012

OPTIMIZING SHADOW RECOVERY ALGORITHMS

by

Jack Kent

Hector Garcia-Molina

TR-CS-012

October, 1985

OPTIMIZING SHADOW RECOVERY ALGORITHMS

Jack Kent

Hector Garcia-Molina

Department of Electrical Engineering and Computer Science

Princeton University

Princeton, N.J. 08544

ABSTRACT

Experiments conducted on a database testbed at Princeton indicate excessive page-table I/O is the major performance drawback of shadow recovery. In light of this, we propose a method for parametrizing shadow recovery that minimizes page-table I/O without sacrificing too much disk utilization. Using a simple model, we analyze and evaluate our mechanism, comparing it to two conventional ones.

September 20, 1985

OPTIMIZING SHADOW RECOVERY ALGORITHMS

Jack Kent

Hector Garcia-Molina

Department of Electrical Engineering and Computer Science

Princeton University

Princeton, N.J. 08544

1. Introduction

A *crash recovery algorithm* implements the software precautions a database management system (DBMS) or a file system must observe to guarantee that transactions (i.e. user requests) are executed atomically in the presense of failures. (Henceforth, we refer to the section of code responsible for this task as the crash recovery manager - *CRM*.) Different recovery algorithms have been proposed: among them logging [GRAY][ELH], differential files [SEV] and shadowing [LORIE][REUTa].

As one might expect, no one solution is always preferable; each has some particular drawback. For example, to perform reasonably, logging recovery requires a dedicated disk arm or tape unit, differential file recovery demands long idle periods (to merge differential file and database), and shadow recovery needs

enough memory to maintain a very large data-structure (the page-table) in core. It is not our intent in this paper to compare the three classes of recovery algorithms. Other work has been done in this area [DEW][REUTb], including a recent experimental study conducted at Princeton where various mechanisms were implemented and evaluated [KENTb].

In this paper we focus exclusively on shadow recovery and how to improve its performance. One such enhancement was suggested by our experimental work; we observed that the major performance drawback of shadow recovery is excessive page-table I/O. (The page-table maps the location of the database or file pages to blocks on disk.) Except for small databases, the page-table is too large for main-memory, and instead must be swapped out to disk.

Our new shadow recovery technique effectively "shrinks" the size of the page-table, while keeping the size of the database fixed. Unlike other schemes [REUTa], this is achieved **without** compromising disk utility. Moreover, our shadow recovery mechanism can be customized for a desired disk utilization and application. We believe that this mechanism, with its added flexibility and improved performance, extends the range of applications where shadowing is the preferred alternative for crash recovery.

Overview of the Paper

We organize the paper as follows: In section two, we review how shadow recovery works. We also present four metrics to evaluate different shadowing schemes and discuss how these metrics apply to two specific existing schemes. Section three follows with a description of our new shadow mechanism, CRM_{TRIM}

(for Tunable Recovery IMplementation). †

Sections four and five discuss the tradeoffs involved in tuning CRM_{TRIM}, and analyze a simple performance model that can be used to evaluate and customize the mechanism. In section six we extend the model, compare the efficiency of this new mechanism with conventional ones, and demonstrate how a few hypothetical applications might use the tuning procedure. Finally, in section seven, we present our conclusions.

2. A Review of Shadow Recovery Algorithms

A DBMS is comprised of many software components, e.g., access methods, high-level schema, etc. The crash recovery manager lies at the inner-most layer of the DBMS software.

CRM communicates with low-level software (a device driver) to read and write pages to and from disk. The ***physical database*** consists of Q such pages (called physical pages), B_0, B_1,..., B_Q.

Any application (be it a file system, a DBMS, or a mail server) desiring recoverable actions is layered on top of *CRM*. *CRM* provides these applications with a 'virtual disk', such that designated groups of reads and writes (called *transactions*) to this disk are atomic. The virtual disk is broken down into N ($N \leq Q$) ***logical pages***, L_0, L_1, ...L_N. These N pages comprise the ***logical database***. (Henceforth, we use the terms database and *logical database* interchangeably.)

† In nautical lingo, 'trimming' and 'tuning' are synonymous, e.g. a yachtsman trims his sails to optimize for wind speed and angle.

2.1 The Shadowing Data Structures

The *page table* (PT) is a linear structure consisting of N entries, p_1, p_2, ..., p_N. The *ith* entry of this structure (p_i), is the number of the physical page holding L_i, called the *global* version of L_i. At all times, the page-table defines a consistent (transaction-wise) logical database.

PT must survive a system crash and for this reason it is maintained in non-volatile storage, usually disk. On disk, the page-table is broken down into units of *mapping pages* ($M_0, M_1, \cdots M_m$); each mapping page fits in a physical page. (We will say M_i cover L_j, if M_i contains the number of the physical page holding L_j.) To access a mapping page, the system must read it into a memory resident *PT cache*. To update a mapping page, it must first be read in, modified in-core, and then flushed back to the PT on disk.

At any time, a physical page in the physical database is either free or busy. A physical page is *busy* if it:

(1) holds the committed version of a logical page

or

(2) holds the local version of a logical page that is being (or has been) updated by an uncommitted transaction.

Otherwise the physical page is *free*. CRM_{SHADOW} uses a bitmap to record the availability of physical database physical pages. The bitmap need not be maintained in non-volatile memory; it can be reconstructed after a crash by consulting the page-table.

2.2 Updates with Shadow Recovery

In figure 2-1, we see a portion of the logical and physical database. M_0 maps the first three logical pages, M_1 maps the second three logical pages, etc.

Now, assume a transaction T wants to modify logical pages L_0 and L_1. To do this, CRM_{SHADOW} reserves the once free physical pages B_5 and B_7, as T's local version of L_0 and L_1 and then copies B_1 and B_2 to B_5 and B_7 (refer to figure 3-3b). For the duration of T, any modifications made to L_0 and L_1 are written to B_1 and B_3.

Handling transaction failures is easy with shadows. To recover the database from a failed transaction T, CRM_{SHADOW} simply releases T's local versions. Remember, though T might have changed the physical database, it has not affected the logical database.

CRM_{SHADOW} commits T as follows. First, it guarantees that T's two local versions have been flushed from the cache to the database. Then, CRM_{SHADOW} modifies M_0 accordingly and flushes M_0 back to the page-table (refer to figure 2-1c). In this case, the process is easy since a single mapping page covers both logical pages that T modified.

Unfortunately, if the changes instigated by T span multiple mapping pages the process is non-trivial. The reason is that PT must be updated atomically. Nevertheless, we can still solve the problem by "recursively" treating PT as a database and maintaining a higher level page-table for it. Thus, we have a tree of page-tables, each level being much smaller than the previous one. At the highest level is a table that resides on a single disk block and can be updated atomically

[LAMP]. As another option, the page-table can be atomically modified using logging [DEW][GRAY].

2.3 Evaluating Different Shadowing Schemes

We believe a shadow recovery scheme should be evaluated on the basis of four (interdependent) metrics: ***Disk utility, memory cost, scrambling cost, and I/O cost.***

Disk Utility is the ratio of the ***logical database*** size to the ***physical database*** size. Computing disk utility is trivial, e.g. if a database is configured with N logical pages and Q physical pages, then database disk utility is $\frac{N}{Q}$. For this metric, we ignore the page-table disk space, since it is many orders magnitude smaller than the database.

Memory Cost accounts for the extra memory that CRM_{SHADOW} reserves for data-structures, most significantly (by far) the ***page–table.*** We measure this cost in bits.

Scrambling Cost accounts for the decrease in database disk throughput resulting from CRM_{SHADOW}'s physical scrambling of the logical database. Think of scrambling cost as any extra movement of the database disk arm or additional rotations of the database disk that were caused by not updating in-place.

I/O Cost accounts for the additional I/O burden imposed by the shadow recovery mechanism. This cost has two components. The first one is page-table I/O, and can be measured by the number of mapping pages that must be read into memory or written to disk per transaction executed. The second component

represents the overhead of storing the page-table itself reliably. If PT updates are logged, it is the cost of writing to the log. If PT is accessed via a tree, it is the cost of reading and writing the non-leaf pages of the tree.

We will focus exclusively on the first component of I/O cost for the following reasons:

(1) In most applications, the first component dominates the second. The page-table is much smaller than the database, so managing it is cheaper. If a tree is used, usually two levels are sufficient, and a significant fraction of the non-leaf nodes can fit in memory. This reduces the number of disk reads further. When updating the higher levels of the tree, it is likely that concurrent transactions will modify the same page (this is especially true of the root) and the I/O cost is amortized. Similarly, if logging is used, many transactions can write their page-table modifications in a single write to the page-table log.

(2) The second component depends on the particulars of the shadowing scheme (e.g. tree of logging), so it is difficult to study in general. However, if it were necessary to study these costs, they could be examined independently of the first component. For example, if a tree were used, the I/O cost of managing the non-leaf levels could be computed in the same way we will compute page-table (i.e. leaf level) I/O cost.

In the next two sections, we will briefly summarize the page-table strategy of two shadow recovery methods and will discuss how they fare with respect to the above metrics. One such mechanism is called *TWIST*† [REUTa]; the other has

† We are actually describing a variant of TWIST.

been proposed by Lorie [LORIE]. We will refer to them as CRM_{TWIST} and CRM_{LORIE}.

2.4 TWIST Shadow Recovery

CRM_{TWIST} allocates two contiguous physical pages for every logical page. For all i, L_i will (always) map to either B_{2*i} or B_{2*i+1}, so p_i (the mapping for L_i) will require only a single bit.

Since the TWIST page-table is very small, it quite possibly could fit in a very small page-table cache without swapping. Thus, both TWIST I/O cost and memory cost should be small.

CRM_{TWIST} maintains continuity between the logical and physical database. Adjacent logical pages are separated by at most two physical pages. Thus, TWIST scrambling cost should be negligible.

Of course, CRM_{TWIST} does have its drawbacks, it wastes disk space. At any time, no more than half the database contains 'real' information; disk utility is a low $\frac{1}{2}$. And this may be unacceptable for many applications where the data is relatively static.

Lorie's mechanism increases disk utility at the expense of more page-table I/O and a slightly more muddled logical to physical mapping.

2.5 LORIE'S Shadow Recovery

Under CRM_{LORIE}, shadow disk locality is maintained on disk in units of *logical clusters*. A logical cluster associates a region of localized reference in the

logical database, with localized access in the physical database. For example, a logical cluster might hold a number of consecutive logical pages that upper level software regards as a file or a relation. This cluster, in turn, might reside on adjacent cylinders, to ensure that successive references within a file can be satisfied quickly. We assume C such clusters are allocated across the physical database. Each cluster will house L logical pages, and $L + F$ physical pages where $(L)C = N$ and $(L + F)C = Q$.

During an update (of L_i) CRM_{LORIE} tries to shadow L_i on its logical cluster. Failing this, the nearest logical cluster with a free physical page is used. Thus, a logical page can be shadowed to any physical page and so each page-table mapping for L_i (p_i) requires $\lceil \log_2 Q \rceil$ bits.

How does Lorie's shadow scheme compare to TWIST? First, note the increase in disk utility. So long as $F < L$, then $\frac{N}{Q} > \frac{1}{2}$. ‡

CRM_{LORIE} scrambling cost should also be small as information that will be referenced together is kept in a region of fast access on the disk.

What about I/O cost? This could be a problem, as the page table requires $N \lceil \log_2 Q \rceil$ bits. And unless large amount of memory are allocated to the PT cache (which will increase memory cost), this may cause an I/O operation for many of the logical to physical mappings.

In the next section, we describe an alternative way to maintain the page-

‡ Note that, by allocating fewer free pages than TWIST, fewer 'simultaneous' updates can be tolerated. However, in most real applications, only a fraction of the database will be updated at any given time.

table. This new method (CRM_{TRIM}) will provide I/O cost comparable to TWIST and disk utility comparable to LORIE.

3. Our Shadow Recovery Scheme

As in Lorie's strategy, we divide the logical database into clusters, each having L logical pages and $L+F$ physical pages. We refer to the logical cluster housing L_i as $CLUSTER(i)$.

The main problem with Lorie's page allocation scheme is that, to better allow sharing of free pages, each PT entry p_i is large. We will address this problem by maintaining two page-tables, S and P. In P, each entry p_i is small and can only map a logical page to a small subset of physical pages. If the logical page is not found in this subset, then a secondary page table S, with full size entries s_i, can map the logical page to any location. Intuitively, P serves as a filter and (ideally) maps the majority of the logical page requests. As we will later see, in order to properly "tune" this mechanism, we must optimize the size of the filter (i.e., the size of a p_i entry) to minimize page table I/O.

Specifically, each s_i entry contains $b_2 = \lceil \log_2 R \rceil$ bits, enough to map any logical page L_i to any physical page. Each P entry consists of b_1 bits where $b_1 < b_2$. And so, each p_i can take on values from 0 to $2^{b_1} - 1$. Associated with P is a function ϕ that maps the value p_i (for page L_i) onto a $CLUSTER(i)$ page. In the case where $p_i = 2^{b_1} - 1 = CODE$, ϕ is undefined and s_i contains the address of L_i.

Two main memory caches are maintained by CRM_{TRIM}, the primary cache

for P and the secondary cache for S. Mapping pages are swapped exclusively from their respective caches. Although page-table P is smaller than S, the primary cache should dwarf the secondary cache since P (by itself) will map the majority of the logical page requests. (It would also be possible to treat P and S as two parts of a single table with a single cache. We do not discuss this option here.) Let's see how CRM_{TRIM} recovery works via a simple example.

Example 3-1

Assume that we are working on cluster 0; henceforth we refer to this as **C**. Also, assume that **C** holds 8 physical pages and 6 logical pages, and that $b_1 = 2$ (so $CODE = 3$). Of course, this is only an example. In reality, we would expect clusters to be much larger. Finally, we define $\phi(i,p_i) = i + p_i \ mod \ 8$.

At system initiation, the database is configured so that logical page i (L_i) resides on page i (B_i) of **C**. (refer to figure 3-1) We see this by looking at page-table P. Since none of the $p_i = CODE$, all of the logical pages on **C** are addressed using P. In this example, CRM_{TRIM} forms the physical page address of L_i by adding p_i to i.

Now, assume that a transaction T wishes to update L_0 and L_6. First, CRM_{TRIM} must find the physical locations of these pages. So the P mapping page that covers L_0 and L_6 is read into the page-table cache (if it is not already there). Then, CRM_{TRIM} must acquire free pages (shadows) for these pages.

Checking L_0 first, CRM_{TRIM} examines the (availability of the) three pages that can be addressed from p_0; they are B_0,B_1 and B_2 (obtained by setting

$p_0 = 0,1$ *or* 2 respectively). Block 0 holds the 'old' copy of L_0, so it cannot be used for a shadow. Blocks 2 and 3 contain L_1 and L_2; they also cannot be used. † So, finding no primary mapped shadow, CRM_{TRIM} searches for the first free page it can find on **C**; in this case it's B_7. The L_0 information at B_0 is then copied to B_7 and this page functions as L_0's shadow.

Next, CRM_{TRIM} must shadow L_6. After examining B_6 and B_7 (two of the pages that can be addressed from p_6) and finding both are busy, CRM_{TRIM} finds B_8 is free and copies B_6 to B_8.

Before committing T, CRM_{TRIM} must record the new locations of L_0 and L_8. Recording the latter's new location requires only a simple modification; p_8 must be changed from 0 to 2 (6+2=8). Changing L_0's locations requires two modifications. First, CRM_{TRIM} sets p_0 to *CODE*, and then s_0 to 7. Next, the modified P and S mapping pages are flushed to disk. † T is now committed (refer to figure 3-2). ○

The above example is only one specific implementation of CRM_{TRIM}. Below, we give a general description of this recovery scheme. Let a be a logical page number on a cluster **C** ($0 \leq a \leq L$), b be the value of a non-CODE entry in P ($0 \leq b \leq \omega = 2^{b_1} - 2$), and c be the number of a physical page on **C** ($0 \leq c \leq L+F$). CRM_{TRIM} uses a special function ϕ, combined with page-table P to map logical pages to physical pages, $\phi(a,b) = c$. We will say that physical page c

† We assume a bit-map is maintained in core for this purpose. See section 2.1.
† Remember, there are different ways to guarantee that the page-table is atomically flushed.

primary maps logical page a, if $\exists\ b$ such that $\phi(a,b) = c$. And we refer to the primary mapping set, *PMS*, of L_a as $\{\ c: \exists$ b such that $\phi(a,b) = c\ \}$.

CRM_{TRIM} uses P (in conjunction with ϕ), and S to map any logical page L_a to a physical page. The algorithm is described below.

If $p_a = CODE$

Physical location of L_a is s_a

Else

Physical location of L_a is $\phi(a,p_a)$

During a mapping, it's possible that p_a will not be in-core (in the primary cache), and this will cause a page-table I/O; CRM_{TRIM} must bring in the P mapping page covering a. We say CRM_{TRIM} *primary faults* on a. An additional I/O may be necessary if P does not contain the location of the physical page housing L_a (i.e., $p_a = CODE$). In this case, we say CRM_{TRIM} *secondary faults* on a.

The above fragment adequately describes how reads are performed, but how does an update of L_a proceed? As in a read (of L_a), CRM_{TRIM} first finds the physical location of L_a (using the above code). Then, by probing the free-page bit map, CRM_{TRIM} checks if there is a **free** physical page available (call it c_1) that primary maps L_a. If there is, c_1 is used to shadow L_a, and we say L_a is *primary shadowed*. If no primary mapping of L_a is free, CRM_{TRIM} searches for another free page on-cluster, and failing this, a free page on a nearby cluster. Either way, we say L_a is *secondary shadowed*. Note that if logical page L_a goes from a primary mapping to a secondary mapping after an update, then both p_a

and s_a must be modified. As in Lorie's mechanism, a logical page that is stored off-cluster will remain there until the next time it is updated, at which time another effort will be made to primary map it, or at least move it on cluster.

How should ϕ be constructed? First, it should be a simple function (i.e. easy to compute). And also, ϕ should satisfy the below conditions.

(1) If $b_1 \neq b_2$, then $\phi(a,b_1) \neq \phi(a,b_2)$.

(2) *Let* $SIZE(c) = |\{\ a \text{ where } \exists b \text{ such that } \phi(a,b) = c\ \}|$.

$|SIZE(c_i) - SIZE(c_j)| \leq 1$ for all physical pages c_i, c_j $(i \neq j)$ on a cluster.

Intuitively, condition (1) says that ϕ shouldn't waste a coding by allowing two ways to map a given logical page to a given physical page. And condition (2) says that ϕ should distribute the physical pages in a cluster evenly among the logical pages as candidates for primary mappings. The motivation behind (2) is analogous to the construction of a good hashing function. †

4. ANALYZING AND TUNING CRM_{TRIM}

In the remainder of this paper, we will develop and analyze a performance model of CRM_{TRIM}. Our goal is to quantify the advantages of CRM_{TRIM}, while at the same time providing tools for tuning the algorithm to a specific application.

4.1 A General Description of the Model

† Condition (2) presumes uniform access across a cluster.

The DBA specifies a CRM_{TRIM} configuration by adjusting the following main system parameters: N - the number of logical pages in the database, F - the number of free pages in a cluster, L - the number of logical pages in a cluster, b_1 - the size of a P mapping, and M - the size of the primary cache (specified in bits). Of course, there are trade-offs involved in tweaking each parameter.

For example, by increasing F (or decreasing L), keeping all other parameters fixed, more logical pages become primary mapped. Thus, there are fewer secondary faults and so I/O cost decreases. On the minus side, increasing F decreases disk utility.

Likewise, increasing the size of the primary cache (M) allows more of P to fit in core (resulting in fewer primary faults) and I/O cost again decreases. Of course, performance picks up at the expense of memory cost.

Increasing b_1 affects only I/O cost, but it's unclear in which direction. For example, by increasing b_1, P grows, so a smaller percentage of P fits in-core and there will be more primary faults. On the other hand, as b_1 increases, more logical pages become primary mapped and so there will be fewer secondary faults.

Our objective in the next section will be to develop a model that quantifies the aforementioned tradeoffs. Using this model, the DBA can optimize b_1 (i.e. minimize page-table I/O) for a given L,F,N,M configuration. Or he can assess the reductions in I/O cost afforded by extra memory or additional free pages.

4.2 Assumptions Relating to the Model

Implicit in our model, are several assumptions about the CRM_{TRIM} environ-

ment

that serve to simplify the analysis. These assumptions are listed below:

(A1) We assume that the probability two transactions are working within the same cluster simultaneously is small. (In fact, if transactions lock large granules, this assumption is incidentally satisfied.) This means that if a transaction T is working on cluster **C**, T will be the only transaction vying for free pages (for use as shadows) on **C**. (Note that the effect of concurrent transactions within a cluster can be approximated by a single transaction accessing more pages.)

(A2) We assume that when a transaction starts accessing cluster **C**, the cluster will have L busy physical pages and F free ones. In reality, a cluster may have fewer free pages (if a foreign logical page was shadowed here) or more (if a cluster page was shadowed off-cluster). However, it is reasonable to assume there will be F free pages, since F is the average or expected value, and deviations should be small.

(A3) We assume that ϕ is chosen randomly from all possible ϕ's satisfying condition (1). Since condition (2) is relaxed, our analysis will estimate the worst case performance of CRM_{TRIM}.

(A4) We assume the secondary cache is very small compared to the size of the logical database. That is, on a transaction's first secondary fault to a given cluster, it will never find the needed mapping page in the secondary cache.

5. Analyzing a Simple Model

In this section, we describe how to 'tune' CRM_{TRIM}, by analyzing a simple model. This presentation should also motivate the need for a more general model, which we later develop in section 6.

We start by assuming the parameters L (the number of logical pages in a cluster), F (the number of free physical pages in a cluster), N (the number of logical pages in the database) and M (the size of the primary cache) are all given. Further, we assume that all transactions read and update a *single* logical page within a cluster. We will compute the average amount of page-table I/O generated per transaction. Or, more formally, let A be a random variable that represents the amount of page-table I/O performed on behalf of a transaction. We will compute $\overline{A}$.

5.1 Computing the Page-Table I/O Cost per Transaction

Consider an arbitrary update transaction T about to access logical page L_i on cluster **C**. For now, let's assume **C** is configured such that V of its logical pages are primary mapped, and $L - V$ are secondary mapped. (We will compute V later)

We will compute the following cost components of $\overline{A}$ with respect to T. Each component corresponds to the probability CRM_{TRIM} performs a specific type of page-table I/O.

(1) R_P^T - the probability transaction T reads a primary mapping page.

(2) W_P^T - ' ' writes ' '

(3) R_S^T - the probability transaction T reads a secondary mapping page.

(4) W_S^T - ' ' writes ' '

First, let's compute R_P^T. This is simply the probability that T primary faults.

$$R_P^T = 1 - \frac{M}{b_1 N} \quad \text{for } M < b_1 N$$

$$= 0 \qquad \textit{otherwise}$$

Computing W_P^T is slightly more difficult. To do so, we must consider all those scenarios where p_i is modified (and thus the mapping page covering L_i is updated).

Clearly, if L_i is primary shadowed, then p_i is modified. Also, if L_i is primary mapped and secondary shadowed, then p_i is changed to *CODE*. On the other hand, if L_i is secondary mapped and secondary shadowed, p_i doesn't change; $p_i = CODE$ before and after T commits. Thus, the probability W_P^T that CRM_{TRIM} updates a primary mapping page on behalf of T is the probability that L_i is **not both** secondary mapped **and** secondary shadowed.

$$W_P^T = 1 - \delta_{\text{sec}}\, \delta_{sec\_sec}$$

Let's first compute the probability δ_{sec} that L_i is secondary mapped:

$$\delta_{\text{sec}} = 1 - \frac{V}{L}$$

And for future reference, we also compute δ_{prim}:

$$\delta_{prim} = \frac{V}{L}$$

Now, given that L_i is secondary mapped, what is the probability δ_{sec_sec} that CRM_{TRIM} **cannot** shadow L_i to a primary mapping?

Since L_i is currently secondary mapped, there are $\left\{ {L+F-1 \atop \omega} \right\}$ ways to choose the PMS for L_i. To see this, note that any of the are $L+F-1$ cluster address (all except s_i) can form L_i's PMS.

Now, CRM_{TRIM} can primary shadow L_i, so long as L_i's PMS contains at least one free page. Since F of the $L+F$ physical pages on **C** are free, of all the possible PMS's for L_i, there are $\left\{ {L-1 \atop \omega} \right\}$ different PMS (for L_i) that contain no free pages. And so, given that each ϕ is equally likely:

$$\delta_{sec\_sec} = \frac{\left\{ {L-1 \atop \omega} \right\}}{\left\{ {L+F-1 \atop \omega} \right\}}$$

From the above, we can compute W_P^T. However, for future reference we also compute δ_{sec_prim}, the probability that CRM_{TRIM} can primary shadow a secondary mapped L_i.

$$\delta_{sec\_prim} = 1 - \delta_{sec\_sec}$$

And also for future reference, we compute δ_{prim_sec}, the probability that T cannot shadow L_i to a primary mapping, given that L_i is primary mapped. Note that in this case, CRM_{TRIM} cannot shadow L_i to $\phi(p_i,L_i)$; L_i is already there. So, if L_i is to be primary shadowed, one of its **other** $\omega - 1$ primary mappings must be free. In this case, there are $\left\{ {L+F-1 \atop \omega - 1} \right\}$ possible ways to construct ϕ, $\left\{ {L-1 \atop \omega - 1} \right\}$ of which will force CRM_{TRIM} to secondary shadow L_i.

$$\delta_{prim\_sec} = \frac{\left\{ {L-1 \atop \omega - 1} \right\}}{\left\{ {L+F-1 \atop \omega - 1} \right\}}$$

And similarly:

$$\delta_{prim\_prim} = 1 - \delta_{prim\_sec}$$

Example 5-1

Given L=101,F=10 and b_1=2 (so ω=3), we compute δ_{prim_prim} and δ_{sec_prim}.

$$\delta_{prim\_prim} = 1 - \delta_{prim\_sec} = 1 - \frac{\left\{ {100 \atop 2} \right\}}{\left\{ {110 \atop 2} \right\}} = 0.17$$

$$\delta_{sec\_prim} = 1 - \delta_{sec\_sec} = 1 - \frac{\left\{ {100 \atop 3} \right\}}{\left\{ {110 \atop 3} \right\}} = 0.25$$

From the above, we can see that, if a logical page L_i is already secondary

mapped, it's likelier that it will be primary shadowed. ○

Let's return to our analysis, and compute R_S^T - the probability that T reads in a secondary mapping page. Such will be be the case if L_i is secondary mapped (since CRM_{TRIM} must read the S mapping page covering L_i) or secondary shadowed (since CRM_{TRIM} must modify and therefore read in this same mapping page). In that case :

$$R_S^T = \delta_{\text{sec}} + \delta_{prim}\,\delta_{prim\_sec}$$

Finally, we must compute W_S^T. This is the probability that CRM_{TRIM} updates a secondary mapping page; alternatively it's the probability that L_i is secondary shadowed.

$$W_S^T = \delta_{\text{sec}}\,\delta_{sec\_sec} + \delta_{prim}\,\delta_{prim\_sec}$$

And so , the expected amount of page-table I/O generated by a single read/write transaction ($\overline{A}$) is $R_P^T + W_P^T + R_S^T + W_S^T$.

5.2 Computing the Steady State of a Cluster - simple case

In the previous sub-section, we assumed that when a transaction T entered the system ready to update **C**, it would 'see' the cluster with F free pages, V primary mapped pages (L–V secondary mapped pages) where F, V and L were fixed. In truth however, V is non-deterministic ($0 \leq V \leq L$). When T enters the system , it might see any of L+1 different 'states' of **C**. We refer to a possible state of **C** (STATE(**C**)) as S_V meaning that **C** contains V primary mapped logical

pages.

An update transaction T can cause a cluster to change state. For example, assume that STATE(**C**) $= S_V$ immediately before T enters the system. Now, assume that T secondary shadows a logical page that was primary mapped. It should be clear that, after T has completed, STATE(**C**) $= S_{V-1}$.

Our intent is to compute the steady state probabilities for each S_i. In this way, we can compute the 'true' value of $\overline{A}$, by summing up the respective A's for each cluster state, weighted by the probability of being in that state. Of course, computing the steady state probabilities is no problem provided we construct the transition matrix, i.e. the probabilities of going from state to state. This is our next task.

Let us compute the transitions out of an arbitrary cluster in state S_V. To do this, we consider four possible scenarios (refer to figure 5-1) that may occur when a transaction T updates a logical page from this cluster. In scenario (1), T secondary shadows a primary mapped logical page. In the next scenario, T secondary shadows a secondary mapped logical page. In scenario (3), T primary shadows a primary mapped logical page. And in the last scenario, T primary shadows a secondary mapped logical page. The transition probabilities δ_{prim_prim}, δ_{prim_sec}, δ_{sec_prim} and δ_{sec_sec} were computed earlier.

From the state diagram, we see that in the first (fourth) scenario, T causes a net-loss (net-gain) of one primary mapped page on **C**. The two other cases result in no net-loss or gain of primary mapped pages, and thus take cluster **C** from S_V to S_V.

In the top of figure 5-1, we show explicitly how the two dimensional S transition matrix is constructed from the state diagram. The V^{th} row in $S[][]$ represents the flow out of S_V.

In this way, we evaluate the transition probabilities from other states. Then, we can compute the steady state probabilities and the true value of $\overline{A}$.

Example 5-2

GENERIC SAVINGS BANK clusters its database information alphabetically on 140 physical page clusters, with 20 free physical pages per cluster. Their logical database consists of N=128,000 pages and a 256K bit primary cache. (When $b_1 = 2$, the cache holds precisely the entire P table.) Most of the *GSB* transactions read and update a single logical page.

We use our model to evaluate $\overline{A}$ for each possible value of b_1. In theory, $1 \leq \overline{A} \leq 4$ since a transaction must (at least) write out a primary or secondary mapping page and possibly could read in and update both a primary and secondary mapping page. Graph 5-1 describes the actual results. From the graph, we can see that b_1^{opt} (the optimal value of b_1) equals five.

6. Extending the Model

The simple model, while enlightening, is lacking in two respects. It handles only one 'type' of transaction (a single record update) and provides us with only one metric relating to disk cost (recovery I/O per transaction). To offset these limitations, we extend the model to allow different transaction types and extend the analysis to include another, sometimes more relevant metric.

6.1 The Transaction Environment

We generalize our description of transaction behavior as follows. A transaction of type $T_{(i,j)}$ reads i ($i \leq F$) pages within a single logical cluster and updates j ($j \leq i$) of these pages. We also assume that a single primary and a single secondary mapping page cover an entire cluster. For this reason, each transaction will 'pay' no more than four page-table I/O's.

We let $T(i,j)$ represent the probability transaction $T_{(i,j)}$ will be run by the application. Accordingly, we require that the DBA specify the composition of his transaction environment, in terms of the transaction types that we provide.

Transactions are still restricted to accessing a single cluster, one transaction at a time. Transactions that access multiple clusters can be studied by considering them as a group of single cluster transactions. Similarly, concurrent access to a cluster by several transactions can be approximated by a single transaction that accesses more than one page. For these reasons, we believe the new model is not unduly restrictive.

6.2 Two Metrics for Evaluating Mapping Speed

In sub-section 5-1, we computed $\overline{A}$, the average amount of recovery I/O per transaction. Given our new generalization of transaction types, we extend this notation by letting $A_{T_{(i,j)}}$ be a random variable, representing the amount of recovery I/O caused by transaction $T_{(i,j)}$.

As before, one way that we can evaluate CRM_{TRIM}'s mapping speed is by the average amount of recovery I/O generated per transaction or

$\nu = \sum_{i=0}^{i=L} \sum_{j=0}^{j=i} T(i,j)\ \overline{A_{T_{(i,j)}}}$. This metric seems especially appropriate for environments geared to throughput (batch applications) since it treats all transactions equally, without regard to transaction size.

In an interactive environment, minimizing response time is the goal, so ν may not be the most appropriate metric. Rather, we want a metric that penalizes the shorter transactions more than the long ones for recovery I/O. After all, from the perspective of *time added* as a percentage of total transaction time, small transactions are more affected by a single recovery I/O. In light of this, we propose another metric - the average amount of recovery I/O per database I/O or $\mu = \sum_{i=0}^{i=L} \sum_{j=0}^{j=i} \frac{T(i,j)}{i+j}\ \overline{A_{T_{(i,j)}}}$.

6.3 General Analysis

We refer the reader to [KENTb] for details of the general analysis. In it, we show how to compute $A_{T_{(i,j)}}$ for a given cluster state S_V. And we also describe how the steady state probabilities can be computed efficiently, using dynamic programming.

6.4 Using the Extended Model

In this section, we go through a few sample applications that show how to apply the analysis and tune CRM_{TRIM}.

Example 6-1

Again, we use the *GENERIC SAVINGS BANK* example, but we now con-

sider more low-level implementation details.

In reality, a *GSB* transaction works as follows: The system first accesses a special 'customer to cluster' directory that is maintained in core. Then within the selected cluster at logical page L_0, the database system reads in the logical page directory; this maps customers to logical pages within the cluster. Finally, the database system reads in the appropriate logical page from the cluster.

The majority of transactions at *GSB* fall into one of two categories.

(1) Debit-Credit: Most of these transactions change only the leaf page and not the cluster directory. We model *D–C* transactions as type $T_{(2,1)}$ (95%) and $T_{(2,2)}$. (2.5%)

(2) Query: Retrieve customer's balance for inspection. We model these transactions as type $T_{(2,0)}$ (2.5%).

As stated earlier, each *GSB* cluster holds 140 physical pages and there is a 256K bit primary cache. We tried three different cluster configurations, F=20 and L=120, F=40 and L=100, and F=60 and L=80. † In graph 6-1, we plot mapping speed as a function of b_1 and F. For example, the triangled graph represents a configuration with F=20 and L=120.

It's interesting to note how the optimal value of b_1 decreases as F increases (from $b_1^{opt} = 5$ at F=40 to $b_1^{opt} = 3$ at F=60). This stands to reason; with more free pages on a cluster, fewer bits are needed for a primary mapping address. We can also see from the graph that, by decreasing file utility from $\frac{120}{140}$ to $\frac{80}{140}$,

† We assume that changing the cluster configuration won't affect transaction behavior as transactions are very small.

I/O cost decreases about 17%. Compare the low points on the triangled curve and the pentagonaled curve.

In figure 6-1, we see the recovery I/O for the F=40/L=100 configuration broken down into its components cost for each value of b_1. $P1$ ($P3$) represents the overall probability that a transaction primary (secondary) faults. $P2$ ($P4$) represents the overall probability that a transaction writes out a primary (secondary) mapping page. Also shown is the average number of primary mapped pages in the steady state. Looking at this table, it's easy to see why CRM_{TRIM} performs so poorly with $b_1 = 2$; on average, only 56 of 100 pages are primary mapped. So an average transaction will almost certainly secondary fault ($P3 = 0.843$).

In graph 6-2, we vary the memory size for an F=40/L=100 configuration. When m=i, the primary cache can fully hold P with b_1 set to i. The graph's tree-like structure can be explained by the fact that, at low b_1, extra memory can't be used. For example, with $b_1 = 2$ P fits in core at m=2, so adding memory won't affect this configuration's performance. And looking at this same graph, we see that increasing m to more than 4 is probably not wise. After all, b_1^{opt} decreases by less than 5% from m=4 to m=6.

In graph 6-3, we compare the I/O costs of CRM_{LORIE}, CRM_{TRIM} and CRM_{TWIST} as a function of disk utility. We fixed m at 2. The triangled curve represents the minimal I/O cost for different CRM_{TRIM} configurations as measured by the analysis. The hexagonaled horizontal line is CRM_{LORIE} I/O cost; note that this mechanism is unaffected (from the perspective of page-table I/O)

by changes in disk utility as each logical page can address any physical page (addressing a disk page requires 18 bits). The pentagonaled point measures CRM_{TWIST} I/O cost; as TWIST requires 2 physical pages for every logical page, it's defined only at a single point.

Comparing the three mechanisms, we see for example, that CRM_{TRIM} (as expected) is especially useful when the disk should be around three quarters utilized. In this case, CRM_{TWIST} is unacceptable and CRM_{LORIE} wastes bits.

From the graphs, we can see there is a relatively large difference in I/O cost between CRM_{TRIM} (about 1.35 I/O's per transaction) and CRM_{TWIST} (about 1.00 I/O per transaction) when disk utility is $\frac{1}{2}$. We thought the gap might be due to simplifications in our analysis that result from assumption (3). So we simulated CRM_{TRIM} using a fixed function ϕ that satisfied condition (2) and measured the minimum page-table I/O cost. The squared curve describes the results. Note that our analysis was somewhat pessimistic, CRM_{TRIM} performs even better than we predicted.

The difference between the CRM_{TRIM} simulation and analysis closes quickly as the ratio $\rho = \frac{L*\omega}{L+F}$ becomes large. The number ρ represents the average number of logical pages that primary map to a given physical page. For $\rho = 1$, the difference between simulation and analysis was about 10%, while at $\rho = 4.28$ the difference was less than 1%. This stands to reason. The larger ρ becomes, the less information is lost by relaxing condition (2).

We see that CRM_{TRIM} underperforms TWIST at 50% utilization In this

case, one bit per P entry would suffice to record the two possible locations of a page (as in TWIST). However, our mechanism also needs a CODE value (unnecessary at 50% utilization), so b_1 must be set to two.

Example 6-2

FLY-BY-NIGHT airlines arranges its data in 60 page clusters, with 20 free pages and 40 logical pages per cluster. After measurements, it was found that the majority of transactions were single page queries ($T(1,0) = 0.62$) with updates accounting for the remaining transactions. Each update transactions reads and updates i pages, where i varies from one to fifteen ($T(i,i) = 0.025$).

In graph 6-4, we plot the values of ν and μ for each value of b_1. (Note that $0 \leq \mu \leq 1$ as we pay anywhere from zero to one recovery I/O's per database I/O.) Observe that no choice for b_1 simultaneously minimizes both μ and ν. For example, setting $b_1 = 5$ minimizes ν but causes almost a 20% increase in μ over the optimal value (0.43 to 0.51). The metric μ favors a small $b_1(=2)$, since shorter transactions are less likely to primary fault. Granted large transactions now secondary fault more often, but μ amortizes each such fault over many database I/O's. ○

One of the advantages of CRM_{TRIM} is that it can exploit information about the application's transaction behavior. At the same time, this may be a problem if CRM_{TRIM} is too sensitive to its parameters and is difficult to tune without knowing the application precisely. Fortunately, in our case, this does not seem to be the case. For example, if we compare the curve of graph 5-1 to the topmost

one of Graph 6-1 (both for L = 120, F = 20), they appear similar and have the same b_1^{opt} value, even though they are for different types of transactions. This appears to indicate that b_1^{opt} is mainly a function of the utilization (which can be fixed) and not of the type of transaction. To verify this, we decided to test CRM_{TRIM}'s sensitivity to variations in both transaction size and the type of transaction access (read or write).

In graph 6-5, we varied transaction size (call this variable i) along the x-axis and considered three scenarios. In the first scenario (the triangled curves) we look at an environment with 100% read/write transactions. A read/write transaction of size i reads and updates i pages in a cluster. In scenario two (the squared curves), we consider an application with 50% read/write transactions and 50% queries. A query of size i reads i pages from a cluster. In scenario three (the pentagonaled curves), we examine a query-intensive transaction environment. For each scenario, there are two (partially overlapping) curves. The lower one is for b_1^{opt}, while the upper one is for a fixed $b_1 = 4$. For the optimal curve, the values of b_1 that yield the optimum are labeled. Unlabelled points have the same b_1 value as the point to their left. Thus, for example, in the query intensive environment with transaction of size 1, a mechanism with $b_1 = 4$ produces a little over 0.5 I/O's per transaction, while the best possible performance (0.4 I/O's per transaction) is obtained with $b_1 = 2$.

As expected, the more update transactions in the system, the more recovery I/O's to be paid per transaction (i.e., the squared curves lie below the triangled curves and above the pentagonaled curves). From the graph, we also see that

even those applications that cannot precisely describe transaction behavior can benefit from our tuning procedure. Specifically, note that the value $b_1 = 4$ is optimal for wide ranges of the parameters. In addition, in cases where it is not, the difference in I/O's between $b_1 = 4$ and the optimal b_1 is rather small. Thus, the value $b_1 = 4$ appears to be a safe one for all the parameter ranges illustrated in the graph.

7. Conclusions

We have presented a page-table management strategy that can significantly reduce the amount of I/O overhead of a shadow recovery strategy. Given that recovery I/O is the major cost component of shadowing, this strategy can dramatically improve overall system performance. And just as important, it is flexible enough to fully exploit different disk utilizations.

There are a number of possible improvements or variations to the basic strategy. For example, if the two page-tables are intelligently placed on disk, both a P and S mapping pages might be flushed in a single disk rotation, further improving performance. We can easily account for such behavior in our analysis by charging two mapping page writes as a single page-table I/O.

If a main memory non-volatile buffer is available (implemented in a battery pack, for example), then recovery I/O's can be eliminated. In our case, if P resides in this buffer, only the less likely S I/O will cost. Of course, other strategies could benefit from this hardware, but note that ours is especially well suited for it. That is, the smaller P tables which handles most of the requests can be in the buffer, and the larger S table can still reside on disk.

8. References

[DEW] Dewitt,D., Agrawal, R. "Integrated Concurrency Control and Recovery Mechanisms:Design and Performance Evaluation", *Univ of Wisc Tech Report*, Report #497, Feb. 1983.

[ELH] Elhardt, K. Bayer, R. "The Database Cache for High Performance and Fast Restart in Database Systems", *ACM Transactions on Databases*,4,9,(Dec 1984),pp. 503-525

[GRAY] Gray, J.N., McJones, P. "The Recovery Manager of the System R Database Manager", *ACM Computing Surveys*,13,2,1981.

[KENTa] Kent,J., Garcia-Molina, H. "An Experimental Evaluation of Two Crash Recovery Mechanisms", *ACM PODS Proceedings,(March 1984), pp. 113-123.*

[KENTb] Kent,J., "Performance and Implementation Issues in Database Crash Recovery", Ph.D thesis, Princeton University, (June 1985)

[LAMP] Lampson, B., Sturgis, H. "Crash Recovery in a Distributed Data Storage System", *XEROX Res. Rep. , Palo Alto, Calif.*, Submitted for Publication.

[LOR] Lorie, R.A., "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems*,2,1,(March 1977),91-104.

[REUTa] Reuter, A. "A fast transaction-oriented scheme for UNDO recovery" *IEEE Trans. Software Eng. SE-1*, 6,(July 1980)

[REUTb] Reuter, A. "Performance Analysis of Recovery Techniques", *ACM Transactions on Database Systems*,9,4,pp.526-559.

[SEV] Severance, D. "A Practical Guide to the Design of Differential Files for Recovery of On-Line Databases", *ACM Transactions on Database Systems*,7,4, (Dec 1982),pp. 540-565.

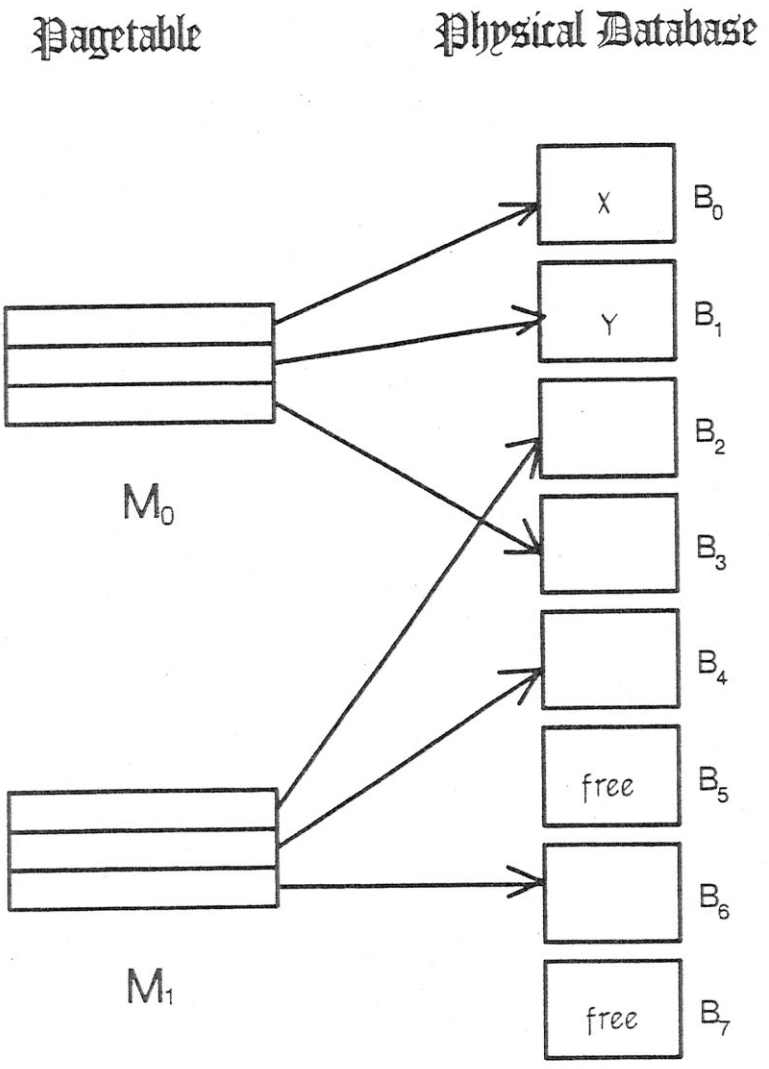

Pagetable
Physical Database
X
Y
free
free
B_0
B_1
B_2
B_3
B_4
B_5
B_6
B_7
M_0
M_1

figure 2-1a

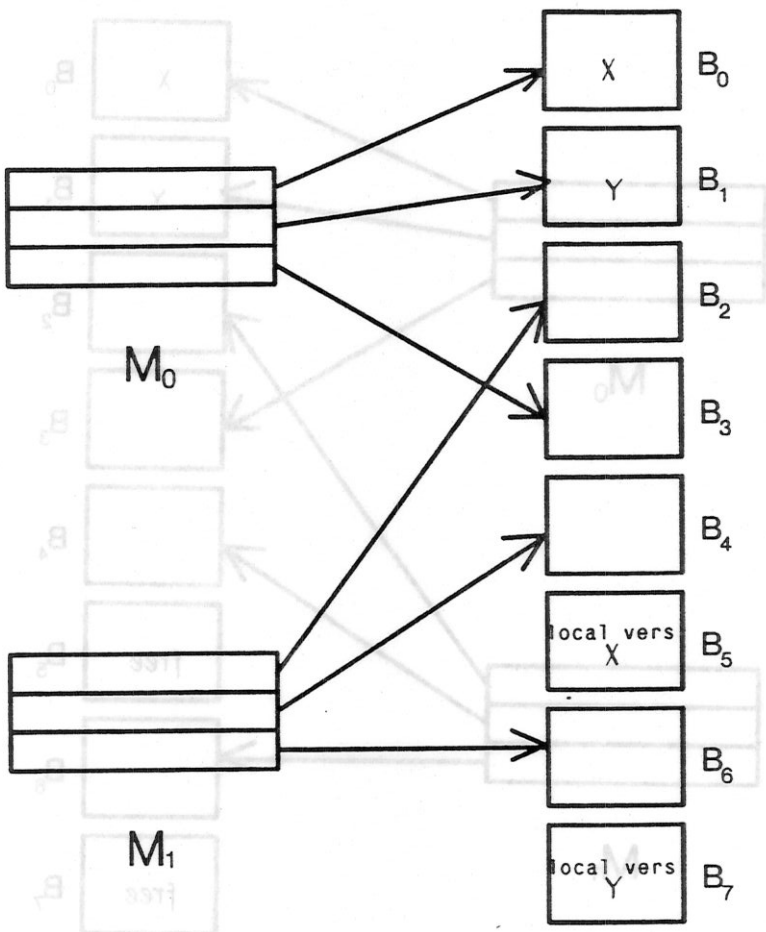

Pagetable
Physical Database
X
Y
local vers X
local vers Y
M_0
M_1
B_0
B_1
B_2
B_3
B_4
B_5
B_6
B_7

figure 2-1b

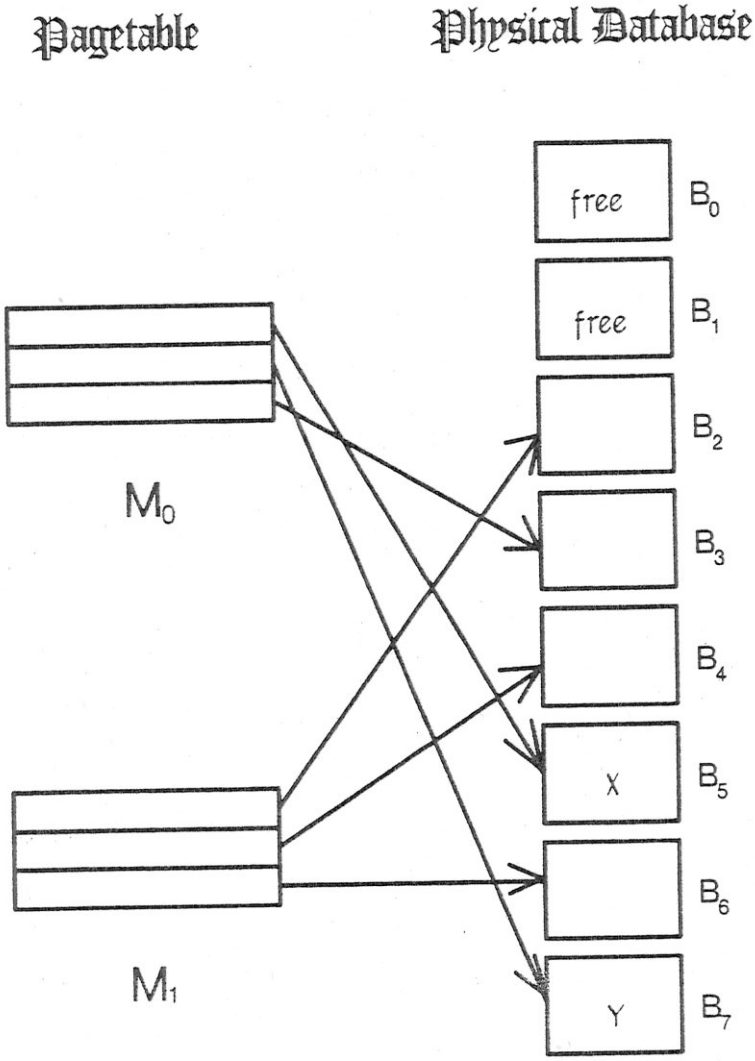

Pagetable
Physical Database
free
free
X
Y
B_0
B_1
B_2
B_3
B_4
B_5
B_6
B_7
M_0
M_1

figure 2-1c

figure 3-1

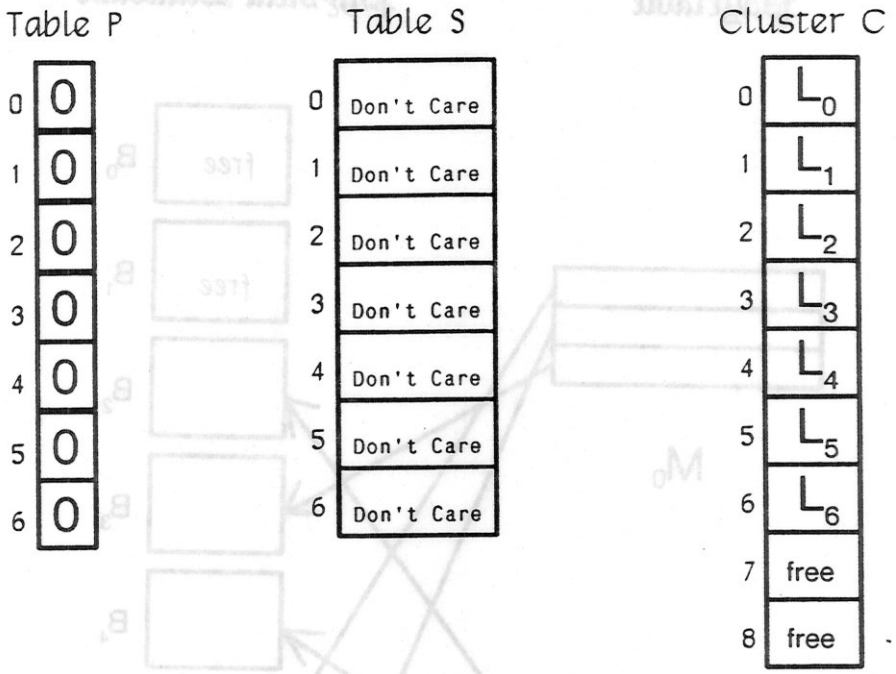


figure 3-2

S_0 S_1 S_2 $\cdots$ S_{V-1} S_V S_{V+1} $\cdots$ S_{L-2} S_{L-1} S_L

S_0

S_1

S_2

.
.
.

S_{V-1}

S_V

a1	a2	a3

S_{V+1}

.
.
.

$$a1 = \delta_{prim}\, \delta_{prim\text{-}sec}$$

$$a2 = \delta_{prim}\, \delta_{prim\text{-}prim} + \delta_{sec}\, \delta_{sec\text{-}sec}$$

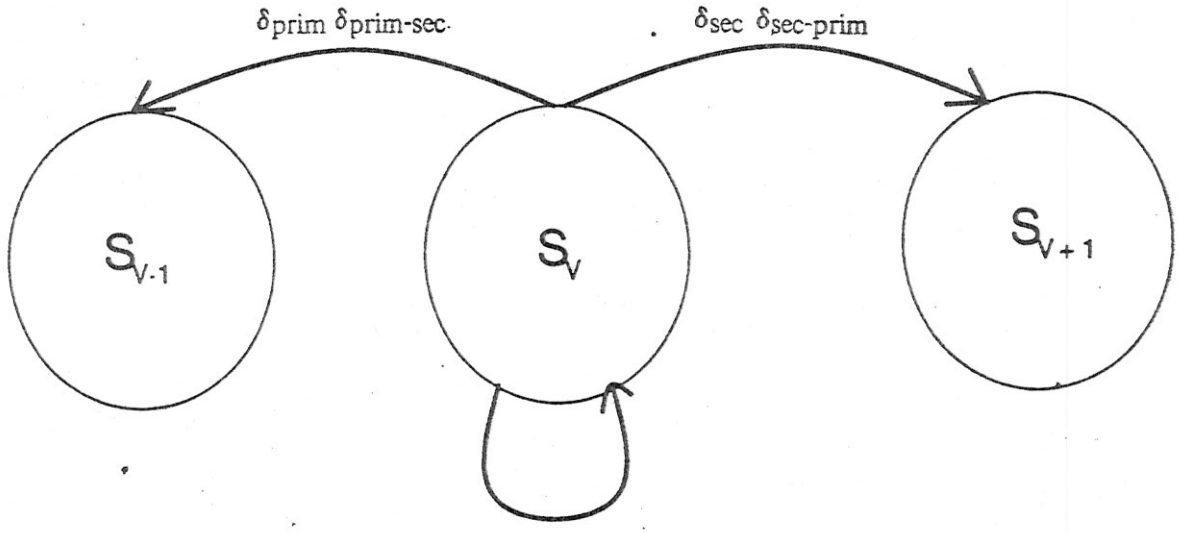
$$a3 = \delta_{sec}\, \delta_{sec\text{-}prim}$$

S_{L-2}

S_{L-1}

S_L

figure 5-1



GENERIC SAVINGS BANK						
No. of bits	Most Likely State	Recovery Cost per Transaction				
		P1	P2	P3	P4	Total
2	56	0.000	0.823	0.843	0.436	2.10
3	88	0.333	0.964	0.324	0.120	1.74
4	100	0.500	0.974	0.019	0.006	1.50
5	100	0.600	0.975	0.000	0.000	1.57
6	100	0.667	0.975	0.000	0.000	1.64

Figure 6-1

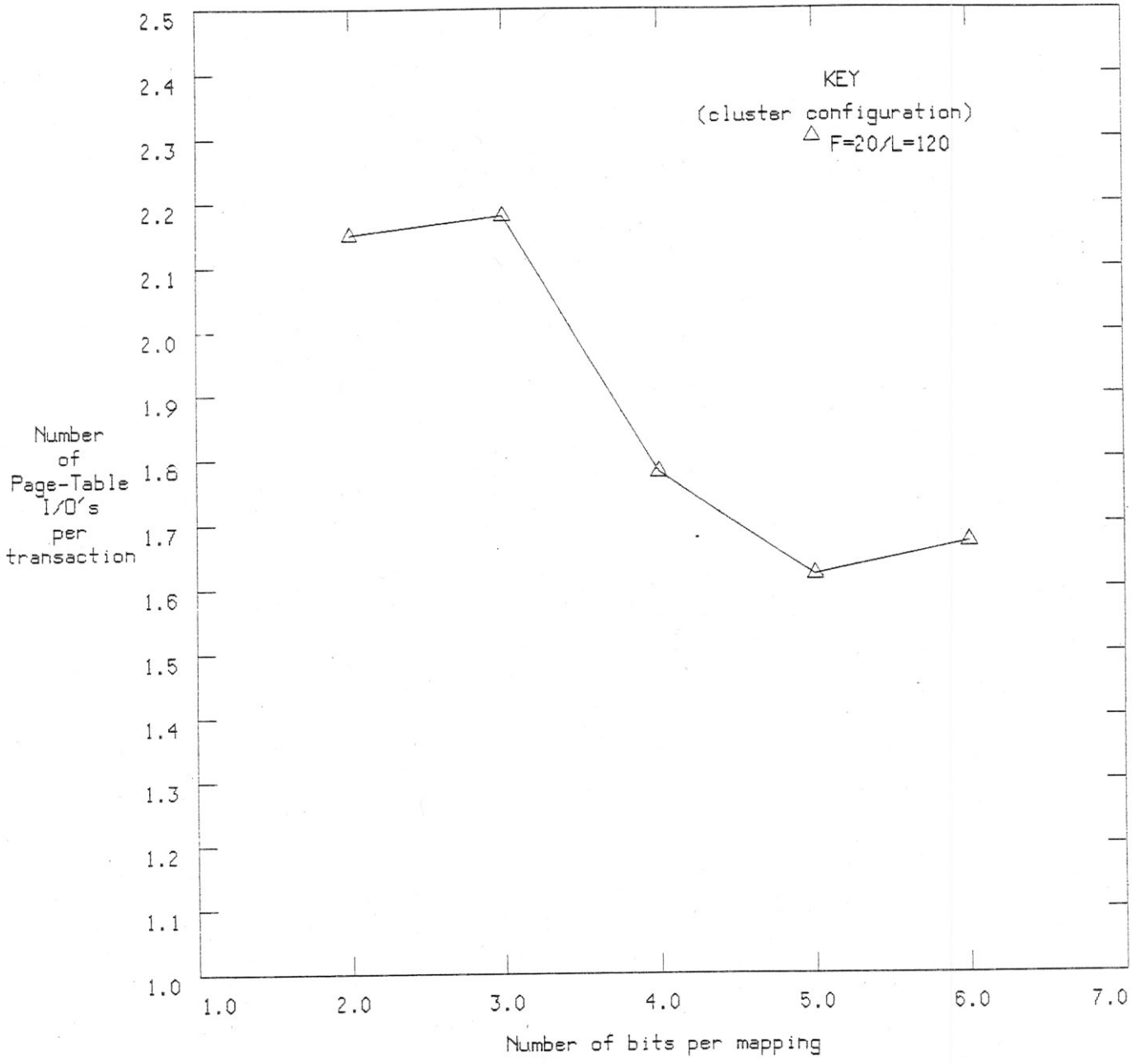
Cluster Configuration: FREE=40/LOGICAL=100

No. of bits = b_1 = size of p_1 entry

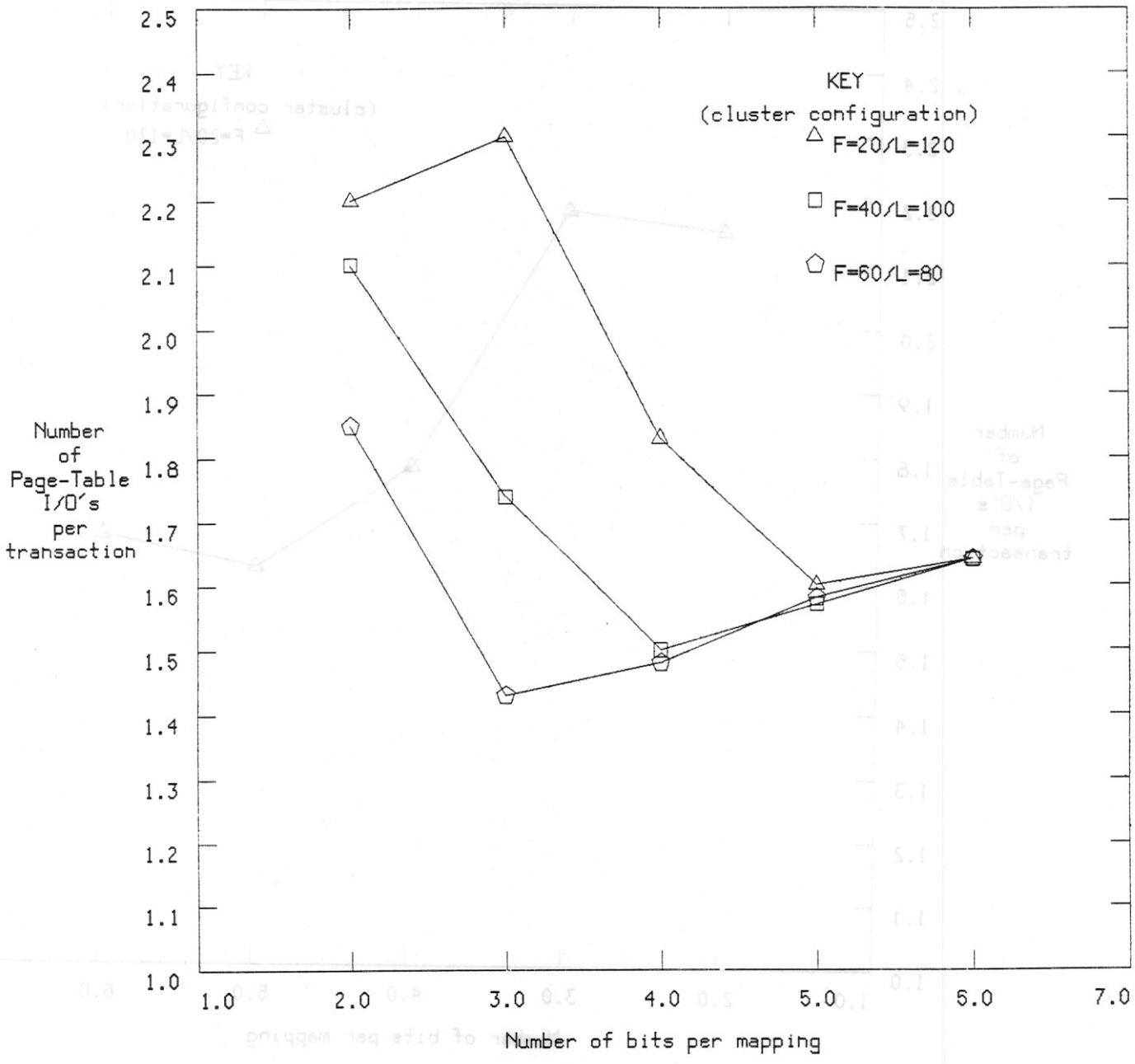
Most Likely State = average number

of primary mapped pages in steady state

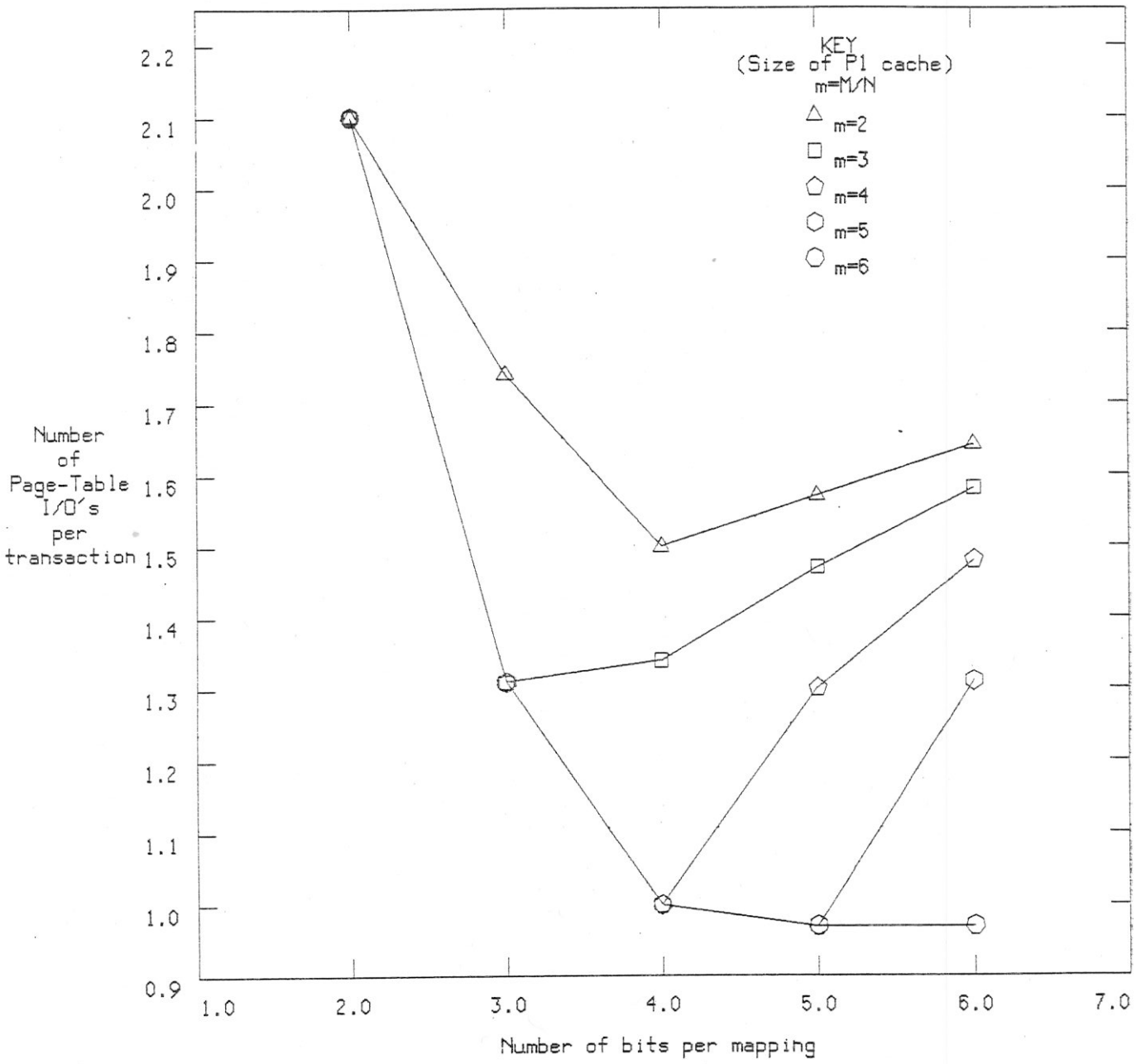
Total = $P_1 + P_2 + P_3 + P_4$ .ps 12


KEY
(cluster configuration)
△ F=20/L=120
Number of Page-Table I/O's per transaction
2.5
2.4
2.3
2.2
2.1
2.0
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
Number of bits per mapping

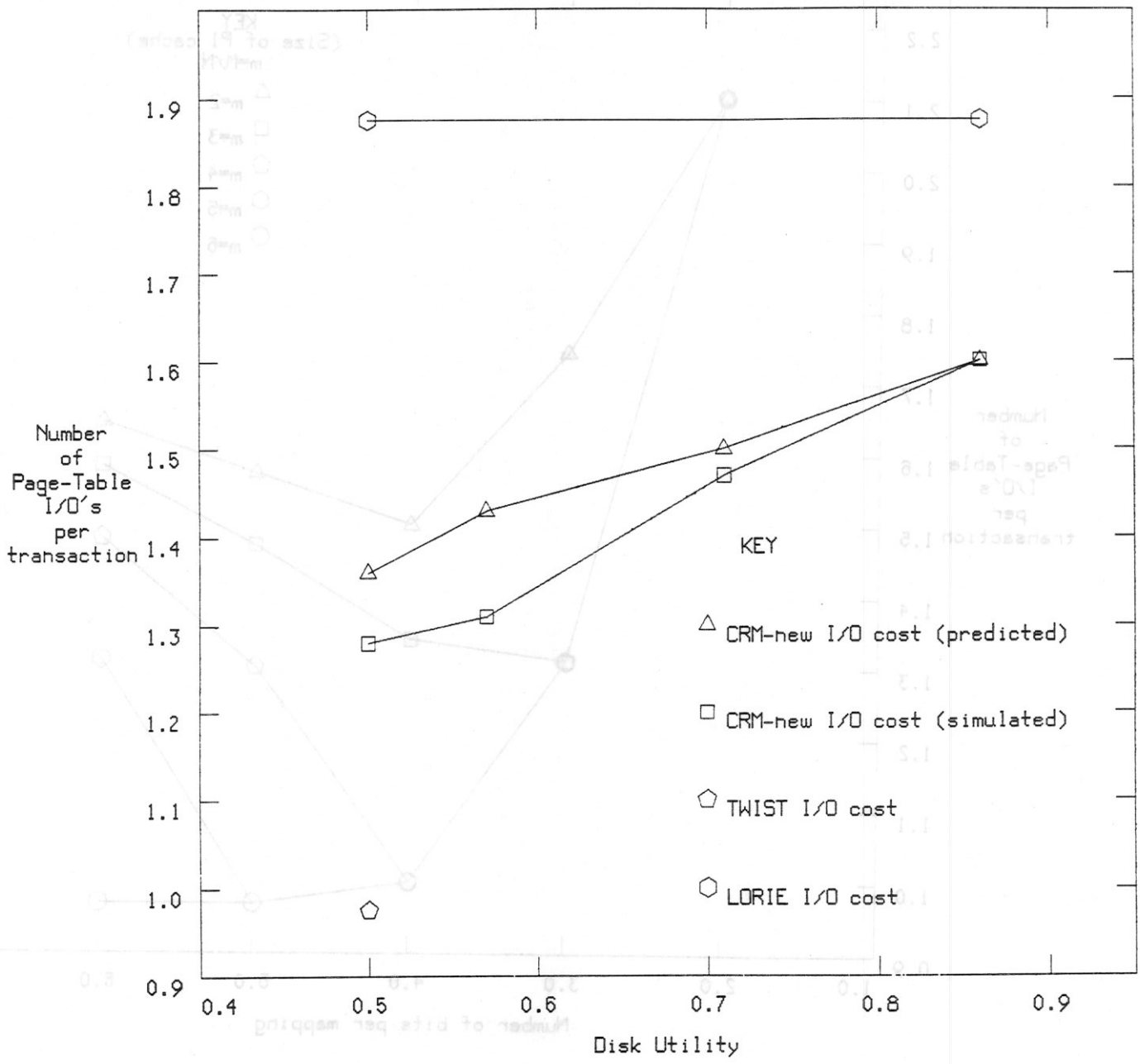
Graph 5-1


KEY
(cluster configuration)
F=20/L=120
F=40/L=100
F=60/L=80
Number of Page-Table I/O's per transaction
2.5
2.4
2.3
2.2
2.1
2.0
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
Number of bits per mapping

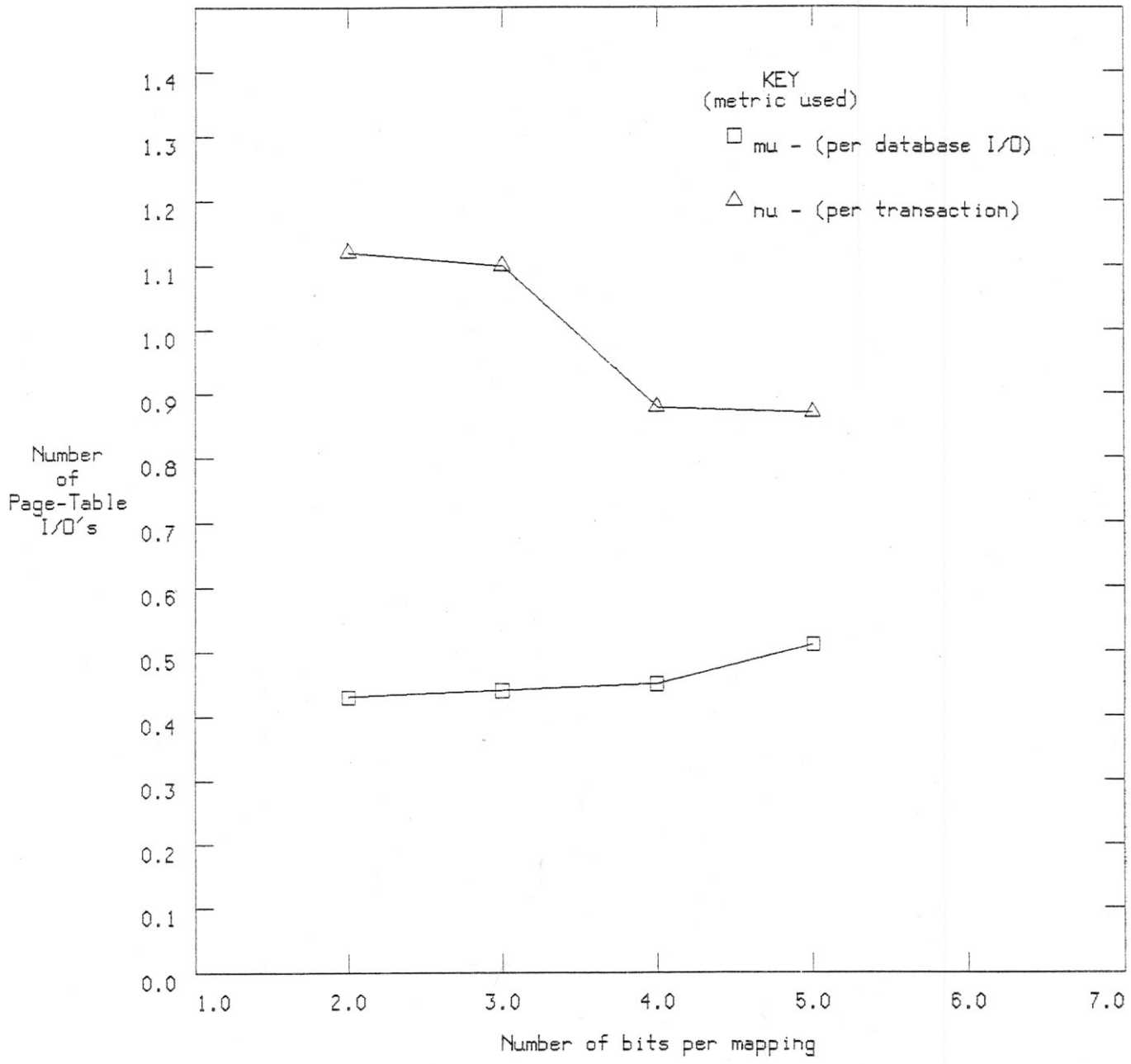
Graph 6-1


KEY
(Size of P1 cache)
m=M/N
m=2
m=3
m=4
m=5
m=6
Number of Page-Table I/O's per transaction
2.2
2.1
2.0
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
1.0
2.0
3.0
4.0
5.0
6.0
7.0
Number of bits per mapping

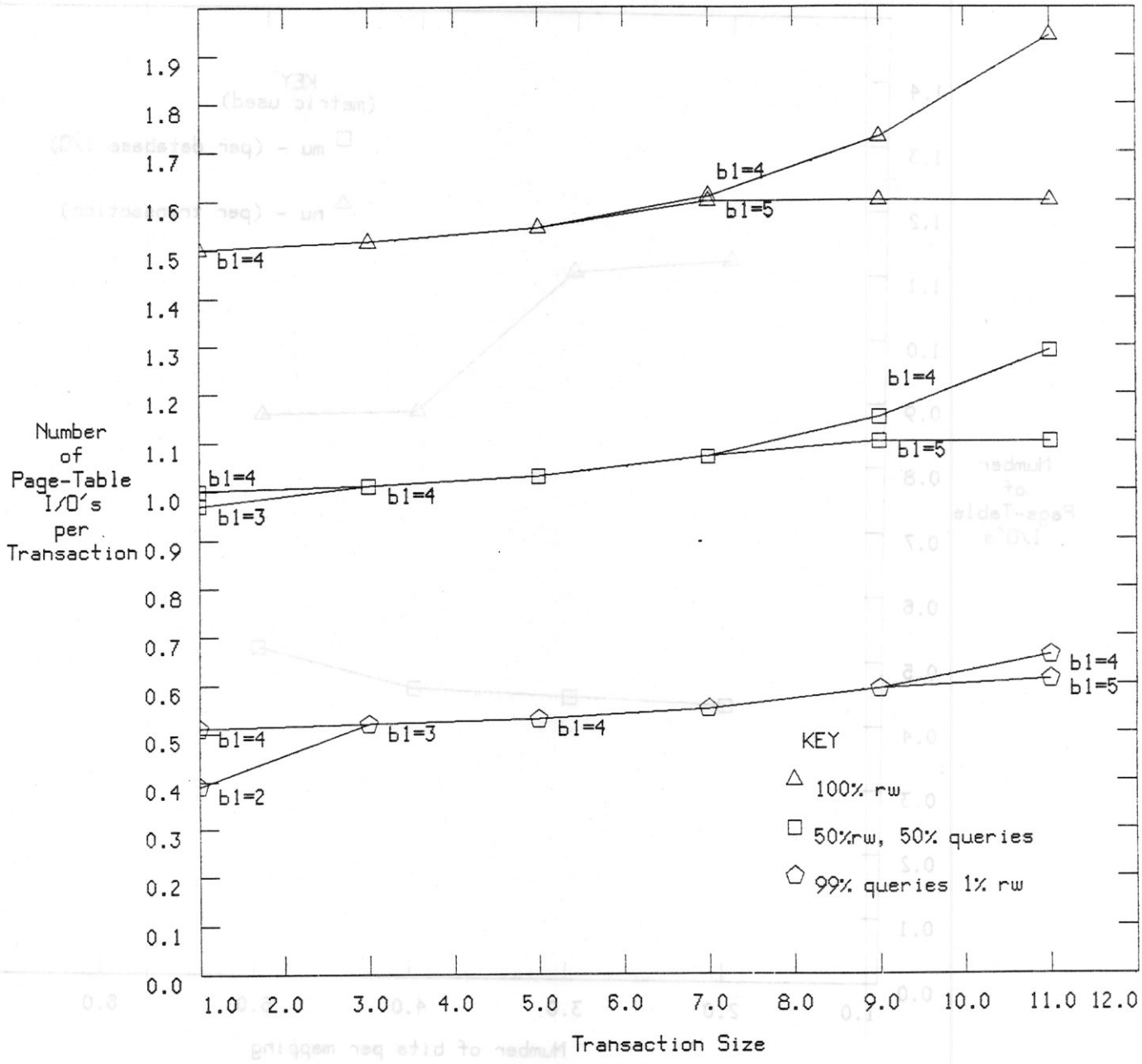
Graph 6-2


Number of Page-Table I/O's per transaction
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.4
0.5
0.6
0.7
0.8
0.9
Disk Utility
KEY
CRM-new I/O cost (predicted)
CRM-new I/O cost (simulated)
TWIST I/O cost
LORIE I/O cost

Graph 6-3


KEY
(metric used)
mu - (per database I/O)
nu - (per transaction)
Number of Page-Table I/O's
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
Number of bits per mapping

Graph 6-4


Number of Page-Table I/O's per Transaction
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
Transaction Size
b1=4
b1=5
b1=4
b1=4
b1=5
b1=4
b1=4
b1=3
b1=4
b1=5
b1=4
b1=3
b1=4
b1=2
KEY
100% rw
50%rw, 50% queries
99% queries 1% rw

Graph 6-5