Load Control for Locking: The 'Half-and-Half' Approach

by

Michael J. Carey
Sanjay Krishnamurthi
Miron Livny

Computer Sciences Technical Report #880
October 1989

Load Control for Locking: The 'Half-and-Half' Approach

Michael J. Carey
Sanjay Krishnamurthi
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was supported in part by the National Science Foundation under grant IRI-8657323 and by grants from the Digital Equipment Corporation and the Microelectronics and Computer Technology Consortium (MCC).

Load Control for Locking: The 'Half-and-Half' Approach

Michael J. Carey
Sanjay Krishnamurthi
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

A number of concurrency control performance studies have shown that, under high levels of data contention, concurrency control algorithms can exhibit thrashing behavior which is detrimental to overall system performance. In this paper, we present an approach to eliminating thrashing in the case of two-phase locking, a widely used concurrency control algorithm. Our solution, which we call the 'Half-and-Half' Algorithm, involves monitoring the state of the DBMS in order to dynamically control the multiprogramming level of the system. Results from a performance study indicate that the Half-and-Half algorithm can be very effective at preventing thrashing under a wide range of operating conditions and workloads.

1. INTRODUCTION

In computer systems with multiple CPU and I/O resources, *multiprogramming* is usually employed in order to permit a number of tasks to run simultaneously. Multiprogramming offers several benefits that make it "the most important concept in modern operating systems" [Pete86]. It enables multiple users to concurrently access the resources of a single computer system. It also provides for increased utilization of computer system resources, enabling systems to accomplish more work per unit of time. Multiprogramming is not without its dangers, however. As described by Denning over two decades ago [Denn68], multiprogramming can also lead to *thrashing*, a situation where an attempt to improve performance by running additional tasks leads instead to excessive overhead and severe performance degradation. In the context of a computer system with virtual memory, this can occur when physical memory becomes over-utilized so that tasks begin stealing pages from one another; under these conditions, the number of page faults incurred by each task increases dramatically. Various solutions to this problem have been developed in the operating system community over the years [Pete86].

This research was supported in part by the National Science Foundation under grant IRI-8657323 and by grants from the Digital Equipment Corporation and the Microelectronics and Computer Technology Consortium (MCC).

Like operating systems, most database management systems (DBMSs) are multiprogrammed in order to permit resource sharing and to maximize transaction throughput. And, as in operating systems, the system must be protected against thrashing in order to realize the potential benefits of multiprogramming. In database systems, however, there are at least two distinct potential sources for thrashing behavior. The first source is related to buffer management, and is very similar to the virtual memory thrashing problem faced by operating systems. Transactions use and compete for buffer pages in much the same way that processes compete for virtual memory, so buffer thrashing can occur if the buffer pool becomes overutilized. This problem has been studied in some depth, and several solutions based on intelligently reserving a certain number of buffer pages before admitting a transaction have been proposed [Chou85, Sacc86]. The second potential source of thrashing is very different, and seems to be quite unique to database systems. This source is the concurrency control manager, which coordinates transactions' accesses to shared data in order to prevent data consistency problems that could otherwise result. In this paper we will focus on thrashing in the context of dynamic two-phase locking (2PL), the most widely used algorithm for database concurrency control [Bern87].

In 2PL as it is usually implemented, transactions set shared locks on objects that they read and exclusive locks on those objects that they write.[1] Transactions hold their locks until they either commit successfully or abort. Shared locks are compatible with one another, but an exclusive lock on an object is incompatible with other shared and exclusive locks on the object. In the event of a lock request that conflicts with the current lock holders or waiters for an object, the transaction that made the conflicting request is required to wait until the lock becomes available in the requested mode. A waits-for graph of transactions is maintained, and deadlock detection is performed when a transaction is required to block. In the event of a deadlock, one of the transactions involved (e.g., the youngest one) is chosen as the victim and is aborted.

A number of concurrency control performance studies [Balt82, Care84, Fran85, Tay85, Agra87a] have observed that 2PL can cause thrashing under high levels of data contention. This can occur due to one of two causes. If the system resources are lightly utilized, transaction waiting can lead to thrashing

[1] Exclusive locks are usually acquired through upgrading a previously obtained shared lock to exclusive mode.

[Balt82, Fran85, Tay85, Agra87a]. Such *wait-induced thrashing* occurs when data contention is so high that, due to lock waiting, allowing a new transaction into the system actually leads to a decrease rather than an increase in the number of runnable transactions. In a system with resources that are more heavily utilized, thrashing can occur due to transaction aborts [Care84, Agra87a]. Such *abort-induced thrashing* occurs at the point where admitting a new transaction becomes likely to lead to a deadlock, and results from the fact that a transaction that executes for some time and is later aborted leads to wasted resource capacity. Here, allowing a new transaction into the system ends up increasing the system overhead for permitting shared use of the resources and data managed by the DBMS.

As an example of thrashing due to data contention, Figure 1 presents some DBMS performance results for transactions using 2PL. These results were obtained through simulations, the details of which will be covered shortly. The underlying assumptions include a centralized system with one CPU and five disks, a 1000-page database, and transactions that (on the average) read eight pages and modify two of them. In the absence of a concurrency control mechanism, we see in Figure 1 that performance would increase and then level off as a function of the number of active transactions. This is due to initial improvements in resource utilization followed by resource saturation. However, the performance trend of the system when 2PL is used for concurrency control is an initial increase in throughput followed by a subsequent drop due to thrashing. To avoid such thrashing, we would like the system to cease admitting transactions once the peak throughput is reached, and it is tempting to suggest using a fixed multiprogramming level as a way to accomplish this. Unfortunately, the optimal multiprogramming level is workload-dependent, and the system workload may vary over time. Figure 2 illustrates this point by showing what happens when the optimal multiprogramming level (35) for the workload from Figure 1 is used for that workload and then for a workload where the average transaction size is four times larger. Clearly, a more adaptive solution is required.

In this paper, we present a solution to the thrashing problem for database systems that use locking-based concurrency control. The solution, known as the Half-and-Half load control algorithm, operates by dynamically monitoring the number of running and blocked transactions and using this state information to adjust the multiprogramming level of the system. The Half-and-Half algorithm is described in Section 2 of the paper. A simulation model of a DBMS is used to study the performance of the algorithm; the structure

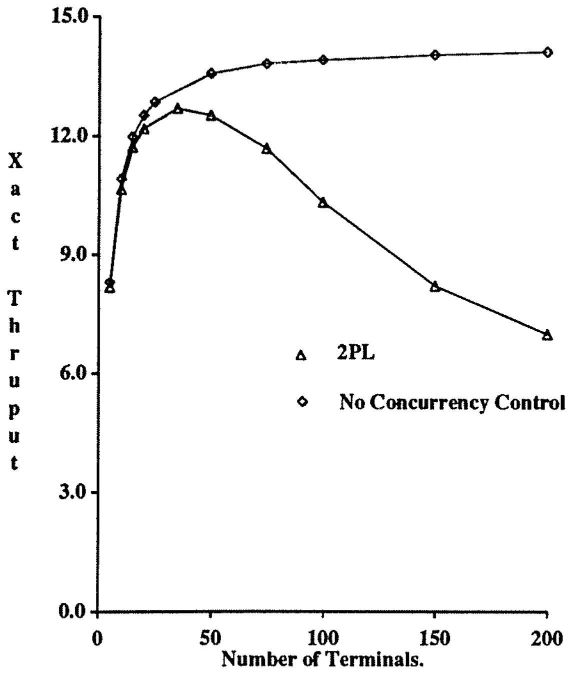

Figure 1: Example of Thrashing in 2PL.

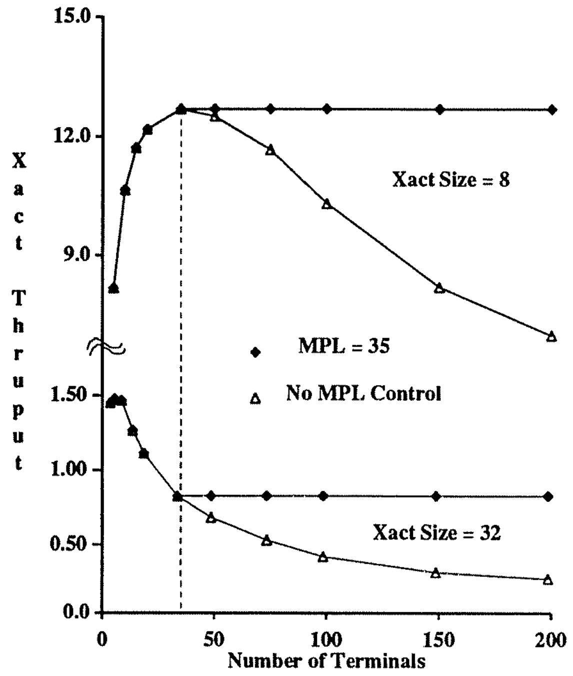

Figure 2: The Fixed MPL Approach.

and salient characteristics of this model are discussed in Section 3. Section 4 presents the results of a number of simulation experiments that test the Half-and-Half algorithm while varying such critical DBMS workload factors as the size of the database, the size of transactions, the transaction read/write ratio, and the transaction mix. Also presented in Section 4 are experiments that explore the robustness of the algorithm to changes in key assumptions about the problem and one experiment that briefly compares the Half-and-Half scheme with two other possible approaches. Finally, Section 5 summarizes the paper and discusses future work possibilities.

2. THE HALF-AND-HALF ALGORITHM

The Half-and-Half algorithm is based on the addition of a *load control* component to the standard software architecture of a DBMS. The job of this component is to control the set of active transactions — transactions that have been admitted into the system and are thus competing for DBMS resources — by dynamically monitoring the state of the DBMS. In the Half-and-Half algorithm, the load controller is integrated with the concurrency control algorithm. It is responsible for deciding when (and when not) to admit new transactions into the system based on its knowledge of the current system state; when a new transaction arrives, the load controller decides whether to admit the transaction immediately or to place it in an external ready queue for later admission. It is also given the authority to abort one or more active transactions as a corrective action if it perceives the system as being overloaded. The algorithm operates with no built-in assumptions about the system or the workload[2], so in some sense it has to experimentally determine the system state by observing how the state changes in response to its control actions.

In making its transaction admission and abort decisions, the goal of the Half-and-Half algorithm is to maximize the performance of the system (e.g., transaction throughput). In a DBMS with multiple physical resources, running more transactions leads to higher resource utilizations, so increasing the number of running transactions is desirable if the overhead per transaction does not increase as a result. Conversely, decreasing the number of running transactions is undesirable if it leads to resources becoming underutilized. What makes the load control problem non-trivial is the fact that, given 2PL's potential for thrashing, it is possible that admitting a new transaction will increase system overhead (through abort-induced thrashing) or actually cause the number of running transactions to decrease (due to wait-induced thrashing). Predicting the outcome of an admission decision can thus be rather difficult. Note also that one cannot judge the impact of a given admission decision right away, as it takes time for a new transaction to begin making lock requests (and thus to affect the state of the system).

Based on this line of reasoning, the Half-and-Half algorithm divides the DBMS state space into three mutually exclusive regions: *Underloaded*, where admitting new transactions is desirable; *Overloaded*, where too many transactions have been admitted; and *Comfortable*, where no load control action at all is warranted. If the algorithm detects that the system is in either of the first two states, it takes an appropriate corrective action (described shortly). In order to categorize the current operating region of the system, the following state classification is applied to each active transaction:

Running: An active transaction is said to be *running* if it is not currently waiting for a lock; otherwise it is said to be *blocked.*

Mature: An active transaction is said to be *mature* after it has completed 25% of its estimated number of lock requests.

[2]Actually, the Half-and-Half algorithm does expect each transaction to provide an estimate of the number of locks that it will request, but we will see later that the algorithm's performance is not very sensitive to the accuracy of this estimate.

State	Running	Mature
State 1	Yes	Yes
State 2	Yes	No
State 3	No	Yes
State 4	No	No

Table 1: Possible Transaction States.

The motivation for the *running* versus *blocked* classification should be clear. The motivation for the *mature* transaction notion is the observation that it takes time for a newly admitted transaction to have an impact on the system state. Based on these two classifications, each admitted transaction can be in one of the four states indicated in Table 1. The Half-and-Half algorithm keeps track of the current number of transactions in each state as well as the overall number of active transactions in the system. This information is used to identify the current operating region of the system.

Figure 3 shows, for the workload of Figure 1, how the number of mature, running (i.e., State 1) transactions varies with the number of terminals that are simultaneously attempting to run transactions. Also shown in Figure 3 is the number of other transactions (i.e., States 2-4). We focus on mature, running transactions here since these are the unquestionably "good" ones. Notice how the number of such transactions increases and then decreases, as we would expect. The initial increase occurs as more and more transactions become available to run, and the subsequent decrease happens when the level of data contention becomes so heavy that serious lock thrashing occurs. There is also something else to notice in Figure 3 — interestingly, the curves for the State 1 and States 2-4 transaction populations cross at 35 terminals, which is just where the performance peak appeared in Figure 1. This means that half of the transactions are in State 1 at the performance peak, with the other half being in States 2-4 at that point. When we first noticed this, we checked to see if it was generally true for other workloads and parameter combinations as well; we found that the crossover indeed tends to occur at approximately the maximum performance point in many cases, particularly when this point occurs at a relatively large (i.e., two digit) number of terminals. Figure 4 shows the same pair of curves for the larger transaction workload of Figure 2. While the crossover and maximum performance points don't coincide exactly in this case, they are still quite close. These empirical observations led us to define the *Underloaded* and *Overloaded* regions of operation based on the following *50% rule* (for which the Half-and-Half algorithm is named):

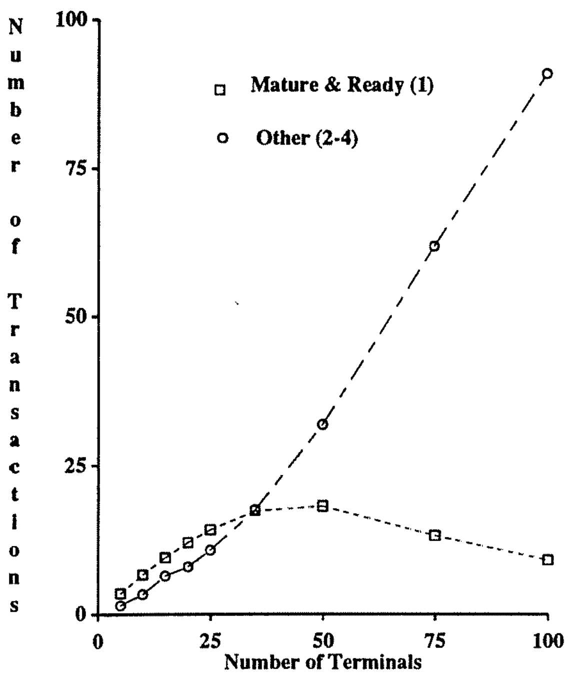

Figure 3: Transaction States (Size = 8).

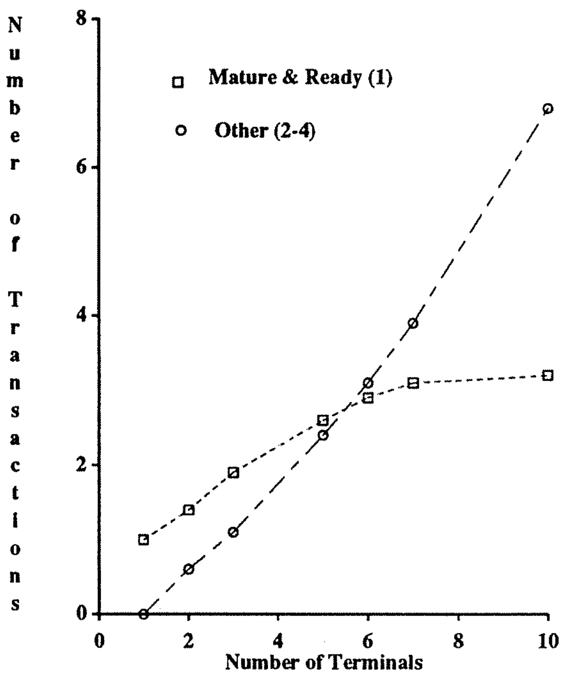

Figure 4: Transaction States (Size = 32).

$$Underloaded \equiv (\text{\# in State 1 / \# Active}) > (0.5 + \delta) \qquad (1)$$

$$Overloaded \equiv (\text{\# in State 3 / \# Active}) > (0.5 + \delta) \qquad (2)$$

These conditions can be interpreted as follows: Condition (1) of the 50% rule says that if more than about 50% of the transactions inside the system[3] are mature and running, then conditions are favorable for admitting one or more new transactions. Condition (2) says that if more than about 50% of the active transactions are mature but blocked, then one or more transactions should be aborted in order to reduce the level of data contention in the system. Note that, for safety, each condition considers only mature transactions. The goal of the Half-and-Half algorithm, which is invoked whenever a transaction arrives, makes a lock request, or commits, is to respond to these conditions in such a way as to keep the system in the *Comfortable* state if possible (or the *Underloaded* state if not). It does so by responding as follows to the system state transitions of interest:

Arrival: When a transaction arrives, if the current system state is *Underloaded* or a decision was previously made to admit the next arrival (see **Commit** section below), then the newly arrived transaction is admitted. Otherwise the transaction is made to wait in the external ready queue.

[3]The Half-and-Half algorithm uses a tolerance value δ in order to obtain added stability through hysteresis. We found δ=0.025 (i.e., a 5% overall tolerance window for conditions (1) and (2) together) to work well in practice.

Lock Request: If a lock request leads to blocking of the requester and the system is in the *Overloaded* region, then blocked transactions will be aborted until the system leaves this region of operation. Victims are chosen in increasing order of age, so the youngest[4] blocked transaction will be the first victim selected; also, only blocked transactions that are in turn blocking other transactions are considered as potential victims (since aborting these transactions will enable others to run). However, if the lock request is granted and *Underloaded* is the current region of operation, then new transactions will be admitted from the external ready queue until either the system leaves the *Underloaded* region or the ready queue is exhausted.

Commit: When a transaction commits, a new transaction is (unconditionally) admitted to replace it if one is available. Otherwise the algorithm decides to admit the next transaction that arrives and records this decision.

To summarize the main idea behind the 50% rule, it essentially says that it is fine for up to half of the active transactions to be blocked, but that the system should be prevented from exceeding this level of blocking.

3. PERFORMANCE MODEL

Before presenting the results of our experiments with the Half-and-Half algorithm, we first describe the simulation model that was used to obtain the results. Our simulator is based on the closed queuing model of a centralized (i.e., single site) database system shown in Figure 5. This model has been used in a number of earlier studies, including [Care83, Care84, Care86, Agra87a, Agra87b]. Transactions originate from a finite collection of terminals in the model. A transaction is said to be active if it is either receiving service or waiting for service inside the database system, and it is possible to control the number of active transactions in the system through the multiprogramming level limit or *mpl*. When a new transaction originates, if the system decides to postpone its admission for load control purposes, the new transaction will enter the external *ready queue* to wait for admission. Once admitted into the system, the transaction enters the *cc queue* (concurrency control queue) and makes the first of its concurrency control requests. If this request is granted, the transaction proceeds to the *page queue* and accesses its first page. When the next concurrency control request is required, the transaction re-enters the concurrency control queue and makes the request (and so on). Transaction reads and writes are interspersed, with each write request being immediately preceded by a read request for the same page (for modeling convenience as well as realism).

[4]Transactions are timestamped when they first arrive, and retain their timestamps even if aborted (to avoid starvation).

If the result of a concurrency control request is that the transaction must block, it enters the *blocked queue* until it is once again able to proceed. If a request leads to a decision to abort and restart the transaction, it goes to the back of the ready queue. It then begins making all of the *same* concurrency control requests and page accesses over again. Eventually the transaction may complete and the concurrency control algorithm may choose to commit the transaction. If the transaction is read-only, it is finished. If the transaction has written one or more pages during its execution, it first enters the *update queue* and writes its deferred updates into the database before leaving the system.[5]

To further illustrate the flow of transactions in the model, we now describe in more detail how 2PL and the Half-and-Half algorithm are modeled. Each concurrency control request corresponds to a lock request for a page, and lock requests alternate with page processing. Locks are all released together at end-of-transaction (after the deferred updates have been performed). Wait queues for locks and a waits-for graph are maintained in detail by a concurrency control specific portion of the simulator. The Half-and-Half algorithm is also modeled in detail; in addition to being integrated with the concurrency control portion of the simulator, it is given control over the transaction admission decision.

Underlying the logical model of Figure 5 are two physical resources, the CPU and the disk (I/O) resources. Associated with the concurrency control, page access, and deferred update services in Figure 5 are some use of one or both of these two resources. The amounts of CPU and I/O time per logical service are specified as model parameters. The physical queuing model is depicted in Figure 6, and Table 1 summarizes the associated model parameters. As shown, the physical model is a collection of terminals, CPU servers, and disks; model parameters specify the number of each that the system has. Requests in the queue for the pool of CPU servers are serviced FCFS (first-come, first-served), except that concurrency control requests get priority over other service requests. Our I/O system model is a probabilistic model of a database that is declustered across all of the disks. There is a queue associated with each disk; when a transaction needs service, it chooses a disk (at random, with all disks being equally likely) and waits in the queue associated with the selected disk. The service discipline for the disk queues in the model is also FCFS.

[5]Examples of recovery approaches that use deferred updates to minimize the cost of backing out aborted transactions include the Commercial INGRES recovery mechanism [Rowe86] and the database cache of Elhard and Bayer [Elha84].

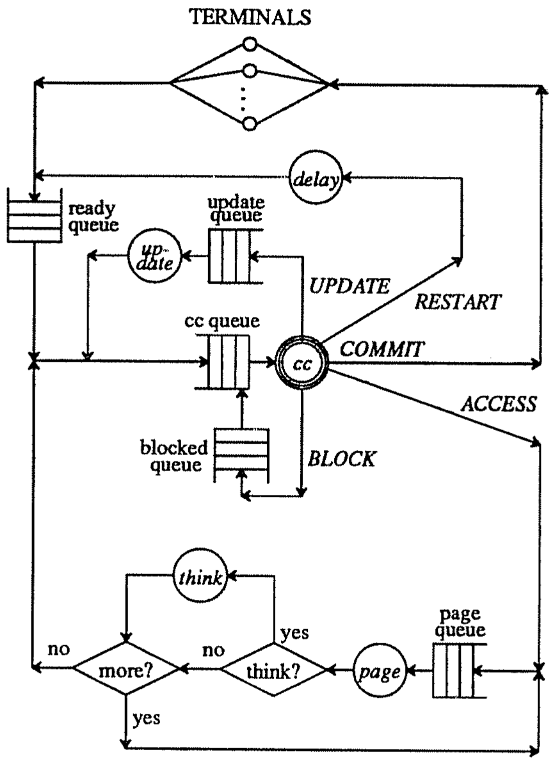

Figure 5: Logical Model of DBMS.

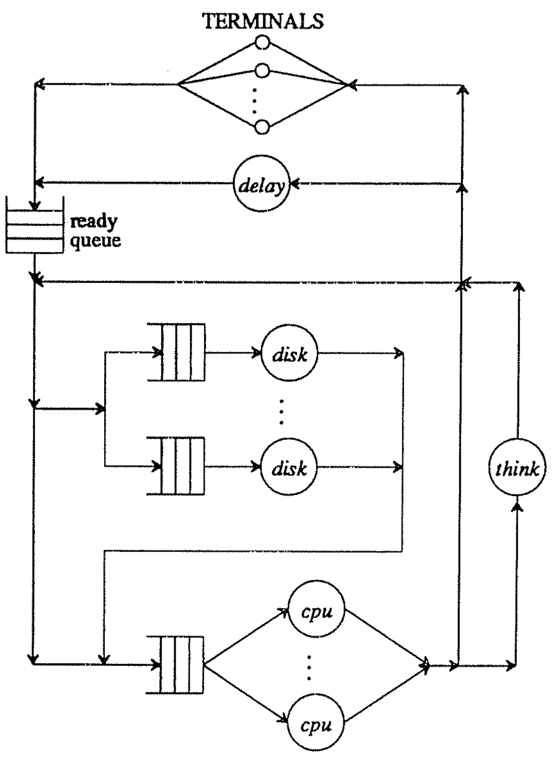

Figure 6: Physical Model of DBMS.

Parameter	Meaning
db_size	Number of pages in DB
tran_size	Mean transaction readset size
write_prob	Pr(page is written | page is read)
num_terms	Number of terminals
think_time	Mean thinking time
mpl	Multiprogramming level
page_io	Page I/O time
page_cpu	Page CPU time
num_cpus	Number of CPUs
num_disks	Number of disks

Table 1: Simulation Model Parameters.

The parameters *page_io* and *page_cpu* in Table 1 are the amounts of I/O and CPU time associated with reading or writing a page. Reading a page takes resources equal to *page_io* followed by *page_cpu*. Writing a page takes resources equal to *page_cpu* at the time of the write request and *page_io* at deferred update time, as it is assumed that transactions maintain deferred update lists in buffers in main memory. These parameters represent constant service time requirements rather than stochastic ones for simplicity. Costs for locking and deadlock detection operations are assumed to be included as a portion of the CPU cost for processing a page.

The size of the database is assumed to be *db_size* pages, and transactions are modeled according to the number of pages that they read and write. The average number of pages read by a transaction is uniformly distributed over the range *tran_size* ± ½ *tran_size*. These pages are randomly chosen (without replacement) from among all pages in the database. Finally, the probability that a page read by a transaction will also be written is given by the *write_prob* parameter.

4. EXPERIMENTS AND RESULTS

In this section, we present the results of a collection of experiments that were designed to test the Half-and-Half algorithm under a wide variety of operating conditions. Variations considered include the number of terminals submitting transactions, the size of transactions, the write probability of transactions, the size of the database, and the nature of the transaction mix (homogeneous or heterogeneous). We also present results from experiments that relax certain modeling assumptions or compare our approach with other alternatives. Before presenting any of the results, however, we briefly review the performance metrics and statistical methods employed in the experiments and describe the base parameter settings for the experiments.

4.1. Performance Methodology

The primary performance metric used in analyzing the experimental results is the *page throughput* of the system, defined as the number of pages processed (i.e., read or written) per second by committed transactions. We chose this metric because it is a workload-insensitive measure of the amount of useful work performed by the system. Like transaction throughput, a larger number indicates higher performance. Unlike transaction throughput, page throughput provides a single, meaningful performance number, even for multi-class workloads containing a variety of transaction sizes. The page throughput of the system is computed in our simulator by recording the number of page reads and page writes done by committed transactions and then dividing their sum by the total simulation time.

To ensure significance of the results, we employed a modified form of the batch means method [Sarg76] to statistically analyze the page throughput results.[6] Each simulation was run for 20 batches with a large batch time to produce sufficiently tight 90% confidence intervals. The actual batch time varied from experiment to experiment, but the confidence intervals were typically in the range of plus or minus a few percent of the mean value. We omit confidence interval details from our graphs for clarity, but we discuss only statistically significant performance differences when summarizing our results.

In addition to page throughput, we will examine the *raw page rate* of the system in analyzing the results. The raw page rate is the number of pages processed per second by *all* transactions (committed or otherwise). It is similar to page throughput in that it is measured in units of pages per second, but it differs in that it also includes the pages read or written by aborted transactions. The raw page rate will be used to provide insight into how hard the system is working, and the difference between the raw page rate and page throughput of the system will provide information on the amount of work wasted due to transaction aborts.

4.2. Base Parameter Settings

Table 2 gives the simulation parameter values that will serve as the base case values for our experiments. Most of our experiments are based on a database size of 1000 pages together with a mean transaction size of 8 pages and a write probability of 0.25; transactions thus read from 4 to 12 pages, updating an average of 25% of the pages that they read. These parameter settings were chosen in order to provide an interesting level of data contention, where our load control scheme can effectively be "stress tested." Throughout our experiments the think time used is zero, so the completion of one transaction leads to the initiation of another. We arranged things this way in order to keep the system under relatively high

Parameter	Value
db_size	1000 pages
tran_size	8 pages (4-12 pages)
write_prob	0.25
num_terms	200
think_time	0
page_io	35 milliseconds
page_cpu	5 milliseconds
num_cpus	1
num_disks	5

Table 2: Default Simulation Parameter Settings.

[6]More information on the details of the modified batch means method may be found in [Care83].

pressure at all times, varying the actual level of pressure by changing the number of terminals submitting transactions. In experiments where the number of terminals is held fixed, the number will be set at 200 terminals. We use 1 CPU with 5 disks as our model hardware configuration for the experiments, with page I/O and CPU times set in such a way as to be plausible while producing a somewhat I/O bound system (given the hardware configuration assumed).

4.3. Basic Performance Issues

The first set of experiments that we present investigate the effectiveness of the Half-and-Half algorithm for the base case and for varied transaction sizes, database sizes, and write probabilities. The base case is described fully by the parameters of Table 2, and it is the case for which results were first given in the Introduction. Figure 7 shows how the Half-and-Half load controller performs for these base case parameter values as the number of terminals (i.e., the number of transactions available to run) is increased from 5 to 200. As shown, the Half-and-Half algorithm is very effective at curtailing the thrashing that otherwise occurs for 2PL in this case. The algorithm successfully keeps the system operating at its peak performance level once the number of terminals exceeds the point where 2PL reaches its maximum page throughput.

In order to test the performance of the algorithm for a range of transaction sizes, we ran a number of simulations with different transaction sizes while holding the other parameters at their base case settings. Transaction sizes considered involved readsets ranging from 4 pages (which is very small for actual transactions) to 72 pages (which is rather large); the number of terminals submitting transactions was held fixed at 200. Figure 8 presents the results of these Half-and-Half simulation runs. In order to provide useful performance reference points, Figure 8 contains three other curves as well — the maximum page throughput for 2PL (determined by running a number of simulations to locate the optimal fixed MPL for each transaction size), the performance provided by 2PL with the multiprogramming level constrained to be 35 or less (which is optimal for the base case), and 2PL's performance with a multiprogramming level limit (MPL) of 20 (chosen simply as another example of a fixed MPL). As shown in the figure, the Half-and-Half algorithm works well over the entire range of transaction sizes, providing performance within a few percent of the optimal MPL performance for all but the very largest size considered.

Comparing the performance of the Half-and-Half algorithm with that of 2PL with a fixed MPL, we see that the Half-and-Half algorithm is significantly more robust. Each of the fixed MPL settings is (obviously) optimal for some point in the range; however, each one suffers when the average transaction size is significantly smaller or larger than the size for which it is the optimal MPL. Figure 9 shows the raw page rate for each case, including pages processed both by committed transactions and by aborted transactions. The figure shows that, as one might expect, the problem with the fixed MPLs for small transactions is that they admit too few transactions and thus fail to take full advantage of the available system resources; as a result, the overall amount of work accomplished by the system is lower in the fixed MPL cases. For large transactions, however, the fixed MPLs admit too many transactions and thus suffer due to thrashing. This is evident from the fact that the fixed MPLs do more work overall for large transactions (Figure 9), but have a lower page throughput (Figure 8) due to transactions being aborted to break deadlocks. Figure 10 shows how the average MPL maintained by the Half-and-Half algorithm varies with transaction size and how it compares to the optimal MPL for each size. As shown, the algorithm tends to be a bit too liberal, overshooting the optimal MPL, which is why it falls slightly below the optimal throughput line in Figure 8. This overshooting is due to the experimental nature of the Half-and-Half algorithm (since it essentially admits transactions and observes what happens as a result).

Figure 11 shows the results of a series of simulations where the size of the database is varied (while fixing the other parameters at their base case values). This variation is of interest because real databases can have "hot spots," and the performance impact of non-uniform data sharing on lock contention can be modeled as a reduction in the effective database size [Tay85]. Figure 11 again presents results for the Half-and-Half load control algorithm, two fixed MPLs, and the maximum achievable throughput for 2PL for each database size. We observe that the Half-and-Half algorithm does a good job of providing performance close to the optimal 2PL performance for each database size, and also that fixing the MPL at a single, predetermined value can lead to performance losses if that value is not chosen correctly. Both fixed MPLs perform poorly at small database sizes, as they admit too many transactions given the high level of data contention there. Conversely, the two fixed MPLs permit too few transactions to run simultaneously for the large database sizes, as one might expect based on our earlier results. We also performed a series of simulations that varied the write probability for transactions from that of the base case. We omit those

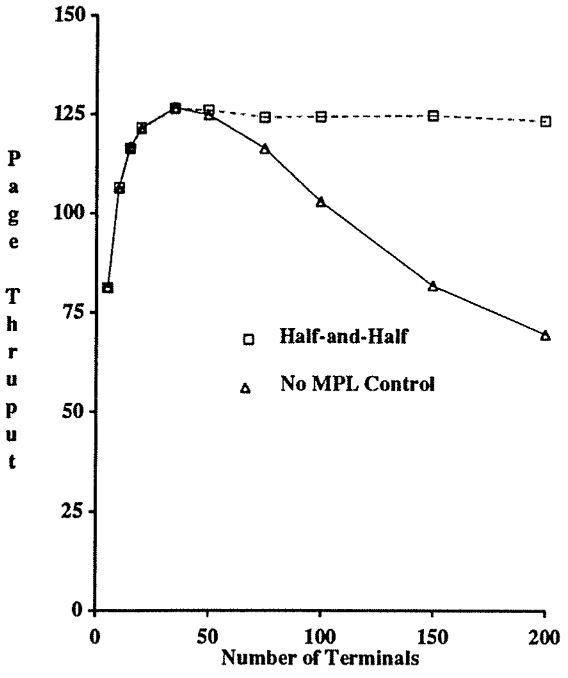

Figure 7: Page Thruput (Base Case).

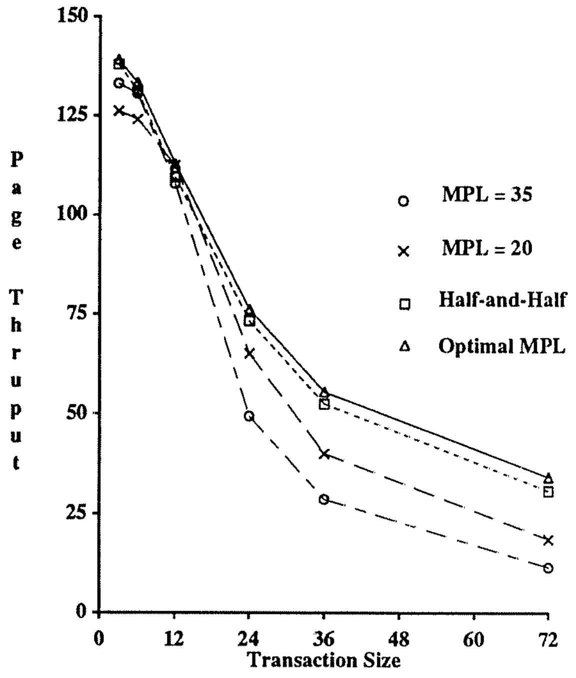

Figure 8: Page Thruput (Transaction Size).

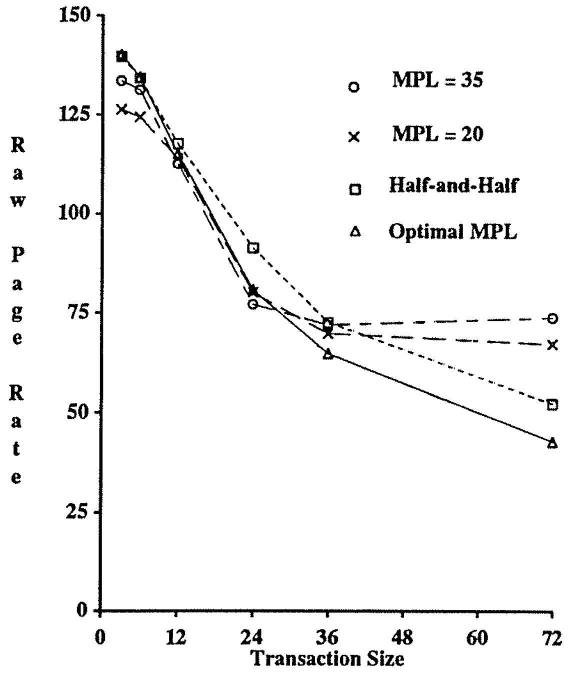

Figure 9: Raw Page Rate (Transaction Size).

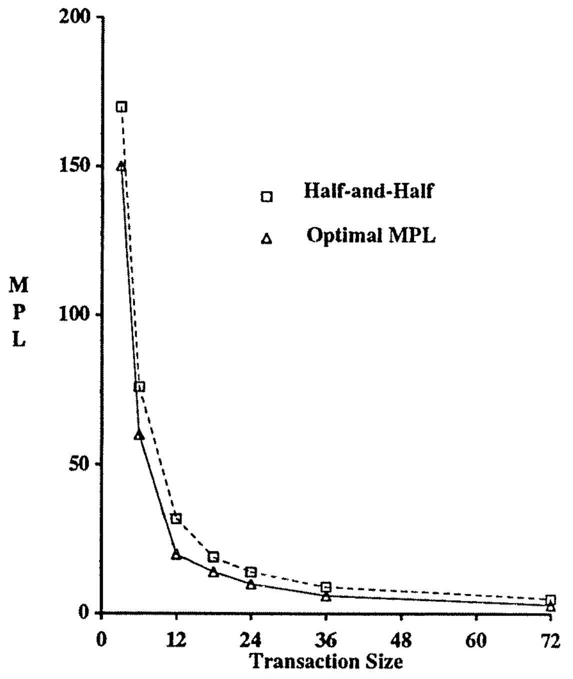

Figure 10: MPL Maintained (Transaction Size).

results here because they contain little additional insight; the Half-and-Half algorithm performed well over the entire range, while each fixed MPL was only optimal or near-optimal for a subset of the range.

4.4. Heterogeneous Workloads

The results presented in the previous subsection appear very favorable for the Half-and-Half algorithm. However, the experiments reported there only considered workloads where the readset and writeset sizes for all transactions were drawn from a common distribution. In this section we consider two more challenging workloads, one that mixes small update transactions and larger read-only transactions, and one where the average transaction size varies over time.

Figure 12 presents results for a mix of read-only and update transactions. The workload used in the simulations that produced this figure was made up of two classes of transactions. The first transaction class contains small update transactions, each reading an average of 4 pages and updating every page read. The other transaction class contains large read-only transactions that simply read an average of 24 pages each. 160 out of the 200 terminals were assigned to submit transactions from the small update transaction class, while the other 40 terminals were assigned to submit large read-only transactions. The resulting average

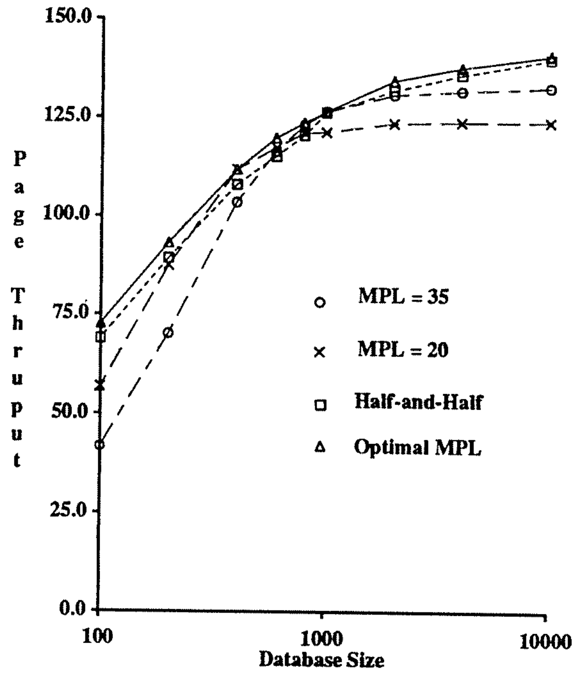

Figure 11: Page Thruput (Database Size).

transaction readset size is 8, as in the base case. The other simulation parameters were fixed at their base settings. Figure 12 shows how the page throughput varies for 2PL as the MPL is increased. Note that the general shape of the curve is similar to that of the base case, and that the Half-and-Half algorithm performs very close to the optimal MPL. (The Half-and-Half result is actually just a single value, produced at the MPL that it dynamically selected.)

In commercial database applications, large read-only transactions are often handled by having them use the *degree 2 lock protocol* in order to provide greater concurrency [Gray79, Moha89]. In this lock protocol, transactions lock each item before reading it, but they unlock the item before reading the next one (instead of holding all locks until transaction commit or abort time). Transactions that utilize this lock protocol see a committed but non-serializable view of the database. In the interest of reality, we repeated the previous heterogeneous workload experiment while having the read-only transactions in the mix run using degree 2 locking. Figure 13 presents the results that were obtained from these simulations. As one would expect, the performance curve for the system running without load control is less sharp and has a higher maximum page throughput since large read-only transactions now act like a series of short transactions. Again, though, thrashing occurs at the highest MPL settings, and we see that the Half-and-Half algorithm does a good job of operating the system near the optimal performance point.

One of the key arguments in favor of dynamic load control over a fixed MPL is that workloads tend change over time (e.g., the mid-morning workload of a DBMS is likely to be quite different from its middle-of-the-night workload). The next experiment that we describe was designed to explore the ability of the Half-and-Half algorithm to adapt to a simple dynamic workload. This experiment involved varying the size of new transactions over the course of each simulation run. In particular, the size of the transactions submitted by the simulator's terminals was varied by repeatedly alternating between the following two phases of operation:

(1) Choose the average transaction size *tran_size* randomly from the range between 4 and 72 pages (inclusive), and randomly choose a number of transactions (N_1) from the set {1000, 2000, 3000, 4000, 5000}. The next N_1 transactions generated by the terminals will have an average readset size of *tran_size*.

(2) Now fix the average readset size at 4 pages, and compute the number of transactions (N_2) of size 4 needed in order to produce an average transaction size of 8 pages when averaged of over phase (1) and phase (2). The next N_2 transactions will have an average readset size of 4 pages.

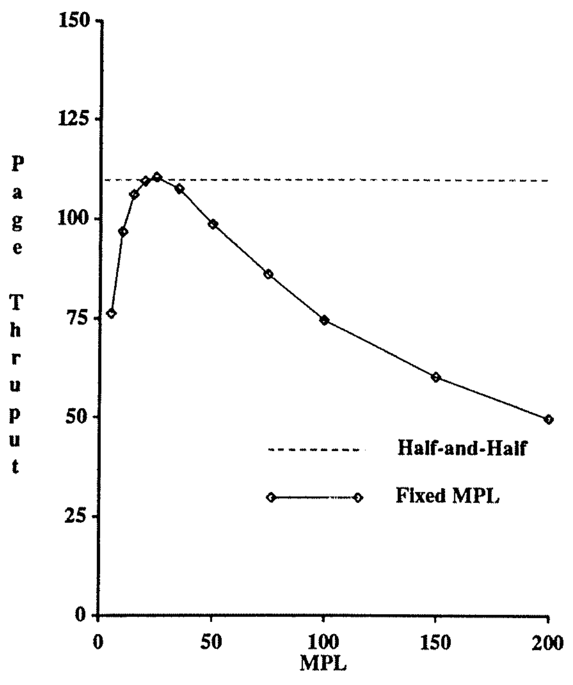

Figure 12: Page Thruput (Mixed Workload).

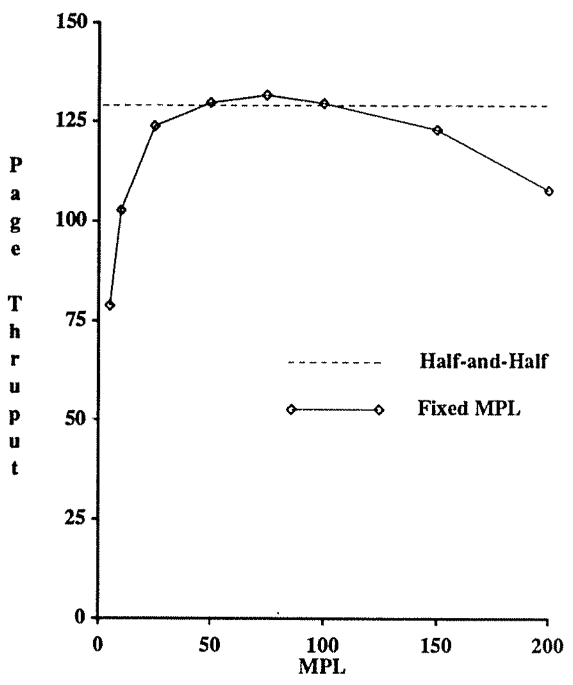

Figure 13: Page Thruput (Mixed Workload, Degree 2).

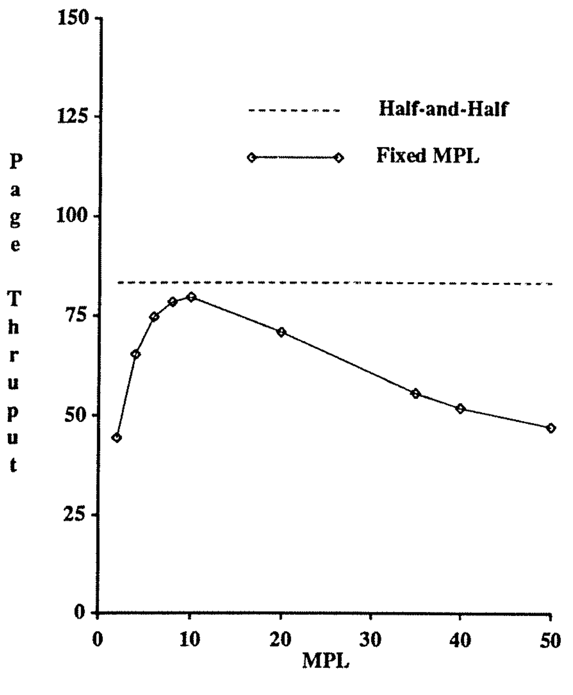

Figure 14: Page Thruput (Slow Workload Variation).

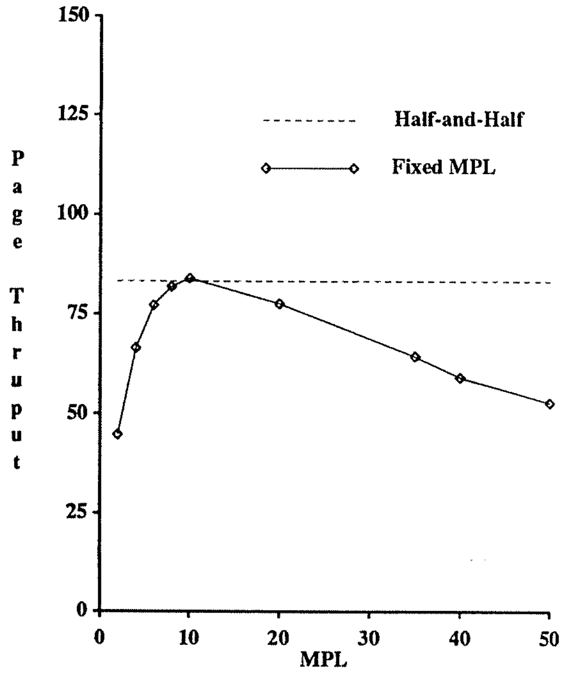

Figure 15: Page Thruput (Fast Workload Variation).

Simulations based on this approach to workload generation were run using 200 terminals for 2PL with a range of fixed MPLs and also for the Half-and-Half algorithm; all other parameters were set to their base case values. Figure 14 presents the results obtained from this experiment. As shown, in this case the Half-and-Half algorithm actually outperforms the best possible fixed MPL. This is because no fixed MPL is able to do a good job of handling both of the phases that the workload alternates between, while the Half-and-Half algorithm is capable of adapting itself to the current operating conditions during each phase. Figure 15 presents a similar set of results, but here the choices for choosing each phase (1) transaction count (N_1) is the set $\{200, 400, 600, 800, 1000\}$. As shown, the relative performance of the Half-and-Half algorithm for this more rapidly varying workload is closer to what we saw in the case of the two-class workload earlier in this subsection. This is to be expected, as the time-varying workload becomes closer in nature to a multi-class workload as the number of transactions in each phase shrinks.

4.5. Other Possible Approaches

While we are unaware of similar attempts to develop an adaptive load control mechanism for 2PL, a few related ideas have appeared in the concurrency control literature. One related idea is Tay's "rule of thumb" for locking [Tay85], which states that thrashing can be avoided by restricting the workload so that the quantity k^2N/D_e is less than 1.5. Tay's k term is the number of pages locked by each transaction, N is the multiprogramming level, and D_e is the effective database size. If the write probability for transactions is w, and transactions set either shared or exclusive locks on pages, then Tay computes the effective database size from the actual database size of D pages as $D_e = D/(1-(1-w)^2)$. If the average transaction size, write probability, and database size are each known, then Tay's rule of thumb can be used to determine a fixed MPL that will prevent thrashing.

Figures 16-17 show, for the transaction size study of Figures 8-10, how well the fixed MPL determined using Tay's rule controls the load of our simulated DBMS. Figure 16 presents the page throughput of the system under the Half-and-Half algorithm, under the MPL dictated by Tay's rule, and under the optimal MPL. As shown, the performance results in all three cases are pretty much the same for transaction sizes of 24 or less, while the Half-and-Half algorithm provides performance closer to that of the optimal MPL over the upper portion of the transaction size range. Figure 17 shows why this is the case, as Tay's rule of thumb appears to be overly conservative at the high end. For example, when the average transaction size is 72, the optimal MPL is 3 while Tay's rule yields an MPL of only 1; the Half-and-Half algorithm overadmits transactions here, yielding an average MPL of nearly 5. At the lower end of the transaction size range, both Tay's rule and the Half-and-Half algorithm are a bit too liberal in admitting transactions, but the resulting performance differences are negligible for this region of operation. The main problem with trying to use Tay's rule for load control is that it requires information about the average transaction size, the write probability for transactions, and the effective size of the database, all of which can vary over time; the Half-and-Half algorithm does not depend on this sort of detailed knowledge.

Another related idea from the concurrency control literature is the "bounded wait queue" algorithm proposed by Berard, Balter, and Decitre [Balt82]. They suggested the idea of preventing thrashing due to lock waits by placing a limit on the length of lock wait queues and aborting transactions whose lock requests would cause the wait queue for a lock to exceed this limit. Experiments with a very simple performance model led them to conclude that a limit of 1 was the best choice. We implemented their scheme[7] in the context of our simulation model, and Figures 18-19 present results from rerunning our base case experiment on their algorithm using wait queue limits of 1 and 2. Figure 18 presents the page throughput results. The limit of 1 performs worse than 2PL with no limit, and a limit of 2 performs just slightly better; neither provides performance comparable to that of the Half-and-Half algorithm. Figure 19 shows the total number of pages processed by committed and aborted transactions, which explains the results of Figure 18. In the limited waiting scheme with a limit of 1, many pages are processed by transactions that are aborted, i.e., resources are wasted due to abort-induced thrashing. This thrashing did not occur in [Balt82] because resource contention was ignored in their performance model (see [Agra87a, Agra87b] for more on this). A limit of 2 provides performance similar to that of 2PL with no limit because lock queue lengths greater than 2 have a low probability of forming anyway [Tay85].

[7]Their proposal only considered exclusive locks, and had to be generalized to handle both shared and exclusive locks. We generalized their "K or fewer waiters" limit to be "K or fewer compatible groups of waiters," where a compatible group of lock waiters is a group of transactions that are requesting the lock in compatible lock modes [Gray79]. For example, our generalization permits several shared lock requests to wait for an exclusively-locked page since all of the shared locks can be granted simultaneously when the exclusive lock is released.

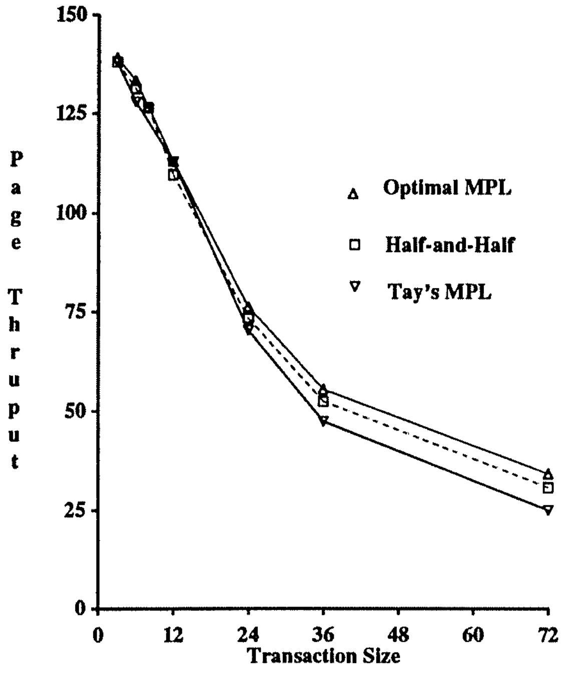

Figure 16: Page Thruput (Tay's Rule).

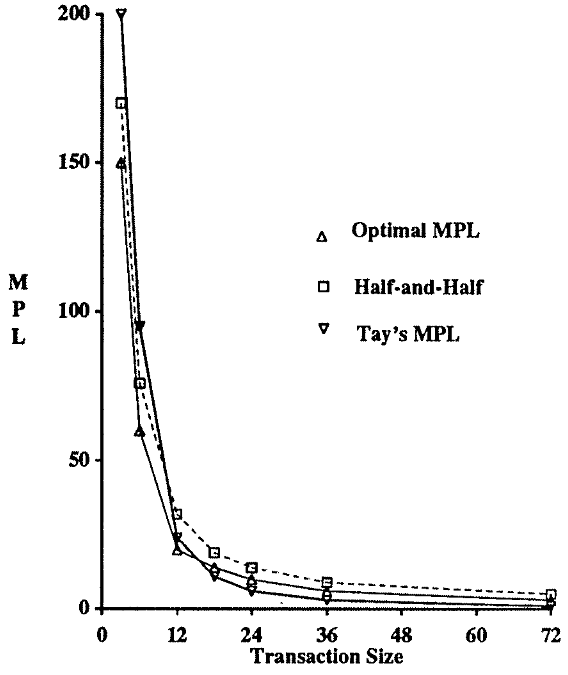

Figure 17: MPL Maintained (Tay's Rule).

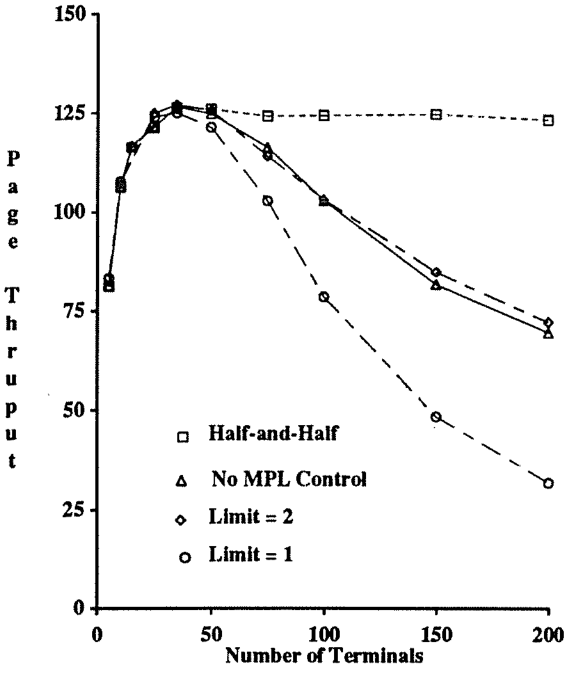

Figure 18: Page Thruput (Bounded Wait Queues).

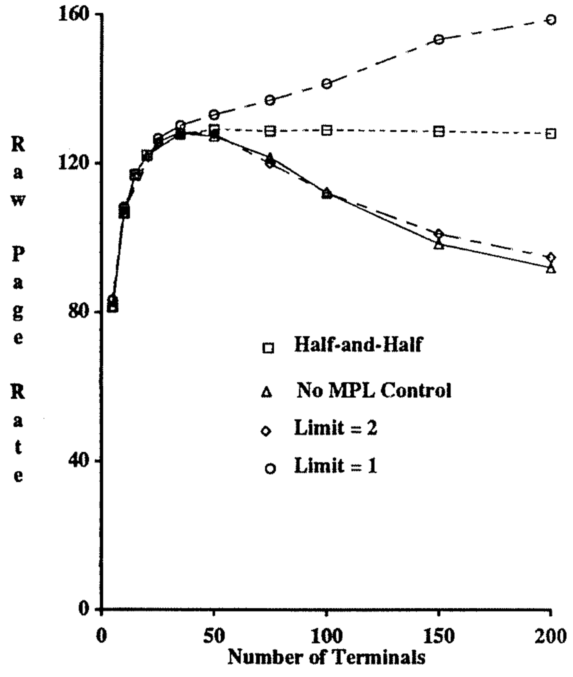

Figure 19: Raw Page Rate (Bounded Wait Queues).

4.6. Sensitivity to Assumptions

From the results presented here, it would appear that the Half-and-Half algorithm offers a very robust approach to load control for 2PL. We now examine the robustness of the algorithm by relaxing two assumptions that may be of concern to DBMS practitioners. First, our specification of the algorithm includes the notion of mature transactions, which are those transactions that have made more than 25% of their lock requests. Unless the workload consists of very simple (canned) transactions, coming up with accurate estimates of the actual sizes of transactions may prove difficult (even if transactions consist of relational queries and updates). Figure 20 shows the base case performance of the Half-and-Half algorithm when the maturity definition is varied from 10% to 50% of a transaction's lock requests. As shown, the algorithm is not particularly sensitive to this parameter, which indicates that it should be capable of tolerating significant estimation errors while providing good performance. Figure 21 considers the possibility of modifying the algorithm's maturity definition to be 25% of a transaction's locks or else X locks, whichever is fewer, and compares the performance of this modified algorithm with that of the basic algorithm and the optimal MPL (for the transaction size experiment of Figures 8-10). As shown, the modified algorithm works almost as well as the basic algorithm until X becomes less than about 15% of the average transaction size. This suggests that the maturity threshold does need to be related to the overall transaction length, but again that this relationship can be rather rough.

One final assumption that we consider here is the I/O model employed by our simulator. Until now, we have ignored buffering, assuming that every page requested by a transaction causes an I/O (which is of course unrealistic). To ensure that our results regarding the Half-and-Half algorithm's behavior are not dependent on this assumption, we added a simple LRU-based buffer manager model to our simulator. This buffer manager has a single parameter, *buf_size*, which is the number of pages in the buffer pool; it keeps a list of the *buf_size* most recently accessed pages, and a read request for a page only causes an I/O if the requested page is not on this list. Figure 22 presents results from rerunning our base experiment with a buffer pool size of 100 pages (10% of the database). The page throughput is higher here than in Figure 7, as expected, but otherwise the results are basically identical. Figure 23 presents similar results for a buffer pool size of 1000 pages (100% of the database). The page throughput is higher still, and again the Half-and-Half algorithm is effective. The only difference between Figure 23 and previous results is that the

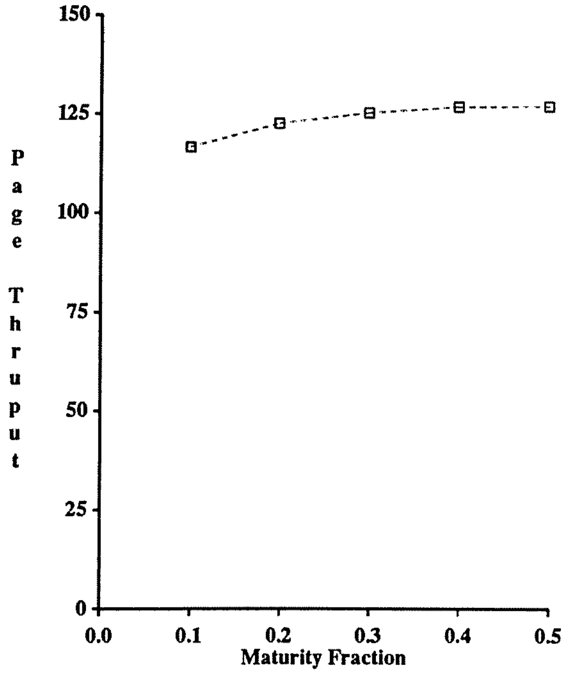

Figure 20: Page Thruput (Maturity Fraction).

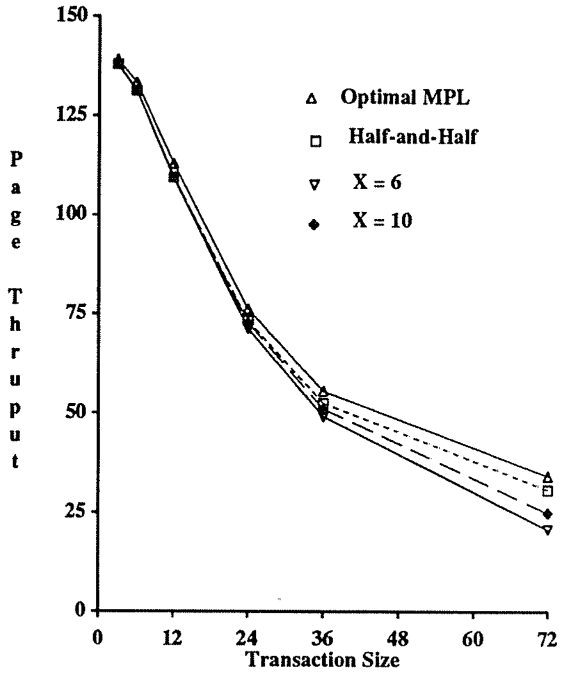

Figure 21: Page Thruput (Maturity Definition).

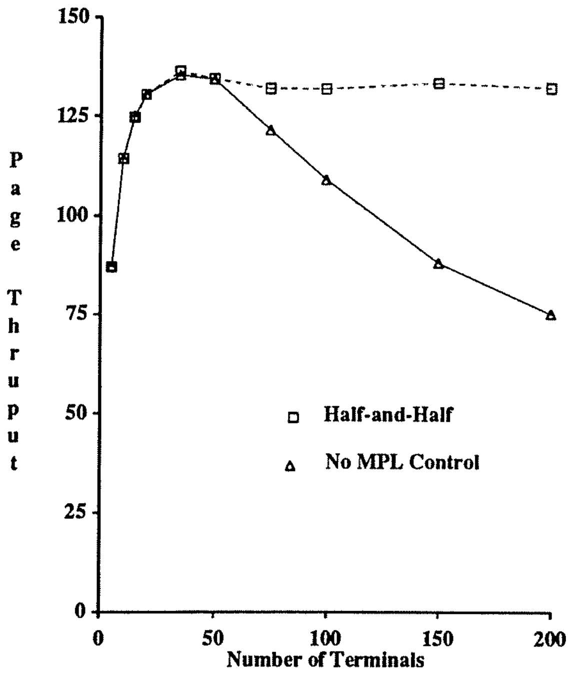

Figure 22: Page Thruput (Buffer Size = 100).

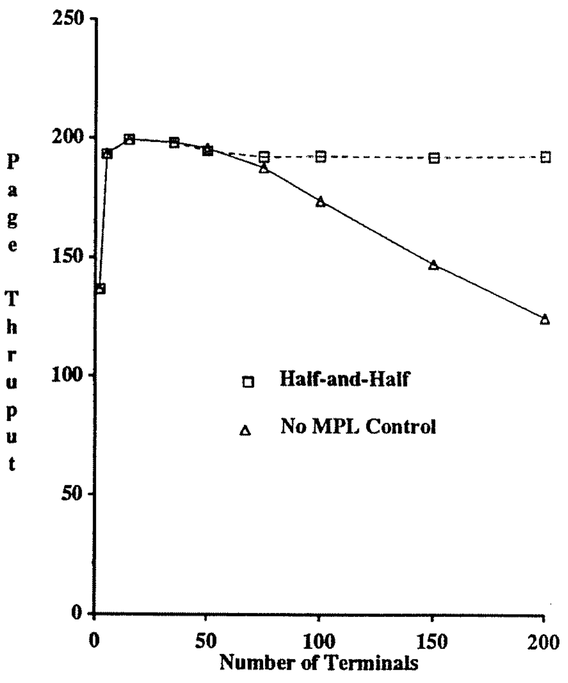

Figure 23: Page Thruput (Buffer Size = 1000).

Half-and-Half algorithm does just a bit worse here when the number of terminals is fairly large. This is due to the fact that here the system is CPU-bound, so the performance bottleneck is a single resource. Since a small number of transactions is sufficient to saturate a single resource, the Half-and-Half algorithm's tendency to overadmit transactions has a slightly more negative performance impact in this case.

5. CONCLUSIONS

In this paper, we have presented a scheme to eliminate thrashing for two-phase locking (2PL). Our scheme, the Half-and-Half load control algorithm, monitors the state of the DBMS and dynamically controls the multiprogramming level of the system. Roughly speaking, if fewer than 50% of the transactions in the system are blocked, the algorithm allows additional transactions to begin running; however, if more than 50% blocking occurs, it stops admitting new transactions and even aborts active transactions in order to correct what it perceives to be a system overload condition. Using simulation, we studied the performance of the Half-and-Half algorithm over a wide range of workloads and operating conditions. Results from this performance study indicate that the Half-and-Half algorithm is a promising solution to the thrashing problem for lock-based concurrency control algorithms.

One issue that we did not explicitly study here is how the Half-and-Half load controller should be integrated with load controllers for other DBMS resources (e.g., that of the buffer manager [Chou85, Sacc86]). This does not seem difficult, however; we conjecture that successful integration simply means asking each component for its opinion of the current workload, and ceasing to admit transactions when any of the components says "enough." The question of which component is likely to stop admitting transactions sooner — e.g., the lock manager or the buffer manager — seems likely to be application-dependent. However, with memory sizes increasing rapidly, buffer management seems less and less likely to be the limiting factor for a high-performance DBMS.

Several avenues remain for future work. First, the Half-and-Half algorithm shows no favoritism for one transaction class over another when faced with a multi-class workload, as it admits waiting transactions in their order of arrival. It might be interesting to consider extending the algorithm to somehow discriminate between transaction classes in order to provide still better performance for multi-class workloads.

Second, we have considered only the case of a single, centralized DBMS. The question of how to add load control to a distributed DBMS with decentralized control seems to be an interesting one, as load control deadlocks must be carefully prevented.

ACKNOWLEDGEMENTS

We would like to thank Bruce Lindsay, C. Mohan, and Hamid Pirahesh of IBM Almaden for raising several of the robustness issues considered in Section 4.

REFERENCES

[Agra87a] Agrawal, R. Carey, M., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. on Database Sys.* 12(4), Dec. 1987.

[Agra87b] Agrawal, R. Carey, M., and McVoy, L., "The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," *IEEE Trans. on Software Eng.* SE-13(12), Dec. 1987.

[Balt82] Balter, R., Berard, P., and Decitre, P., "Why Control of the Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management," *Proc. 1st ACM Symp. on Principles of Dist. Computing*, Aug. 1982.

[Bern87] Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Co., 1987.

[Care83] Carey, M., *Modeling and Evaluation of Database Concurrency Control Algorithms*, Ph.D. Thesis, Computer Science Division (EECS), Univ. of California, Berkeley, Sept. 1983.

[Care84] Carey, M., and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proc. 10th VLDB Conf.*, Singapore, Aug. 1984.

[Care86] Carey, M., and Muhanna, W., "The Performance of Multiversion Concurrency Control Algorithms," *ACM Trans. on Computer Sys.* 4(4), Nov. 1986.

[Chou85] Chou, H-T., and DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proc. 11th VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[Denn68] Denning, P., "Thrashing: Its Causes and Prevention," *AFIPS Conf. Proc., Vol. 33 (Fall Joint Computer Conf.)*, 1968.

[Elha84] Elhard, K., and Bayer, R., "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM Trans. on Database Sys.* 9(4), Dec. 1984.

[Fran85] Franaszek, P., and Robinson, J., "Limitations of Concurrency in Transaction Processing," *ACM Trans. on Database Sys.* 10(1), March 1985.

[Gray79] Gray, J., "Notes On Database Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[Moha89] Mohan, C., personal communication, August 1989.

[Pete86] Peterson, J., and Silberschatz, A., *Operating Systems Concepts*, Addison-Wesley, 1986.

[Rowe86] Rowe, L., and Stonebraker, M., "The Commercial INGRES Epilogue," Chapter 3 in *The INGRES Papers: Anatomy of a Relational Database System*, M. Stonebraker, ed., Addison-Wesley Publishing Co., 1986.

[Sacc86] Sacco, G., and Schkolnick, M., "Buffer Management in Relational Database Systems," *ACM Trans. on Database Sys.* 11(4), Dec. 1986.

[Sarg76] Sargent, R., "Statistical Analysis of Simulation Output Data," *Proc. 4th Annual Symp. on the Simulation of Computer Sys.*, August 1976.

[Tay85] Tay, Y., Goodman, N., and Suri, R., "Locking Performance in Centralized Databases," *ACM Trans. on Database Sys.* 10(4), Dec. 1985.