Technical Report

Exploring Checkpointing and Closed Nesting in Distributed Transactional Memory

Alexandru Turcu
Virginia Tech
talex@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Abstract

Checkpointing and closed nesting are mechanisms typically used for implementing partial roll-back in transactional systems. Closed nesting limits the amount of work to redo on an abort by allowing sub-transactions to abort and retry independently from their parents. Checkpointing goes further and allows a transaction to be rolled back to any previous point where a checkpoint was saved. Checkpointing thus enables very fine-grained rollbacks.

In this paper we focus on understanding the performance considerations of closed nesting and checkpointing in Distributed Transactional Memory (DTM). We extend an existing DTM algorithm, TFA, with support for the two partial rollback models, and implement it in the Hyflow2 open-source DTM framework for the JVM. We then perform a thorough evaluation to determine their behavior, implementation overheads, and favorable conditions.

1. Introduction

Transactional systems based on Software Transactional Memory (STM) are nowadays considered the next generation of software architectures for managing concurrent requests on shared data. In such systems, transactions are typically characterized by an execution time several orders of magnitude smaller than in traditional architectures (e.g, DBMS). Clearly, avoiding the synchronous interaction with the stable storage, STM systems are able to serve many concurrent requests in parallel, guaranteeing great performance in comparison to traditional DBMS systems [20].

Perhaps more importantly, STM frameworks aim to simplify the development of concurrent applications. The difficult task of programming concurrency is shifted away from regular programmers, and into the hands of expert library developers, who can guarantee consistency and a good level of performance. In fact, leveraging an STM library, programmers only have to implement the business logic of the application, without having to manage the details of synchronizing among the different live threads in the system.

The challenge to manually build synchronization mechanisms is exacerbated when the system moves from the centralized setting to distributed. When implementing and debugging concurrency in distributed systems, developers usually face issues like distributed deadlocks, distributed livelocks, distributed conflict detection, and others. These issues are typically difficult to trace and resolve without relying on a physical synchronized clock for accurate ordering events. Distributed Transactional Memory (DTM) systems promise to solve these problems of distributed synchronization, while providing an abstraction, the distributed transaction, that is easy to work with and completely transparent.

In DTM, the transaction execution time is higher than in centralized STM due to the (possibly multiple) interactions with other nodes on the network. Aborting a transaction involves re-executing all the transactional operations performed so far. While in a centralized setting transactions are only comprised of in-memory operations, in DTM many operations require network access, such as retrieving new copies of objects from other nodes. This emphasizes the effect on performance due to aborts.

The typical approach to minimizing the effect of aborts is trying to reduce the probability that such an abort will even take place, using techniques like transaction scheduling. Unfortunately, depending on the underlying transactional protocols, this is not always possible. Often times, two or more concurrent transactions access the same data, and at least one access is a write. This is a *conflict*, and some of the transactions have to be aborted and restarted at a later time.

The aborting transaction can either restart from the beginning and re-execute in its entirety, or, in case some of the operations it executed are still valid, restart from an intermediary location in the code and re-execute only the invalidated portions of the transaction. Even though the first approach (named *flat nesting*) is straightforward to implement and seems appropriate for short transactions, the sec-

ond approach is particularity valuable when the transaction execution time is not negligible, like in DTM. Potential gains include transaction response time and throughput, but they have to exceed any overheads introduced by the mechanism used to implement the partial rollback.

In this paper we focus on the two methods previously proposed for supporting partial rollback, *closed-nested sub-transactions* and *checkpoints*, in the context of DTM. Our aim is to determine which of the two performs better and under what conditions. This question was previously asked for multiprocessor STM [11], and an extension to DTM is only natural.

Closed nesting allows treating transactions as containers for inner transactions. While sub-transactions are executing, they can abort independently from their *parent transactions*, thereby potentially reducing the scope of rollbacks. When a sub-transaction commits, its state is merged into the state of its parent. If a conflict is detected after the sub-transaction commits, the parent will be aborted as well. In closed nesting, the scope of a rollback can only be chosen among enclosing transaction boundaries when the conflict is detected.

The second mechanism, checkpointing, does not require the sub-transaction construct for delimiting partial rollback scopes. Instead, special calls for saving the transaction execution state are used throughout the transaction, either explicitly (*manual checkpointing*) or implicitly (*automatic checkpointing*, when the checkpoint is taken alongside other transactional operations already present in the code). Checkpoints can be seen as a generalization of closed nesting: transactions can be rolled back to any previous checkpoint in order to resolve conflicts.

The granularity of checkpointing is configurable. At one end of the spectrum, a checkpoint is taken before every operation that can generate a conflict. If a conflict does arise, the transaction can be restarted from the exact operation that will resolve the conflict. This is optimal in the sense that no unnecessary operations need to be re-executed. The problem with this approach is that taking checkpoints is an expensive operation in itself. Thus, taking too many checkpoints can turn into a significant overhead. Conversely, reducing the number of checkpoints will reduce overheads, but increases the amount of valid work that needs to be retried in case of a conflict. At the opposite end of the spectrum, the only checkpoint ever taken is at the beginning of the transaction, leading to a behavior identical to flat-nesting, albeit using a different mechanism for implementing it.

Our paper focuses on this trade-off. The main goal is to highlight the settings in which the checkpointing approach outperforms closed nesting and vice-versa, and how both of them compare to flat-nesting.

We start from an existing DTM protocol, the Transactional Forwarding Algorithm (TFA) [21], and extend it to support closed nesting and transactional checkpointing. We name the resulting protocols N-TFA (Nested TFA[1]) and TFA-CP (TFA with Checkpoints). TFA was chosen as a starting point because it provides a strong consistency guarantee, *opacity*. Strong consistency makes it easy for programmers to reason about concurrency in their application.

We implement all the mechanisms for supporting partial rollback within HyFlow2 [24], a high-performance, open-source DTM framework for the JVM, written in Scala. Hyflow2's design is modular and allows for pluggable support for lookup protocols, transactional synchronization and recovery mechanisms, and contention management policies.

Supported by an extensive evaluation study using a set of six micro-benchmarks and one macro-benchmark, we determine that, when applicable, closed nesting consistently outperforms flat nesting, but the average improvement is low at only 3%. On the other hand, checkpointing incurred significant overheads and suffered from repeated aborts, leading to an average performance degradation of 17-18%. Despite that, we identified specific conditions where checkpointing takes the lead: high-contention workloads where the aborts are concentrated around the middle of the transaction.

Our complete implementation is publicly available at http://www.hyflow.org/hyflow/wiki/Hyflow2 .

The rest of the paper is organized as follow: the Section 2 describes TFA and transaction nesting, which represent the background for this work. Section 3 overviews related work. In Section 4 we introduce N-TFA and TFA-CP, our extensions to TFA with support for closed nesting and checkpointing. Section 5 discusses implementation details, focusing on continuations, the mechanism needed to implement checkpoints. In Section 6 we evaluate our implementation. Finally, Sections 7 and 8 recommend future work, and respectively, conclude the paper.

2. Background

In this section we provide a brief introduction to TFA, the base protocol we extend in this paper. We then proceed to describe the different transaction nesting models and transaction checkpointing.

2.1 TFA Protocol

Transactional Forwarding Algorithm (TFA) [21] is a lock-based DTM algorithm with lazy lock acquisition and buffered writes. All read objects are stored in a local *read-set*, so all reads can later be revalidated. Written objects are also stored in a local *write-set*.

TFA uses a variant of the Lamport clocks mechanism [12] to establish "happens before" relationships across nodes. Each distributed node has a local clock lc and atomically increments its local clock on the commit of every transaction

[1] Preliminary results on N-TFA were presented by the authors in the TRANSACT 2012 workshop. TRANSACT does not publish archival proceedings to facilitate resubmission to more formal venues. The implementation and experimental analysis have been completely re-done for this paper.

that changes the shared state (write transactions). All messages sent between nodes piggyback the local clock value of the sender node. Upon receiving such a message, each node compares the remote clock value included in the message with its own local clock. If the remote value is greater, the local clock is updated to this greater value. Otherwise, the remote clock value is ignored.

TFA guarantees the safety property called opacity [8]. Under opacity, a transaction will never observe an inconsistent snapshot of the data. This applies not only to successful transactions (as in *serializability*), but also to failed attempts, in order to avoid potentially unrecoverable situations such as infinite loops [8]. Opacity is ensured by checking for conflicts every time a new object is accessed, using a process called Transactional Forwarding.

Each transaction records the local clock value lc at the time it starts (i.e., starting clock, sc). Then, when it communicates with remote nodes (for the purpose of accessing new objects), it compares the clock value of the remote node (rc) to its own start clock (sc). If $rc > sc$, the transaction undergoes a Transactional Forwarding procedure: it validates its read-set, and, should that be successful, updates its starting time to $sc = rc$. If the validation fails the transaction aborts. Validation is performed by comparing the object's latest version with the current transaction's start clock.

2.2 Nested Transactions and Checkpointing

Transactions are nested when they appear within another transaction's boundary. Transaction nesting makes code composability easy: multiple operations inside a transaction will be executed atomically, regardless of whether the said operations contain transactions or not, and without breaking encapsulation. This is an important advantage of transactional memory when compared to traditional lock-based synchronization.

Three transactional nesting models were proposed in the literature: Flat, Closed [15, 16, 18] and Open [6, 17–19, 27].

Flat nesting is the simplest form of nesting, which simply ignores the existence of transactions in inner code. All operations are executed in the context of the parent transaction. Aborting any inner-transaction causes the parent transaction to abort. Thus, no partial rollback can be performed with this model. Clearly, flat nesting does not provide any performance improvement over non-nested transactions.

Closed nesting allows inner transactions to abort individually. Aborting an inner-transaction does not necessarily lead to also aborting the parent transaction (i.e., partial rollback is possible). However, inner-transactions' commits are not visible outside the parent transaction. An inner-transaction commits its changes only into the private context of its parent transaction, without exposing any intermediate results to other transactions. Only when the parent transaction commits is the shared state modified.

Open nesting considers the operations performed by sub-transactions at a higher level of abstraction, in an attempt to avoid false conflicts occurring at the memory level. It allows inner transactions to commit or abort individually, and their commits are globally visible immediately. In case an enclosing transaction aborts, due to any fundamental conflicts (i.e., not false) at the higher levels of abstraction, all the inner transactions are roll-backed by using compensating actions, which are predefined for each abstract operation.

Transactional Memory systems with support for nested transactions usually employ exceptions to manage the control flow after an abort. Exceptions allow passing execution to a handler associated with the enclosing transaction, and can thus be used to rollback to the boundary of any such ancestor transaction. This behavior closely matches the closed nesting model. However, once a sub-transaction commits, it becomes unavailable as a rollback destination. This model is disadvantageous for workloads that have many sub-transactions nested at the same level, because it increases the amount of potentially valid work that needs to be aborted and uselessly re-executed in case of a conflict.

Checkpointing [10, 11] addresses this issue by allowing execution to return to any previously saved state (*checkpoint*) within the current transaction, regardless of whether the sub-transaction encompassing that checkpoint is still active or not. This allows developing a very fine grained partial rollback mechanism, which can identify the exact operation to rollback execution to, in order to resolve the current conflict. On abort, a checkpoint that can resolve the conflict is located and activated, effectively reverting transaction execution to the state it had at the time the checkpoint was originally taken. The program control flow is managed by saving and restoring the thread's execution state (i.e., CPU registers and activation stack) and employs a mechanism called *continuations* [5]. By themselves, continuations do not affect program data (i.e., the heap).

3. Related Work

Nested transactions (using closed nesting) originated in the database community [3, 4] and were thoroughly described by Moss in [16]. His work focused on the popular two-phase commit protocol and extended it to support nesting. In addition to that, he also proposed algorithms for distributed transaction management, object state restoration, and distributed deadlock detection.

One of the early works introducing nesting to Transactional Memory was done by Moss and Hosking in [18]. They describe the semantics of transactional operations in terms of *system states*, which are tuples that group together a transaction ID, a memory location, a read/write flag, and the value read or written. They also provide sketches for several possible Hardware Transactional Memory (HTM) implementations, which work by extending existing cache coherence protocols. Moss further focuses on open-nested transactions in [17], explaining how using multiple levels of abstractions

can help differentiate between fundamental and false conflicts and thus improve concurrency.

Moravan et al. [15] implement closed and open nesting in their previously proposed LogTM HTM. They implement the nesting models by maintaining a stack of log frames, similar to the run-time activation stack, with one frame for each nesting level. Hardware support is limited to four nesting levels, with any excess nested transactions flattened into the inner-most sub-transaction.

Agrawal et al. combine closed and open nesting by introducing the concept of transaction ownership [2]. They propose the separation of TM systems into transactional modules (or Xmodules), which *own* data. Thus, a sub-transaction would commit data owned by its own Xmodule directly to memory using an open-nested model. However, for data owned by foreign Xmodules, it would employ the closed nesting model and would not directly write to the memory.

Checkpoints were first introduced in the context of database recovery [3, 9]. Here, the focus is on optimizing access to the stable storage, in order to reduce the overhead in saving and restoring states saved on disk, while enabling reliable recovery in case of failures. An overview on the use of checkpointing in distributed databases is available in [13]. For implementing partial rollback, databases use the notion of *savepoints* [7]. While savepoints have been previously studied in the database context [22], our work is focused on an environment with different characteristics, namely main-memory distributed transactions, executing without sandboxing, and destined to be integrated within a general purpose programming language.

Koskinen and Herlihy introduced checkpoints to the (centralized) transactional memory community in [11], and present it as an alternative to closed nesting due to its fine-grained conflict resolution capabilities. They implemented a checkpoint-based STM algorithm in C. Saving and restoring the execution state of the program was done using the getcontext/setcontext function family, while responsibility for backing up data (local variables) fell on the programmer. They evaluated the overhead of checkpointing and showed that performance gains of up to 100% are feasible. Their evaluation however is brief and targets a system where transactions are prioritized, which is not the general case.

Dhoke et al. in [1] conducted a comparison between closed nesting and checkpointing in a fault-tolerant DTM based on a Quorum protocol. The authors assess that exploiting closed-nested transactions to implement partial rollback leads to better performance. The Quorum-based protocol is intrinsically costly in terms of synchronization time among replicas, and employs incremental validation of the read-set at every read operation. In this scenario, checkpointing suffers from continuous abort and restore, nullifying the gain of partial rollback. Our work differs as it targets non-fault-tolerant DTM, where all operations are faster because they do not need to consider replication, thus being a previously unexplored data-point. Moreover, our analysis is more in-depth and manages to identify conditions favorable for checkpointing.

4. Proposed Algorithms

4.1 System Model

Let O = $\{O_1,O_2,...\}$ be the set of objects accessed using transactions. Every such object O_j has an unique identifier, id_j. For simplicity, we treat them as shared registers which are accessed solely through read and write methods, but such treatment does not preclude generality. Each object has an owner node, denoted by owner(O_j). Additionally, they may have cached copies at other nodes and they can change owners. A change in ownership occurs upon the successful commit of a transaction which modified the object.

Let T = $\{T_1, T_2, ...\}$ be the set of all transactions. Each transaction has an unique identifier. A transaction contains a sequence of operations, each of which is a read or write operation on an object on O. An execution of a transaction ends by either a commit (success) or an abort (failure). Thus, transactions have three possible states: active, committed, and aborted. Any aborted transaction is later retried using a new identifier.

Our nesting model is based on Moss and Hosking [18]. With transactional nesting, let $parent(T_k)$ denote the parent (enclosing) transaction of a transaction T_k. A root transaction has $parent(T_k) = \emptyset$. Sub-transactions are executed using either the flat or closed nesting models. For closed nesting, a read operation on an object O_k first looks at the current transaction's (T_k) read and write-sets. If a value is found, it is immediately returned. Otherwise, the read is attempted again from the context of $parent(T_k)$. Read operations are thus recursive, going up T_k's ancestor chain until a value is found. Write operations simply store the newly written value to the current transaction's write-set. The commit of a closed-nested transaction T_k merges $readset(T_k)$ into $readset(parent(T_k))$ and $writeset(T_k)$ into $writeset(parent(T_k))$.

4.2 TFA with Closed Nesting (N-TFA)

In TFA, transactions are immobile. Furthermore, all sub-transactions of a transaction T_k are created and executed on the same node as T_k. We name the commit operation as present in TFA the *top-level commit* model. This is used when a top-level transaction commits the changes from its replay-log to the globally committed memory. This commit is only performed after the successful validation of all objects in the transaction's read-set, as defined by the TFA algorithm. If the validation fails, i.e. at least one of the objects' version is newer than the current transaction's starting time, the transaction is aborted.

We further define a second type of commit, the *merge commit* model. This is used when a sub-transaction commits the changes from its replay-log to the replay-log of its parent.

Besides using the merge commit model, a TFA extension with support for closed nesting needs to address three issues: timestamps, object versioning and early validation.

To address all these issues we design a protocol called *Nested Transactional Forwarding Algorithm* (N-TFA).

N-TFA merge commits do not change object versions. Assume that transaction T_k opened and read an object O_1. Let T_{k2} be a sub-transaction of T_k. Assume that T_{k2} also reads object O_1, and moreover, T_{k2} can successfully commit (O_1 was not modified by any other transaction). Intuitively, T_{k2} should not update the object's lock version when it commits, because, the object as seen by other transactions did not change. If the version was updated at this point, other unrelated transactions would be forced to unnecessarily abort due to invalid read-set even if T_k eventually aborts (due to other objects) without changing O_1 in the globally committed memory.

All objects are validated against the outer-most transaction's starting time. While we could imagine an algorithm where sub-transaction's start times were used to validate objects, doing so would only add unnecessary complexity and would again provide no real benefit. Therefore, all transaction forwarding operations must be operated upon the starting time of the root transaction. This also implies that early validation operations (i.e., checking the consistency of objects in a transaction's read-set before advancing the transaction's starting time) must consider not only the current sub-transaction's read-set, but also all the objects in the read-sets of all the sub-transaction's ancestors.

Summarizing the previous observations, the starting time of sub-transactions is not used for object validity verification and the object versions are not updated upon a sub-transaction's commit. Consequently, merge-commits and the start of new sub-transactions are not globally important events and should not be recorded by incrementing node-local clocks. If the clocks were incremented on such events, remote nodes would need to perform the transaction forwarding operation unnecessarily, only to find that no objects were changed. This is undesirable as the forwarding operation bears the overhead of validating all objects in a transaction's read-set. Additionally, since no global objects are changed at merge-commits, no locks need to be acquired for such commits.

In case one or more objects are detected as invalid, the upper-most transaction that contains an invalid object and all of its children should be aborted. In the original TFA, it was sufficient to stop the validation procedure when the first invalid object is observed. However, with N-TFA, all objects within the root transaction must be validated in order to determine the best point to rollback to.

An example of N-TFA is shown in Figure 1. The top-level transaction T_k is executing on node N_1. A sub-transaction T_{k1} executes and commits successfully. Next, another sub-transaction T_{k2} opens an object O_1, which is located on

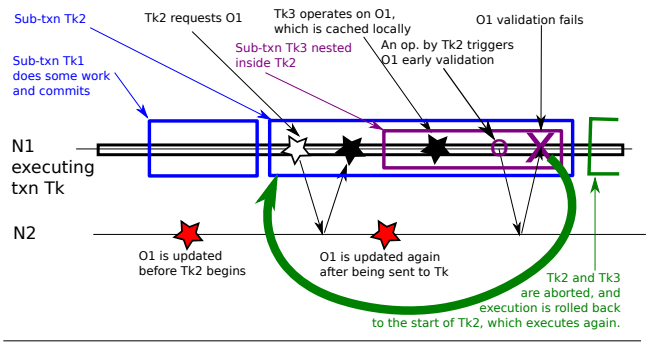

Figure 1. N-TFA Example

node N_2. T_{k2} spawns a further sub-transaction, T_{k3} which operates on O_1. Assume that at this point sub-transaction T_{k3} performs an operation that triggers an early validation, and O_1 is observed to be invalid. Under TFA, this would abort the root transaction T_k, including the work done by sub-transaction T_{k1}. N-TFA on the other hand only aborts as many sub-transactions as needed to resolve the conflict. In this case, only T_{k2} and T_{k3} need to abort. The transaction will be rolled-back to the beginning of T_{k2}, such that the next operation performed is retrieving a new copy of the previously invalid object, O_1.

N-TFA maintains the properties of the original TFA, in particular, opacity and strong progressiveness. Sketches of the proofs are available in [23].

4.3 TFA with Checkpoints (TFA-CP)

We designed and implemented TFA-CP, an extension of the TFA algorithm with support for transactional checkpoints.

Figures 2 and 3 show the key operations of TFA-CP. Of interest is the *startCheckpointedExec* routine which acts as an event-loop: it repeatedly passes execution to a user-supplied block of code, which is to be executed transactionally. The user-code, during its execution, calls DTM library functions, which are potential checkpoint locations. The system may use any of these calls to trigger recording a checkpoint. When it does, execution is passed back to the event-loop thus creating a new continuation. This continuation is stored alongside the current read and write-sets as the new checkpoint within the context of the current transaction. Finally, the execution is passed back to the user-code by resuming the previously created continuation. Except for recording a checkpoint as described above, there are two other occasions when execution is passed from the user-code to the event-loop: on transaction completion and on the detection of a conflict. In the first case, the event-loop is tasked to commence the commit operation, and upon success, the loop is terminated. In the second case, and also if the commit fails, the system determines which checkpoint should the transaction be reverted to, in order to resolve the conflict while aborting a minimal amount of work. The appro-

```
class HaiTxnLevel(val parLevel: HaiTxnLevel, cont: Continuation)
{ //... }
class HaiInTxn extends InTxn {
  // Wrapper for the transaction.
  // Needs to be a Runnable to be started as a continuation.
  class HaiRunCkpt[Z](block: InTxn => Z) extends Runnable {
    def run() {
      // Execute block
      val res = block(HaiInTxn.this)
      // End transaction, return from continuation to commit.
      CheckpointStatus.set(CheckpointSuccess(res))
      Continuation.suspend()
  } }

  // Main entry point for transaction. Takes a function (block)
  // as an argument. Returns the value returned by a successful
  // transactional execution of the given function.
  def startCheckpointedExec[Z](block: InTxn => Z): Z = {
    // This var stores the list of checkpoints.
    var level = new HaiTxnLevel(null, null)

    // Start executing the transaction
    var cont = Continuation.startWith(new HaiRunCkpt(block))
    // Here we returned from the first continuation

    while (true) {
      // Why did the continuation return?
      CheckpointStatus.get match {
        case res: CheckpointInterim =>
          // Transaction requests a new checkpoint, store it.
          level = new HaiTxnLevel(level, cont)
          level.mergeFrom(level.parLevel)
          // Continue
          cont = Continuation.continueWith(cont)
        case res: CheckpointFailure =>
          // Conflict detected. Transaction requests rollback.
          val restartRes = restartAtInvalidLevel(block)
          level = restartRes._1
          cont = restartRes._2
        case res: CheckpointSuccess =>
          // This transaction is over, attempt commit.
          if (tryCommit()) {
            // Success
            return res.result.asInstanceOf[Z]
          } else {
            // Commit failed, rollback to appropriate checkpoint.
            val restartRes = restartAtInvalidLevel(block)
            level = restartRes._1
            cont = restartRes._2
  } } } }
```

Figure 2. Pseudo-code for TFA-CP.

```
  // Rolls back transaction
  def restartAtInvalidLevel[Z](block: InTxn => Z):
          Tuple2[HaiTxnLevel, Continuation] = {
    // Find out how far back we need to abort
    var level = _currentLevel.root
    while (level.status == Txn.Active
          && level.activeChild != null) {
      level = level.activeChild
    }
    // Resume execution at that checkpoint
    var cont = level.cont
    level = new HaiTxnLevel(level.parLevel, cont)
    if (level.parLevel != null) {
      level.mergeFrom(level.parLevel)
    }
    if (cont != null) {
      // If we have a continuation, resume there
      cont = Continuation.continueWith(cont)
    } else {
      // Otherwise, re-start at the beginning
      cont = Continuation.startWith(new HaiRunCkpt(block))
    }
    return (level, cont)
  }

  // Performs transactional forwarding
  def forward(rclk: Long) {
    if (rclk > _currentLevel.root.startTime) {
      // Check for read-set validity
      if (rsValidate) {
        // Valid read-set, update Txn start time
        _currentLevel.root.startTime = rclk
      } else {
        // Invalid read-set, abort
        // Since we abort to the last valid level,
        // it is OK to update starting time
        _currentLevel.root.startTime = rclk
        CheckpointStatus.set(CheckpointFailure())
        Continuation.suspend()
} } } }

// Checkpointing-enabled directory manager
class CpDirectory extends Directory {
  override def open[T](id: String): T = {
    // Record checkpoint checkpoint
    CheckpointStatus.set(CheckpointInterim())
    Continuation.suspend()
    // Continuation returned, open as usual
    return super.open(id)
} }
```

Figure 3. Pseudo-code for TFA-CP. (continued)

priate continuation is then resumed, passing control back to the user-code.

TFA-CP currently stores checkpoints before object retrieval operations. The granularity of checkpointing can be configured using two distinct ways, either by specifying the probability P that each object retrieval would record a checkpoint, or specifying that a checkpoint will be taken every E object opens. Clearly $P = 100$ and $E = 1$ both characterize the finest grained strategy, which always allows resuming execution at the exact operation that would resolve the current conflict. $P < 100\%$ (or $E > 1$) can be used to reduce the total number of checkpoints recorded and thus, any overhead associated with capturing continuations.

In order to avoid superfluous checkpoints, the first object opened in a transaction never triggers a checkpoint. Also exempt are object re-opens, when the current transaction already has a cached copy of the requested object. Checkpoints are stored in a doubly-linked list, with new checkpoints inserted at the head of the list. Each checkpoint stores the complete read and write-sets of the current transaction at the time the checkpoint is taken. This makes read operations fast (they do not have to traverse the list), at the cost of increased memory consumption.

5. Implementation

N-TFA and TFA-CP were implemented in our high-performance DTM framework for the JVM, Hyflow2 [24]. Hyflow2 is written in Scala. Its architecture is modular, and the implementation makes use of the *actor model* extensively. Actors are a concurrency abstraction that encapsulate private data, and communicate externally solely through message passing. The actor library used under the covers is Akka [26], which in turn relies on the Netty asynchronous network li-

brary to provide communication with remote actors in a completely network-transparent manner.

Unfortunately, standard JVMs do not provide any support for continuations. To exploit continuations in Java, one would have to either use a non-standard JVM (e.g., *Avian JVM*, *DaVinci JVM* with the continuation patch) or employ a byte-code rewriting library (e.g., *JavaFlow*, *LightWolf*). We experimented with the *DaVinci JVM* and with the *JavaFlow* library. The former is faster, because continuations are implemented in native code. It requires the users to run a non-standard JVM, which unfortunately is not easily available and needs to be compiled from older source code with several patches applied.

The latter choice, *JavaFlow*, is able to run on stock JVM, but is slower because it implements the continuation mechanism in Java code. *JavaFlow* stores all local variables in a per-thread stack structure which replaces and emulates the regular call stack. This replacement stack is under the control of the library: it is backed-up when suspending a continuation and later restored when resuming it.

Both *JavaFlow* and *DaVinci* proved to support resuming the same continuation multiple times — a feature that is essential for implementing transaction checkpoints. In *DaVinci JVM* this functionality is undocumented and required a small modification to enable. On the other hand, *JavaFlow* turned out to be incompatible with many of Scala's features. At first we attempted to rewrite our code to work around these incompatibilities, but the problems were difficult to trace. Moreover, regular transaction user-code would be severely restricted and hard to debug. We thus decided *DaVinci JVM* was the better platform choice for implementing TFA-CP.

In the early experiments we performed, we noticed it is very easy to skew the results in favor of one rollback model or another, simply by using inconsistent random back-off strategies. Thus, a significant amount of time was spent debugging the back-off implementation, making sure it is correct and consistent across rollback models. We used the same back-off settings for all rollback models. Choosing the best back-off setting for each model could make the topic of a separate study in its own.

6. Evaluation

Our implementation was evaluated on a testbed consisting of up to 24 emulated nodes, communicating over loopback TCP. Each such node spawns two CPU cores available on a 48-core AMD Opteron machine running at 1.7GHz. We verified that the behavior under this setup is very similar to using genuinely distributed nodes connected using Gigabit Ethernet, as long as the network is not driven to saturation. The operating system is Ubuntu Linux 10.04, and the OpenJDK HotSpot version (prerequisite for the continuations patch) is *19.0-b03*, a beta release from circa 2010.

We used a set of five micro-benchmarks (Bank, BST, RBT, Hash-Table and Skip-List), one macro-benchmark

Metric Type	Records
Meter	Event count, mean rate, per interval rate
Histogram	Minimum, maximum, mean, median, standard deviation, 95%, 99%, 99.9%
Timer	Same as a meter for the event's rate. Same as a histogram for the event's duration.

Table 1. Description of recorded metric types.

Transactions by outcome. **Timers**: non-aborting transactions, first aborted attempt, subsequent aborted attempts, final successful attempt.
Checkpoint operations. **Timers**: saving, resuming.
Contention management. **Timers**: random back-off. **Histograms**: aborted nesting levels or checkpoints, nesting level or checkpoint aborting to, previous aborts. **Meters**: aborts by cause (invalid readset, object locked, etc.)
Front-end operations (as seen by the transaction thread). **Timers**: locating an object, retrieving an object
Back-end operations. **Timers**: locating, retrieving, validating, updating, locking, unlocking objects.
JVM metrics. **Histograms**: number of garbage collections, time taken by GC (for 5s intervals).

Table 2. Recorded metrics by category and type.

(a version of TPC-C adapted for running within the restrictions of Hyflow2), and a synthetic counter application where we could control the location within the transaction likely to generate conflicts. Each node spawns transactions back-to-back using two benchmark threads. For the micro-benchmarks, each transaction consists of several data-structure operations. In case of closed nesting, each operation is executed in its own sub-transaction, all nested directly under the root transaction. For TPC-C, closed nesting was not applicable and was omitted.

Each test was allowed to run for 150 seconds in order for the code to be JIT-compiled before the measurements were started. Next, throughput and a collection of other metrics[2] were measured over an interval of 60 seconds. The metrics we collected are summarized in Tables 1 and 2. For each configuration, a number of rollback strategies were investigated. They are (figure legend identifiers in parentheses):
nesting: flat nesting (*flat*), closed nesting (*closed*);
fine checkpointing: checkpoint on every object (*cp100*), checkpoint on every object but always rollback to the beginning (*cp-flat*);
coarse checkpointing: every N^{th} object (*cp-e3*, *cp-e7*, etc);
zero checkpointing: saves no checkpoints (i.e., no partial rollback), but use the continuations mechanism to pass execution out of a transaction in case of an abort (*cp-zero*);
manual checkpointing (*cpman*).
The *cp-flat* strategy was implemented to decouple the effects of partial rollback from the overheads of checkpointing.

[2] Using Coda Hale's Metrics library: http://metrics.codahale.com/

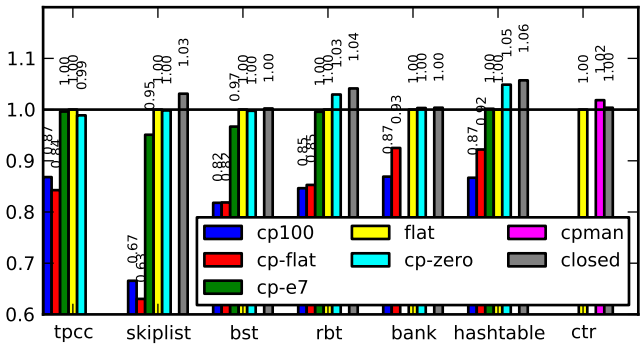

Figure 4. Per-benchmark summary of our results. Plots throughput relative to flat nesting.

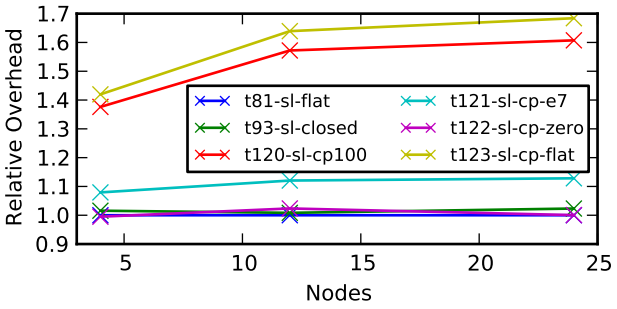

Figure 5. Average time taken by non-aborting transactions, relative to flat nesting, on Skip-List.

Benchmark	Saving a Ckpt	Restoring a Ckpt
Hash-Table	120-145 us	37-44 us
Skip-List	155-180 us	30-40 us
TPC-C	90-165 us	20-45 us

Table 3. Average durations for Saving and Restoring Checkpoints under various benchmark configurations.

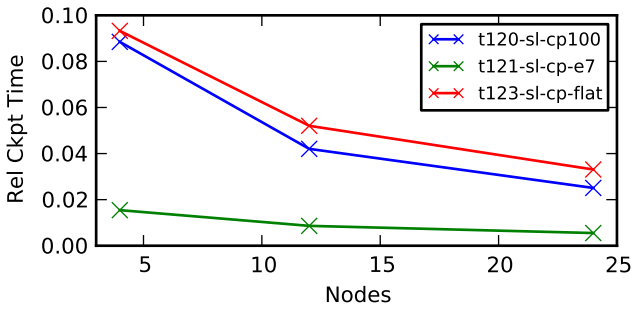

Figure 6. Time spent saving checkpoints relative to the total transaction time, on Skip-List.

	Mean	Median	Stddev
cp100	960	450	6400
cp-e7	725	475	3600
flat	710	500	3200

Table 4. Duration values in microseconds for the front-end object get operation, on Skip-List at 24 nodes.

Figure 4 presents a summary of our results. Two of our rollback strategies were on average just slightly faster than flat nesting: closed nesting by 3% and *cp-zero* by 1%. As *cp-zero* does not enable partial rollback, its benefits are only due to the continuations mechanism being faster than exceptions, in the absence of storing any checkpoints. The best checkpoint strategy in our test was *cp-e7* and only 2% slower than flat nesting. Fine-grained checkpointing strategies, with or without partial rollback, were 17-18% behind flat.

To better understand these results, we analyze the collected data from three perspectives: overheads incurred by the mechanisms used to implement the different strategies, the behavior of the different partial rollback strategies, and the effects of variables characterizing the workload. Additional plots are provided in a technical report [25].

6.1 Overheads

For the purpose of this section, we will focus on the Skip-List benchmark, where checkpointing performed worst. The other data-structure benchmarks and TPC-C followed similar trends, but to a lesser extent.

To understand the overheads of each mechanism, we look at the average time taken by abort-free transactions. Figure 5 shows this metric normalized against flat nesting. We can notice that closed nesting and *cp-zero* are within 2-3% of flat nesting, *cp-e7* within 12%, while *cp100* and *cp-flat* are 60%, and respectively 70% slower. The same trends can also be observed for the duration of the first execution attempt in a transaction with aborts.

We compare these slow-downs with the time spent in checkpoint operations. Table 3 shows typical values for the duration of checkpoint save and resume operations. We use this data and compute the fraction of total transaction time that is spent saving checkpoints. The result is shown in Figure 6: saving checkpoints (using *cp100*) takes 3-9% of the total transaction time. Time spent resuming checkpoints is negligible. Thus, the time taken by checkpoint operations accounts for only a small part of the increase in execution time of each transaction.

To further try and explain this slow-down, we looked at the time taken by the various operations performed during transaction execution, such as locating, retrieving and validating objects. We measured this time from both the transaction's perspective (which is likely sending a request across the network), and from the back-end worker actor's perspective (which services the requests). We noticed the mean duration for the same operation is higher for the fine-grained checkpointing strategies. The difference is small but notice-

	GC runs	GC time
cp100	2.60	133 ms
cp-e7	2.20	85 ms
flat	2.37	70 ms

Table 5. Garbage Collector statistics, on Skip-List, at 24 nodes. Reported are the number of collections and time taken by the collector over a five seconds interval.

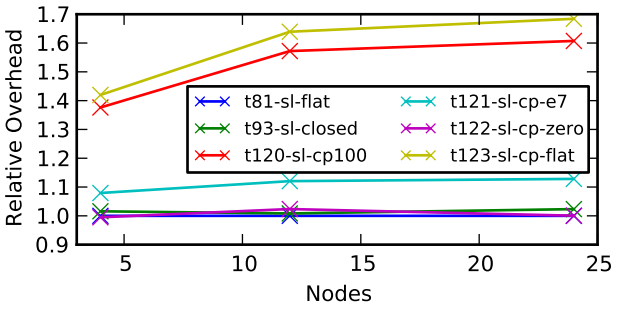

Figure 7. Average time taken by non-aborting transactions, relative to flat nesting, on BST.

able on the back-end, but quite significant (25-30%) on the front-end. Furthermore, the median values do not show such differences, while standard deviation values are very large (Table 4). This could be explained if a small fraction of the operations take significantly more time than the rest.

The issue described above is reminiscent of garbage collection, but we have found no evidence that would support any claims of causality. While fine-grained checkpointing does lead to increased garbage collection activity, the collector runs roughly once every 2 seconds and takes no more than 1.5% of the CPU time (Table 5). The impact on transaction throughput would be minimal, and would not lead to the large standard deviation values observed.

We were not able to pinpoint the cause of this slowdown, but we suspect it may be specific to the particular implementation of continuations that *DaVinci JVM* is using. We were however able to find a parameter that influences it. Figure 7 shows the effect of the length of a transaction on the slowdown relative to flat nesting. Thus, a workload comprising of long transactions which operate on many objects will incur higher overheads compared to workloads consisting of short transactions. This is indeed the case for the Skip-List benchmark, where checkpointing performs worst.

6.2 Partial Rollback Effects

In this section we aim to show the effects of partial rollback on transaction execution, where any benefits originate from, and what are the limiting factors.

We delimited the three execution stages of transactions with aborts: the *first abort* (from the beginning of the transaction until the first conflict is detected), subsequent *failed*

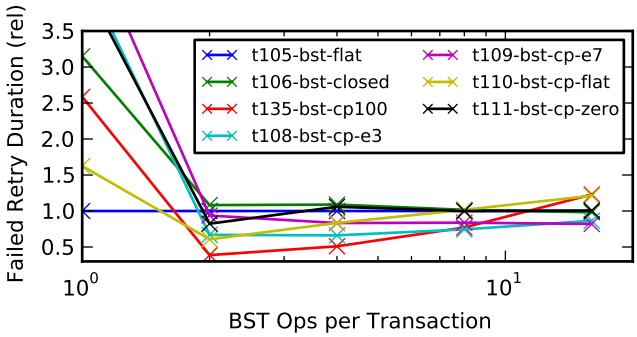

Figure 8. Relative duration of individual failed retries, BST.

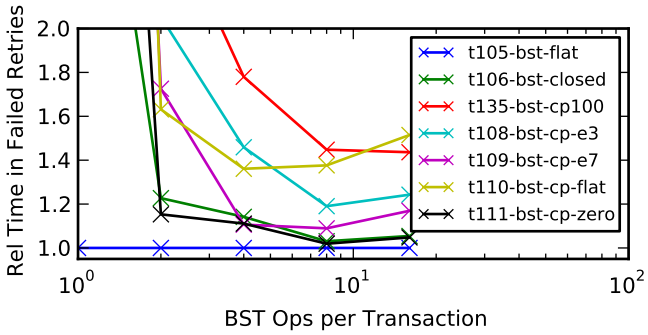

Figure 9. Fraction of total time spent in failed retries, relative to flat nesting, on BST.

retries (in-between two consecutive conflicts), and the final *successful retry* (from the last conflict until commit). We measured the execution rate and duration of each of these stages, and we can thus examine a breakdown of where the execution time is spent.

Figure 8 shows how the average duration of a failed retry generally decreases with more fine-grained partial rollback (except for very long transactions, as explained in Section 6.1, and low-contention workloads, when rollbacks are rare). A similar trend occurs for for the final successful retry. However, when aggregated across all transactions, this trend is reversed and more fine-grained rollback strategies actually spend more time in failed retries (Figure 9). This reversal can be explained by a disproportional increase in the rate of aborts, as confirmed by our data. The benefits due to shorter retries can not keep up with the repeating aborts.

6.3 Workload Characteristics

Throughout our experiments we varied a number of workload parameters, such as the ratio of read-only transactions, number of objects available in the system, number of participating nodes, and length of a transaction. All these parameters affect contention. When contention is low, effects due to partial rollback are negligible, and the transaction throughput depends solely on the overheads of the implementation.

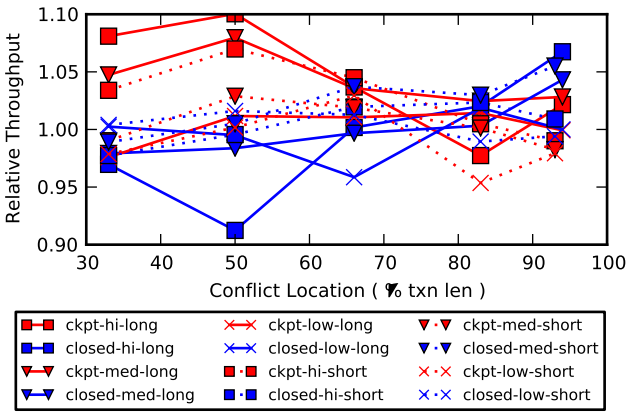

Figure 10. Enhanced Counter results. Labels given as rollback model – contention level – transaction length. Note that contention is very high (72%) even on the *low* setting.

In all other cases, fine-grained partial rollback led to repeated aborts, which with the exception of closed nesting, canceled any benefits gained by shorter retries.

However, our analysis so far only covered workloads where aborts lead to relatively large rollbacks, because the conflicting objects were accessed early in the transaction. Using our Enhanced Counter benchmark, we tried to explore situations where a sizable portion at the beginning of the transaction would be relatively conflict-free. Conflicts would be significantly more likely in later parts of the transaction, where checkpointing could perform better thanks to the short abort and retry cycles. If the partial rollback provides sufficient benefit, it may be able to offset the checkpointing overheads. Automatic checkpointing was disabled in an attempt to reduce overheads, and two manual checkpoints were taken right before and after the high conflict probability operation.

The results of this experiment are presented in Figure 10. Checkpointing was able to match the performance of closed nesting, and to a small extent even exceed it. The maximum speed-up we observed compared to flat nesting was 10%. This speed-up occurs for very high contention (in excess of one abort per commit), for long transactions, when the conflicts are concentrated around the middle of the transaction.

7. Discussion and Future Work

Our analysis was shaped by two design choices we made early in the study: the transactional protocol we based this work on, and the Java platform with a specific implementation of continuations and its intrinsic overheads.

We thus found that checkpointing always started with a 5-100% performance degradation. The overheads can be reduced towards the lower end of the range by increasing the granularity of checkpoints. Despite that, as long as conflicts are either uniformly probable throughout the duration of the transaction, or concentrated early in the transaction, partial rollback using checkpointing did not enable sufficient performance gain to offset the overheads.

However, we believe these overheads could be further reduced, either by using a better optimized implementation of continuations in the JVM, or by changing platforms altogether. Lower level languages like C/C++ have the potential to increase overall performance by eliminating the additional virtual machine layer and uncertainties like the garbage collector behavior. Moreover, C/C++ provide a lighter-weight continuations mechanism, the *getcontext/setcontext* function family, that only saves and restores the CPU registers, leaving the activation stack under the control of the program. This approach would incur smaller overheads that are concentrated only within the checkpointing operation itself.

A C++ DTM framework containing an implementation of the algorithms we proposed was already completed (HyflowCPP [14]). Preliminary results suggest that, in such an environment, both partial rollback models can significantly benefit from the lower overheads of C++. Moreover, checkpointing is able to consistently outperform closed nesting. We leave a thorough evaluation of N-TFA and TFA-CP as implemented in HyflowCPP for future work.

The other important factor affecting our analysis is TFA, our base protocol. TFA applies well for workloads which do not exhibit data locality, and in high contention situations. Moreover, the rollback in itself is passive and cheap: there are no compensating actions to execute. If we would have used an algorithm with different characteristics, our results may be different. For example, checkpointing would bring more benefit if rollbacks are expensive, such as in the case of undo logging or open nesting.

8. Conclusion

This paper addressed the problem of partial rollback in a distributed transactional memory system. We extended the Transactional Forwarding Algorithm with support for closed nesting and checkpointing, naming the resulting protocols N-TFA, and respectively, TFA-CP. We then proceeded with an extensive evaluation study to determine the performance gains enabled by partial rollback, where these gains originate from and what are the limiting factors.

Closed nesting, when applicable, turned out to be consistently faster than flat nesting, but the average improvement was small at only 3%. Checkpointing on the other hand incurred significant implementation overheads that partial rollback was generally unable to overcome. The average performance degradation we observed was 17-18%. While we were unable to pinpoint the source of the overheads, we managed to link them to the length of the transaction as measured in the number of accessed objects. We further determined the specific conditions when checkpointing overheads can be overcome: high-conflict workloads when conflicts are concentrated around the middle of the transaction.

Acknowledgements

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

A. Bank Plots

Note: plots in appendixes have not been curated and may contain errors.

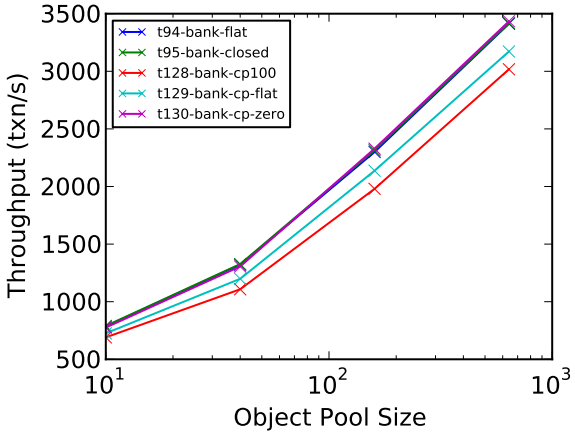

Figure 11. Absolute throughput, in Txn/s.

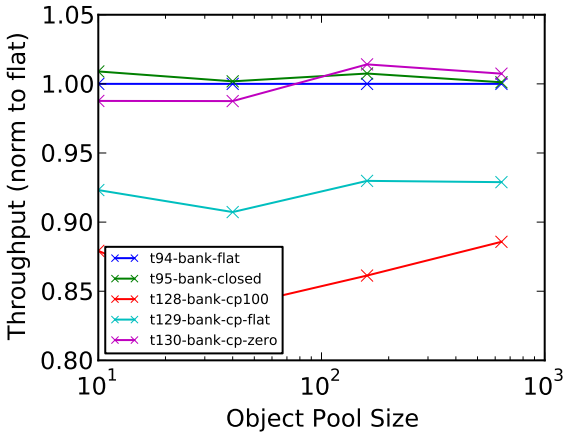

Figure 12. Througput, normalized to flat.

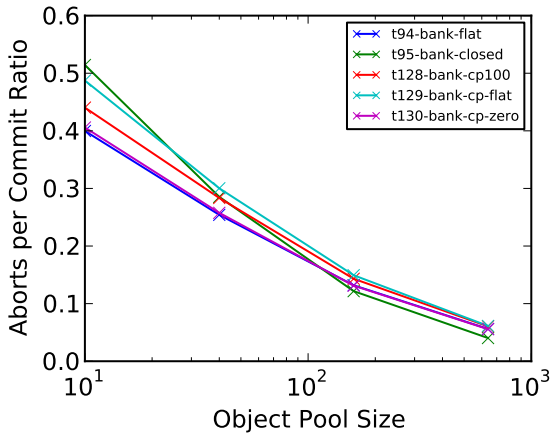

Figure 13. Aborts per commit ratio.

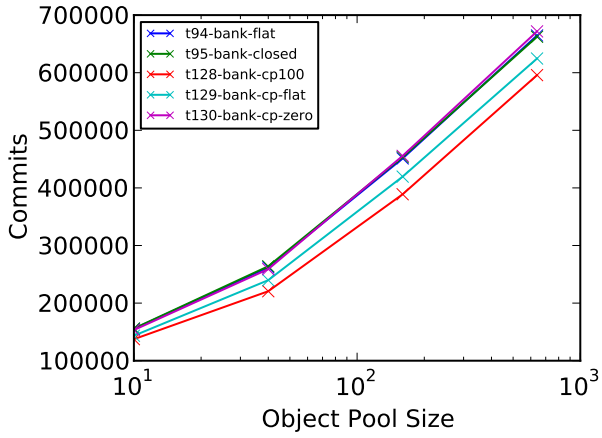

Figure 14. Total number of commits.

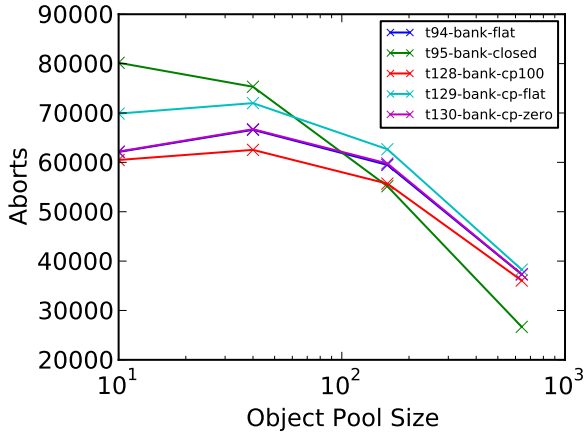

Figure 15. Total number of aborts.

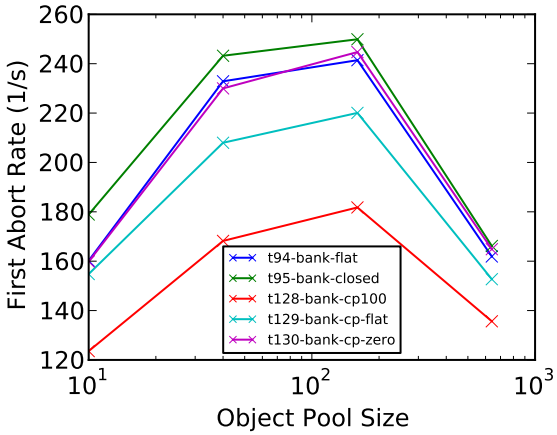

Figure 16. First Abort rate.

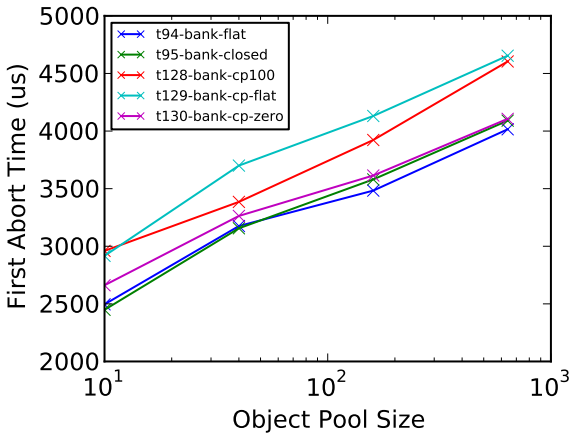

Figure 17. First Abort average duration.

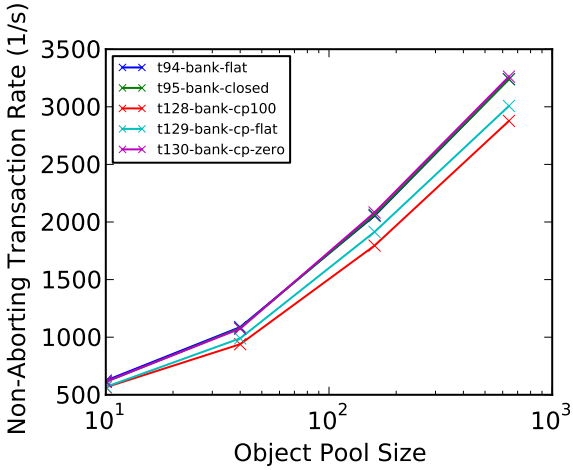

Figure 18. Non-Aborting Transaction rate.

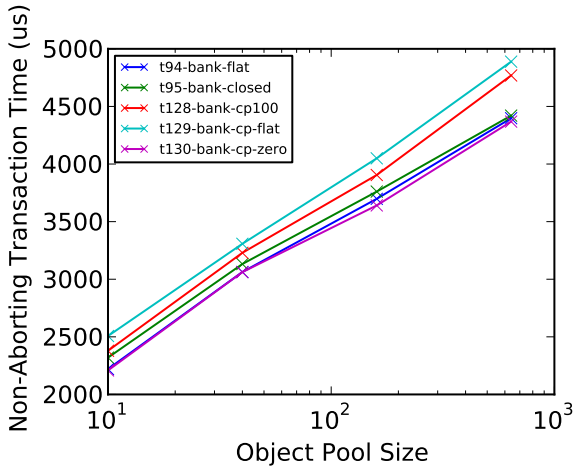

Figure 19. Non-Aborting Transaction average duration.

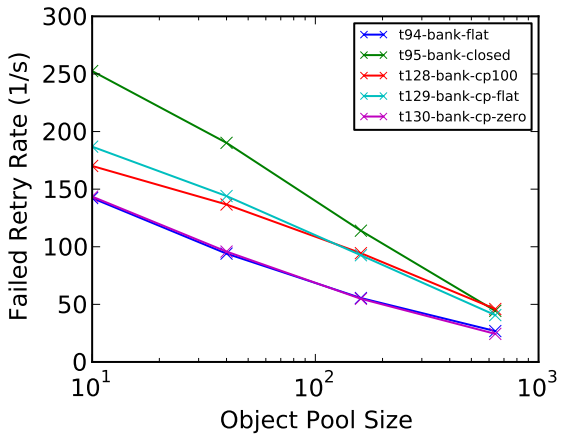

Figure 20. Failed Retry rate.

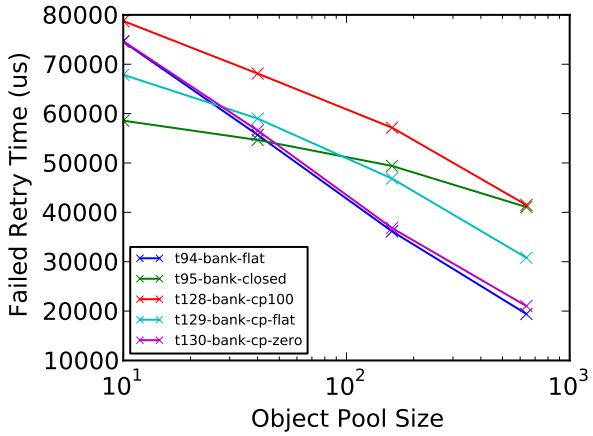

Figure 21. Failed Retry average duration.

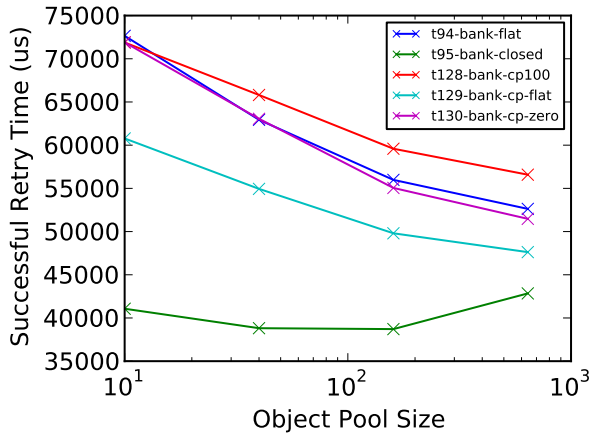

Figure 22. Successful Retry average duration.

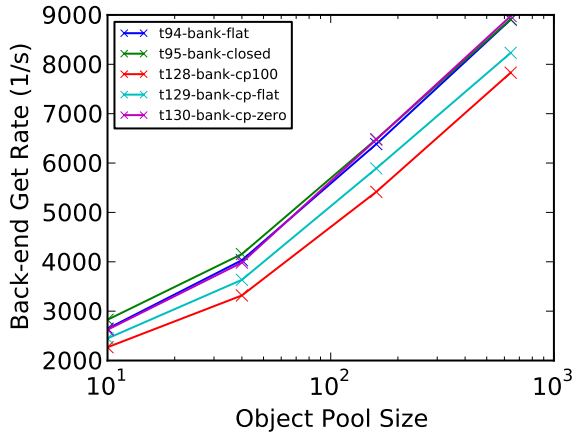

Figure 23. Back-end Get rate.

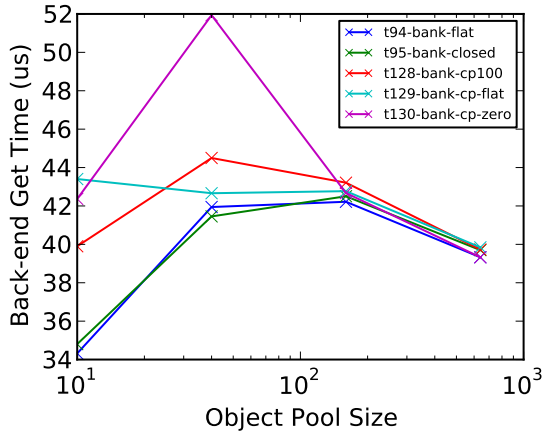

Figure 24. Back-end Get average duration.

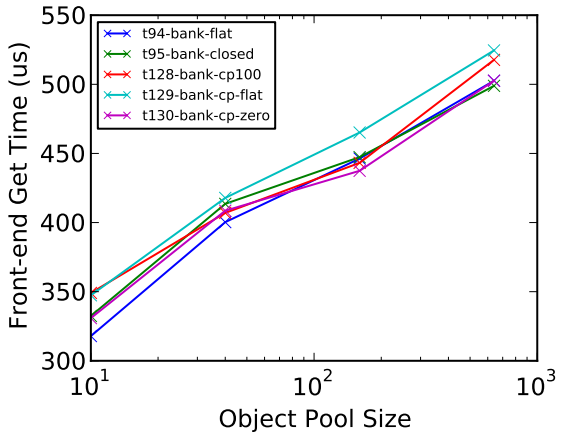

Figure 25. Front-end Get average duration.

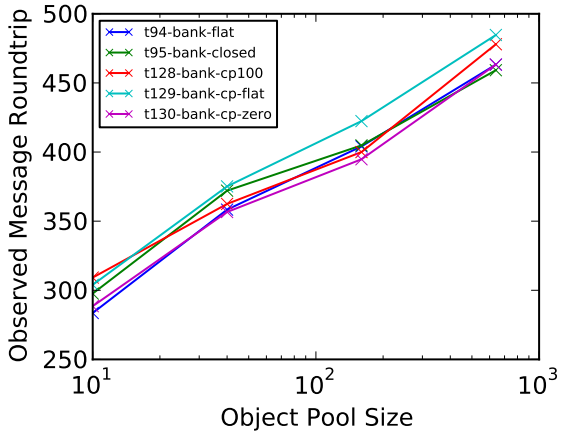

Figure 26. Observed round-trip latency.

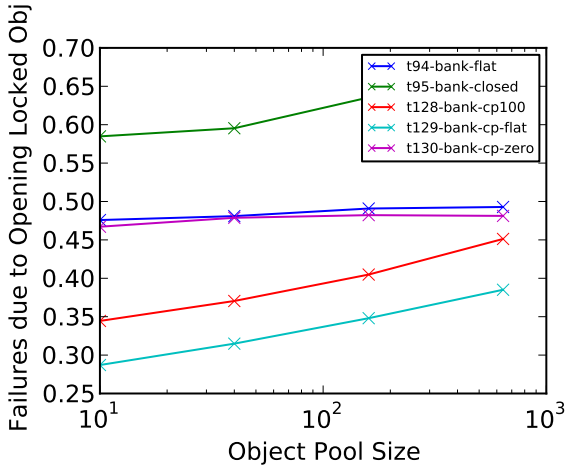

Figure 27. Fraction of aborts due to object locked at open time.

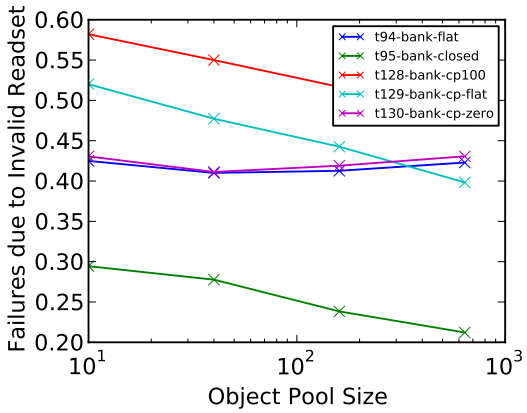

Figure 28. Fraction of aborts due to invalid read-set.

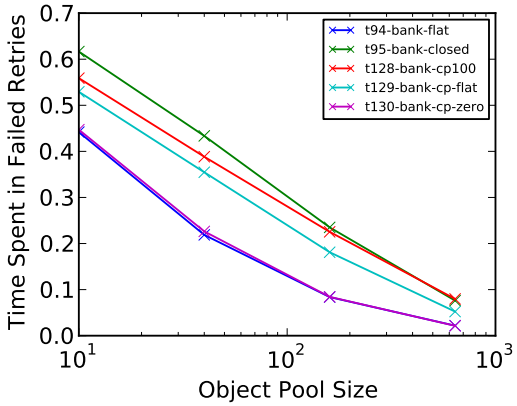

Figure 29. Fraction of time spent in failed retries.

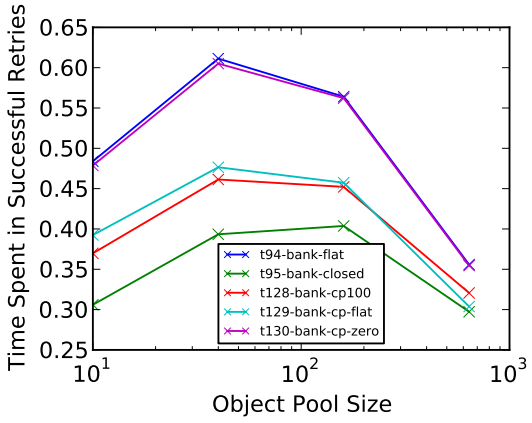

Figure 30. Fraction of time spent in successful retries.

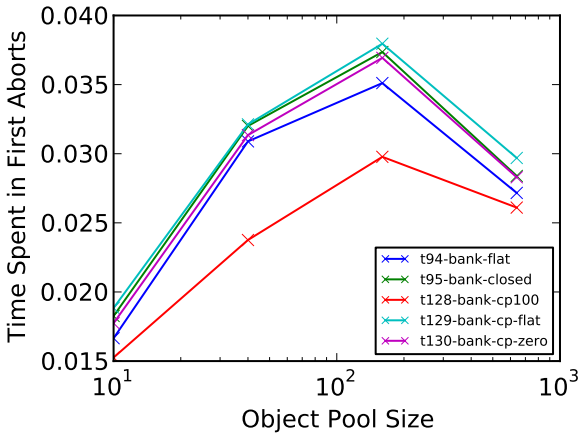

Figure 31. Fraction of time spent in first aborts.

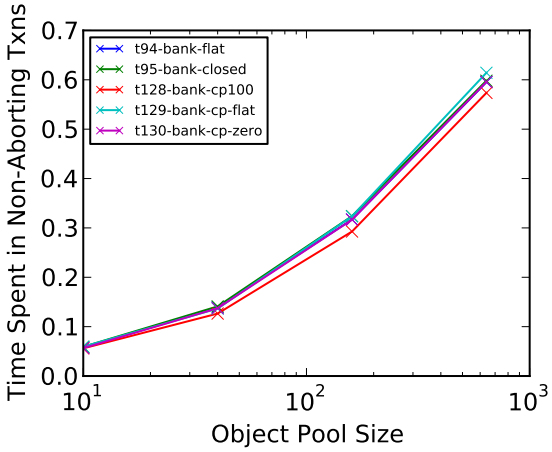

Figure 32. Fraction of time spent in non-aborting transactions.

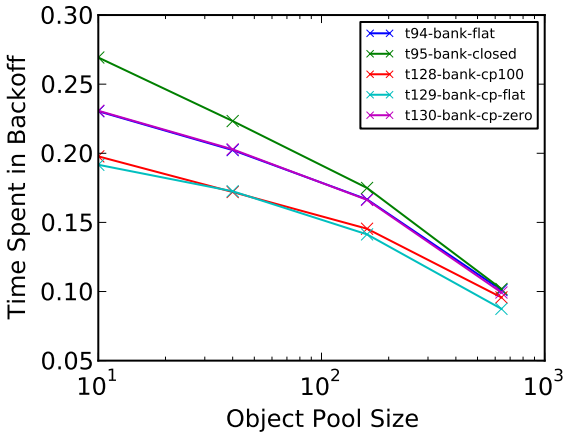

Figure 33. Fraction of time spent in back-off.

B. BST Plots

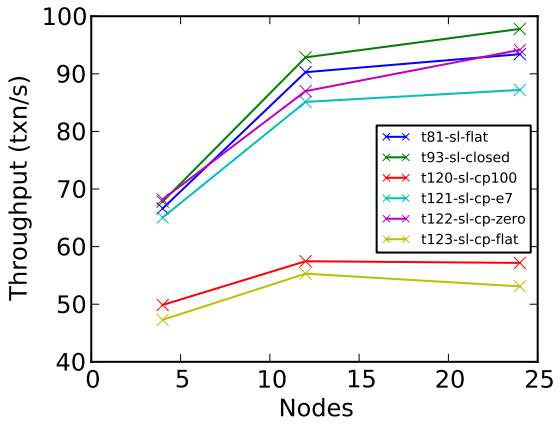

Figure 34. Absolute throughput, in Txn/s.

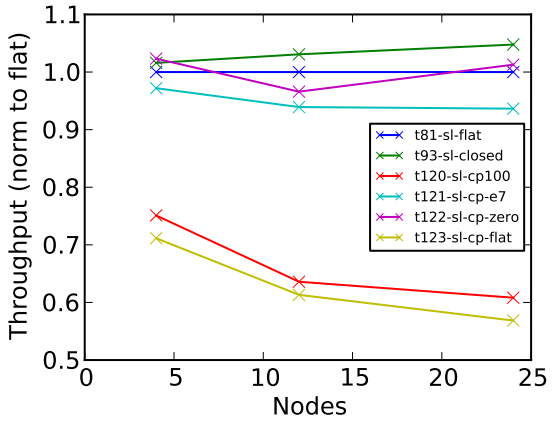

Figure 35. Througput, normalized to flat.

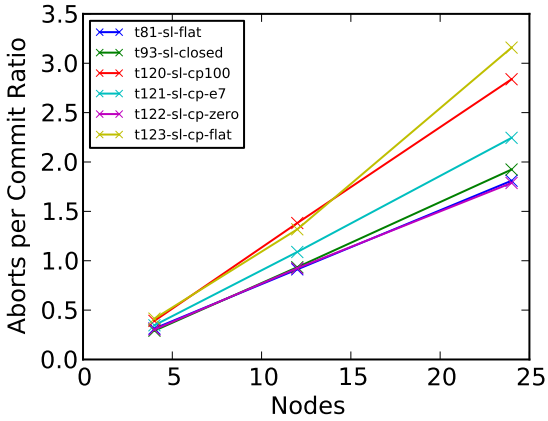

Figure 36. Aborts per commit ratio.

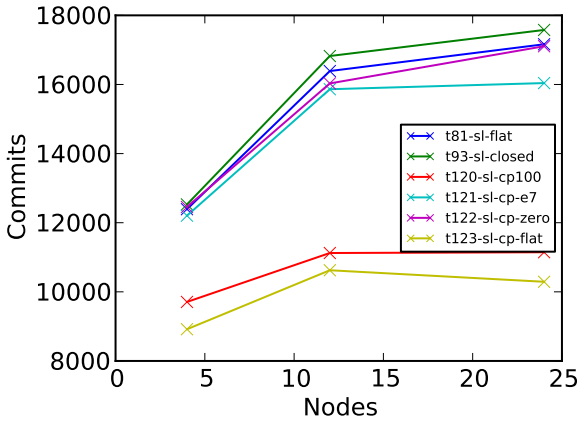

Figure 37. Total number of commits.

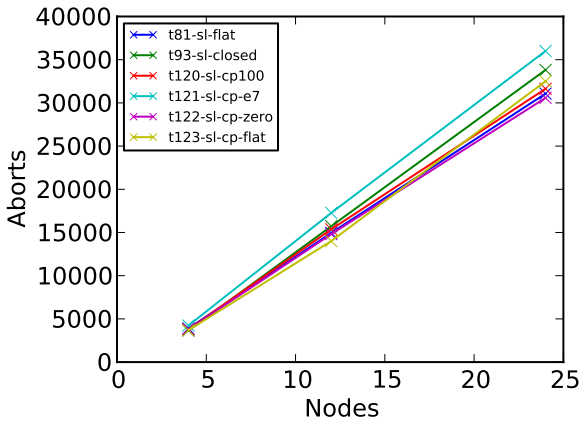

Figure 38. Total number of aborts.

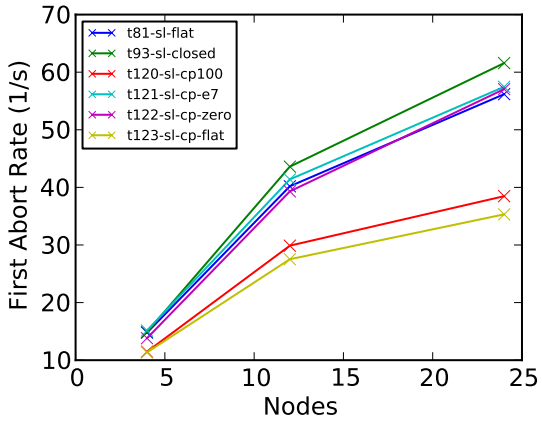

Figure 39. First Abort rate.

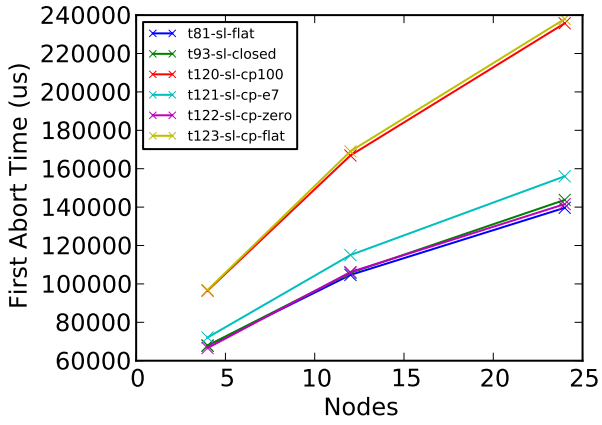

Figure 40. First Abort average duration.

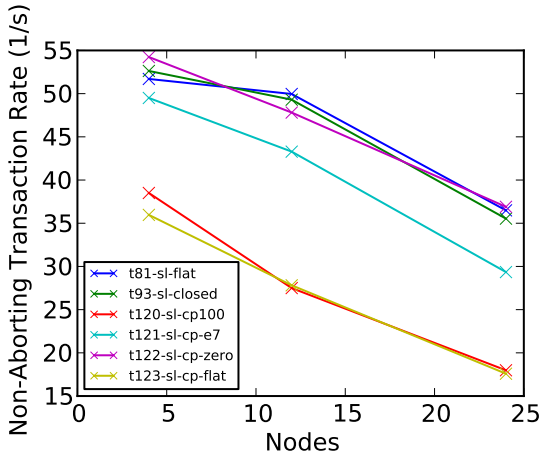

Figure 41. Non-Aborting Transaction rate.

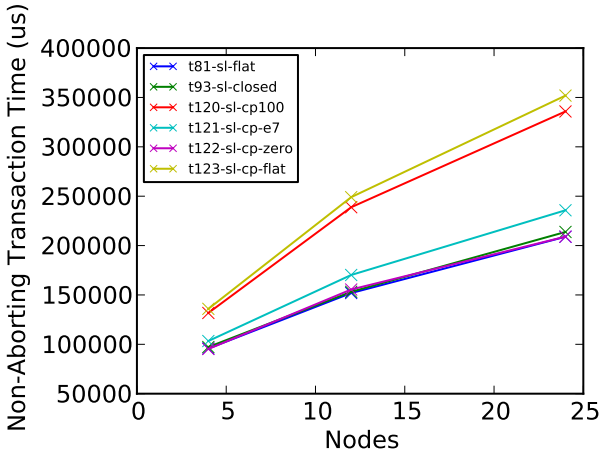

Figure 42. Non-Aborting Transaction average duration.

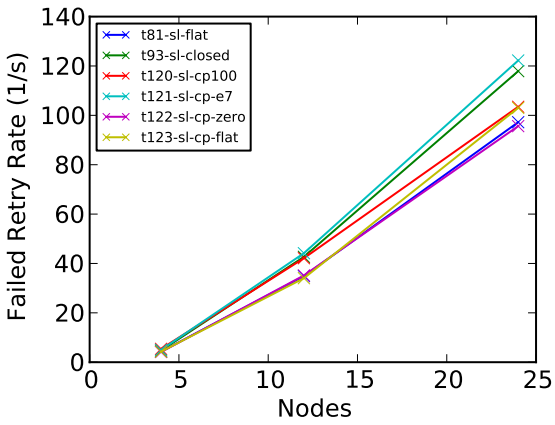

Figure 43. Failed Retry rate.

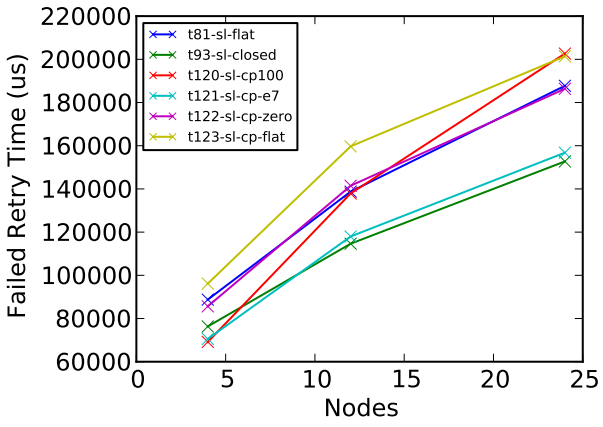

Figure 44. Failed Retry average duration.

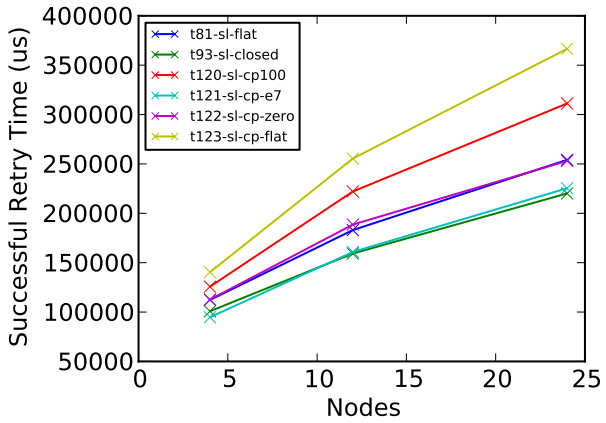

Figure 45. Successful Retry average duration.

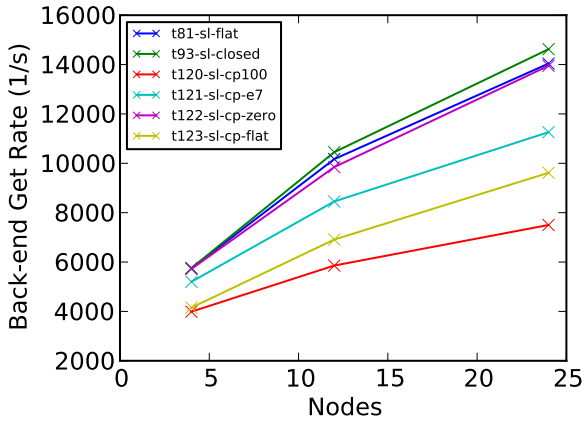

Figure 46. Back-end Get rate.

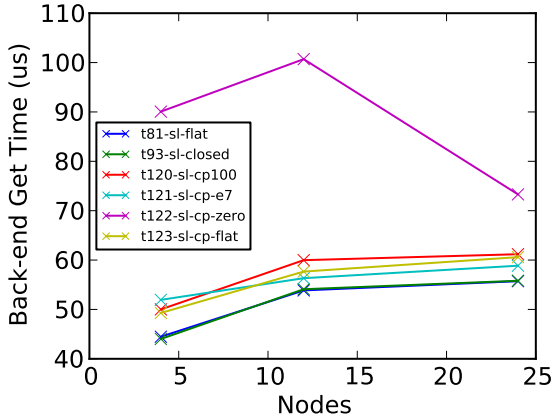

Figure 47. Back-end Get average duration.

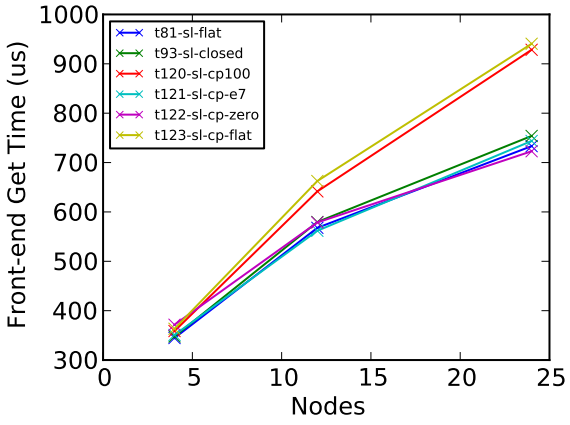

Figure 48. Front-end Get average duration.

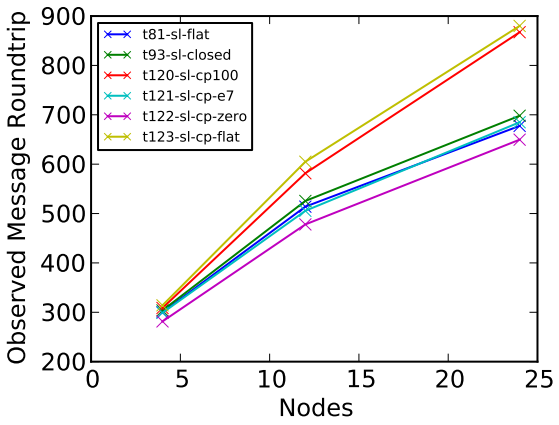

Figure 49. Observed round-trip latency.

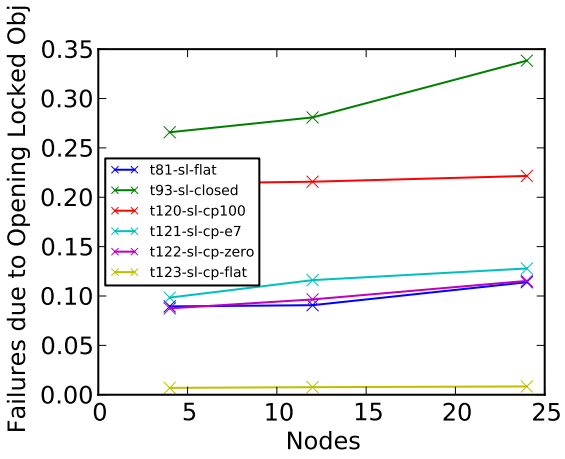

Figure 50. Fraction of aborts due to object locked at open time.

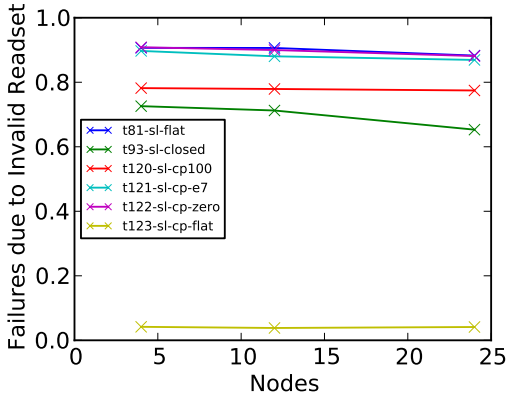

Figure 51. Fraction of aborts due to invalid read-set.

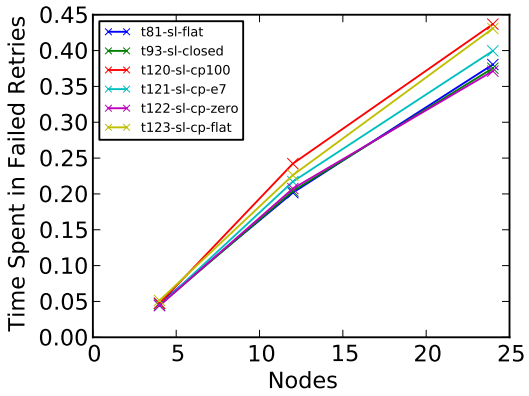

Figure 52. Fraction of time spent in failed retries.

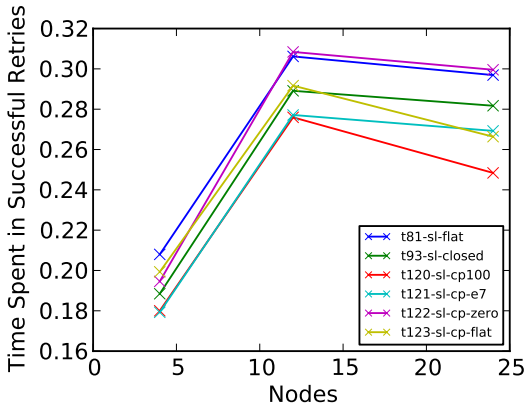

Figure 53. Fraction of time spent in successful retries.

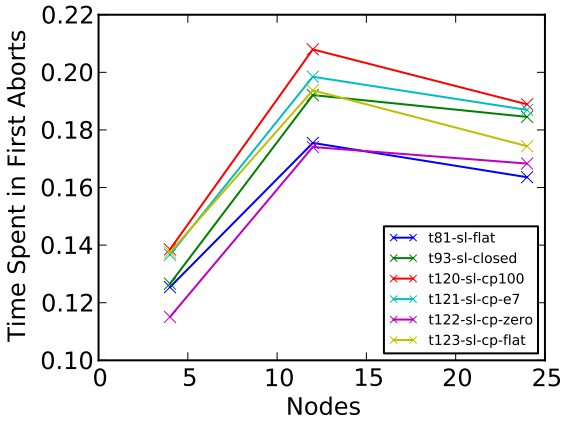

Figure 54. Fraction of time spent in first aborts.

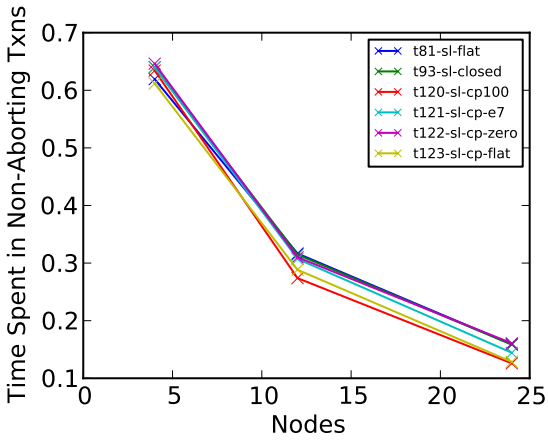

Figure 55. Fraction of time spent in non-aborting transactions.

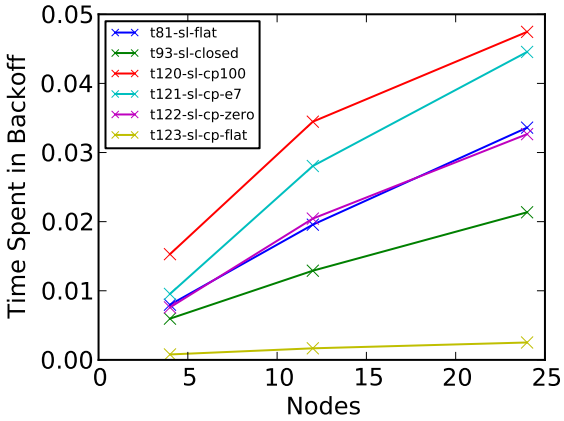

Figure 56. Fraction of time spent in back-off.

C. TPCC Plots

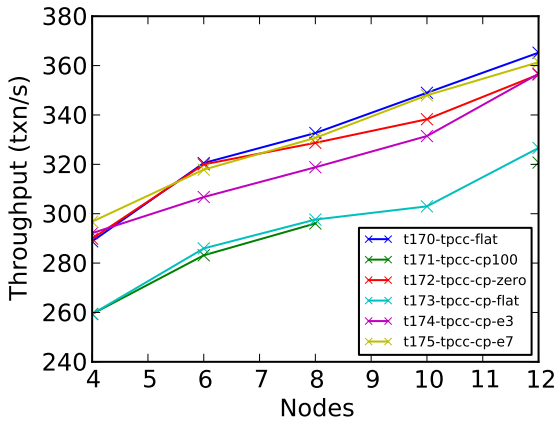

Figure 57. Absolute throughput, in Txn/s.

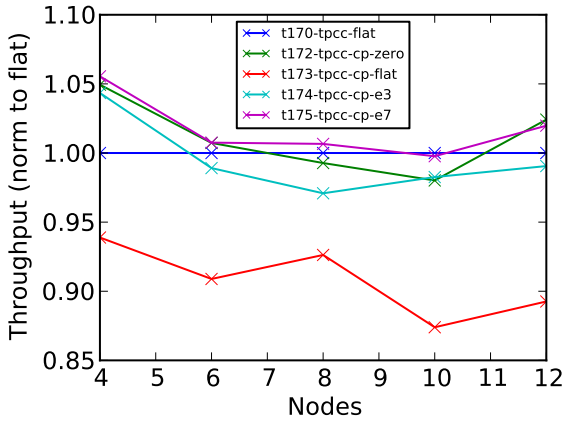

Figure 58. Througput, normalized to flat.

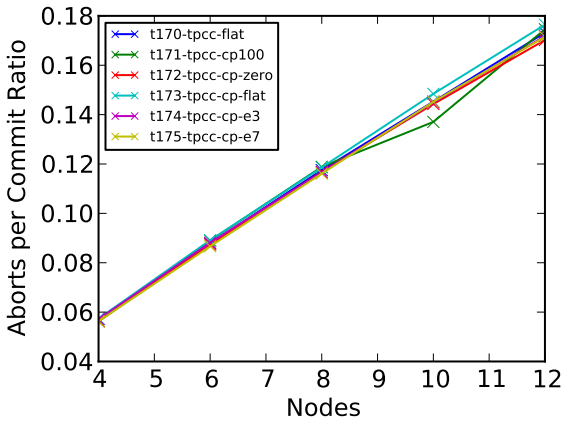

Figure 59. Aborts per commit ratio.

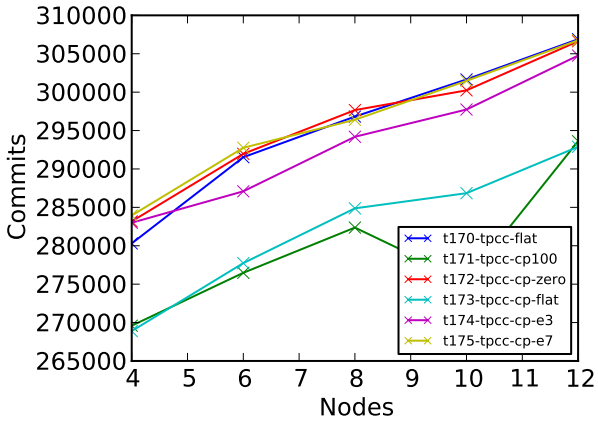

Figure 60. Total number of commits.

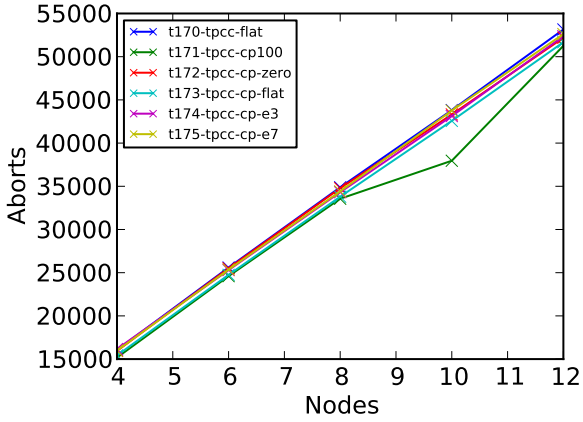

Figure 61. Total number of aborts.

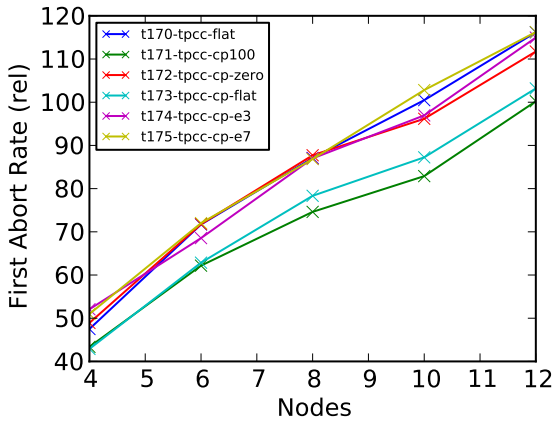

Figure 62. First Abort rate.

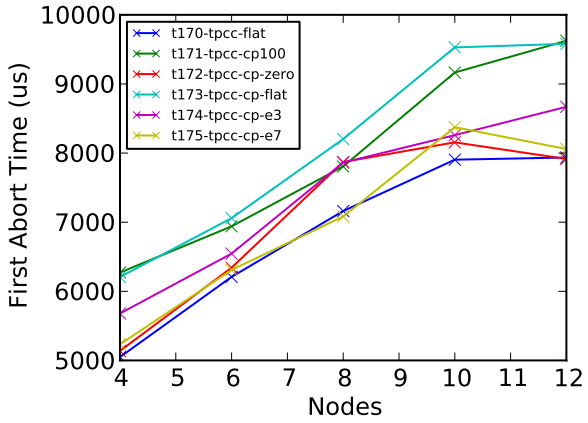

Figure 63. First Abort average duration.

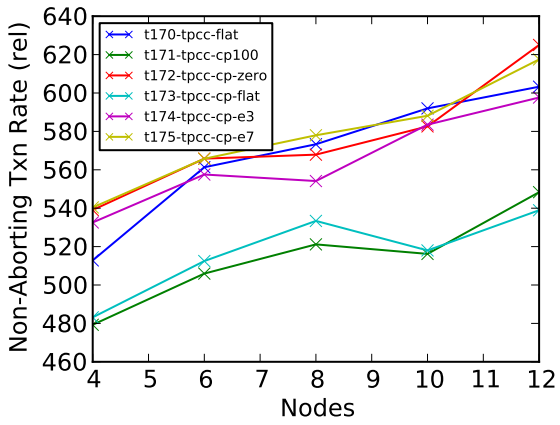

Figure 64. Non-Aborting Transaction rate.

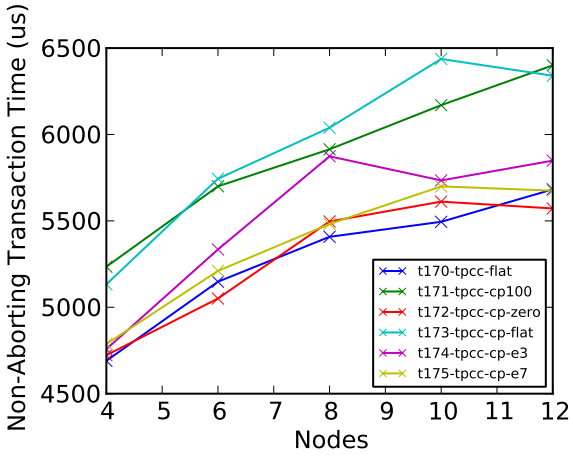

Figure 65. Non-Aborting Transaction average duration.

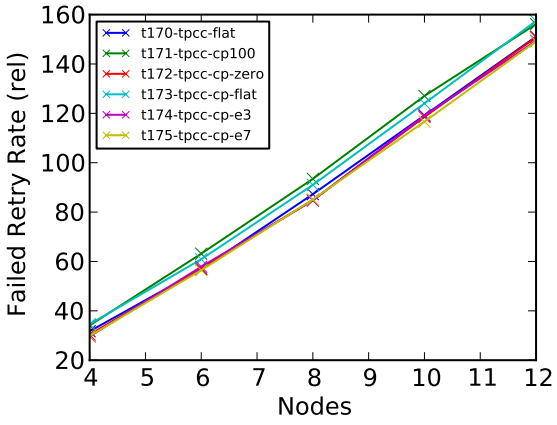

Figure 66. Failed Retry rate.

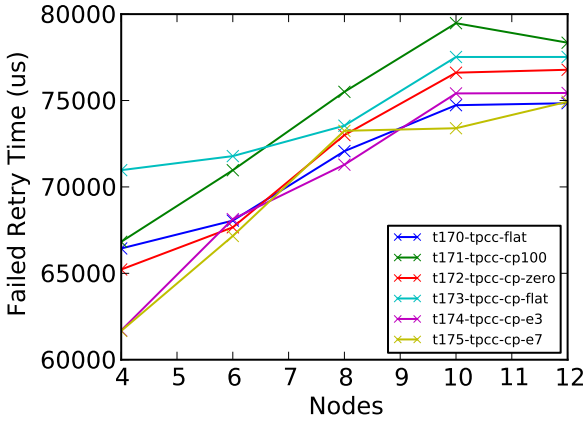

Figure 67. Failed Retry average duration.

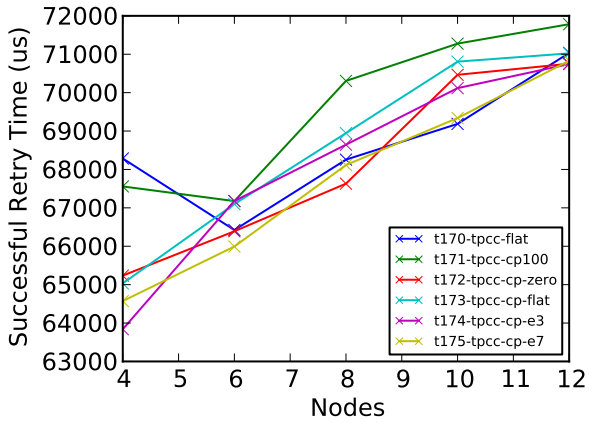

Figure 68. Successful Retry average duration.

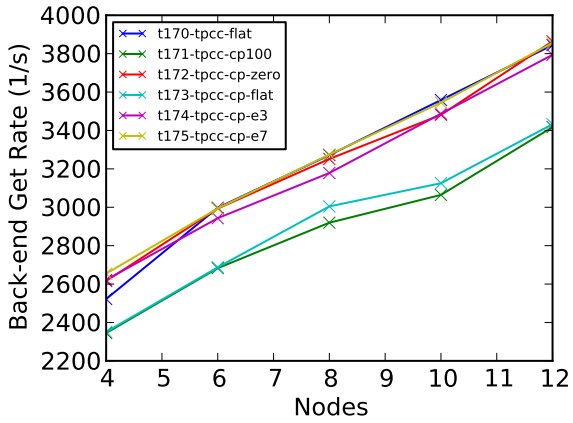

Figure 69. Back-end Get rate.

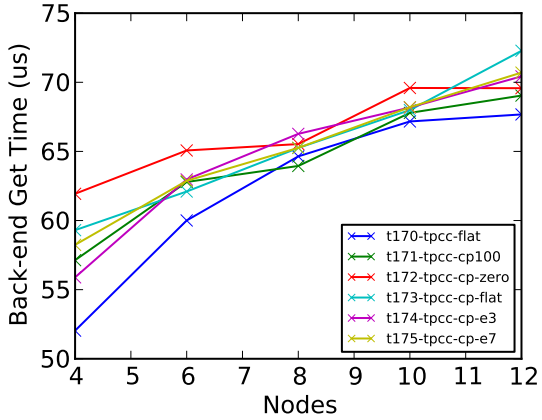

Figure 70. Back-end Get average duration.

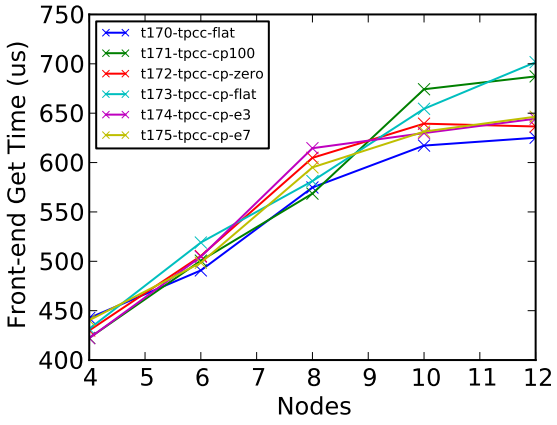

Figure 71. Front-end Get average duration.

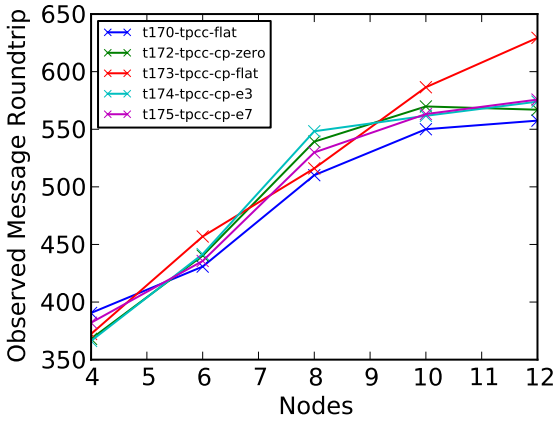

Figure 72. Observed round-trip latency.

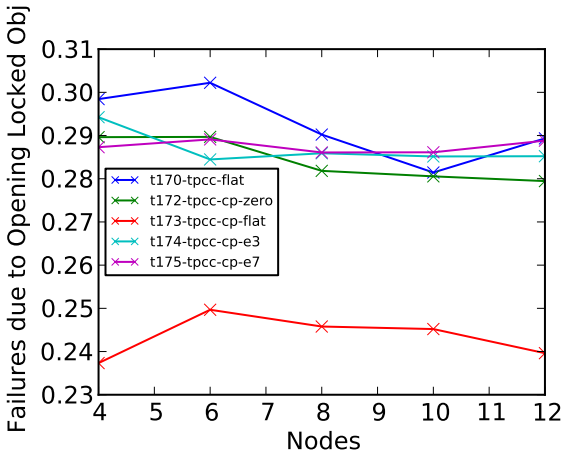

Figure 73. Fraction of aborts due to object locked at open time.

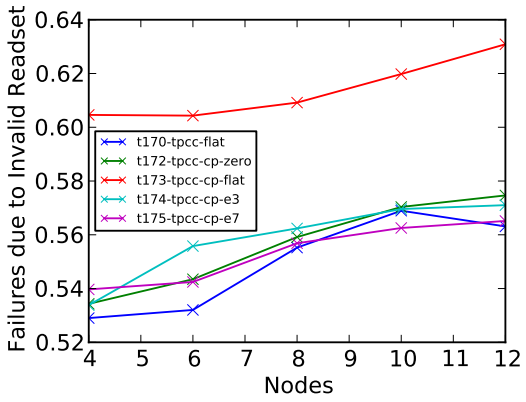

Figure 74. Fraction of aborts due to invalid read-set.

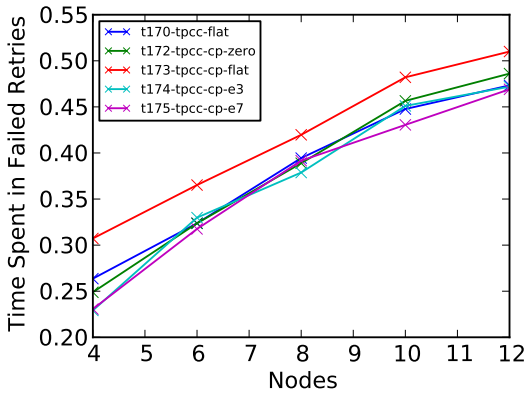

Figure 75. Fraction of time spent in failed retries.

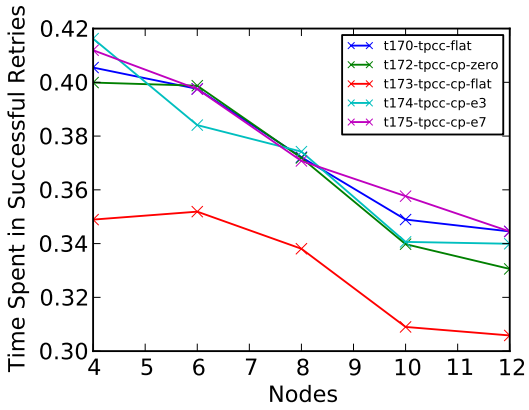

Figure 76. Fraction of time spent in successful retries.

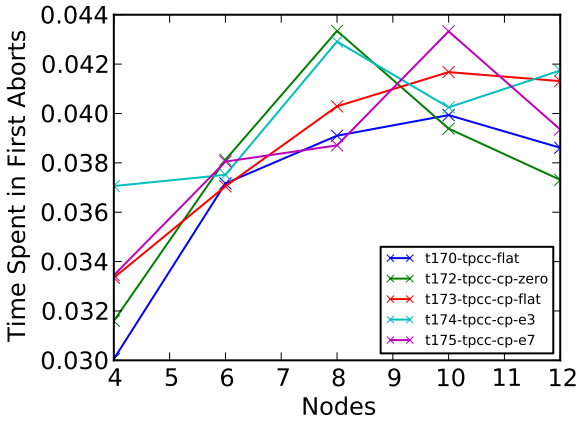

Figure 77. Fraction of time spent in first aborts.

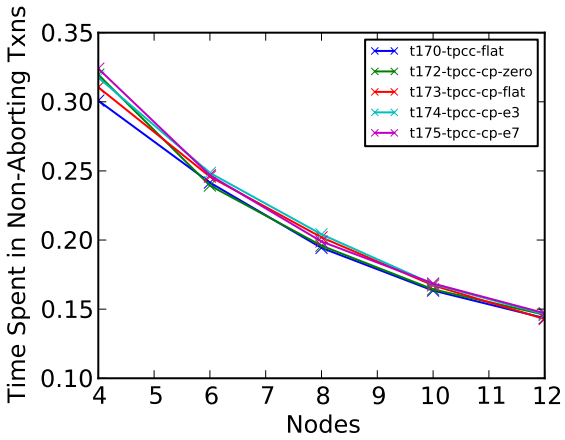

Figure 78. Fraction of time spent in non-aborting transactions.

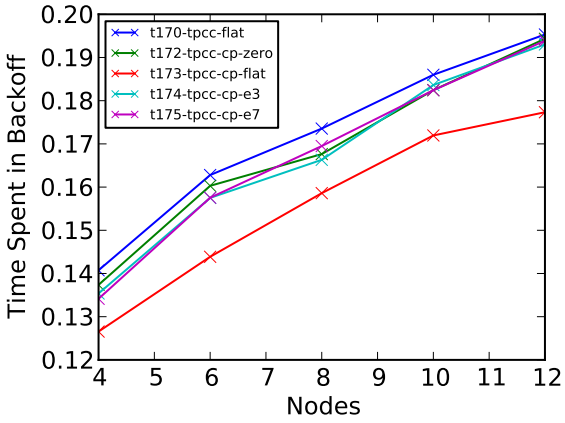

Figure 79. Fraction of time spent in back-off.

D. Hashtable Plots

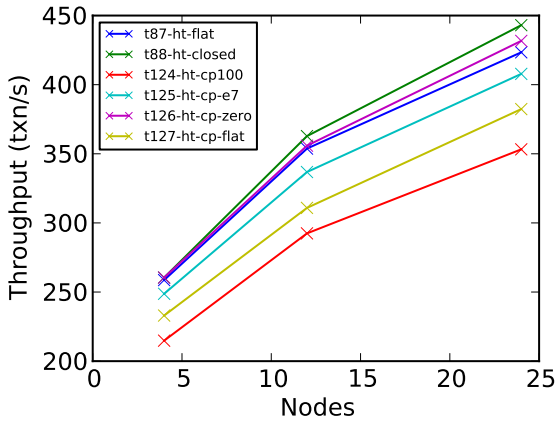

Figure 80. Absolute throughput, in Txn/s.

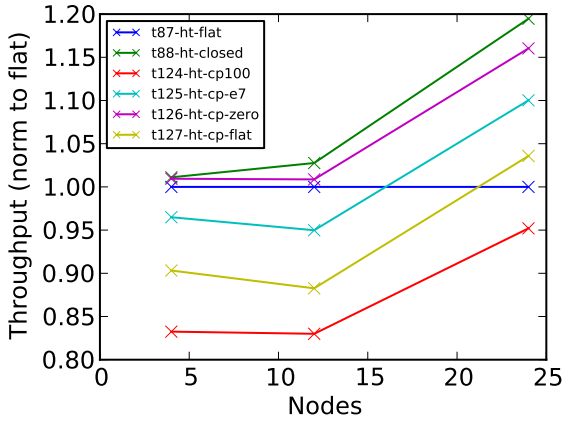

Figure 81. Througput, normalized to flat.

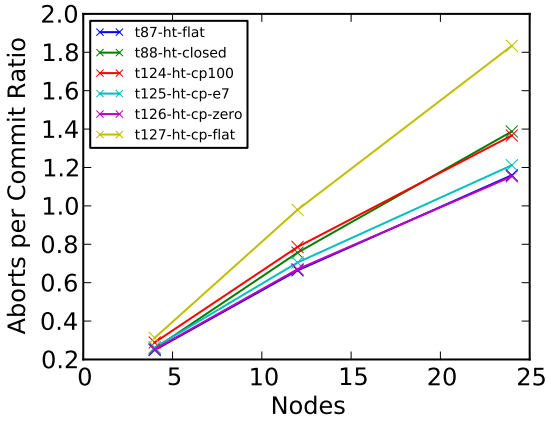

Figure 82. Aborts per commit ratio.

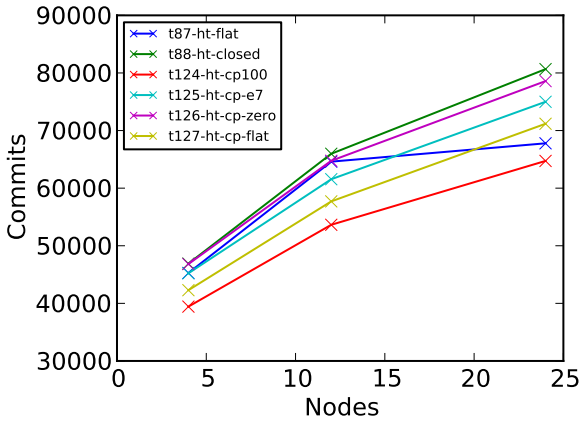

Figure 83. Total number of commits.

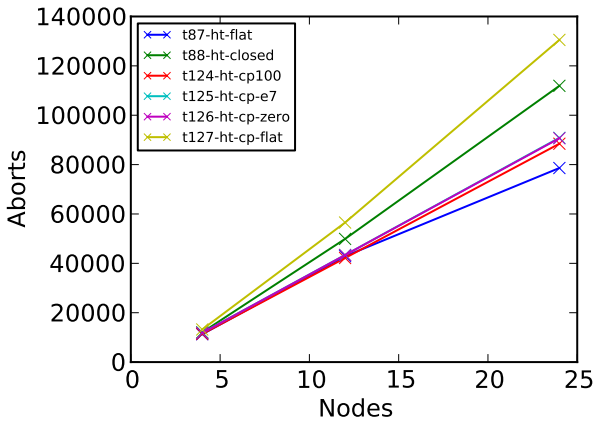

Figure 84. Total number of aborts.

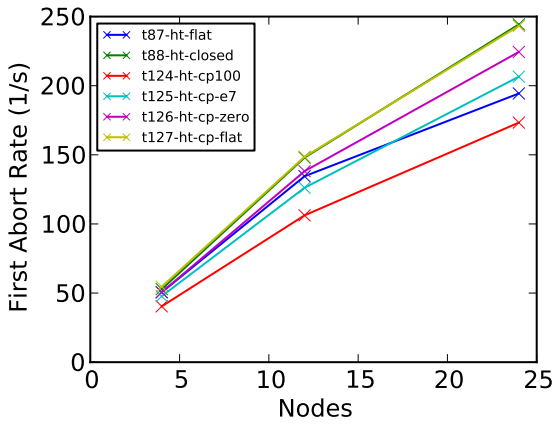

Figure 85. First Abort rate.

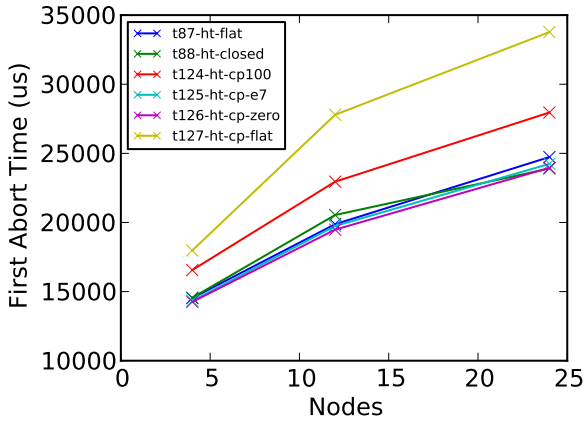

Figure 86. First Abort average duration.

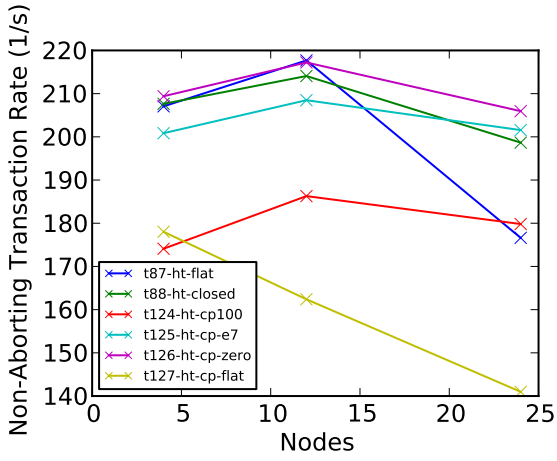

Figure 87. Non-Aborting Transaction rate.

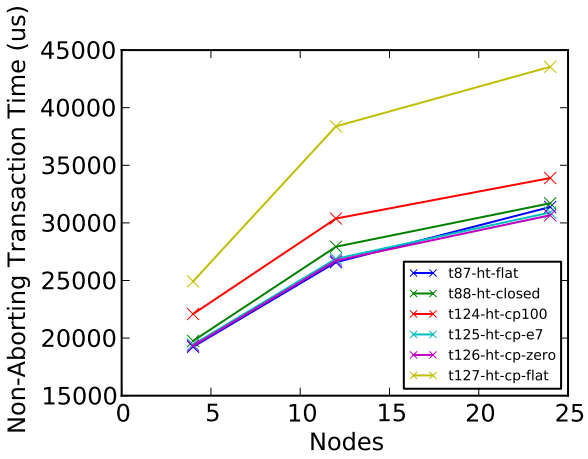

Figure 88. Non-Aborting Transaction average duration.

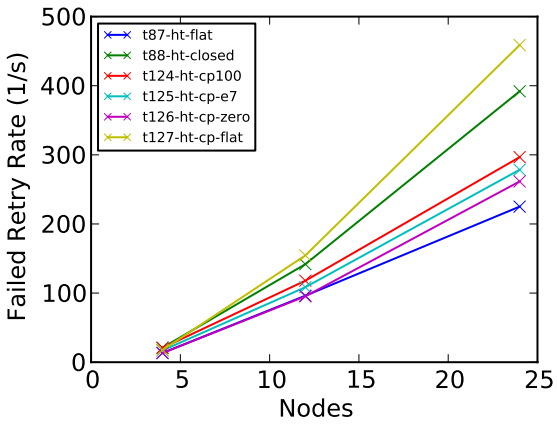

Figure 89. Failed Retry rate.

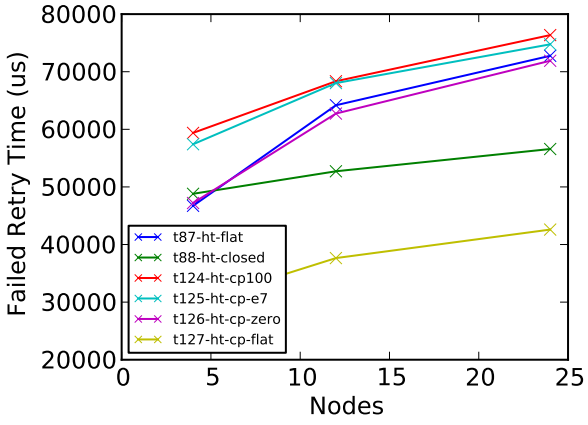

Figure 90. Failed Retry average duration.

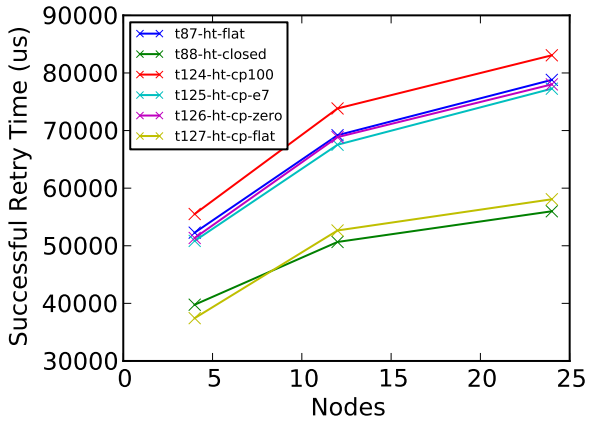

Figure 91. Successful Retry average duration.

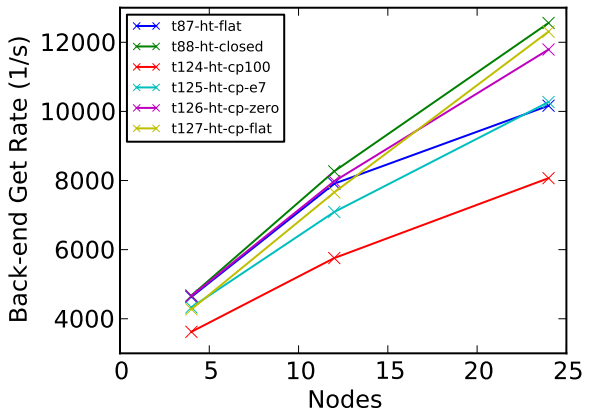

Figure 92. Back-end Get rate.

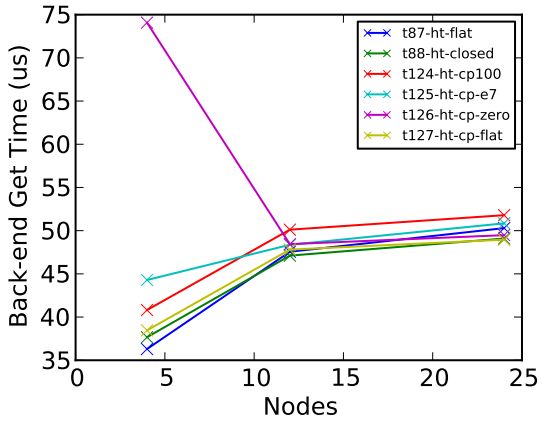

Figure 93. Back-end Get average duration.

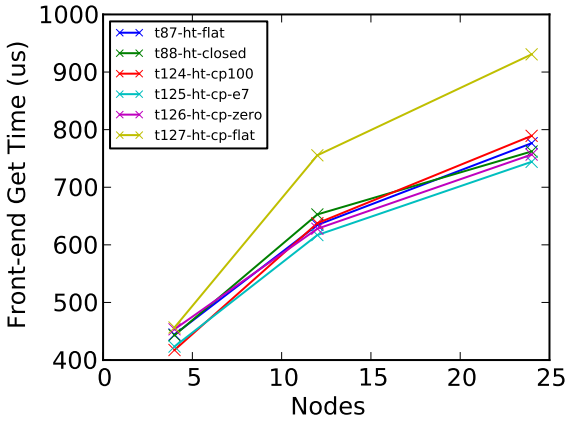

Figure 94. Front-end Get average duration.

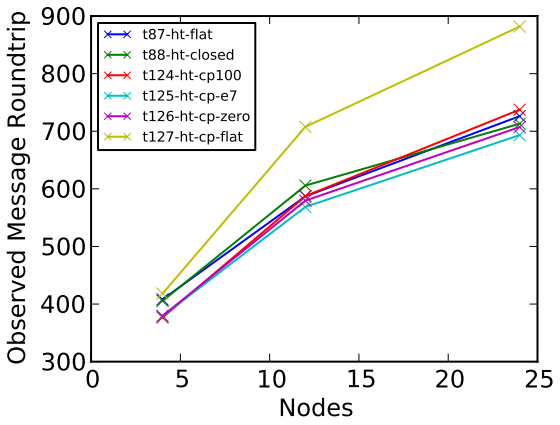

Figure 95. Observed round-trip latency.

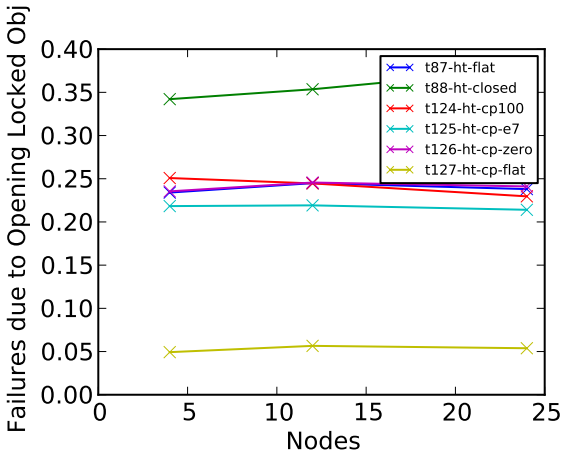

Figure 96. Fraction of aborts due to object locked at open time.

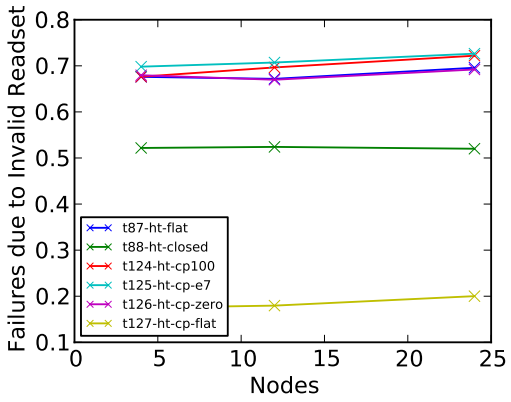

Figure 97. Fraction of aborts due to invalid read-set.

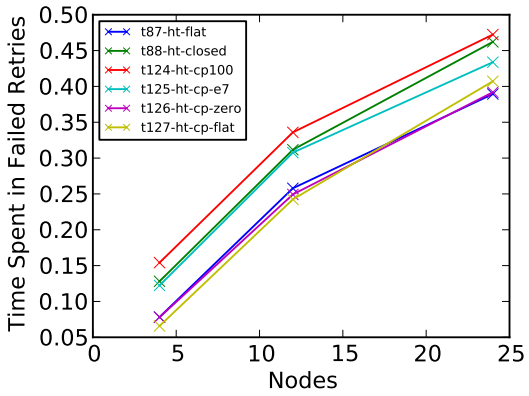

Figure 98. Fraction of time spent in failed retries.

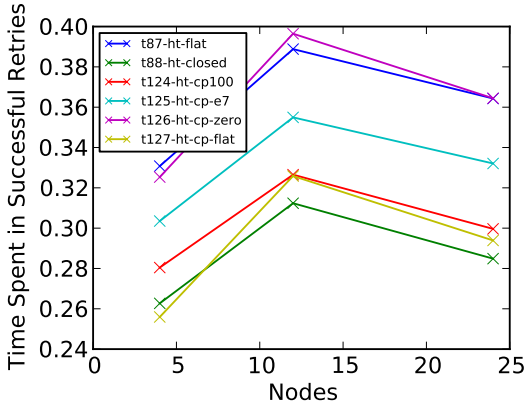

Figure 99. Fraction of time spent in successful retries.

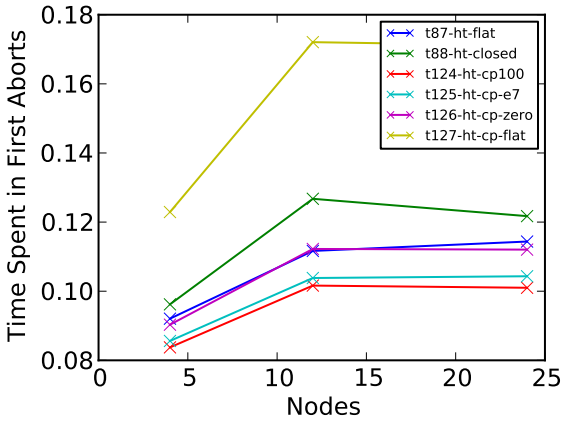

Figure 100. Fraction of time spent in first aborts.

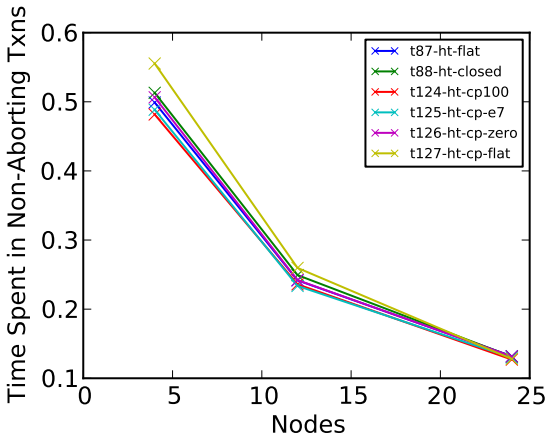

Figure 101. Fraction of time spent in non-aborting transactions.

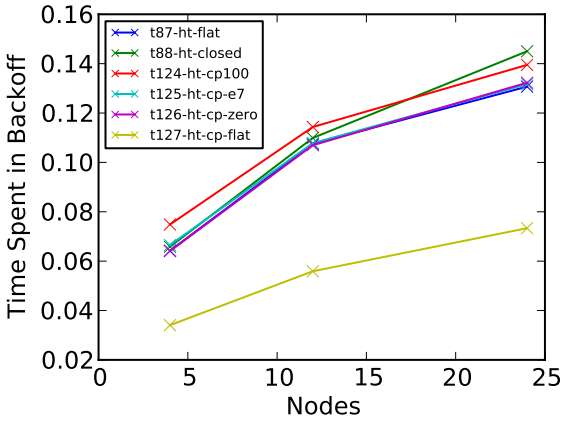

Figure 102. Fraction of time spent in back-off.

E. Skip-List Plots

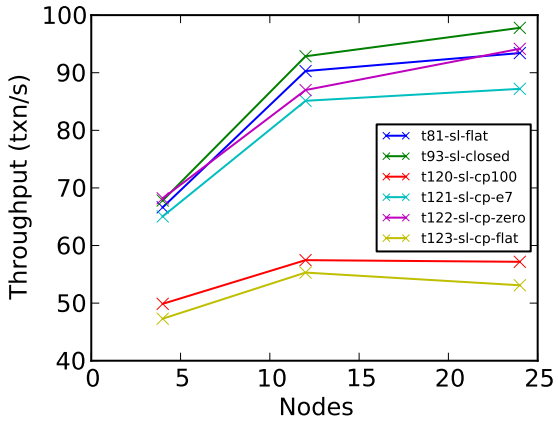

Figure 103. Absolute throughput, in Txn/s.

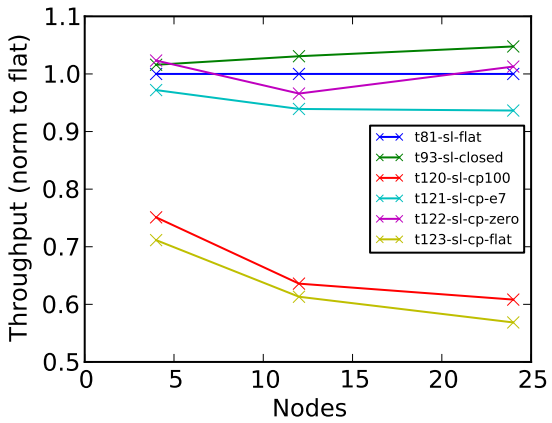

Figure 104. Througput, normalized to flat.

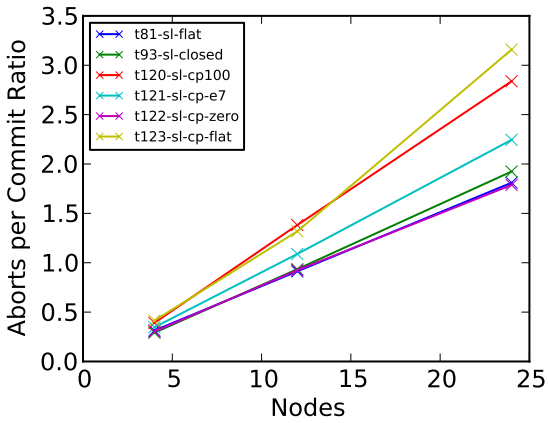

Figure 105. Aborts per commit ratio.

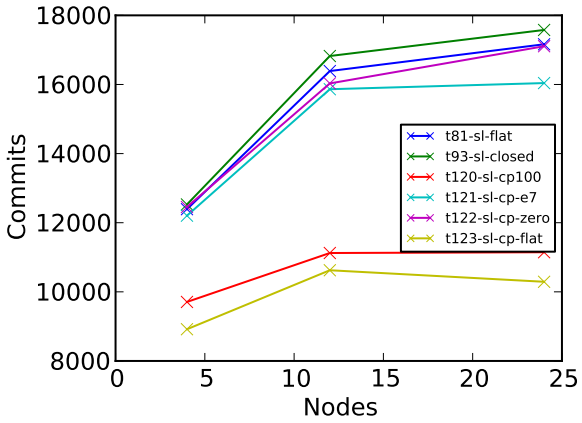

Figure 106. Total number of commits.

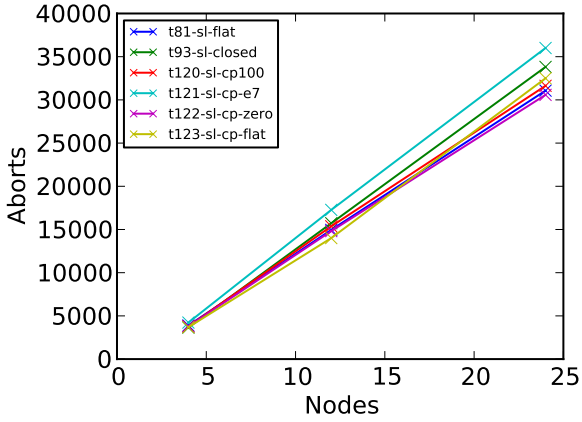

Figure 107. Total number of aborts.

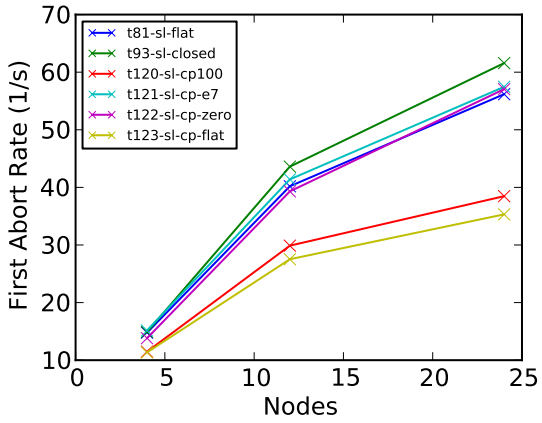

Figure 108. First Abort rate.

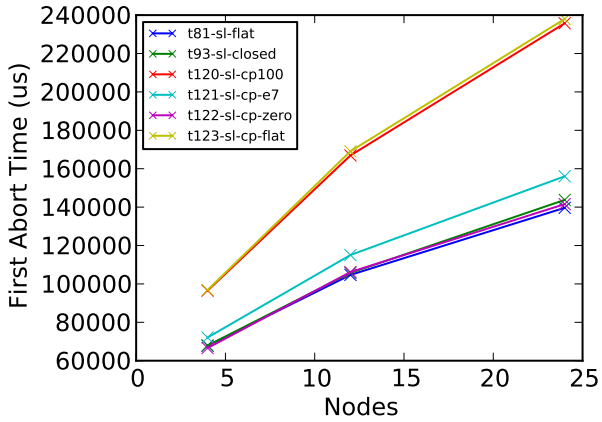

Figure 109. First Abort average duration.

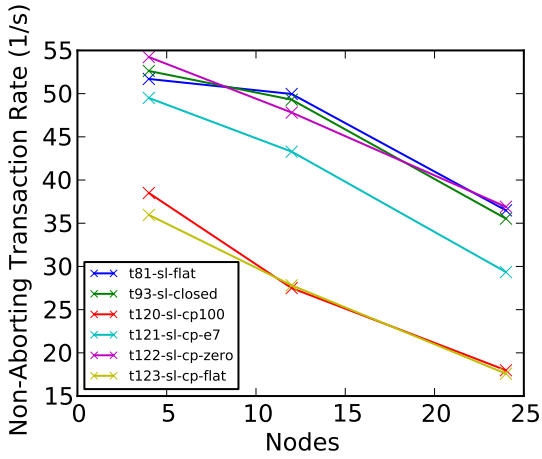

Figure 110. Non-Aborting Transaction rate.

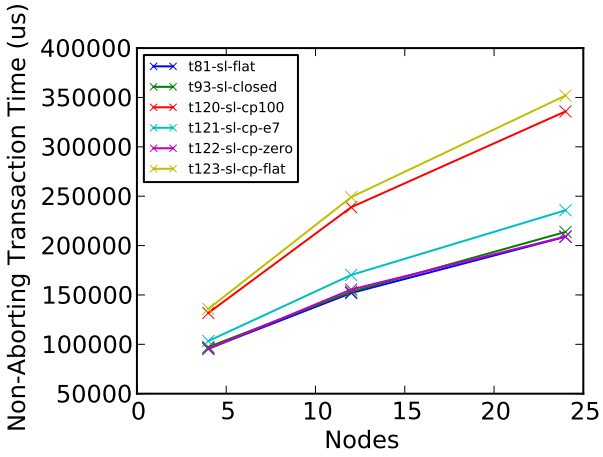

Figure 111. Non-Aborting Transaction average duration.

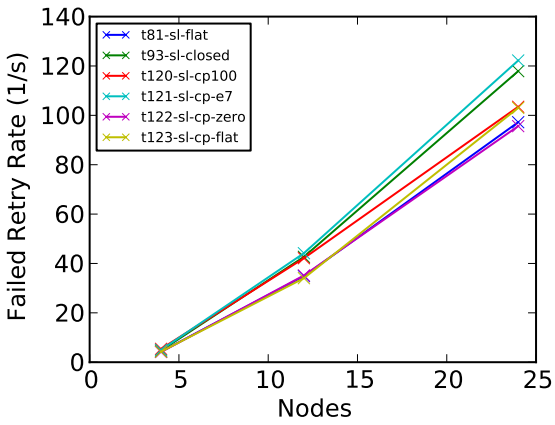

Figure 112. Failed Retry rate.

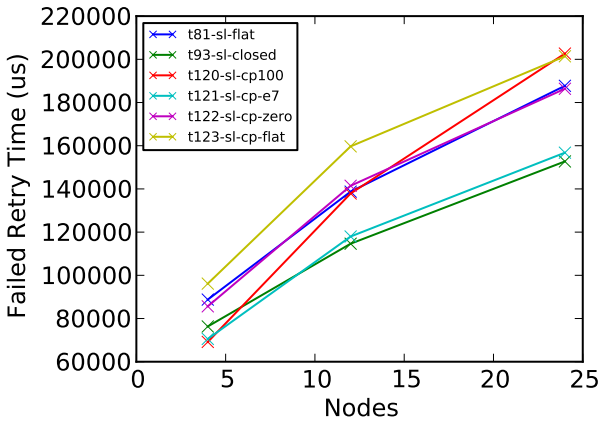

Figure 113. Failed Retry average duration.

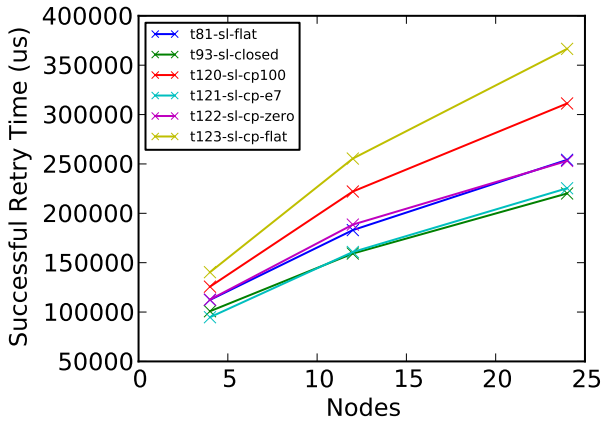

Figure 114. Successful Retry average duration.

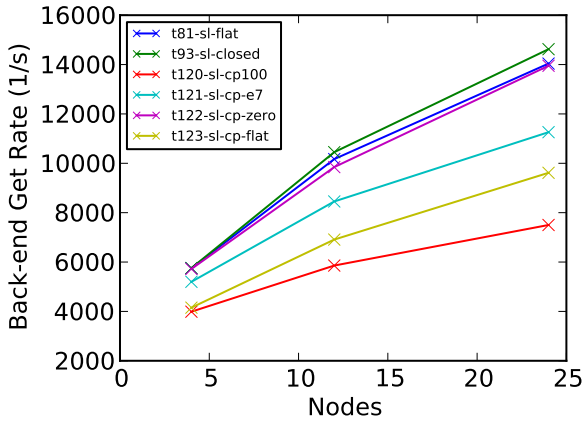

Figure 115. Back-end Get rate.

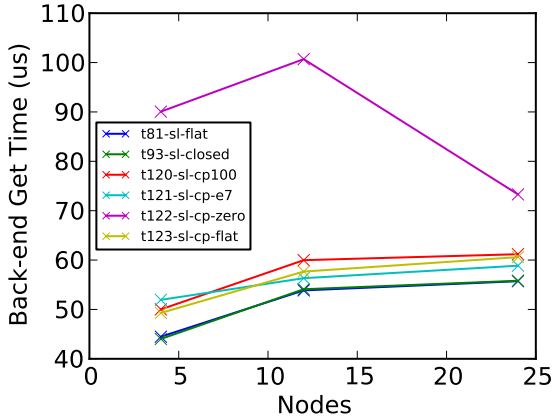

Figure 116. Back-end Get average duration.

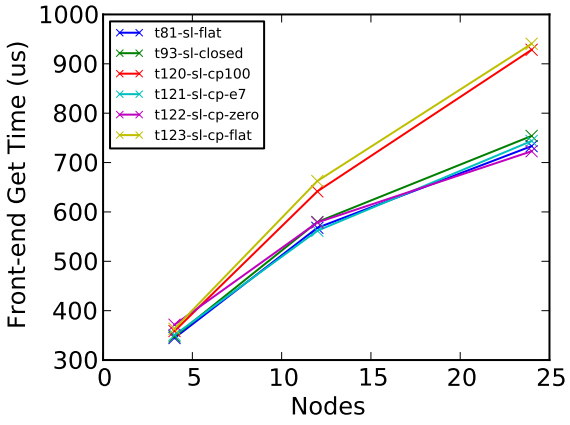

Figure 117. Front-end Get average duration.

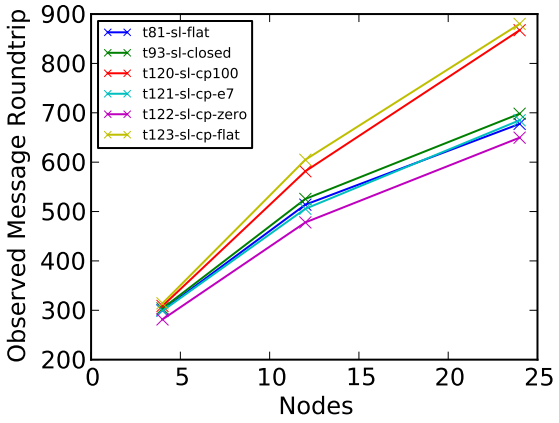

Figure 118. Observed round-trip latency.

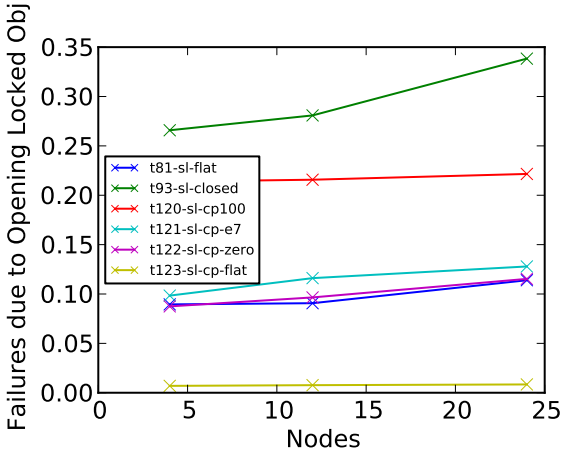

Figure 119. Fraction of aborts due to object locked at open time.

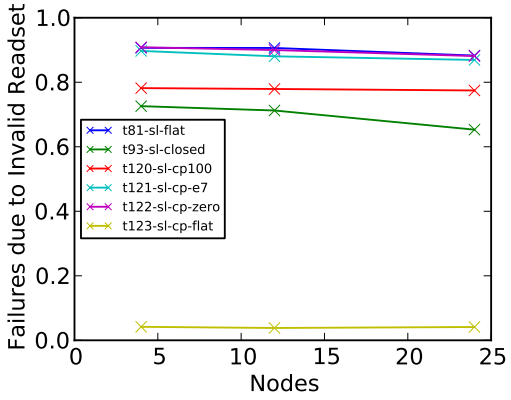

Figure 120. Fraction of aborts due to invalid read-set.

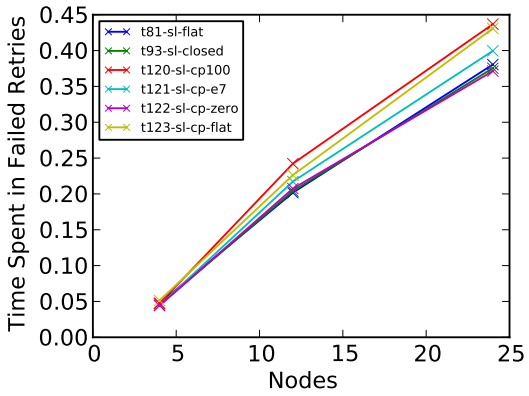

Figure 121. Fraction of time spent in failed retries.

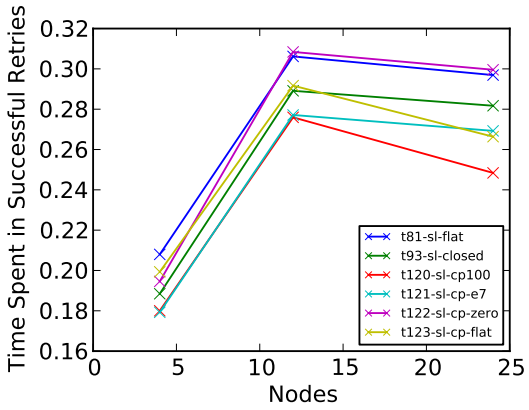

Figure 122. Fraction of time spent in successful retries.

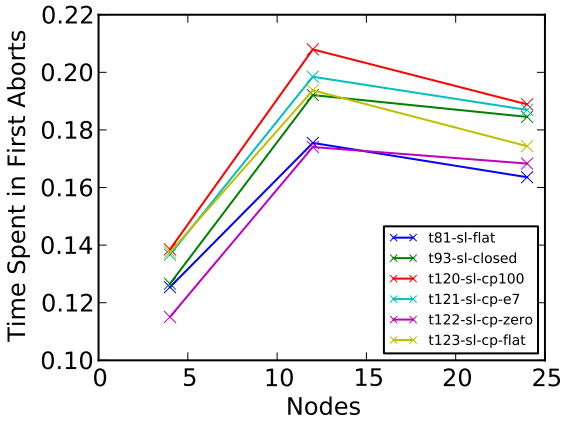

Figure 123. Fraction of time spent in first aborts.

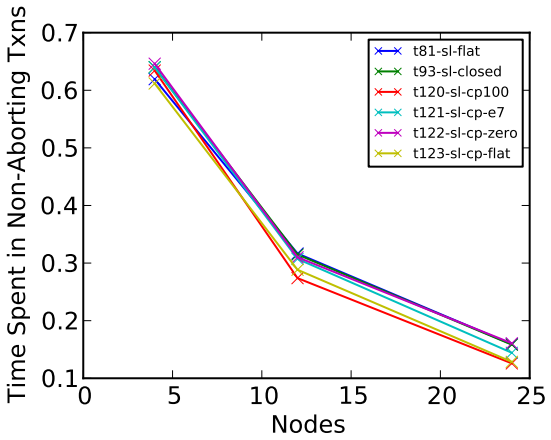

Figure 124. Fraction of time spent in non-aborting transactions.

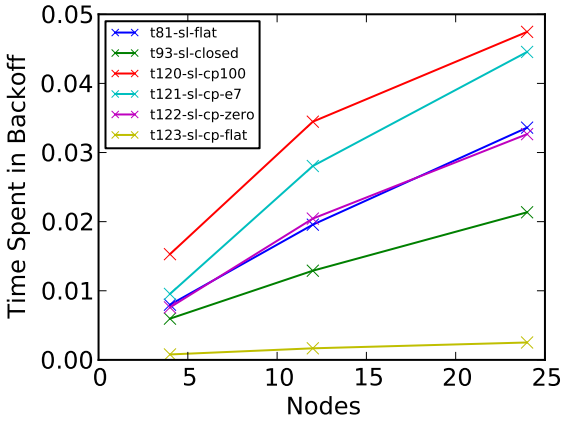

Figure 125. Fraction of time spent in back-off.

F. RBT Plots

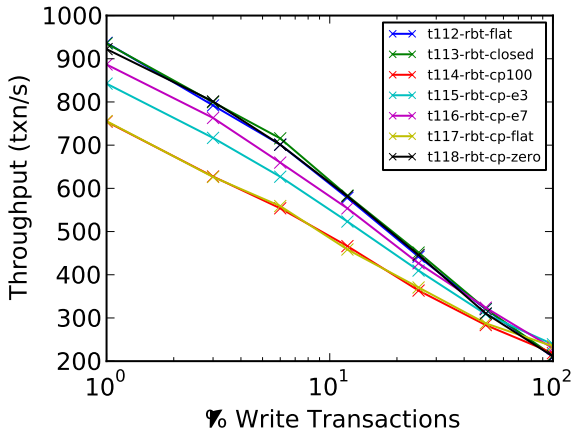

Figure 126. Absolute throughput, in Txn/s.

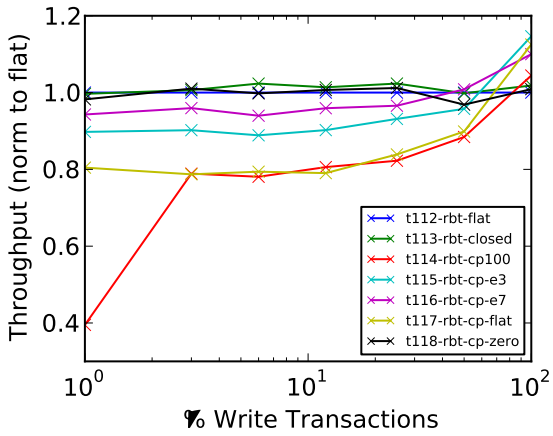

Figure 127. Througput, normalized to flat.

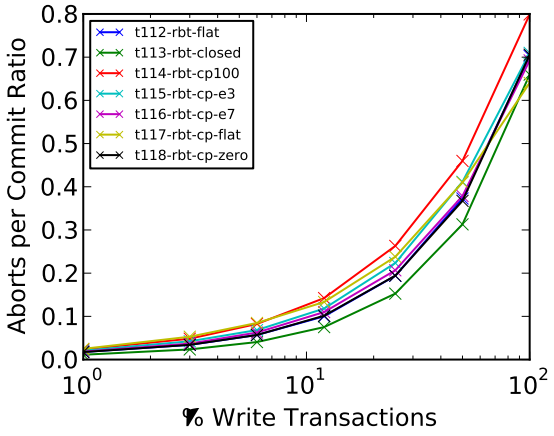

Figure 128. Aborts per commit ratio.

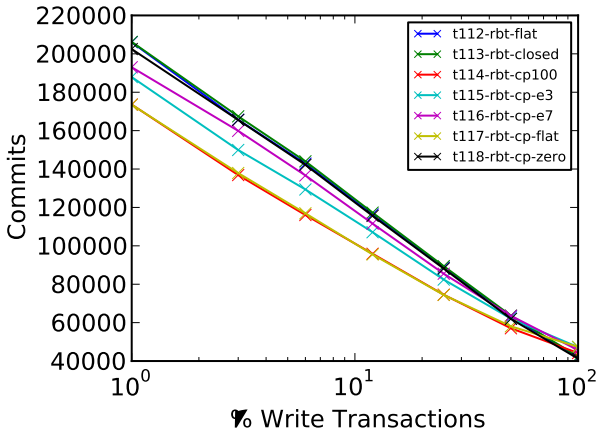

Figure 129. Total number of commits.

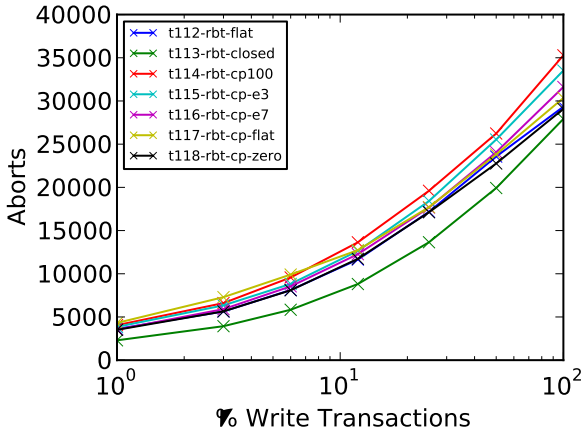

Figure 130. Total number of aborts.

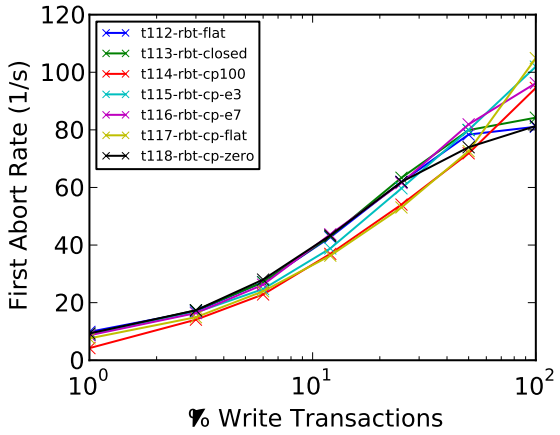

Figure 131. First Abort rate.

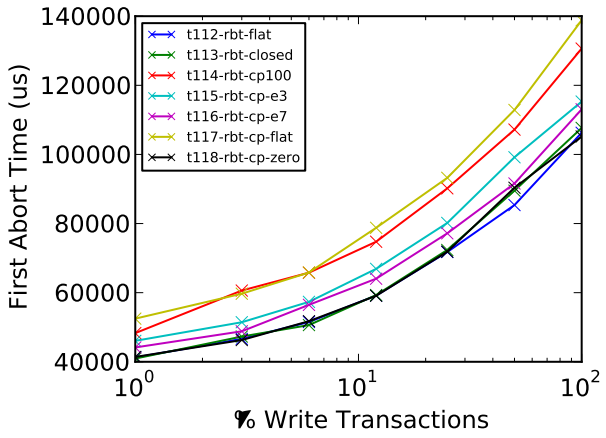

Figure 132. First Abort average duration.

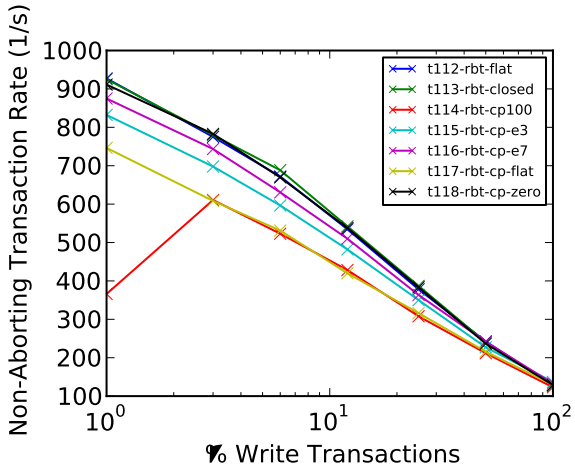

Figure 133. Non-Aborting Transaction rate.

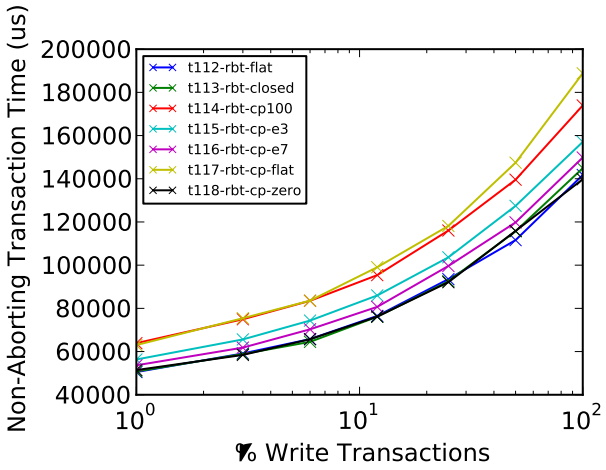

Figure 134. Non-Aborting Transaction average duration.

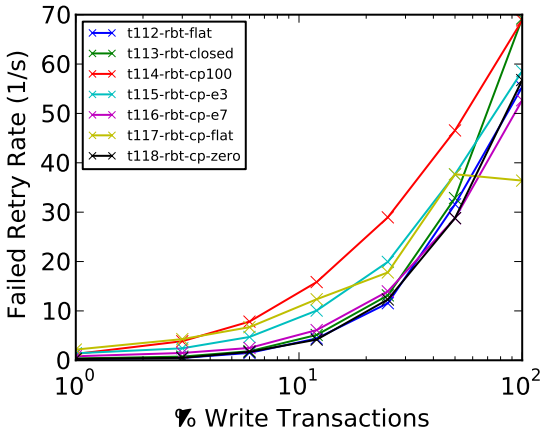

Figure 135. Failed Retry rate.

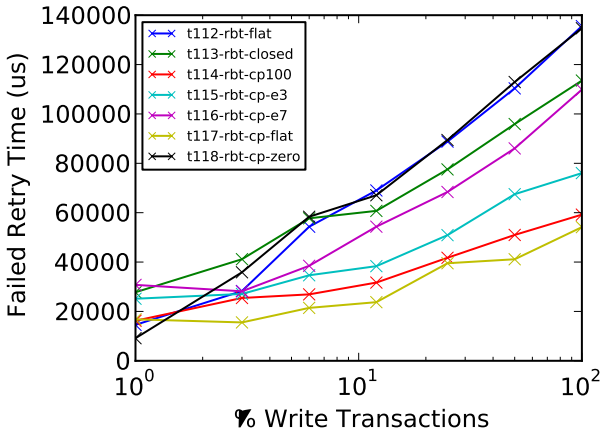

Figure 136. Failed Retry average duration.

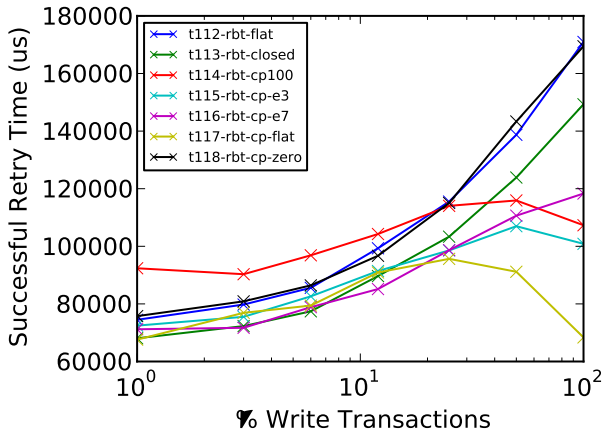

Figure 137. Successful Retry average duration.

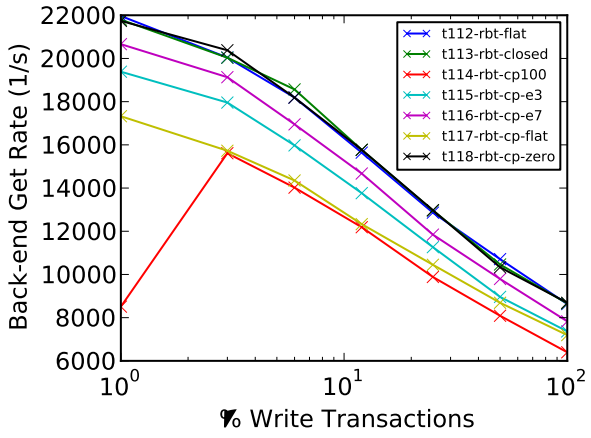

Figure 138. Back-end Get rate.

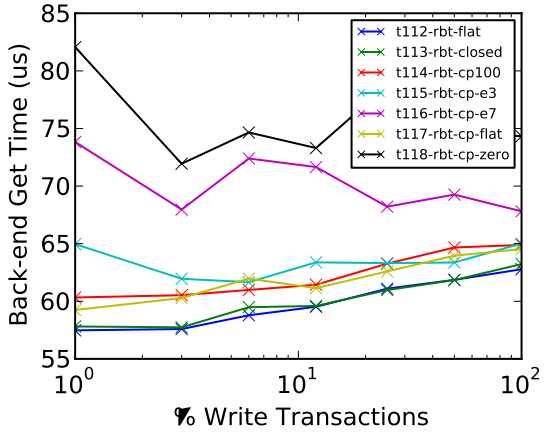

Figure 139. Back-end Get average duration.

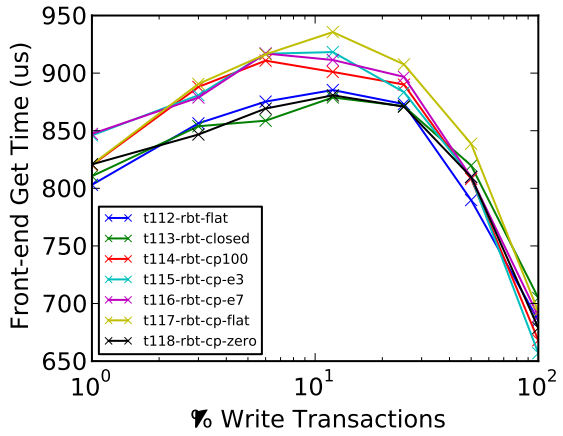

Figure 140. Front-end Get average duration.

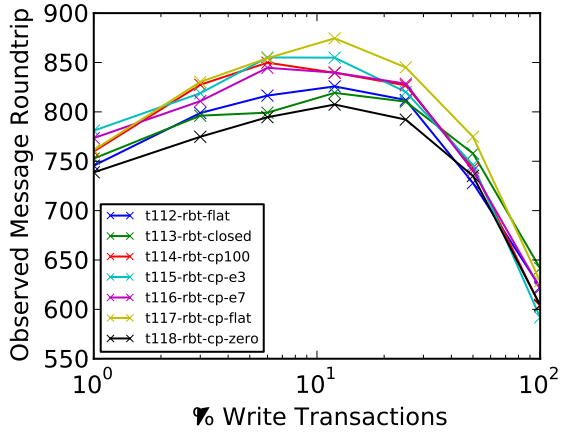

Figure 141. Observed round-trip latency.

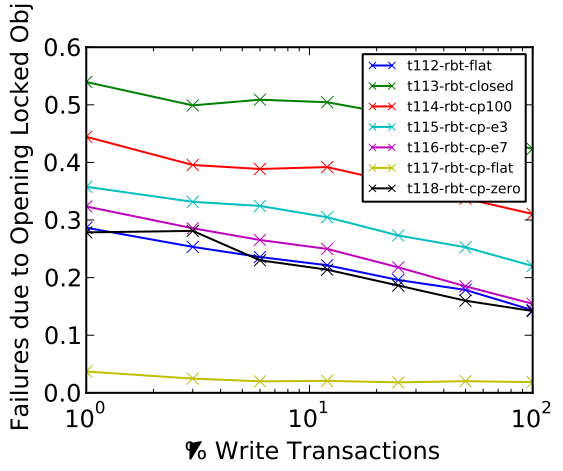

Figure 142. Fraction of aborts due to object locked at open time.

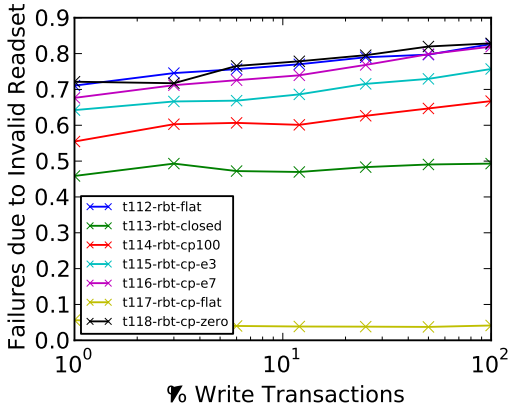

Figure 143. Fraction of aborts due to invalid read-set.

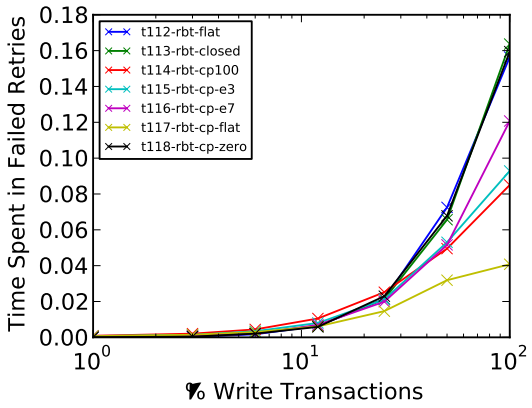

Figure 144. Fraction of time spent in failed retries.

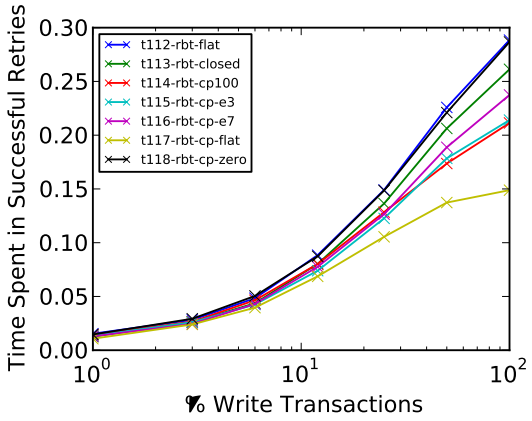

Figure 145. Fraction of time spent in successful retries.

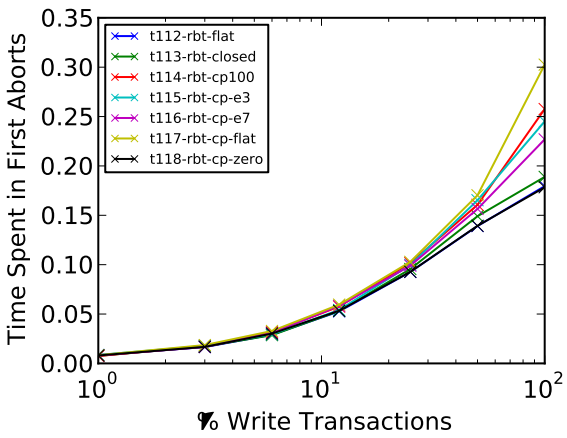

Figure 146. Fraction of time spent in first aborts.

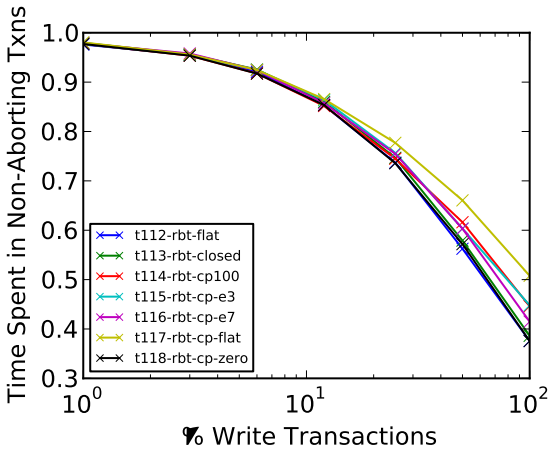

Figure 147. Fraction of time spent in non-aborting transactions.

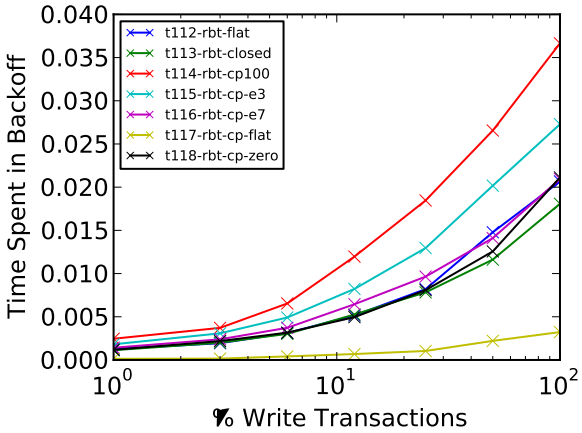

Figure 148. Fraction of time spent in back-off.

References

[1] B. Z. Aditya Dhoke and B. R. Ravindran. On closed nesting and checkpointing in replicated distributed transactional memory. Technical report, Technical report, Virginia Tech, October 2012. URL http://www.ssrg.ece.vt.edu/papers/ipdps.pdf.

[2] K. Agrawal, I.-T. A. Lee, and J. Sukha. Safe open-nested transactions through ownership. In D. A. Reed and V. Sarkar, editors, *PPOPP*. ACM, 2009. ISBN 978-1-60558-397-6.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. ISBN 0-201-10715-5.

[4] E. Boertjes, P. W. P. J. Grefen, J. Vonk, and P. M. G. Apers. An architecture for nested transaction support on standard database systems. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, DEXA '98, pages 448–459, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64950-6. URL `http://dl.acm.org/citation.cfm?id=648311.761611`.

[5] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Notices*, volume 28, pages 237–247. ACM, 1993.

[6] H. Garcia-Molina. Using semantic knowledge for transaction processing in distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, 1983.

[7] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the system r database manager. *ACM Comput. Surv.*, 13(2):223–242, June 1981. ISSN 0360-0300. doi: 10.1145/356842.356847. URL `http://doi.acm.org/10.1145/356842.356847`.

[8] R. Guerraoui and M. Kapaka. Opacity: A correctness condition for transactional memory, 2007.

[9] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983. ISSN 0360-0300. doi: 10.1145/289.291. URL `http://doi.acm.org/10.1145/289.291`.

[10] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, SE-13(1):23 – 31, jan. 1987. ISSN 0098-5589. doi: 10.1109/TSE.1987.232562.

[11] E. Koskinen and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA*, 2008.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[13] J.-L. Lin and M. H. Dunham. A survey of distributed database checkpointing. *Distrib. Parallel Databases*, 5(3):289–319, July 1997. ISSN 0926-8782. doi: 10.1023/A:1008689312900. URL `http://dx.doi.org/10.1023/A:1008689312900`.

[14] S. Mishra. Hyflowcpp: A distributed transactional memory framework for c++. Master's thesis, Virginia Tech, 2013.

[15] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In J. P. Shen and M. Martonosi, editors, *ASPLOS*, pages 359–370. ACM, 2006. ISBN 1-59593-451-0.

[16] J. E. B. Moss. Nested transactions: An approach to reliable distributed computing, 1981.

[17] J. E. B. Moss. Open nested transactions: Semantics and support (poster). In *Workshop on Mem Perf Issues*, 2006.

[18] J. E. B. Moss and A. L. Hosking. Nested tm: Model and architecture sketches. *Sci Comp Prog*, 63(2):186–201, 2006.

[19] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPOPP*, 2007.

[20] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *Proc. of DPDNS*, 2010.

[21] M. M. Saad and B. Ravindran. Transactional forwarding algorithm. Technical report, Virginia Tech, January 2012.

[22] A. Thomasian. Checkpointing for optimistic concurrency control methods. *Knowledge and Data Engineering, IEEE Transactions on*, 7(2):332–339, 1995. ISSN 1041-4347. doi: 10.1109/69.382303.

[23] A. Turcu. On improving distributed transactional memory through nesting and data partitioning. PhD Thesis Proposal, November 2012.

[24] A. Turcu and B. Ravindran. Hyflow2: A high performance distributed transactional memory framework in scala. Technical report, Virginia Tech, April 2012. URL `http://hyflow.org/hyflow/chrome/site/pub/hyflow2-tech.pdf`.

[25] A. Turcu, R. Palmieri, and B. Ravindran. Exploring checkpointing and closed nesting in distributed transactional memory. Technical report, Virginia Tech, March 2013. URL `http://hyflow.org/hyflow/chrome/site/pub/ckpt-tech.pdf`.

[26] TypeSafe. Akka (toolkit and runtime for building highly concurrent, distributed and fault tolerant event-driven applications on the jvm), April 2012. URL `http://akka.io/`.

[27] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Trans. Database Syst.*, 16(1):132–180, 1991.