROLLARY 8. *Suppose $G(2m, s(s+1)/2)$ is partitioned into $s(s+1)/2$ spanning subgraphs H_i, $1 \leq i \leq s(s+1)/2$. Then*

$$\max_{1 \leq i \leq s(s+1)/2} S(H_i) \geq 2m.$$

Now we proceed to prove our main result. Assume, without loss of generality, that ω^*, the makespan of the optimal execution, is 1. Assume, by way of contradiction, that $\omega > \frac{s(s+1)}{2} + 1$. Recall that each unit of time is divided into m discrete ticks, and tasks start and stop at times k/m, where k is a positive integer. It follows that $k \leq m$, since $\tau_i \leq \omega^* = 1$.

Notice that for $0 \leq t_0 < t_1$, where $t_1 - t_0 \geq 1$,

$$\max_{1 \leq i \leq j \leq s} r'_{ij}(t_0) + r'_{ij}(t_1) > 1, \tag{2}$$

because otherwise any task running at time t_1 should have been running at time t_0 or earlier.

For each $1 \leq i < j \leq s$, construct a graph H_{ij} as follows:

$$V(H_{ij}) = \left\{0, \ldots, \left(\frac{s(s+1)}{2} + 1\right) m - 1\right\} \tag{3}$$

$$(a, b) \quad \text{is an edge of} \quad H_{ij} \quad \text{iff} \quad r_{ij}(a/m) + r_{ij}(b/m) > 1. \tag{4}$$

If $|a - b| \geq 2m$, then (a, b) is an edge in at least one H_{ij} (Equation 2), so $G(2m, s(s+1)/2) \subseteq \cup_{i,j} H_{ij}$. Notice that if $G \subseteq G'$, then $S(G) \leq S(G')$. Because Equation 2 is a strict inequality, Lemma 7 implies that:

$$\max_{i,j} \sum_{k=0}^{(\frac{s(s+1)}{2}+1)m-1} r'_{ij}\left(\frac{k}{m}\right) > \max_{i,j}(S(H_{ij})) \geq 2m.$$

Equivalently,

$$\max_{i,j} \frac{1}{m} \sum_{k=0}^{(\frac{s(s+1)}{2}+1)m-1} r'_{ij}\left(\frac{k}{m}\right) > 2.$$

This sum is just the area under the resource usage curve:

$$\max_{i,j} \frac{1}{m} \sum_{k=0}^{(\frac{s(s+1)}{2}+1)m-1} r'_{ij}\left(\frac{k}{m}\right) = \int_0^\infty r'_{ij}(t)dt$$

Putting these equations together implies that for some $1 \leq i < j \leq s$,

$$\int_0^\infty r'_{ij}(t)dt > 2. \tag{5}$$

which contradicts Corollaries 4 and 6.

We have just shown the following:

THEOREM 9. *Any contention manager that satisfies the pending commit property produces a makespan within a factor of $s(s+1)/2$ of optimal.*

5. EXPERIMENTAL RESULTS

The makespan argument characterizes the manager's worst-case performance, and says little about how it behaves for common cases. In this section, we report some experimental results comparing the greedy manager against other managers that appear in the literature. The point of the experiments is not to establish one or more contention managers as superior (that would take much more extensive benchmarking) but to show that the greedy contention manager has respectable practical performance, i.e., average case behavior.

The experiments were done using the SXM software transactional memory system, a system written in C# currently under development at Microsoft Research Cambridge [21]. The SXM package differs from the earlier Java-based Dynamic STM package [10] by using run-time code generation instead of programmer convention to synchronize shared objects. The contention managers from Scherer and Scott [23] were ported to C# either from the original Java code, or from their written descriptions. The benchmarks were run on a machine with four Intel Xeon processors with 2-way hyperthreading turned on. Each processor runs at 2.0 GHz and has 1 GB of RAM.

Figures 1 to 4 compare several contention managers defined in the literature [23, 24], for various applications, and under low and high contention scenarios. We use four data structures used as benchmark applications: a list, a skip-list [10], a red-black tree [10], and a red-black forest (a data structure made of fifty red-back trees, in which insertions and removals of elements proceed in either one or all trees on a random basis; the distribution of the lengthes of the transactions produced which a red-black forest thus exhibits a high variance).

We consider constant size transactions, in which a number of threads ranging from 1 to 32 continuously insert and remove elements taken from a small set of 256 integers, hence forcing contention to happen, and an update rate of 100%. All figures depict the number of committed transactions per second, as a function of the number of threads.

Interestingly enough, as conveyed by Figures 1 to 4, the best contention manager for a given transaction is not necessarily the best contention manager for another transaction. Which contention manager suits best for a given transaction depends on various factors, such as the contention pattern with conflicting transactions, the length of the contented vs. uncontented periods within the transaction, etc.

As depicted by Figures 1 and 2, *karma* [23] and *polka* [24] contention managers provide a better throughput in a contention-intensive scenario.

Figure 3 exhibits a low contention pattern among transactions by making threads perform computations unrelated to the effective transactions at the end. In our SXM implementation, when there is no contention among transactions at the end of the computation, for various length of uncontented periods, and when transactions are approximately of the same size, the *greedy* contention manager provides the best throughput.

Figure 4 exhibits an intensive contention pattern among transactions of irregular length. In this case, the best contention management policies seem uncertain. More precisely, it highly varies depending on the number of threads

In short, and to to the best of our understanding, the *greedy* contention manager performs well when there is no or few contention. This is because the *greedy* contention manager does not maintain costly data structures for assigning priority to transactions. On the other hand, *karma* performs well when contention is intensive, because the priority it assigns to transactions, though more costly to update, reveals itself a very good estimator of the importance of the transactions.

6. CONCLUDING REMARKS

Contention managers were first proposed in [10], and a comprehensive survey on the subject is due to Scherer and Scott [23]. The notion that waiting transactions should not obstruct active transactions has been used by McWherter et al. [17] to improve the throughput of web-based on-line transaction systems.

Using timestamp-based priorities to guarantee progress is an old trick: it goes at least as far back as 1978 [22], and has been used in more recent transactional memory proposals [20].

None of the contention managers explored in [23] satisfies the pending commit property, i.e., at any time, some running transaction will commit, which our theoretical analysis reveals to be fundamental for worst-case analysis. Clearly, none of the *polite* or *randomized* managers provide any deterministic guarantee. As explained in [23], the *queueOnBlock* manager is prone to dependency cycles whereas the *aggressive* manager is prone to livelocks. Aborting enemies after a time-out, as in the *killBlocked*, *kindergarten*, and *timestamp* managers, diminishes the probability of livelocks without however canceling it. The *karma* and *eruption* contention managers account for the work performed by a conflicting transaction and the number of times it already aborted, before aborting it. In theory however, any transaction A might get repeatedly aborted due to newcomer transactions that, between two aborts of A, get aborted more often and access more objects.

Some STM implementations, as described by Harris and Frasier [7] and Harris et al. [8], discover conflicts when transactions commit, not while they are executing. Contention managers do not seem well-suited to these kinds of STMs, and the question of ensuring progress for this kind of STM design remains largely unexplored.

With the exception of Scherer and Scott's *timestamp* manager [23], none of the contention managers we know of ensure progress if transactions can stop prematurely. Interestingly, our greedy contention manager can be extended to cope with failures in much the same way as the timestamp manager: add a rule which stipulates that whenever a transaction A waits for a higher priority transaction B, A does so only until a time-out expires; choose the time-out period to be proportional to the number of times A had to wait for B and then aborted B. This growth can simply be performed by doubling the time for each such new discovery.

We close with some open problems. We do not know if the $s(s+1)/2$ bound for the greedy contention manager is tight. Are there other contention managers that provide similar, provable properties? For example, can one use randomization to implement a contention manager that is proved to behave well with high probability?

We evaluated the greedy contention manager's efficiency by comparing its makespan for n concurrent transactions against the same makespan for an optimal off-line list scheduler. An open problem is to do the same for threads that ex-

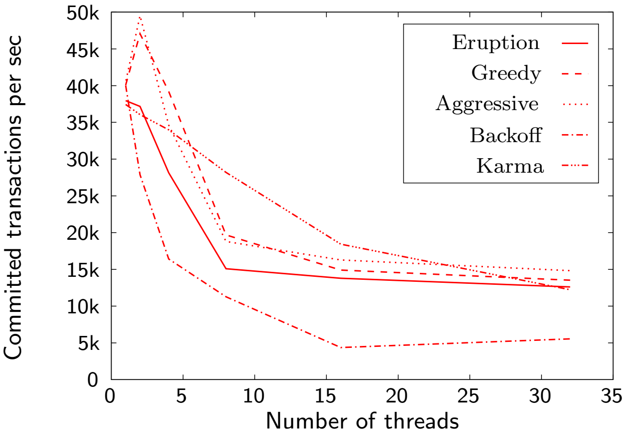

Figure 1: List application

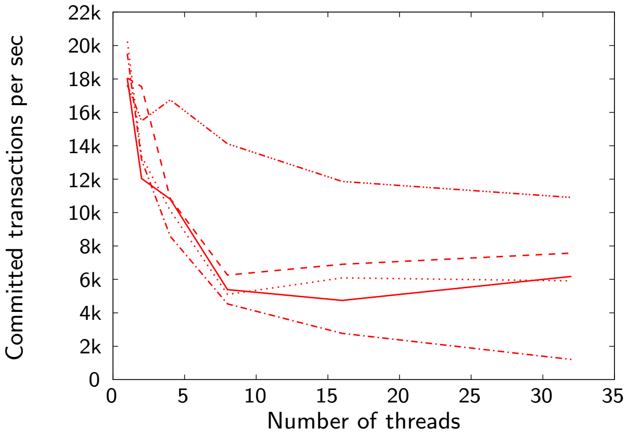

Figure 2: Skiplist application

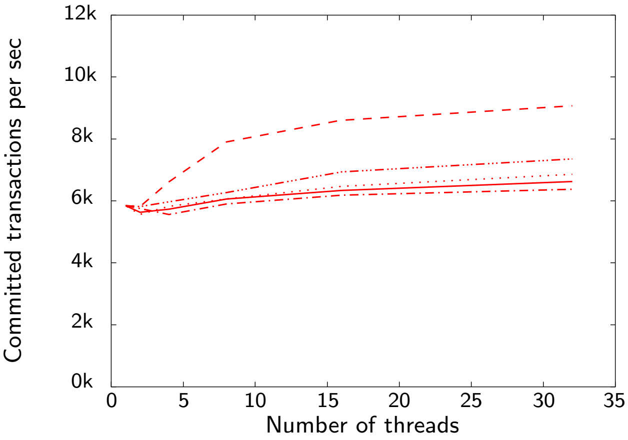

Figure 3: Red-black application (low contention)

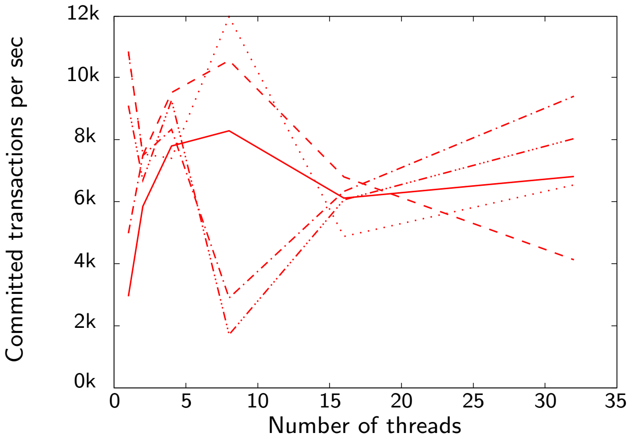

Figure 4: Red-black forest application

ecute a sequence of transactions, instead of just one. Similar analyses for off-line task scheduling exist for unit-execution time (UET) tasks [3, 14]. The UET assumption reduces the problem to multidimensional bin-packing, an approach not easily adapted to transactions of unknown duration.

Acknowledgments

We are grateful to Bill Scherer for sharing his contention manager source code.

7. REFERENCES

[1] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[2] Steve J. Chapin. Distributed and multiprocessor scheduling. *ACM Comput. Surv.*, 28(1):233–235, 1996.

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

[4] Mario J. Gonzalez, Jr. Deterministic processor scheduling. *ACM Comput. Surv.*, 9(3):173–204, 1977.

[5] R. Guerraoui. Indulgent algorithms. In *Proceedings of the nineteenth annual symposium on Principles of distributed computing*, pages 289–299. ACM Press, 2000.

[6] Lance Hammond, Vicky Wong, Mike Chen, Ben Hertzberg, Brian D. Carlstrom, John D. Davis, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the thirty-first Annual International Symposium on Computer Architecture*, June 2004.

[7] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the eighteenth ACM Conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[8] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming*, 2005. To appear.

[9] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the twenty-third International Conference on Distributed Computing Systems*, pages 522–529, May 2003.

[10] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on*

Principles of distributed computing, pages 92–101. ACM Press, 2003.

[11] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the seventeenth Conference on Very Large Databases*. Morgan Kaufman, 1991.

[12] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160. ACM Press, 1994.

[13] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[14] Errol L. Lloyd. Critical path scheduling of task systems with resource and processor constraints (extended abstract). In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 436–446. ACM Press, 1980.

[15] V. J. Marathe, W. N. Scherer, III, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, October 2004.

[16] José F. Martínez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the tenth international conference on architectural support for programming languages and operating systems*, pages 18–29. ACM Press, 2002.

[17] David McWherter, Bianca Schroeder, Natassa Ailamaki, and Mor Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proceedings of the twentieth International Conference on Data Engineering*, April 2004.

[18] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228. ACM Press, 1997.

[19] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the tenth international conference on architectural support for programming languages and operating systems*, pages 184–196. ACM Press, 2002.

[20] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the tenth international conference on architectural support for programming languages and operating systems*, pages 5–17. ACM Press, 2002.

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

[22] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

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

[24] W. Scherer, III and M. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, July 2005.

[25] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM Press, 1995.

[26] Janice M. Stone, Harold S. Stone, Phil Heidelberger, and John Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.