Characterizing the Performance of Algorithms for Lock-free Objects

Theodore Johnson
Dept. of CIS
University of Florida
Gainesville, Fl 32611-2024

Abstract

Concurrent access to shared data objects must be regulated by a concurrency control protocol to ensure correctness. Many concurrency control protocols require that a process set a *lock* on the data it accesses. Recently, there has been considerable interest in *lock-free* concurrency control algorithms. Lock-free algorithms offer the potential for better system performance because slow or failed processes do not block fast processes. Process "slowdowns" can occur due to cache line faults, memory and bus contention, page faults, context switching, NUMA architectures, heterogeneous architectures, or differences in operation execution time. Much work has been done to characterize the performance of locking algorithms, but little has been done to characterize the performance of lock-free algorithms. In this paper, we present a performance model for analyzing lock-free algorithms that studies the effects of slowdowns on performance. We find that lock-free algorithms are better than locking algorithms if the slowdowns are *transient*, but worse if the slowdowns are *permanent*. One implication of this result is that lock-free concurrent objects are appropriate for UMA architectures, but NUMA architectures require special protocols.

1 Introduction

Processes (or tasks, threads, etc.) in a concurrent system often access *shared objects* to coordinate their activities, whether performing a user computation or maintaining system resources. We regard a shared object to be a shared data structure and a set of operations on the data structure (in this paper we don't allow nested calls or inheritance). The processes that access shared data objects must follow a *concurrency control* protocol to ensure correct executions. Concurrent access to shared data is often moderated with *locks*. A data item is protected by a lock, and a process must acquire the lock before accessing the data item. The type of lock that a process requests depends on the nature of the shared data access, and different lock types have different compatibilities and different priorities. For example, read-only access to a data item can be granted by the acquisition of a shared lock, while read and write access requires an exclusive lock. Shared locks are compatible with each other, but an exclusive lock is compatible with no other lock.

Locking protocols for concurrent database access are well-known [10]. In addition, locking protocols for concurrent access to a wide variety of specialized data structures have been proposed. Examples include binary search trees [33, 37], AVL trees [15], B-trees [8, 53], priority queues [12, 46, 30] and so on. Shasha and Goodman [54] have developed a framework for proving the correctness of lock-based concurrent search

structure algorithms.

The analytical tools needed to study the performance of lock-based data structure algorithms have been established [27, 28, 47]. A general analytical model for modeling the performance of lock-based concurrent data structure algorithms has been developed [29, 28]. The performance of locking protocols also has been well studied. Tay, Suri, and Goodman [57], and Ryu and Thomasian [52] have developed analytical models of the performance of Two-phase Locking variants in database systems.

Herlihy has proposed general methods for implementing *non-blocking* concurrent objects (i.e., concurrent data structures) [21]. In a non-blocking object, one of the processes that accesses the object is guaranteed to make progress in its computation within a finite number of steps. A non-blocking algorithm is fault-tolerant, since a failed process will not make the object unavailable. In addition, fast processes execute at the expense of slow operations, which (hopefully) improves the performance of the object. A typical non-blocking algorithm reads the state of the object, computes its modifications, then attempts to *commit* its modification. If no conflicting operation has modified the object, the commit is successful, and the operation is finished. Otherwise, the operation tries again. The operation typically uses the *compare-and-swap* [65, 9, 43] atomic read-modify-write instruction to try to commit its modifications (one work uses the *load-locked/store-conditional* instruction [22], and several special architecture that support lock-free algorithms have been developed [23, 56]). While many additional non-blocking and lock-free algorithms have been proposed, most have this essential form. Herlihy has also proposed methods for *wait-free* concurrent objects, in which every operation is guaranteed of completion within a bounded number of steps. We do not address the performance of wait-free objects in this paper.

Considerable research on lock-free concurrent algorithms has been done lately [25, 22, 58, 2, 23, 56]. The researchers who work on lock-free algorithms claim that lock-free algorithms can improve the performance of concurrent systems because fast operations execute at the expense of slow operations. Process "slowdowns" can occur due to cache line faults, memory and bus contention, page faults, context switching, NUMA architectures, heterogeneous architectures, or differences in operation execution time. While some work has been done to measure the performance of lock-free algorithms [22, 23, 45], the performance of lock-free algorithms relative to that of blocking algorithms has received little study [45]. In this work, we develop a performance model of lock-free algorithms. Our model studies the effects of both transient and permanent slowdowns in the speed of operation execution. We find that lock-free algorithms are better than locking algorithms if the slowdowns are transient, but worse if the slowdowns are permanent. We extend the explanatory model to a model that accurately predicts the utilization of the shared object.

2 Lock-free Algorithms

Herlihy [21] introduced the idea of a *non-blocking* algorithm for implementing concurrent data structures. A concurrent algorithm is nonblocking if it is guaranteed that some processor makes progress in its computation in a finite number of steps. If a process sets a lock and then fails, no process can make progress. Hence, non-blocking algorithms must avoid conventional locks. Herlihy describes a method for transforming a sequential implementation of an object into a concurrent, non-blocking implementation. An object is represented by a pointer to its current instantiation. A process performs an operation on an object by taking a snapshot of the object, computing the new value of the object in a private but shared workspace (using the sequential implementation), then committing the update by setting the object pointer to the address of the newly computed object.

If there is no interference, then the operation should succeed in its commit. If an interfering operation modified the object, the commit should fail. Since the object is updated by changing the object pointer, a process should set the object pointer to the address of its updated object only if the object pointer has the value that the process read in the initial snapshot. This action can be performed atomically by using the *compare-and-swap* (CNS) instruction. The CNS instruction is available on the IBM/370, the Cedar, the BBN, the Motorola 68000 family, and on the Intel 80486. The CNS instruction is equivalent to the atomic execution of the program in Code 1.

```
CNS(point,old,new)
   object **point,*old,*new {
      if(*point=old){
         *point := new
         return(success)
      }
      else return(failure)
}
```

Code 1 *Compare-and-swap operation.*

A typical non-blocking algorithm has the form of Herlihy's small-object protocol, which is shown in Code 2. In this paper, we are abstracting away the memory management problems that can result in the A-B-A problem [26].

```
object_access(point,[parameters])
   object **point {
   object *old_object, *new_object
      while(True) {
         old_object := snapshot(point)
         new_object := serial_update(old_object,[parameters])
```

```
         if(CNS(point,old_object,new_object) = True)
            break;
      }
}
```

Code 2 *Herlihy's small-object lock-free protocol.*

One problem with the protocol in Code 2 is that the entire object must be copied, wasting time and memory. Herlihy also proposed a *large object* protocol that more efficiently updates a serial object. The large-object protocol is similar to the shadow-page technique used to atomically update a disk-resident index. Often, only the modified portions of the object must be copied and replaced. The large-object protocol has the same essential form as the small-object protocol.

Herlihy's algorithms serialized access to the shared object. Other researchers propose algorithms that permit concurrent access to a non-blocking object. Stone [55] proposes a queue that permits concurrent enqueues and dequeues. An enqueuer that puts a record into an empty queue can block dequeuers, so we categorize the algorithm as *lock-free* instead of non-blocking. Stone's algorithm has the performance characteristics of a non-blocking algorithm. Prakash, Lee, and Johnson [44, 45] give an algorithm for a non-blocking queue that permits concurrent enqueues and dequeues. Their solution is based on classifying every possible queue configuration into one of a finite number of states. The current state is defined by an *atomic snapshot* of the value of the head pointer, the tail pointer, and the next-record pointer of the tail record (the authors provide a protocol for taking the atomic snapshot). When an operation executes, it might find the queue in a valid state. In this case, the operation tries to commit its updates with a *decisive instruction* (via a compare-and-swap). If the queue is in an invalid state, the operation takes the queue to a valid state, then starts again. The execution of the PLJ queue is shown in the program in Code 3.

```
object_access(object_instance,[parameters])
   object *object_instance {
   boolean done; obj_state object_state
      done=False
      while(not done) {
         object_state := snapshot(object_instance)
         if(object_state is valid)
            compute action object_instance
            apply action to object_instance
            if(successful)
               done := True
            else
               cleanup(object_instance)
      }
      cleanup(object_instance)
}
```

Code 3 *The PLJ Concurrent lock-free protocol.*

Valois [59] has developed similar non-blocking algorithms for queues, linked lists, and binary search trees. Herlihy and Moss [25] present non-blocking algorithms for garbage collection. Anderson and Woll [3] present wait-free algorithms for the union-find problem.

Turek, Shasha, and Prakash [58] have techniques for transforming concurrent objects implemented with locks into concurrent non-blocking objects. Every operation keeps its 'program' in a publicly available location. Instead of setting a lock on a record, a process attempts to make the 'lock' field of the record point to its own program. If the attempt fails, the blocked process executes the program of the process that holds the lock until the lock is removed. The contention for setting the lock is similar to the phenomena modeled in this work.

Some researchers have investigated hybrid techniques that are primarily locking, but can force processes to release their locks when the process experiences a context switch [2, 11]. These methods use non-locking algorithms to ensure correctness.

Several architectures that support lock-free algorithms have been proposed [56, 23]. The cache coherence mechanism allows a processor to reserve several words in shared memory, and informs the processor if a conflict occurs.

3 Processor Slowdowns

Since the claimed advantage of lock-free algorithms is superior performance in spite of processor slowdowns, we must examine the possible causes of variations in the time to execute an operation.

The first type of processor slowdowns are 'small' slowdowns. Small slowdowns can be caused by cache line faults, contention for the memory module, and contention for the bus or interconnection network [13]. Another source of small slowdowns lies in the dependence of the execution time of an operation on the data in the data structure. For example, a priority queue might be implemented as a sorted list. An enqueue is slow when the list is big, but fast when the list is small. Lock-free algorithms can take advantage of small slowdowns by giving temporarily fast operations priority over temporarily slow operations. For example, a lock free algorithm would give preference to dequeue operations when the priority queue is large, and to enqueue operations when the priority queue is small, permitting a greater overall throughput.

The second type of processor slowdowns are 'large' slowdowns. These slowdowns are caused by page faults or by context switches in multitasking parallel computers. If the process holds a critical lock and experiences a context switch, all processes that compete for the lock are delayed until the lock holding

process regains control of its processor. Many researchers have worked on avoiding the problems caused by long slowdowns. One approach is to delay the context switch of a process while the process holds a lock [5, 38, 64]. These authors report a large improvement in efficiency in multitasking parallel processors by avoiding large slowdowns. However, this approach has several drawbacks. It requires a more complex kernel, it requires a more complex user/kernel interaction, and it allows a user to grab control of the multiprocessor by having the processes lock "dummy" semaphores. Alemany and Felton [2] and Bershad [11] have proposed hybrid schemes that are primarily locking, but which force processes to release their locks on a context switch (using a technique similar to non-locking protocols to ensure correctness). While these schemes avoid the possibility of a user grabbing processors, they still require additional kernel complexity and a more complex user interface. In contrast, lock-free algorithms solve the large slowdown problem without operating system support.

The types of slowdowns that have been discussed in the literature are *transient* slowdowns. The cause of the slowdown is eventually resolved, and after that the process executes its operation as fast as all other processes in the system. Another type of slowdown is a *permanent* slowdown, in which a process that is executing an operation on a shared object is always slower than other processes in the system that access the object. A permanent slowdown can occur because a processor, and hence all processes executing on it, executes at a slower rate than other processors in the system. The multiprocessor might contain heterogeneous CPUs, perhaps due to incremental upgrades. The multiprocessor architecture might be a *Non-Uniform Memory Access* (*NUMA*) architecture, in which some processors can access a memory module faster than others. In a typical NUMA architecture, the globally shared memory is co-located with the processors. In addition, the topology of the multicomputer is such that some processors are closer together than others (for example, in a hierarchical bus or a mesh topology). In a NUMA architecture, the shared object can be accessed quickly by processors that are close to it, but slowly by processors that are far from it. A process might experience a permanent slowdown while executing an operation because of the operation itself. Different operations on a shared object might require different times to compute. For example, Herlihy [22] observed that enqueues into a priority queue experienced discrimination because they take longer to compute.

In an earlier work [45], we ran several simulation studies to compare the performance of our non-blocking queue to that of a lock-based implementation under different conditions. We expected that the non-blocking queue would perform better than the equivalent lock-based queue if the execution times of the operations varied considerably. In the simulation studies, the operations arrived in a Poisson stream and were assigned a processor to execute the operation's program. In our first set of experiments, we assigned a fast processor

90% of the time and a slow processor 10% of the time. Thus, we simulated permanent slowdowns. We were surprised to find that the locking queue has substantially better performance than the non-blocking queue when the processors experience permanent slowdowns.

In a second set of experiments, all operations are assigned identical processors, but the processors occasionally become slow. Thus, we simulated transient slowdowns. Under transient slowdowns, the non-blocking algorithm has substantially better performance than the locking algorithm.

The key observation is that the performance of lock-free algorithms relative to blocking algorithms depends on the nature of the slowdown that the processes experience. Lock-free algorithms work well when transient slowdowns occur, but poorly when permanent slowdowns occur. The models that we develop in this work will explore this phenomenon.

4 Previous Work

Considerable work has been done to analyze the performance of synchronization methods. Many analyses of synchronization methods have examined the relative performance of shared memory locks. Mellor-Crummey and Scott [39] present performance measurements to show the good performance of their algorithm relative to that of some test-and-set and ticket-based algorithms. Agrawal and Cherian [1] present simulation results and a simple analytical model to explore the performance of adaptive backoff synchronization schemes. Anderson [4] presents measurement results of the performance of several spin locks, and suggests a new ticket-based spin lock. Woest and Goodman [61] present simulation results to compare queue-on-lock-bit synchronization techniques against test-and-set spin locks, and the Mellor-Crummey and Scott lock. Graunke and Thakkar [18] present performance measurements of test-and-set and ticket based locks.

Other authors have examined particular aspects of synchronization performance. Lim and Agrawal [36] examine the performance tradeoffs between spinning and blocking. They present analytical models to derive the best point for a blocked process to switch from spinning to blocking. Glenn, Pryor, Conroy, and Johnson [16] present analytical models which show that a thrashing phenomenon can occur due to contention for a synchronization variable. Anderson, Lazowska, and Levy [6] present some simple queuing models of critical section access to study thread management schemes. Zahoran, Lazowska, and Eager [64] present a variety on analytical and simulation models to study the interaction of synchronization and scheduling policies in a multitasking parallel processor.

Previous analytic studies of multiprocessor synchronization do not address the effects of slowdowns on the performance of shared objects (the work of Zahoran, Lazowska, and Eager [64] uses simulation to study the

effect of scheduling policies). Furthermore, most spin lock algorithms are of an essentially different nature than lock-free algorithms. In many algorithms (i.e, ticket locks, the MCS lock, QOLB locks), competition occurs when the lock is free, and afterwards blocked processes cooperate perform the synchronization. The lock is granted in an atomic step in test-and-set locks. Hence, the analyses have primarily been queuing models, or have counted the number of accesses required to obtain the lock. Lock-free algorithms have a different nature, because a process attempting to perform an operation must complete its operation before another process performs a conflicting operation. Hence, the synchronization is competitive but non-atomic. Only two synchronization algorithms have a similar form. In Lamport's "Fast Mutual Exclusion" algorithm [35], processes compete to obtain a lock using only read and write operations. However, the algorithm is not used in practice and its performance has not been studied by analytical or simulation models. The test-and-test-and-set lock [50] is similar to lock-free algorithms in that blocked processors receive a signal that the lock is free (a cache line invalidation), then compete for the lock. The effect of slowdowns on the test-and-test-and-set lock has never been analyzed, though the methods described in this paper can be applied. However, the result is not likely to be of great interest because the test-and-test-and-set lock is not widely used, and the discrimination due to a NUMA architecture is not likely to have a great effect on system performance.

Considerable work has been done to analyze the performance of concurrent data structure algorithms [29, 28]. These techniques assume that the algorithm is lock-based, and concentrate on analyzing waiting times in the lock queues. Since there is no queuing in lock-free algorithms, these techniques do not apply.

Researchers [22] have observed that non-blocking data structure algorithms are similar to to *optimistic concurrency control* (OCC) in databases [10]. Optimistic concurrency control is so named because it makes the optimistic assumption that data conflicts are rare. A transaction accesses data without regard to possible conflicts. If a data conflict does occur, the transaction is aborted and restarted. Given the relationship between OCC and non-locking algorithms, we can try to apply performance models developed to analyze OCC to analyze non-locking algorithms.

Menasce and Nakanishi [40] present a Markov chain model of OCC in which aborted transactions leave, then reenter the transaction processing system as new transactions. Morris and Wong [41, 42] note that generating new transactions to replace aborted ones biases the transaction processing system towards executing short fast transactions. These authors provide an alternative solution method that avoids the bias by requiring that the transaction that replaces the aborted transaction be identical to the aborted transaction. Ryu and Thomasian [51] extend this model of OCC to permit a wide variety of execution time distributions and a variety of OCC execution models. Yu et al. [63, 62] develop approximate models of OCC and locking

concurrency control to evaluate their performance in transaction processing systems.

Of these models, the approach of Ryu and Thomasian is the best suited for application to analyzing non-locking algorithms. Previous models of a similar nature [40, 41, 42] are not as general. Other analyses [63, 62] focus on issues such as buffering and resource contention, and assume that data conflicts are rare. In contrast, the Ryu and Thomasian abstracts away the operating environment and focuses on analyzing the effects of data conflicts only. Furthermore, the Ryu and Thomasian model produces accurate results when the rate of data conflict is high.

Our approach is to extend the simple but flexible model of Ryu and Thomasian [51] to analyze lock-free algorithms. The Ryu-Thomasian model requires that if a transaction is aborted, its execution time is identical to the first execution. However, we explicitly want to account for variations in the execution time in our work load model (since lock-free algorithms are intended to be fast in spite of temporarily or permanently slow processors). Therefore, we start by extending the Ryu-Thomasian performance model to account for two new workload models. We next apply the performance models to analyze several lock-free algorithms. We show how the closed-system model of Ryu and Thomasian can be converted into an open system model. We validate the analytical tools and use them to explore the relative performance of the algorithms.

5 Model Description

Data access conflicts in OCC are detected by the use of *timestamps*. Each *data granule*, g, (the smallest unit of concurrency control) has an associated timestamp, $t(g)$, which contains the last time that the data granule was written to. Each transaction, T, keeps track of its read set $R(T)$ and write set $W(T)$. We assume that $R(T) \supset W(T)$. Every time a new data granule is accessed, the time of access is recorded. If at the commit point a data granule has a last write time greater than the access time, the transaction is aborted. Otherwise, the transaction is committed and the last write time of each granule in $W(T)$ is set to the current time. The procedure used is shown in Code 4.

```
read(g,T)
   read g into T's local workspace
   access_time(g)=Global_time


validate(T)
   for each g ∈ R(T)
      if access_time(g)<t(g)
         abort(T)
   for each g ∈ W(T)
      t(g)=Global_time
```

```
commit(T)
```

Code 4 *OCC validation*

As has been noted elsewhere [22], lock-free protocols of the types described in Code 2 and 3 are essentially similar to the OCC validation described in Code 4. Both types of algorithms read some data values, then commit if and only if no interfering writes have occurred. Although many of the implementation details are different (OCC and lock free algorithms detect conflicts with different mechanisms, and an 'abort' in a lock free algorithm only makes the operation re-execute the `while` loop), an analysis that counts conflicts to calculate the probability of 'committing' applies equally well to both types of algorithms.

Because an operation that executes a non-blocking algorithm acts like a transaction that obeys OCC, we develop the analytical methods in the context of transactions, then apply the methods to analyzing operations. Following Ryu and Thomasian, we distinguish between *static* and *dynamic* concurrency control. In static concurrency control, all data items that will be accessed are read when the transaction starts. In dynamic concurrency control, data items are read as they are needed. We also distinguish between *silent* and *broadcast* concurrency control. The pseudo-code in Code 4 is silent optimistic concurrency control: an operation doesn't advertise its commit, and transactions that will abort continue to execute. Alternatively, a transaction can *broadcast* its commit, so that conflicting transactions can restart immediately [48, 20].

We model the transaction processing system as a closed system in which V transactions each execute one of C transaction types. When a new transaction enters the system, it is a class c transaction with probability f_c, $\sum_C f_c = 1$. A class c transaction is assumed to have an execution time of $\beta(V)b_c(x)$, where $\beta(V)$ is the increase in execution time due to resource contention. Factoring out $\beta(V)$ is an example of a *resource contention decomposition approximation* [57, 51, 28], which lets us focus on the concurrency control mechanism, and which allows the analysis to be applied to different computer models. We will assume that $\beta(V) = 1$ in the analysis (i.e., one processor per operation).

As a transaction T executes, other transactions will commit their executions. If a committing transaction conflicts with T, then T must be aborted. We denote by $\Phi(k, c)$ the probability that a committing class k transaction conflicts with an executing class c transaction. We model the stochastic process in which committing transactions conflict with an executing transaction as a Poisson process. Ryu and Thomasian [51] show that this assumption, which makes the analysis tractable, leads to accurate model predictions under a wide variety of conditions.

We differentiate between three models depending on the actions that occur when a transaction aborts. In [51], a transaction samples its execution time when it first enters the system. If the transaction is aborted, it is executed again with the same execution time as the first execution time. We call this transaction model the *fixed time/fixed class* model, or the FF model[1]. The FF model avoids a bias for fast transactions, permitting a fair comparison to lock-based concurrency control when analyzing transaction processing systems.

The variability of the execution time of a operation could be due to resource contention, to decisions the operation makes when as it executes, or a combination of both. In these cases, the execution time of a operation changes when a operation is re-executed after an abort. However, some processors might be slower than others, and some operations might take longer to compute than others. We introduce the *variable time/fixed class*, or VF, model to represent the situation in which processors can experience both transient and permanent slowdowns. In the VF model, an aborted transaction chooses a new execution time for its next execution. However, the new operation is still of the same class (i.e, on the same processor and the same type of operation).

We might want to model a situation in which processors experience only temporary slowdowns (i.e, a UMA processor and all operations require about the same amount of computation). then fast on the next execution. In the *variable time/variable class*, or VV model, a new transaction type is picked to replace an aborted transaction (possibly a transaction of the same type).

5.1 Model Solution Methods

For a given transaction model, we can solve the system for any of the OCC models in the same way. The method for solving the system depends on the transaction model: the FF and the VF models use the same method, but the VV model is solved by using a different method.

5.1.1 Solving the FF and VF Models

The solution method for the FF and VF models involves taking the system utilization U (the portion of time spent doing useful work) and finding the per-class utilizations U_c. The system utilization U is then computed from the per-class utilizations. Ryu and Thomasian show that the equations can be solved quickly through iteration.

The mean useful residence time of a class c transaction is denoted by $R_a^c(V)$. A transaction might be required to restart several times due to data conflicts. The expected time that a transaction spends executing aborted attempts is denoted by $R_c^d(V)$, and the total residence time of a class c transaction is

[1] Most of the results that we present for the FF model have been taken from [51].

$R^c(V) = R^c_a(V) + R^c_d(V)$. The *utilization* of a class is the proportion of its expected residence time spent in an execution that commits: $U_c = R^c_a(V)/(R^c_a(V) + R^c_d(V))$. The expected residence time of a transaction ($R_a(V)$, $R_d(V)$, and $R(V)$) is calculated by taking the expectation of the per-class expected residence times. The system efficiency, U, is calculated by taking the expectation of the per-class utilizations:

$$U(V) = R_a(V) / \left(\sum_{c=1}^{C} f_c R^c_a(V)/U_c(V) \right) \tag{1}$$

In order to calculate the per-class efficiencies, we need to calculate the probability that a transaction aborts due to a data conflict. We define $\Phi(k, c)$ to be the probability that a class k transaction conflicts with a class c transaction. We know the proportions of committing transactions, so we can calculate the probability that a committing transaction conflicts with a class c transaction Φ_c by:

$$\Phi_c = \sum_{k=1}^{C} \Phi(k, c) f_k \tag{2}$$

We can calculate the rate at which a committing transactions conflict with a class c transaction, γ_c, by setting γ_c to be the proportion of committing transactions that conflict with a class c transaction:

$$\gamma_c = \frac{(V-1)\Phi_c}{b} U(V)$$

where b is the expected execution time of all transactions.

Given the system utilization, we can calculate the per-class conflict rate. From the per-class conflict rate, we can calculate the per-class utilizations, and from the per-class utilizations, we can calculate the system utilization. The output system utilization is a decreasing function of the input system utilization. In the FF model, the utilization is bounded by 1, so the unique root in [0..1] can be found using a binary search iteration. In the VF model, it is possible for the utilization to be greater than 1 (because of the bias towards fast executions), so the root finder must use one of the standard nonlinear equation solution methods [7].

5.1.2 Solving The VV Model

In the VV transaction model, when a transaction aborts, it leaves the system and a new transaction enters. As a result, the proportion of committing class c transactions is no longer f_c, and instead depends on the probability that a class c transaction commits, p_c, and the average execution time of a class c transaction. The solution method for the VV model is based on iteratively finding a root for the vector $\vec{p}$.

In order to calculate the conflict rate, we need to know the proportion of transactions S_k that are executing a class k transaction. When a process is executing a class k transaction, it executes for an expected b_k seconds.

If one was to observe a very large number of transaction executions, say M, then a class k transaction would be executed about Mf_k times. Thus, the observation period would take $\sum_{i=1}^{C} Mf_ib_i$ seconds, during which a class k transaction would be executed for Mf_kb_k seconds. By the theory of alternating renewal processes [49], we have

$$S_k = f_kb_k/b \tag{3}$$

If the process is executing a class k transaction, it will finish at rate $1/b_k$. When the transaction completes, it will commit at rate p_k, and if it commits, it will conflict with a class c transaction with probability $\Phi(k,c)$. Therefore,

$$\begin{aligned}\gamma_c = & \quad (V-1)\sum_{i=1}^{C} S_kp_k\Phi(k,c)/b_k \\ = & \quad (V-1)\sum_{k=1}^{C}(f_kb_k/b)p_k\Phi(k,c)/b_k \\ = & \quad \tfrac{V-1}{b}\sum_{k=1}^{C}\Phi(k,c)f_kp_k \end{aligned} \tag{4}$$

Given the probability that transactions of each transaction class commits, $\vec{p_c}$, we can calculate conflict rate γ_c for each transaction class. Given the conflict rate for a transaction class γ_c, we can calculate the probability that the transaction will commit p_c.

Unlike the case with the FF and the VF models, for the VV model, we need to iterate on a vector. We make use of a property of the system of equations to find a rapidly converging iterative solution: if F is the transformation $F(\vec{p}_{old}) = \vec{p}_{new}$, then $F(p_1,\ldots,p_c+\epsilon,\ldots,p_C) \leq F(p_1,\ldots,p_c,\ldots,p_C)$, where $\epsilon > 0$ and the vector relation $\leq$ refers to component-wise comparison. In other words, the Jacobian of F is strictly nonpositive. The algorithm that we use to find a solution of the VV calculates the i^{th} value of p_c to be $p_c^i = (p_c^{i-1} + F(\vec{p}^{\,i})_c)/2$.

6 Analysis

In this section, we present the calculations needed for solve the systems discussed in the previous section. For each of the four types of optimistic concurrency control, we present the calculation for each of the three transaction models.

6.1 Analysis of Silent/Static OCC

In this section, we examine the simplest OCC scheme. In the silent/static scheme, transactions access their entire data sets when they start their executions, and detect conflicts when they attempt to commit.

6.1.1 Fixed Time/Fixed Class

In [51], if a transaction executes for t seconds, then aborts, it will execute for t seconds when it restarts. If an operation requires t seconds, the probability that it will be commit is $e^{-\gamma t}$, since we assume that conflicts form a Poisson process. Therefore, the number of times that a class c transaction with running time t must execute has the distribution

$$P_e^c(k|t) = (1 - e^{-\gamma_c t})^{k-1} e^{-\gamma_c t}$$

and has mean $e^{\gamma_c t}$. A class c transaction with running time t therefore has a mean residence time of $te^{\gamma_c t}$, and class c transactions have a running time of

$$\begin{aligned} R^c(V) &= \int_0^\infty te^{\gamma_c t} b_c(t) dt \\ &= -\mathcal{B}_c^1(-\gamma_c) \end{aligned}$$

where $\mathcal{B}_c^1$ is the first derivative of the Laplace transform of $b_c(t)$ [32]. Finally, the per-class utilization can be calculated for the iteration to be

$$U_c = R_a^c(V)/R^c(V) \tag{5}$$

$$= -b_c/\mathcal{B}_c^1(-\gamma_c) \tag{6}$$

We note that $b_c(t)$ must be $o(t^{-1}e^{-\gamma_c t})$ for the integral to converge.

6.1.2 Variable time / Fixed Class

In the variable time/fixed class model, every time a class c transaction executes its running time is sampled from $b_c(t)$. Therefore, the unconditional probability that the operation commits is:

$$p_c = \int_{t=0}^\infty e^{-\gamma_c t} b_c(t) dt \tag{7}$$

$$= \mathcal{B}_c(\gamma_c) \tag{8}$$

The number of times that the operation executes has a geometric distribution, so an operation will execute $1/p_c$ times. The first $1/p_c - 1$ times the operation executes, it will be unsuccessful. Knowing that the operation is unsuccessful tells us that it probably required somewhat longer than average to execute, since slow operations are more likely to be aborted. Similarly, successful operations are likely to be faster. In particular, an operation will be successful only if it reaches its commit point before a conflict occurs, and will be unsuccessful only if a conflict occurs before it reaches its commit point. The distributions of the execution times of the successful and unsuccessful operations are calculated by taking order statistics [14]:

$$b_c^s(t) = K_s e^{-\gamma_c t} b_c(t) \tag{9}$$

$$b_c^f(t) = K_f (1 - e^{-\gamma_c t}) b_c(t) \tag{10}$$

where K_s and K_f are normalizing constants computed by

$$K_s = \left(\int_0^\infty e^{-\gamma_c t} b_c(t) dt \right)^{-1}$$

$$K_f = \left(\int_0^\infty (1 - e^{-\gamma_c t}) b_c(t) dt \right)^{-1}$$

If b_c^s and b_c^f are the expected values of $b_c^s(t)$ and $b_c^f(t)$, respectively, then the expected time to complete a class c operation is

$$R_c(V) = b_c^s + (1/p_c - 1) b_c^f \tag{11}$$

$$= b_c^s + p_c^{-1}(1 - p_c) b_c^f \tag{12}$$

We observe that we only need to calculate b_c^s, because

$$b_c = p_c b_c^s + (1 - p_c) b_c^f \tag{13}$$

so that by combining (11) and (13) we get:

$$R_c(V) = b_c / p_c \tag{14}$$

Therefore, we find that

$$U_c = R_c^a(V) / R_c(V)$$

$$= b_c^s / b_c$$

$$= p_c \tag{15}$$

We note that in the variable time model, the only restriction on the distributions $b_c(t)$ is that they have finite means.

6.1.3 Variable Time / Variable Class

For the silent/static VV model, we calculate the conflict rate from formula (4) and the probability that a class c transaction commits from formula (8).

6.2 Analysis of Static/Broadcast OCC

In static/broadcast OCC, transactions access their entire data sets when they start execution, and abort whenever a conflicting transaction commits.

6.2.1 Fixed/Fixed

The probability that a transaction restarts is calculated in the same way as in the silent/static model, given the same conflict rate. The wasted time per transaction now has a truncated exponential distribution:

$$b_w^c(\tau|t) = \frac{\gamma_c e^{-\gamma_c \tau}}{1 - e^{-\gamma_c t}}$$

As a result,

$$U_c(V) = \frac{\gamma_c b_c}{1 - \mathcal{B}_c(-\gamma_c)} \tag{16}$$

6.2.2 Variable/Fixed

The probability that a transaction commits, p_c, and the expected execution time of transactions that commit b_c^s are calculated in the same way as in the silent/static model. The execution time of the aborted transactions is different, since a transaction will abort after t if some other transaction conflicts with it t seconds after it starts, and it has not yet committed:

$$b_c^f(t) = K_c^f \left[\gamma_c e^{-\gamma_c t}(1 - B_c(t))\right]$$

where

$$\begin{aligned} K_c^f &= \frac{1}{\int_0^\infty \gamma_c e^{-\gamma_c t}(1-B_c(t))dt} \\ &= \frac{1}{1-\gamma_c \int_0^\infty e^{-\gamma_c t} \int_0^t b_c(\tau)d\tau\, dt} \\ &= \frac{1}{1-\mathcal{B}_c(\gamma_c)} \end{aligned}$$

Since a conflict aborts a transaction early, we can not make use of equation (13) to simplify equation (11). Instead, we must actually calculate the expected values b_c^s and b_c^f:

$$\begin{aligned} b_c^s &= (1/p_c) \int_0^\infty t e^{-\gamma_c t} b_c(t) dt \\ &= -\mathcal{B}_c'(\gamma_c)/p_c \end{aligned} \tag{17}$$

$$b_c^f = K_c^f \int_{t=0}^\infty \gamma_c t e^{\gamma_c t}(1 - B_c(t))dt$$

$$= \frac{1}{1-\mathcal{B}_c(\gamma_c)} \left(1/\gamma_c - \gamma_c \left(\frac{d}{ds}\mathcal{B}_c(s)/s\Big|_{s=\gamma_c}\right)\right)$$
$$= \frac{1+\gamma_c\mathcal{B}_c'(\gamma_c)-\mathcal{B}_c(\gamma_c)}{\gamma_c(1-\mathcal{B}_c(\gamma_c))} \tag{18}$$

Putting these formulae into equation (11) for $R_c(V)$, we find that

$$R_c(V) = \frac{1-\mathcal{B}_c(\gamma_c)}{\gamma_c\mathcal{B}_c(\gamma_c)} \tag{19}$$

and,

$$U_c(V) = \frac{b_c\gamma_c\mathcal{B}_c(\gamma_c)}{1-\mathcal{B}_c(\gamma_c)} \tag{20}$$

We note that if $b_c(t)$ has an exponential distribution, then $U_c = 1$. This relation can be used to directly solve a system where all execution times are exponentially distributed, or to simplify the calculations when some execution time distributions are exponentially distributed and some are not.

6.2.3 Variable/Variable

In the silent/static case, a class k transaction executes for an expected b_k seconds. In the broadcast/static case, a transaction terminates early if it is aborted. The average amount of time that a transaction spends executing a class k transaction, $\bar{b}_k$, is the weighted average of the execution time depending on whether or not the transaction commits. By using equations (17) and (18), we find that:

$$\bar{b}_k = p_k b_k^s + (1-p_k)b_k^f$$
$$= (1-\mathcal{B}_k(\gamma_k))/\gamma_k \tag{21}$$

Therefore, the proportion of time that a process spends executing a class k transaction is

$$S_k = f_k\bar{b}_k / \sum_{i=1}^{C} f_i\bar{b}_i \tag{22}$$

and the conflict rate of a class c transaction is

$$\gamma_c = \frac{V-1}{\bar{b}} \sum_{k=1}^{C} \Phi(c,k) f_k p_k \tag{23}$$

where $\bar{b} = \sum_{i=1}^{C} f_i\bar{b}_i$. Given a conflict rate γ_c, we calculate p_c by using equation (8).

6.3 Analysis of Silent/Dynamic

In dynamic optimistic concurrency control, a transaction accesses data items as they are needed. A class c transaction that requests n_c data items has $n_c + 1$ phases. As the transaction accesses more data items, it acquires a higher conflict rate. We redefine the conflict function Φ to model the different phases of the

transactions. If a class k transaction commits, it conflicts with a class c transaction in stage i with probability $\Phi(k,c,i)$. The probability that a committing transaction conflicts with a class c transaction in stage i is:

$$\Phi_{c,i} = \sum_{k=1}^{C} f_k \Phi(k,c,i) \tag{24}$$

The conflict rate for a class c transaction in stage i is:

$$\gamma_{c,i} = \frac{(V-1)\Phi_{c,i}}{b} U(V) \tag{25}$$

The amount of time that a class c transaction spends in stage i has the distribution $b_{c,i}(t)$ with mean $b_{c,i}$, and the average time to execute the transaction is $b_c = \sum b_{c,i}$.

6.3.1 Fixed/Fixed

As a transaction moves through different stages, it encounters different conflict rates. The conflict rate for a class c transaction is a vector:

$$\vec{\gamma_c} = (\gamma_{c,1}, \gamma_{c,2}, \ldots, \gamma_{c,n_c+1})$$

Similarly, the execution time of a class c transaction is a vector $\vec{x} = (x_1, x_2, \ldots, x_{n_c+1})$, where x_i is a sample from the distribution with density $b_{c,i}(x)$. The probability that a class c transaction aborts is therefore

$$P_G^c = 1 - e^{\vec{\gamma_c} \cdot \vec{x}}$$

By taking expectations over the times for the processing stages, Ryu and Thomasian find that

$$R^c(V) = -\left(\prod_{i=0}^{n_c} \mathcal{B}_c(-\gamma_{c,i})\right) \sum_{i=0}^{n_c} \frac{\mathcal{B}_c^1(-\gamma_{c,i})}{\mathcal{B}_c(-\gamma_{c,i})}$$

6.3.2 Variable/Fixed

We use the same transaction model as in the Fixed/Fixed case. A transaction will commit only it completes every stage without conflict. We define $p_{c,i}$ to be the probability that a class c transaction completes the i^{th} stage without a conflict. We can calculate $p_{c,i}$ by using formula (8), and substituting $\mathcal{B}_{c,i}$ for $\mathcal{B}$ and $\gamma_{c,i}$ for γ_c. Given the $p_{c,i}$, we can calculate p_c by

$$\begin{aligned} p_c &= \prod_{i=0}^{n_c+1} p_{c,i} \\ &= \prod_{i=0}^{n_c+1} \mathcal{B}_{c,i}(\gamma_{c,i}) \end{aligned} \tag{26}$$

As in the case of silent/static concurrency control, the unconditional expected time spent executing a class c transaction is b_c, so that

$$U_c = p_c \tag{27}$$

6.3.3 Variable/Variable

For the VV model, we use formula (4), appropriately modified to calculate the conflict rates, and formula (26) to calculate p_c.

6.4 Dynamic/Broadcast

6.4.1 Fixed/Fixed

The analysis of dynamic/broadcast concurrency control under the fixed/fixed model uses a combination of the previously discussed techniques. Ryu and Thomasian show that

$$R_d^c = R_{d1}^c + R_{d2}^c$$

$$R_{d1}^c = \mathcal{B}_c^1(0) \sum_{k=0}^{n_c} \left[k \left(\prod_{i>k}^{n_c} \mathcal{B}_c(-\gamma_{c,i}) \right) (1 - \mathcal{B}_c(-\gamma_{c,k})) \right]$$

$$R_{d2}^c(V) = \sum_{k=0}^{n_c} \left[\frac{1}{\gamma_{c,k}} \left(\prod_{i>k}^{n_c} \mathcal{B}_c(-\gamma_{c,i}) \right) (\mathcal{B}_c(-\gamma_{c,k}) - 1 - \gamma_{c,k} b_c) \right]$$

6.4.2 Variable/Fixed

We can use formula (26) to calculate p_c. For each processing phase, we can use formulae (17) and (18) to calculate $b_{c,i}^s$ and $b_{c,i}^f$. If a transaction commits, then it successfully completed each phase, so that

$$b_c^s = \sum_{i=0}^{n_c+1} b_{c,i}^s \tag{28}$$

If a transaction fails to commit, then it might have failed at any one of the $n_c + 1$ stages. We define $q_c = 1 - p_c$ to be the probability that a transaction aborts, and $q_{c,i}$ to be the probability that a transaction aborts at stage i, given that it aborts. A transaction that aborts at stage i must have successfully completed the previous $i - 1$ stages, and a transaction aborts at exactly one of the stages, so

$$q_{c,i} = \frac{1 - p_{c,i}}{q_c} \prod_{j=1}^{i-1} p_{c,j}$$

If a transaction aborts at stage i, then its expected execution time is:

$$b_{c,i}^f + \sum_{j=1}^{i-1} b_{c,j}^s$$

Therefore, b_c^f is the unconditional expected execution time:

$$b_c^f = \sum_{i=0}^{n_c+1} q_{c,i} \left(\sum_{j=0}^{i-1} b_{c,j}^s + b_{c,i}^f \right) \tag{29}$$

We then use formulae (28) and (29) in formula (11) to find $R_c(V)$.

6.4.3 Variable/Variable

We use formula (26) to calculate p_c, and formulae (28) and (29) in formulae (21) and (23) to calculate the conflict rate.

7 Model Validation and Experiments

We wrote an OCC simulator to validate our analytical models. A parameterized number of transactions executed concurrently, and committing transactions conflicted with other transactions depending on a sample from Φ. We ran the simulation for 10,000 transaction executions, then reported statistics on throughput, execution time, and commit probabilities.

Ryu and Thomasian have already validated the F/F model, so we present a validation only of the V/F and V/V models (we also simulated the F/F model, and found close agreement between the simulation and analysis). In our first validation study, we modeled a system with a single transaction type. If there is only one transaction type, the V/F and the V/V models are the same, so we present results for the V/F model only (we also ran simulations and analytical calculations for the V/V model, and obtained nearly identical results). We calculated Φ by assuming that the transactions randomly accessed data items from a database that contained $N = 1024$ data items, and that transactions with overlapping data sets conflict. Ryu and Thomasian provide the following formula for the probability that two access sets of size n and m overlap in a database with N data items:

$$\Psi(n, m|N) = 1 - \binom{N-n}{m} / \binom{N}{m}$$

We report the probability that a transaction commits for a variety of access set sizes and degrees of concurrency in Table 1. The execution times in the static concurrency control experiments and the phase execution times in the dynamic concurrency control experiments were exponentially distributed. The experiments show close agreement between analytical and simulation results, though the calculations are least accurate for the dynamic concurrency control when the level of conflict is high.

We also performed a validation study for a system with two transaction classes. The first transaction class accesses four data items, and the second accesses eight data items. To save space, we reports results for Dynamic/Broadcast OCC only, it being the least accurate of the models. Table 2 reports simulation and analytical results for the V/F and the V/V transaction models for a variety of degrees of concurrency. In these experiments, $f_1 = .6$ and $f_2 = .4$. We found close agreement between the simulation and the analytical predictions.

V		Static/Silent			Static/Broadcast		
access set size		4	16	32	4	16	32
5	sim	.9391	.6418	.4655	.9378	.5270	.2807
	ana	.9444	.6366	.4586	.9414	.5272	.2797
15	sim	.8399	.4249	.2735	.8113	.2439	.0997
	ana	.8446	.4272	.2822	.8212	.2416	.0999
25	sim	.7716	.3474	.2152	.7201	.1569	.0614
	ana	.7755	.3481	.2241	.7281	.1567	.0608
		Dynamic/Silent			Dynamic/Broadcast		
5	sim	.9700	.7020	.4404	.9686	.6811	.3937
	ana	.9704	.7189	.4879	.9700	.6967	.4325
15	sim	.9025	.4587	.2627	.9012	.4126	.1736
	ana	.9071	.4733	.2627	.9039	.4187	.1971
25	sim	.8481	.3588	.1645	.8405	.2988	.1156
	ana	.8554	.3703	.1910	.8479	.3078	.1319

Table 1: Validation study of the V/F model. p_c is reported for a single transaction class and exponentially distributed execution times.

V	Varying/Fixed				Varying/Varying			
	analytical		simulation		analytical		simulation	
	class 1	class 2	class 1	class 2	class 1	class 2	class 1	class 2
5	.9692	.8495	.9684	.8941	.9692	.8946	.9672	.8897
15	.9069	.7081	.9069	.7088	.9066	.7073	.9009	.6919
25	.8589	.5864	.8552	.5797	.8574	.5828	.8511	.5628
35	.8203	.5011	.8167	.4906	.8168	.4935	.8014	.4660
45	.7884	.4379	.7886	.4282	.7822	.4263	.7675	.3986
55	.7576	.3812	.7612	.3892	.7521	.3736	.7294	.3495

Table 2: Validation study for Dynamic/Broadcast OCC and two transaction classes. p_c is reported. Execution phase times are exponentially distributed.

8 Analysis of Nonblocking Data Structures

In this section, we apply the analytical framework to model the performance of non-blocking data structures and explore several performance implications. Our analytical framework can be used to model non-blocking data structure algorithms that have the basic form described in section 2 in Codes 2 and 3. While some non-blocking algorithms use a different mechanism [24, 17, 34], most of the recently proposed methods [45, 58, 19, 55, 59, 60, 56, 23] are similar to these techniques.

8.1 Atomic Snapshot

We examine first the algorithms similar to Code 2 in which taking the snapshot consists performing one read (i.e., reading the pointer to the object). This approach is used by Herlihy [21], is a step in Turek's algorithms [58] and is an approximation to the algorithms proposed by Prakash et al. [45], Valois [59, 60], and Harathi and Johnson [19].

We want to model both transient and permanent slowdowns. The V/F model accounts for transient and permanent slowdowns, and the V/V model permits transient slowdowns only. We are modeling algorithms in which the snapshot is performed atomically, so the operations execute SS transactions.

In Herlihy's algorithms, every operation conflicts with every other, so $\Phi = 1$. In our experiments, we use two transaction classes to model the fast and slow processors. The first transaction class models the fast processors. Its execution time is chosen uniformly randomly in $[.8, 1.2]$, and $f_1 = .9$. The execution time of the second transaction class, which represents the slow processors, is chosen uniformly randomly in $[8, 12]$, and $f_2 = .1$.

We plot the throughput of the nonblocking queue for the permanent and transient slowdown models (VF and VV) against increasing V in Figure 1. For comparison, we also plot the throughput of the locking algorithm algorithm, which is a constant $1/b = 1/1.9$. The nonblocking queue in the permanent slowdown model has a lower throughput than the locking queue, in spite of the preference shown towards fast executions. This phenomena occurs because of the extremely long times required for the completion of the operations executed on the slow processors. These running times are shown in Figure 2. The throughput of the transient slowdown model increases with increasing V, and is considerably greater than that of the locking queue. These model predictions are in agreement with our simulation results [45].

The Ryu and Thomasian models assume a closed system and calculate the throughput and response time as a function of the the number of competing operations. Access to a shared data structure can be better modeled as an open system, in which operations arrive, receive service, then depart. We can use the results

from the closed-system model to approximate the performance measures of an open system. The throughput values for the closed system are used for the state-dependent service rates in a flow-equivalent server [31]. The steps to compute open system response times in the FF and the VF transaction models are:

1. For $V = 1, \ldots, MAX$, calculate the per-class and average response times.

2. Model the number of jobs in the system as a finite-buffer queue. Use the average response times (across all transaction types) as the state-dependent service times. Given the arrival rate λ, calculate the state occupancy probabilities.

3. Use the state occupancy probabilities to weight the per-class response times and compute the average response time by taking the sum.

In the VV model, per-class execution times aren't meaningful. Instead, one calculates the average transaction execution time. The expected probability that a VV transaction commits is: $P_c^{VV} = \sum_{k=1}^{C} f_k b_k p_k / b$. A transaction re-executes until it commits. Thus, the number of executions has a geometric distribution, with expected value $1/P_c^{VV}$. Therefore, the expected time to execute a transaction is

$$\begin{aligned} R(V) = & \quad b/P_c^{VV} \\ = & \quad 1/(\sum_{k=1}^{C} f_k b_k p_k) \end{aligned}$$

Using the parameters from the previous experiment, we plot the response time of the single-snapshot algorithm under the permanent and the transient slowdown processor models against an increasing arrival rate in Figure 3. We also report the results of a simulation for both of the processor models. The chart shows that the VV analytical model accurately predicts response times of the transient slowdown model, but that the VF model is overly optimistic. Figure 4 compares analytical and simulation predictions of the probability that the system is idle for both processor models. Here we can see again that the VV model makes accurate predictions, while the VF model is too optimistic. We include in Figure 3 a plot of the response time of an equivalent locking algorithm (modeled by a M/G/1 queue [32]). The locking algorithm has a considerably better response time than the non-blocking algorithm under the permanent slowdown model. The non-blocking algorithm under the transient slowdown model has a similar response time under a light load, but a lower response time under a heavy load.

In observing the simulations, we noticed that the response time of operations that are alone in the system when they complete is close to response times when there are two operations in the system. This occurs because the jobs that complete when they are alone in the system are often slow jobs that had been forced to

restart several times. We therefore make an approximation (which we call *VF approx*) to the flow-equivalent by setting the service rate when there is one operation in the system to that when there are two jobs in the system. The predictions made by this approximation for the VF model are labeled **VF approx** in Figures 3 and 4. The VF approx makes poor predictions of response times, but accurate predictions of the system utilization.

To test the robustness of our models in the face of different service time distributions, we ran the experiments with the permanent slowdown processor model where the service time distributions have an exponential distribution. The results of these experiments are shown in Figures 5 and 6. These figures also show that the VF model is too optimistic, and that the VF approx model makes poor predictions of the response times but good predictions of the system utilization.

8.2 Composite Snapshot

Several non-blocking algorithms take a snapshot of several variables to determine the state of the data structure [45, 59, 60, 19, 22]. While taking an atomic composite snapshot requires a more complex algorithm, it reduces the amount of copying needed to perform an operation, which improves performance. In addition, architectures that support lock-free algorithms have been proposed [23, 56]. These architectures allow a process to reserve several words of shared memory, and inform the processor if a conflicting write occurs.

Code 5, taken from [45], shows a typical protocol to take an atomic snapshot for an algorithm that implements a non-blocking queue. The nonblocking queue needs to determine the simultaneous values of the three variables in order to determine the state of the queue. We call the three variables `A`, `B`, and `C`, and the protocol reads their simultaneous values into `my_A`, `my_B`, and `my_C`.

```
repeat
   my_A=A
   repeat
      my_B=B
         my_C=C
   until(B == my_B)
until(A == my_A)
```

Code 5 *Composite snapshot.*

During the time that an operation is taking a snapshot, a modification to the data structure can cause the snapshot to fail. Further, as the snapshot is taken, different modifications can cause the snapshot to fail. Thus, while the snapshot is in progress, the operation uses DB optimistic concurrency control. After the snapshot is successfully taken, the operation calculates its update, then attempts to commit its update.

The operation will not abort during the time that it calculates its update, so this stage of the operation uses SS optimistic concurrency control.

Since the optimistic concurrency control used for composite-snapshot non-blocking algorithms is a variation of the DB concurrency control, we use the methods similar to those discussed in section 6.4 to calculate the execution times and the probability of success. The last stage in the calculation will not terminate early when a conflicting commits. Therefore, the value of b^f_{c,n_c+1} in (29) should be calculated using the method described in section 6.1.2:

$$b_c^{f,SS} = \frac{b_c + \mathcal{B}'_c(\gamma_c)}{1 - \mathcal{B}_c(\gamma_c)} \tag{30}$$

We assume that an operation is equally likely to be an enqueue or a dequeue operation, and that the queue is usually full. In this case, when an enqueue operation commits, it kills all other enqueue operations, and the same applies to the dequeue operations. Therefore, one operation kills another upon commit with probability 1/2. We start counting the operation's execution from the point when it executes the statement `my_A=A`. The first stage ends when the first `until` statement is executed, and requires 4 instructions. The second stage ends when the second `until` statement is executed, and requires 1 instruction. The third stage ends when the operation tries to commit its operation, and requires 8 instructions. Fast processors require a time uniformly randomly chosen in $[.8, 1.2]$ to execute the instructions in a stage, and slow processors require a time uniformly randomly chosen between in $[8, 12]$. That is, the time to execute a stage is the number of instructions in the stage multiplied by a sample uniformly randomly selected from $[lo, hi]$.

The results of the experiments are shown in Figures 7 and 8. These figures show the response times and idle probability, respectively. Again we draw the conclusions that the VV model makes accurate predictions, that the VF model is too optimistic, and that the VF approx model makes poor predictions of response times but good predictions of the idle probability.

9 Conclusion

In this work we present a model for analyzing the performance of a large class of non-locking algorithms. This model is an extension of the Ryu and Thomasian model of Optimistic concurrency control. Our extensions allow operations to resample their execution time if they abort (VF transaction model), and also to change that change their operation class (VV transaction model). We validate our models in a closed system under a variety of concurrency control models.

We next apply the analytical tools to compare the performance of non-locking and locking algorithms for shared objects. We use two processor models. In the permanent slowdown model, the execution speed

of the processor is fixed, modulo small variations. In the transient slowdown model, the execution speed of a processor changes between executions. We use the VF transaction model for the permanent slowdown processor model and the VV transaction model for the transient slowdown processor model. Permanent slowdowns can occur due to NUMA architectures, heterogeneous architectures, or differences in operation execution time. Transient slowdowns can occur due to cache line faults, memory and bus contention, page faults, context switching, or data-dependent operation execution times.

We compared the performance of the non-locking and the locking algorithms in a closed system, and found that non-locking algorithms in the variable speed model have significantly better throughput than the locking algorithm, but that non-locking algorithms in the permanent slowdown model have significantly worse throughput. While the closed system model does not give direct performance results for a real system, it indicates the relative performance of the algorithms and it provides a bound on the rate at which operations can execute.

We extend the closed system model to an open system by using a flow-equivalent approximation. The analytical results of this approximation show the same performance ranking with respect to response times as exists in the closed system. Further, the VV model is slightly pessimistic, while the VF model is very optimistic, making us more confident in our performance ranking. We describe a further approximation that lets us accurately calculate the utilization of the concurrent object in the VF model. The analytical models are accurate enough to be useful in predicting the impact of a non-locking concurrent object on system performance.

This work indicates that non-locking algorithms have the potential to provide better performance than locking algorithms when the processors executing the operations experience transient slowdowns only. Thus, lock-free algorithms are appropriate on UMA architectures when all operations on the data require about the same processing time. However, our work shows that lock-free algorithms have poor performance when the processors can experience permanent slowdowns. Slow processors receive significant discrimination, reducing overall throughput. Thus, lock-free algorithms are not appropriate on heterogeneous or NUMA architectures, or when some types of operations require significantly more computation than others. In these cases, non-blocking algorithms must incorporate a fairness mechanism to provide good performance. Approachs to such mechanisms are described in [2, 11].

References

[1] A. Agrawal and M. Cherian. Adaptive backoff synchronization techniques. In *Int'l Symposium on*

Computer Architecture, pages 396–406, 1989.

[2] J. Alemany and E.W. Felton. Performance issues in non-blocking synchronization on shared memory multiprocessors. In *Proc. ACM Symp. Principles of Distributed Computing*, 1992.

[3] R. Anderson and H. Woll. Wait-free algorithms for the union-find problem. In *Proc. ACM Symp. on Theory of Computation*, pages 370–380, 1991.

[4] T. E. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[5] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. on Computer Systems*, 10(1):53–79, 1992.

[6] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Trans. on Computers*, 38(12):1631–1644, 1989.

[7] K.E. Atkinson. *An Introduction to Numerical Analysis*. John Wiley and Sons, 1978.

[8] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[9] Inc. BBN Advanced Computers. Tc2000 programming handbook.

[10] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[11] B. Bershad. Practical considerations for non-blocking concurrent objects. In *Int'l Conf. on Distributed Computing Systems*, pages 264–273, 1993.

[12] J. Biswas and J.C. Browne. Simultaneous update of priority structures. In *Proceedings of the International Conference on Parallel Processing*, pages 124–131, 1987.

[13] I.Y. Bucher and D.A. Calahan. Models of access delays in multiprocessor memories. *IEEE Trans. on Parallel and Distributed Systems*, 3(3):270–280, 1992.

[14] H.A. David. *Order Statistics*. John Wiley, 1981.

[15] C.S. Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, c-29(9):811–817, 1980.

[16] R.R. Glenn, D.V. Pryor, J.M. Conroy, and T. Johnson. A bistability throughput phenomenon in a shared-memory mimd machine. *The Journal of Supercomputing*, 7:357–375, 1993.

[17] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Coordinating large numbers of processors. In *Proceedings of the International Conference on Parallel Processing*. IEEE, 1981.

[18] G. Graunke and S. Thakkar. Synchronization mechanisms for shared-memory multiprocessors. *IEEE Computer*, 26(3):60–69, 1990.

[19] K. Harathi and T. Johnson. A priority synchronization algorithm for multiprocessors. Technical Report tr93.005, UF, 1991. available at ftp.cis.ufl.edu:cis/tech-reports.

[20] T. Harder. Observations on optimistic concurrency control schemes. *Inform. Systems*, 9(2):111–120, 1984.

[21] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceeding of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206. ACM, 1989.

[22] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. on Programming Languages and Systems*, 15(5):745–770, 1993.

[23] M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. Int'l Symp. on Computer Architecture*, pages 289–300, 1993.

[24] M. Herlihy and J. Wing. Axioms for concurrent objects. In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 13–26, 1987.

[25] M.P. Herlihy and J.E.B. Moss. Lock-free garbage collection for multiprocessors. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 229–236, 1991.

[26] IBM T.J. Watson Research Center. *System/370 Principles of Operations*, 1983.

[27] T. Johnson. Approximate analysis of reader and writer access to a shared resource. In *ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 106–114, 1990.

[28] T. Johnson. *The Performance of Concurrent Data Structure Algorithms*. PhD thesis, NYU Dept. of Computer Science, 1990.

[29] T. Johnson and D. Shasha. The performance of concurrent data structure algorithms. *Transactions on Database Systems*, March 1993.

[30] D. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, 1989.

[31] K. Kant. *Introduction to Computer System Performance Evaluation.* McGraw Hill, 1992.

[32] L. Kleinrock. *Queueing Systems*, volume 1. John Wiley, New York, 1975.

[33] H.T. Kung and P.L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.

[34] L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systems*, 5(2):190–222, 1983.

[35] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[36] B.H. Lim and A. Agrawal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Trans. on Computer Systems*, 11(3):253–294, 1993.

[37] U. Manber and R.E. Ladner. Concurrency control in a dynamic search structure. In *Principles of the ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 268–282, 1982.

[38] C. McCann, R. Vaswami, and J. Zahoran. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 11(2):146–176, 1993.

[39] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Computer Systems*, 9(1):21–65, 1991.

[40] D. Menasce and T. Nakanishi. Optimistic vs. pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[41] R. Morris and W. Wong. Performance of concurrency control algorithms with non-exclusive access. In *Performance '84*, pages 87–101, 1984.

[42] R. Morris and W. Wong. Performance analysis of locking and optimistic concurrency control algorithms. *Performance Evaluation*, 5:105–118, 1985.

[43] Motorola. M68000 family programmer's reference manual.

[44] S. Prakash, Y.H. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. In *Proc. Int'l Conf. on Parallel Processing*, pages II68–II75, 1991.

[45] S. Prakash, Y.H. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Trans. on Computers*, 43(5), 1994.

[46] V. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37(12):1657–1665, 1988.

[47] M.I. Reiman and P.E. Wright. Performance analysis of concurrent-read exclusive-write. In *Proc. ACM Sigmetrics Conference on Measuring and Modeling of COmputer SYstems*, pages 168–177, 1991.

[48] J.T. Robinson. Experiments with transaction processing on a multiprocessor. Technical Report RC9725, IBM, Yorktown Heights, 1982.

[49] S.M. Ross. *Stochastic Processes*. John Wiley, 1983.

[50] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *Proc. Int'l Symp. on Computer Architecture*, pages 340–347, 1984.

[51] I.K. Ryu and A. Thomasian. Performance analysis of centralized database with optimistic concurrency control. *Performance Evaluation*, 7:195–211, 1987.

[52] I.K. Ryu and A. Thomasian. Analysis of database performance with dynamic locking. *J. ACM*, 37(3):491–523, 1990.

[53] Y. Sagiv. Concurrent operations on B^*-trees with overtaking. In *4th ACM Symp. Principles of Database Systems*, pages 28–37. ACM, 1985.

[54] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, 1988.

[55] J. Stone. A simple and correct shared-queue algorithm using compare-and-swap. Technical Report RC 15675, IBM TJ Watson Research Center, 1990.

[56] J.M. Stone, H.S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the oklahoma update. *IEEE Parallel and Distributed Technology, Systems and Applications*, 1(4):58–71, 1993.

[57] Y.C. Tay, R. Suri, and N. Goodman. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4):415–462, 1985.

[58] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *ACM Symp. on Principles of Database Systems*, pages 212–222, 1992.

[59] J.D. Valois. Analysis of a lock-free queue. Submitted for publication, 1992.

[60] J.D. Valois. Concurrent dictionaries without locks. Submitted for publication, 1992.

[61] P. J. Woest and J. R. Goodman. An analysis of synchronization mechanisms in shared-memory multiprocessors. In *International Symposium on Shared Memory Multiprocessing*, pages 656–659, Tokyo, Japan, April 1991.

[62] P.S. Yu, D.M. Dias, and S.S. Lavenberg. On modeling database concurrency control. Technical Report RC 15368, IBM Research Division, 1990.

[63] P.S. Yu, H.U Heiss, and D.M Dias. Modeling and analysis of a time-stamp history based certification protocol for concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):525–537, 1991.

[64] J. Zahoran, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, 1991.

[65] C.-Q Zhu and P.-C. Yew. A synchronization scheme and its applications for large multiprocessor systems. In *Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 486–493, 1984.

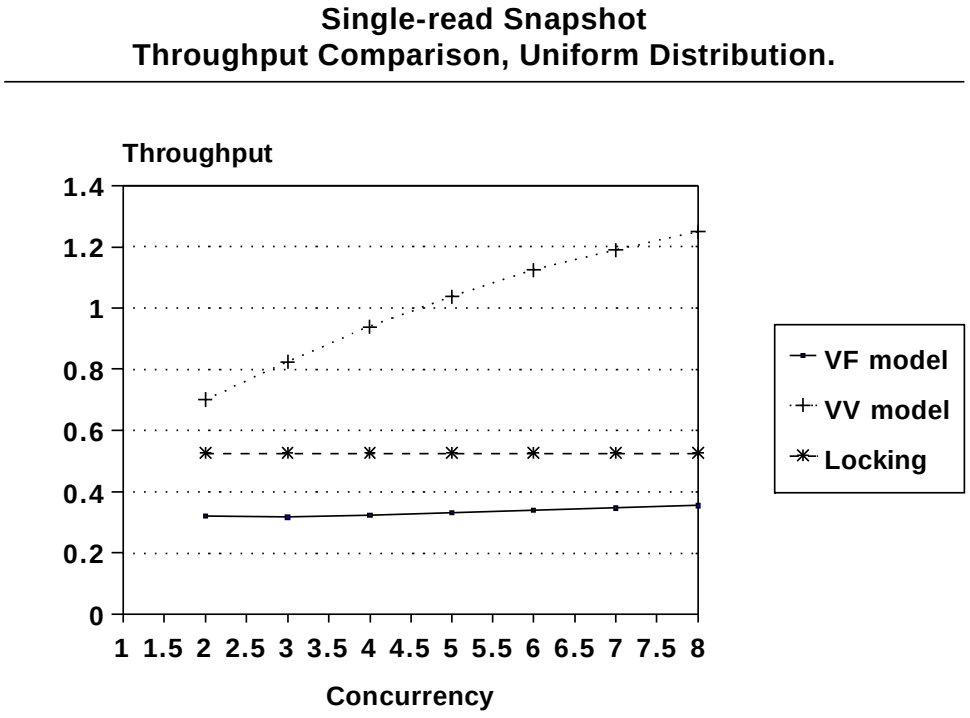

Figure 1: Throughput of the locking queue and the nonblocking queue with transient and permanent slowdowns.

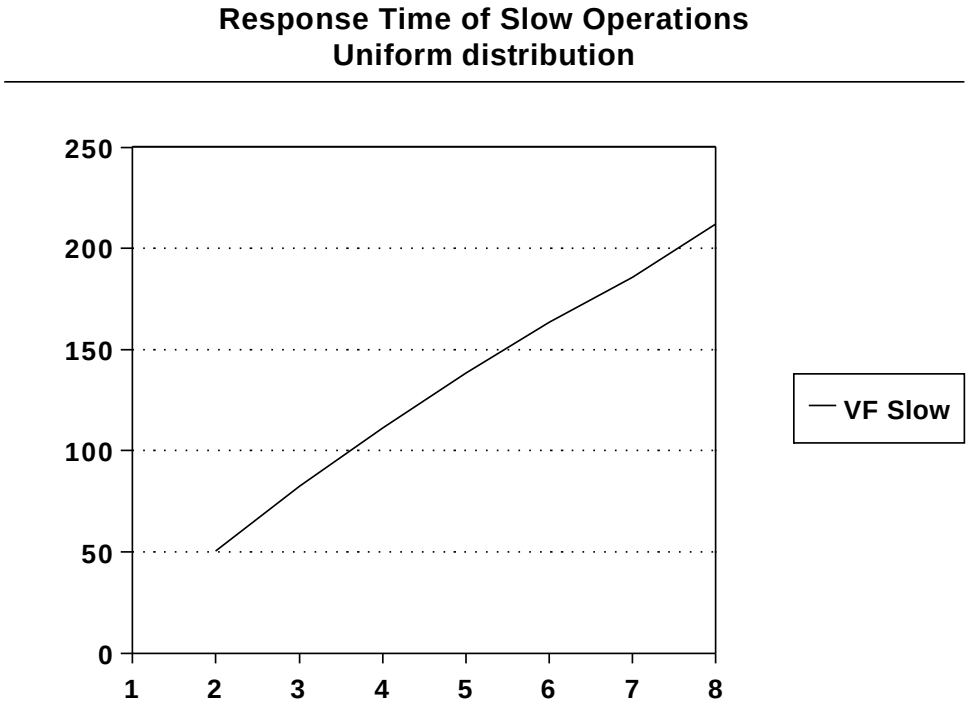

Figure 2: Response time of the slow operations in the permanent slowdown model.

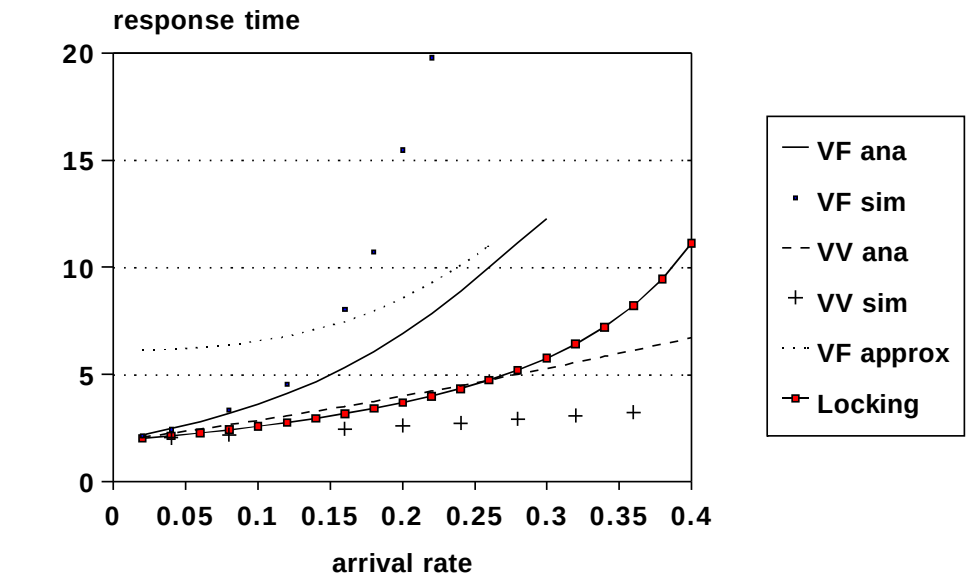

Figure 3: Comparison of analytical and simulation results

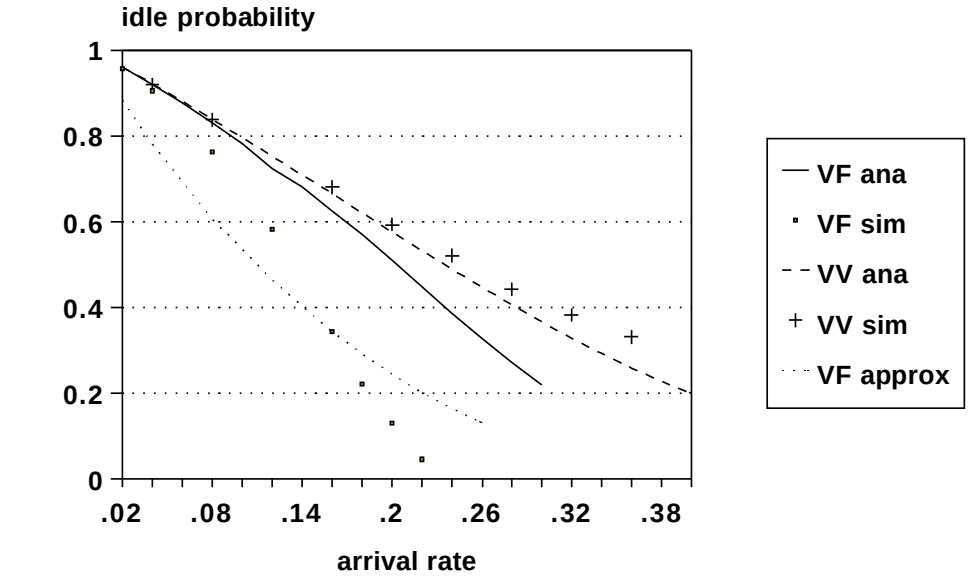

Figure 4: Comparison of analytical and simulation results

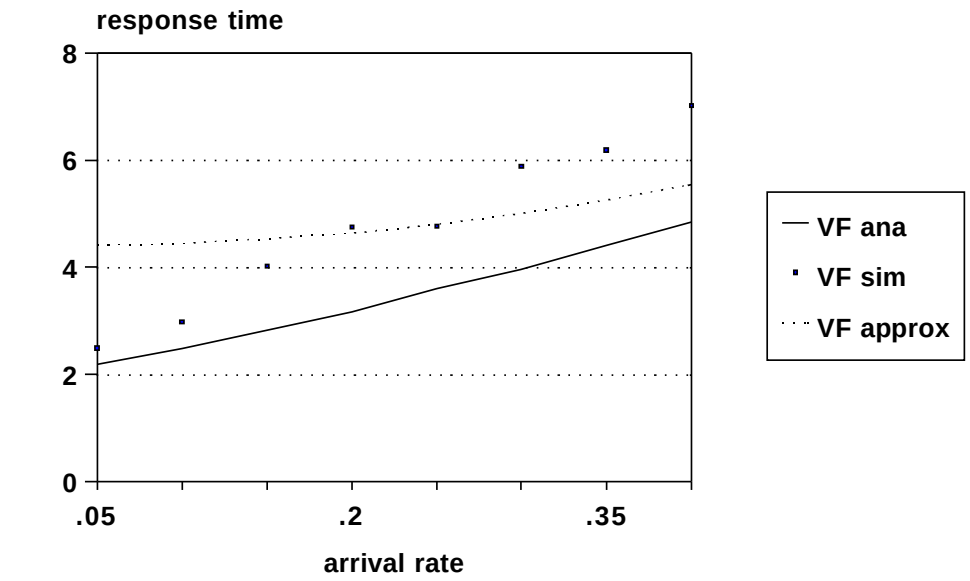

Figure 5: Comparison of analytical and simulation results

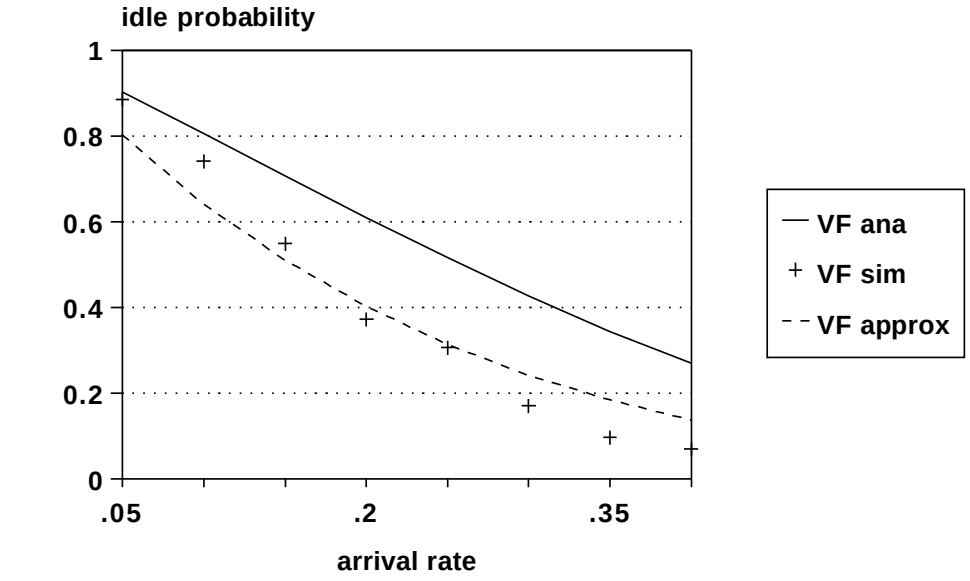

Figure 6: Comparison of analytical and simulation results

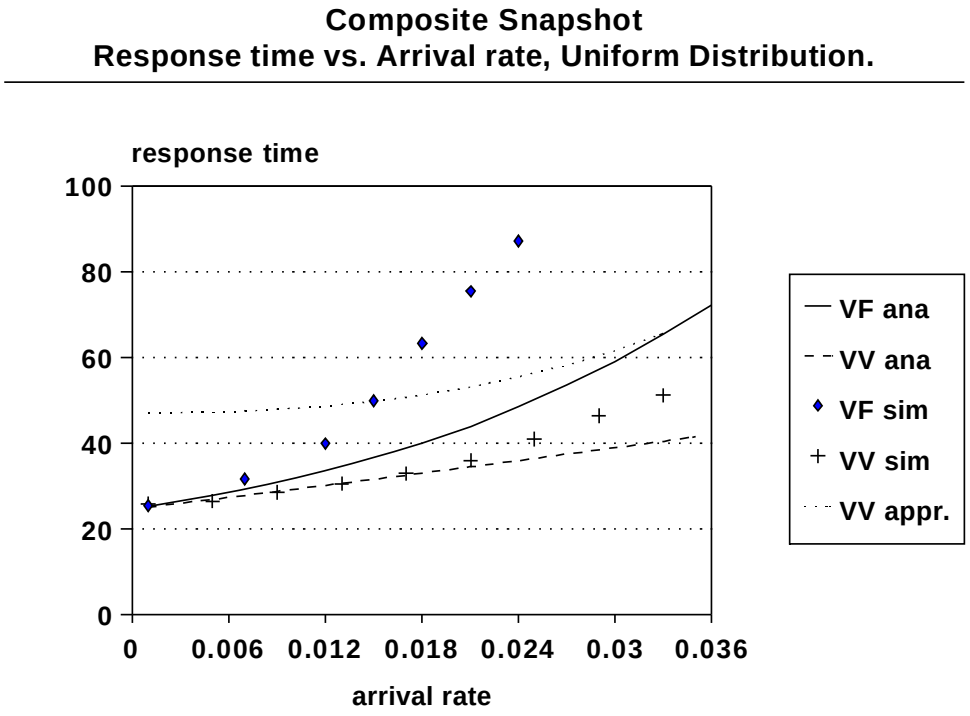

Figure 7: Comparison of analytical and simulation results

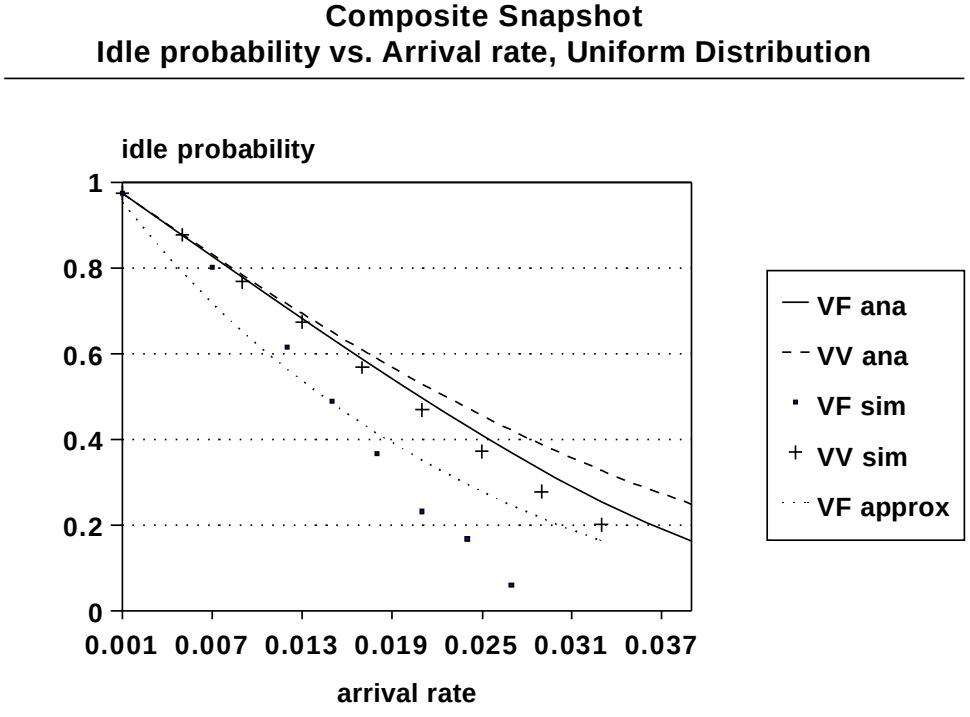

Figure 8: Comparison of analytical and simulation results