Project Number: MLC-EMC1

Performance Analysis of the Linux Buffer Cache While Running an Oracle OLTP Workload

A Major Qualifying Project Report
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

____

Michael Narris

____

Joshua Obal

Date: January 9, 2002

Approved:

____

Professor Mark Claypool, Major Advisor

____

Corporate Sponsor Doug Sullivan, EMC Corp.

Abstract

This project deals with the performance analysis of the Linux buffer cache while running an Oracle OLTP workload. The Linux buffer cache was studied and tests were conducted to gather buffer cache hit rates and test run times. The results of this analysis have lead to a better understanding of the complex operations of this system and may help to inspire further research on this topic.

Table of Contents

Executive Summary

This report was prepared for EMC Corporation and WPI to provide better understanding for the performance of the Linux buffer cache while running an Oracle Online Transaction Processing (OLTP) workload. Improved usage of the Linux buffer cache can lead to faster transaction processing. System performance was tested and monitored by developing and using a micro-benchmark suite that simulates the workload of an actual OLTP server.

The motivation for concentrating on the Linux operating system is due to the desire expressed by EMC to learn more about the Linux buffer cache. Also, Linux is governed by an open source policy, which permits direct modification of the kernel source code, thus providing a great deal of flexibility for project work. The use of an Oracle OLTP database is motivated by the popularity of this type of system in industry, especially in EMC systems.

The Linux buffer cache is used to store recently used data in RAM so that it can be quickly available again rather than needing to be reread from the hard drive. Just as there is a buffer cache for Linux, Oracle uses its own buffer cache to store recently used data.

The workload that was used as a basis for analyzing system performance was generated by micro-benchmarks that were designed for this purpose. These micro-benchmarks consisted of the following five OLTP transactions: new-order, payment, check stock, check order, and delivery. Each of these transactions acted upon separate tables in the database, with the new-order and payment transactions occurring ten times as often as the other three. This design closely resembles the design of the TPC-C benchmark for OLTP systems. Three separate databases of this design were created. The

sizes of these were approximately half, equal, and double the size of the system RAM, which was 768MB. A Linux kernel module and variables inserted into the source code were used to keep track of the frequency with which requested data was found in the buffer, thus providing the buffer cache hit ratio.

The results of the performance analysis show that several factors can influence the Linux and Oracle buffer cache hit ratios. The larger the database size, the greater the number of cache misses, and thus the lower the hit ratio. The Oracle buffer cache hit ratio increases by increasing the size of the Oracle buffer cache. However, this increase only slightly affects the Linux buffer cache hit ratio - decreasing slightly if the Oracle buffer cache is larger than the size of the physical memory. The total completion time for 100 transactions, as shown in Figure 4.9, indicates that the systems performs fastest when Oracle cache sizes between 256MB and 736MB are used.

It was determined that Linux is able to swap part or all of the Oracle cache to swap space on disk if it is not being used. This is undesirable because it causes additional delays to obtain data that is in the Oracle buffer cache. Currently, the best way to alleviate this is to properly adjust the size of the Oracle buffer cache so that it is large enough to store as much data as possible, but still small enough that it does not cause the system to slow down while being swapped in and out of Linux swap space.

The objective of this project was to conduct a performance analysis of the Linux buffer cache, and, in doing so, discover any Linux kernel changes or system configurations that would improve this performance. This analysis has shown that properly adjusting the size of the Oracle buffer cache can improve the hit ratio for both caches as well as the time required to process transactions.

Chapter I. Introduction

There are many useful applications for Oracle including Online Transaction Processing (OLTP). OLTP systems, which involves small, frequent transactions that are typical of an order-entry system, has become a very important function of database systems. OLTP workloads are used in any application that calls for data entry and retrieval transactions such as e-commerce applications. Oracle has become a popular database management software and can be used to run an OLTP workload. Oracle needs an operating system to run on, and one of the leading operating systems for small servers is Linux. The popularity of the Linux operating system is due in part to the open source policy that allows users to modify the operating system code and distribute it for free. This policy provides a great deal of flexibility and makes project work on Linux ideal. However, Linux is not extensively used as an operating system on leading OLTP servers, which motivates performance studies for use in this type of application.

The Linux buffer cache is of particular interest because reading and writing data to disk requires a greater deal of time than accessing data stored in the buffer cache. Whenever data can be found in cache, it is not necessary to read it from disk. Understanding the Linux buffer cache algorithm can help tune a system for optimal performance.

The goal of this project was to gain a better understanding of the Linux buffer cache through performance analysis of a system running an OLTP workload on Oracle. It was also of great interest to discover any means of modifying and improving the Linux buffer cache algorithms and the system configuration for improved performance.

We built micro-benchmarks that simulated multiple simultaneous users of an Oracle OLTP system. These benchmarks created large OLTP workloads and recorded performance information pertaining to the Linux and Oracle buffer caches. Several variables were used in order to better understand the results that were recorded including database size, number of transactions, and Oracle cache size. The results from these tests were then graphed for comparison so that trends and optimum values could be determined.

The results of this project include statistics that were gathered from both Oracle and the Linux kernel. Hit ratios from the Linux and Oracle buffer caches are compared to show the effect of resizing the Oracle buffer cache. Test run times are also used to help understand the Linux and Oracle buffer caches.

This paper describes our methods of analyzing the performance of the Linux buffer cache while running an Oracle OLTP workload. Chapter II, Background, discusses Oracle database systems, OLTP, micro-benchmarks, the TPC-C model for benchmarking OLTP, and the Linux buffer cache. Chapter III, Approach, describes the steps used for developing the micro-benchmark suite and the databases used in the experiments, and also describes the process of running the experiments and modifying the buffer cache algorithm. Chapter IV, Results and Analysis, presents the experimentation results in the form of graphs and describes any trends that were found. Chapter V, Conclusions, presents the conclusions drawn from the analysis of the results. Chapter VI, Future Work, discusses how the results of this project can be applied to further studies involving performance of Linux caching algorithms.

Chapter II. Background

This section describes detailed information about the central topics of the project. The background begins by introducing Oracle database systems as well as some fundamental database concepts. The second topic is online transaction processing (OLTP), which is a popular type of database transaction involving small, frequent transactions. Next is an explanation of micro-benchmark programs to simulate a large load of OLTP transactions and recording performance information. Fourth, there is a description of the TPC-C model for benchmarking OLTP systems. Finally, this chapter presents a description of the buffer cache used by Linux for storing data in memory and writing it to disk.

2.1 Oracle Database Systems

An Oracle database, like any database, allows for the storage, organization, retrieval, and protection of related information. These databases organize related data for easy retrieval. The two types of databases that currently dominate information systems are relational and object-oriented databases. Relational databases store and present all information in tables and clearly define relations between data, making them very easy to conceptualize and use. Object-oriented databases store data in objects that model real world entities, and data relations are controlled through methods that control which actions can be performed on the data. Oracle databases support both of these models. However, this project exclusively uses relational databases because they have much more dominance in the business world [Bobrowski 2000].

An Oracle database can only be accessed once an instance of the Oracle server is started up. This server is responsible for making data accessible to users while maintaining security at the same time. It must also ensure integrity and consistency of data for multiple concurrent users and handle data backup and recovery features. For a user to gain access to the database the user must run a client program such as “sqlplus”. The user must also have an account with adequate permissions to perform the desired operations. This provides the user with a way to request, update, enter and delete information. It also allows the user to alter, create, or drop tables in the database [Bobrowski 2000]. The select and update operations were used in this project to develop transactions, while the other operations were used for database administration.

2.2 Online Transaction Processing (OTLP) and Decision Support Systems (DSS)

The two main types of transactions that can be performed on databases are online transaction processing (OLTP) and decision support systems (DSS). OLTP is used in systems that process many small transactions of retrieving and updating data. Examples of possible applications for this kind of system are banking, e-commerce, and airline reservations. DSS are used by applications that generally make large queries for data that is used for data analysis. These systems do not perform nearly as many write operations to the database as an OLTP. Also, OLTP systems perform far more transactions than DSS [Keeton 1999].

The Keeton thesis (1999) presents several major problems with studying database workload performance in OLTP and DSS systems. The existing standardized benchmarks for these workloads, such as TPC described in Section 2.4, are complex and require the researcher to perform tedious tuning of the system. Reporting results for these systems requires almost flawless system configuration and having a certified auditor audit the benchmark configuration. Because of the cost involved in this, it is common for researchers to instead report results as being TPC-like in nature. Also, studying these systems can require large amounts of expensive equipment in order to accurately simulate systems that are used to process these workloads. The quick growth in complexity of DSS workloads makes it difficult to design systems that have fast enough processors and enough storage capacity to support these tests.

This project exclusively uses OLTP transactions in order to measure the performance of the Linux buffer cache, however it is possible to also conduct a similar experiment using DSS instead. Because of the difficulties associated with using standardized benchmarks, custom micro-benchmarks as described in the following section were used to monitor system performance.

2.3 Micro-Benchmarks

The purpose of the benchmarks in this project was to create a large OLTP workload on the Oracle server and monitor its performance. Due to the numerous combinations of possible settings for each platform, it is nearly impossible for Oracle or any other software manufacturer to test every possible setting. Instead, a database administrator can run a micro-benchmark to detect problems that the software manufacturer overlooked, and to configure the system for better performance.

The micro-benchmarks used in this project were composed of a custom set of transactions to test the Linux buffer cache algorithm. The tests conducted by micro-benchmark programs can be used for everything from mass reads and writes to the database as well as tasks such as logging in and out of the database and numerous other database interactions detailed in the Scaling Oracle8i™ book [Morle 2000]. In this project the benchmark design reflected the need to simulate as closely as possible how a database would operate with a real OLTP application.

2.4 TPC-C Model for Benchmarking OLTP

The specifications for the TPC-C benchmark provide a guideline for implementing a benchmark to generate an OLTP workload. TPC-C was designed by the Transaction Processing Performance Council (TPC) – one of the leading authorities on benchmarking systems running an OLTP workload on databases. TPC-C is an OLTP benchmark that defines a set of functional requirements for a benchmark that can be run on any OLTP system, regardless of the hardware or operating system. The TPC-C benchmark simulates a large number of terminal operators executing transactions against a database, thus generating a large OLTP workload.

TPC-C models the principal transactions found in an order-entry system for any industry that must manage, sell or distribute a product or service. However, it does not include less frequent transactions, which may be important to the functioning of an actual OLTP system, but do not have a large impact on the performance of the system. The five transactions that are included in TPC-C are: entering new orders, delivering orders, recording customer payments, monitoring the stock level in a warehouse, and checking the status of an order. The frequency of these transactions, like the transactions themselves, is also modeled after real world scenarios. The new-order and payment transactions occur ten times as often as delivery, stock, and checking order status transactions. The new-order transaction is the performance limiting transaction in this type of system. The TPC-C database consists of nine different tables and a wide range of population and record sizes.

The results recorded by the TPC-C benchmarks are most useful if they are reproducible and verifiable. In order to report results of the TPC-C benchmark important system information must also be reported in order to make the results reproducible.

TPC-C permits any physical database design technique such, as partitioning data, in order to improve performance. TPC-C's performance metric, tpm-C, evaluates the number of complete business operations that can be processed within one minute on the system being evaluated [TPC 2001].

The TPC-C specification for benchmarking OLTP workloads served as a backbone for designing the micro-benchmarks that were used in this project. The micro-benchmarks also followed the probability for most common transactions as specified by TPC-C.

2.5 Linux Buffer Cache

The purpose of the buffer cache is to avoid having to perform slow disk accesses every time that data is modified or needs to be read by storing data in buffers in memory. The buffer cache keeps track of which data has been modified (has become dirty) in memory and needs to be written to disk, and writes pieces of that data at regular intervals. The buffer cache also provides a faster means of subsequent reads to the same data. In this way, I/O operations produce minimal slowdowns on the system [Bovet 2001].

The Linux buffer cache consists of two kinds of data structures: a set of buffer heads that describe the buffer in the cache; and a hash table that is used to quickly determine which buffer head describes the buffer described by a particular device and block number. The buffer cache is implemented in the Linux kernel and is governed by several functions that are responsible for determining when to write data to and from memory to disk, where in memory to put data that is read from disk, and how to free up space by replacing used memory. The source files "fs/buffer.c" and "include/linux/fs.h" contain many of these functions. Some of these functions include: getblk(), brelse(), refile_buffer(), bforget(), grow_buffers(), refill_freelist(), bdflush(), and kupdate() [Bovet 2001].

The getblk() function in buffer.c is called every time a block must be retrieved from memory. If the desired block is already in the buffer cache (called a cache hit) then it can be quickly read from the memory. If it is found that the block is not in the buffer cache (called a cache miss) then it must be read from disk and stored in a free buffer if any are available. Cache misses are more time consuming than cache hits because the hard disk from which the block must be read has a much slower access time than the system memory.

The methods in which modified (i.e. dirty) buffers are written to disk enable the system to operate much faster by avoiding constant disk operations. There are three methods that are used to write dirty buffers to disk. First, there are three system calls (synch, fsynch, and fdatasynch) that allow user applications to flush dirty buffers to disk. The synch call is the only function that flushes all dirty buffers to disk. It is called periodically while the system is running and is usually called before shutting down the system. Fsynch allows all buffers belonging to a particular file to be flushed to disk. Fdatasynch also flushes all buffers belonging to a particular file, but it does not flush the inode block of the file [Bovet 2001].

The next method that allows buffers to be flushed to disk is the *bdflush()* kernel thread. This kernel thread is responsible for selecting marked, dirty buffers from the buffer cache and forcing the corresponding blocks on disk to be updated. *Bdflush()* is only called when the number of dirty buffers in the buffer cache exceeds the threshold value Nfract. The parameters for *bdflush()* are located in /proc/sys/vm/bdflush in the *b_un* and *bdf*_prm tables [Bovet 2001]. These parameters are described in table 2.1 (1 tick corresponds to approximately 10 milliseconds).

Parameter	Default	Min	Max	Description
Age_buffer	3000	100	60,000	Time-out in ticks of a normal dirty buffer for being written to disk
Age_super	500	100	60,000	Time-out in ticks of a superblock dirty buffer for being written to disk
Interval	500	0	6,000	Delay in ticks between *kupdate* activations
Ndirty	500	10	5,000	Maximum number of dirty buffers written to disk during an activation of *bdflush*
Nfract	40	0	100	Threshold percentage of dirty buffers for waking up *bdflush*

Table 2.1 Buffer Cache Tuning Parameters [Bovet 2001]

The *kupdate()* kernel thread is used to flush older dirty buffers which may not be flushed by *bdflush* because the threshold limit was not hit. The parameter *age_buffer* defines the time that kupdate will wait to flush a dirty buffer to disk, with a default value of 30 seconds. The *kupdate* thread runs at a frequency that is defined by the *interval* parameter. Any buffers whose *b_flushtime* value is greater than or equal to the *jiffies* parameter (specified number of clock ticks) are written to disk by *sync_old_buffers()* which is called by *kupdate()*. Unlike *bdflush(), synch_old_buffers()* does not limit the number of buffers checked on each activation [Bovet 2001].

Chapter III. Approach

The approach chapter deals with the issues involved in setup, design, implementation, and experimentation of the test-data and the micro-benchmark suite for studying the performance of the Linux buffer cache. In order to tune and gather results with the micro-benchmark suite, test databases were created. Linux kernel modifications were made to enable keeping track of buffer cache hits and misses, and a module was developed to access the hit and miss counts in the kernel.

The computer used for this project consisted of the following hardware, software, and configurations: Dual Pentium III Xeon 550 MHz, 768 MB RAM, 1536MB Linux swap space, 6 SCSI hard drives 80 GB total, SuSE Linux 7.2, 2.4.4-64GB-SMP kernel, and Oracle 9i Enterprise Edition.

3.1 Database design

The databases used for testing the Linux buffer cache were composed of randomly generated data with preserved Primary and Foreign keys loaded into multiple tables. A program was designed to create each table of the database to a specified size with randomly generated numbers and/or characters depending on the column. The data for each table was stored in files formatted for input into Oracle. Once the data was in a file, “sqlldr,” Oracle’s bulk database loader was used to load the data contained in the files into an Oracle SQL table.

Three databases of varying size were created relative to the system memory. The smallest was approximately half the size of the system memory, the medium database was approximately equal to the system memory and the large database was approximately twice the system memory. See Figure 3.1 for the database sizes. Each database consisted of nine tables, each with several attributes. See Figure 3.2 for tables and attributes[1]. Figure 3.3 lists the number of entries, or tuples per table in each sized database.

System RAM	768 MB
Small Database	350 MB
Medium Database	700 MB
Large Database	1400 MB

Figure 3.1 System memory and physical size of each database

Table Name	**Table attributes**
Customer accounts	Account →customer information account number, balance
Customer information	Unique account number, customer name, address, telephone number, credit
Deliveries	Account →customer information account number, item number→stock item number, shipping number, estimated time of arrival, actual time of arrival, estimated time of departure, actual time of departure, address, name
Items	Item number→stock item number, item name, description, cost
Orders	Unique order number, Account →customer information account number, order date, ship date, item list and quantity
Payment	Account →customer information account number, amount, payment date
Pending orders	Account →customer information account number, total spent, current items, order date
Stock	Unique item number, quantity, manufacturers cost
Transaction history	Account →customer information account number, total spent, date of first order, date of last order

Figure 3.2 Tables and table attributes

[1] "Unique" indicates a Primary Key, "→" indicates a Foreign Key to specified table and Primary Key attribute.

Table Name	Small Database	Medium Database	Large Database
Customer accounts	2,500,000	5,000,000	10,000,000
Customer information	2,500,000	5,000,000	10,000,000
Deliveries	25,000	50,000	100,000
Items	50,000	100,000	200,000
Orders	25,000	50,000	100,000
Payment	25,000	50,000	100,000
Pending orders	25,000	50,000	100,000
Stock	50,000	100,000	200,000
Transaction history	2,500,000	5,000,000	10,000,000

Figure 3.3 Number of entries per table in each test database

The database sizes in Figure 3.1 were chosen because a system is not able to cache more data than system memory. A database size of half the memory can be completely cached in memory and only produce cache misses when data is first loaded into memory. A database equal to the size of the system memory should mostly be cached but not completely as there are other processes using the system at the same time such as Oracle, which use system resources. A database twice the size of the available system memory cannot all fit in the cache and should produce the most cache misses and cache replacements. To reduce contention, the database files for the large database were placed on a different disk that those of the small and medium database and the Oracle install was on a third separate disk [Bobrowski 2000].

3.2 Micro-benchmark suite design

This section deals with the design and implementation of the micro-benchmark suite. The micro-benchmarks were used to generate a simulated OLTP workload to measure the cache hit ratio on the Linux buffer cache.

3.2.1 OLTP Transactions

The micro-benchmarks for this project needed to closely simulate a real online transaction processing system. To achieve this, the TPC-C model for online transaction processing benchmarking was used.

The primary user interface for Oracle databases is “sqlplus”. Several programming languages and other programs have built-in features, which allow direct access to the database. For this project, the scripting language Perl was selected. Using the Perl DBI module, a simple script can perform queries on the database as easily as using SQL. The scripts are also able to generate random search strings and run other Perl query scripts to better simulate OLTP. Five common OLTP transactions as specified by TPC-C were designed to connect to the database and perform queries on the data. These queries include “check order” which selects a random order number and retrieves the data associated with that order number from the table Orders; “deliveries” which inserts a new randomly generated delivery into the deliveries table; “new order” which selects a random customer account, retrieves the customer and account information, inserts a new order into table Pending, updates the customers transaction history, and returns the information on the item ordered; “payment” which inserts a payment into the table Payment from a random customer; and “stock” which returns information about a randomly selected stock number. None of the returned data is actually printed out. These scripts are meant to simulate OLTP clients retrieving data from a warehouse.

3.2.2 Cache hit rate monitoring

In order to monitor the number of cache hits and misses on the buffer cache while the scripts are running, kernel “hooks” were added. These “hooks” were two variables placed in the Linux source code file “buffer.c”. The first, emc_hits is incremented whenever getblk() is called and the block requested is in the hash table. The occurrence of this event represents a cache hit. The emc_misses variable is incremented whenever the block is not in the hash table. This represents a cache miss. The hash table contains information about all of the blocks in the buffer cache. To ensure that the cache hits and misses were caused due to the micro-benchmark, a device check was added to the getblk() function which only allowed the counters to be incremented if the cache hit or miss was on a block from one of the disks which contained the Oracle database files. The cache hit ratio was then obtained by dividing the number of hits by the sum of the hits and misses.

Accessing the values of these variables posed a problem since user space programs cannot access kernel space variables directly. The solution was to develop a kernel module, which can be loaded and unloaded by use of insmod and rmmod, to access the values of these variable counters. The values were reset whenever the kernel module was loaded and printed when the kernel module was removed. EMCmodule was the module developed as part of the micro-benchmark suite to print the hit and miss counts to the /var/log/messages log file where the statistics can be retrieved by an overall control script.

3.2.3 Control program

Rather than requiring the user to run several programs to gain statistics out of the micro-benchmark suite, an overall control script was created. This Perl script, control.pl, allows the user to execute all of the scripts and data gathering tools sequentially through one command.

When a user runs the control script, they are prompted for the number of transactions to run and how many times they want to run tests with this number of transactions. The kernel statistics module is loaded, one of the nine transaction scripts is chosen at random and run in series with the TPC-C probability (as mentioned in Chapter II Background) of most common transactions, the kernel statistics module is removed, and the statistics are printed to the screen. The specified number of tests is run and results are printed out each time. The script can also be configured to run the scripts in parallel, however in series was chosen because in parallel, the time to run one test cannot be monitored.

3.3 Experiments

Experiments were run with several different sized Oracle caches. The Oracle cache size is specified in 2 files as the value of "db_cache_size" in "initmqp1.ora" and "spfilemqp1.ora". The values of this variable must be the same in each file to ensure the correct cache size is used. The cache size can be a minimum of 16 megabytes up to a maximum size of the Linux swap space.

A typical run of an experiment goes as follows. The cache size is set and the database is started by using the command "dbstart". Inside SQLplus, "@tune" is run which executes the query in the file "tune.sql" which returns information about the Oracle buffer cache and hit rate generated during the startup of the database. Next the test is executed by running "control.pl" which executes the micro-benchmark on a database. The micro-benchmark returns information about the Linux buffer cache and the running time while running "@tune" in SQLplus again will return information about the Oracle buffer cache. At this point the two values can be computed and compared.

These tests were run three times on each of the databases with each of the Oracle buffer cache sizes. The averaged value of the results was then calculated to give more accurate statistics.

3.4 Buffer cache algorithm modifications

An additional aspect of this project was to determine the effects of a change to the buffer cache algorithm. The changes tested for this project were to the bdflush_param structure which controls how often kupdate and bdflush run.

Bdflush_param Parameters	**Description**
nfract	Percentage of buffer cache dirty to activate bdflush
ndirty	Maximum number of dirty blocks to write out per wake-cycle
nrefill	Number of clean buffers to try to obtain each time refill is called
dummy1	Unused
interval	Clock tick delay between kupdate flushes
age_buffer	Time for normal buffer to age before flushing it
nfract_sync	Percentage of buffer cache dirty to activate bdflush synchronously
dummy2	Unused
dummy3	Unused

Figure 3.4 bdflush_param parameters in buffer.c

Tests were run with changes to the interval, making it longer so that kupdate runs less frequently, making bdflush write out dirty buffers only when it needs to. Nfract and nfract_sync percentages were changed to cause them to let the buffer cache fill up more before writing out to disk and ndirty was increased to write out more to disk when bdflush is called. It was believed that since disk access is much slower than RAM access and requires much overhead that can slow down system performance, making disk access occur less often, overall system performance would increase.

3.5 Getblk() Walkthrough

The following code is the Linux source code for the getblk() function. This code also contains the code that was added to monitor buffer cache hits and misses and comments to help describe the code that was added.

```
struct buffer_head * getblk(kdev_t dev, int block, int size){
       struct buffer_head * bh;
       int isize;
       int major, minor;

repeat:
       major = (dev >> 8);
       minor = (dev & 0xff);

       spin_lock(&lru_list_lock);
       write_lock(&hash_table_lock);
       bh = __get_hash_table(dev, block, size);
       if (bh){
           // Increment cache hit variable here
           if ((major == 8) && ((minor == 49) ||
                                (minor == 65))){
              if ((buffer_uptodate(bh)) && (buffer_mapped(bh)))
                  emcmqp_hits++;
              else{
                  // count this as a miss because it will have
                  // to be retrieved from disk
                  emcmqp_misses++;
              }
           }
           goto out;
       }
       // Increment cache_miss variable here
       if ((major == 8) && ((minor == 49) ||
                            (minor == 65))){
           emcmqp_misses++;
       }
       isize = BUFSIZE_INDEX(size);
       spin_lock(&free_list[isize].lock);
       bh = free_list[isize].list;
       if (bh) {
              __remove_from_free_list(bh, isize);
              atomic_set(&bh->b_count, 1);
       }
       spin_unlock(&free_list[isize].lock);
       /*
        * OK, FINALLY we know that this buffer is the only one of
        * its kind, we hold a reference (b_count>0), it is unlocked,
        * and it is clean.
        */
       if (bh) {
              init_buffer(bh, NULL, NULL);
              bh->b_dev = dev;
              bh->b_blocknr = block;
              bh->b_state = 1 << BH_Mapped;

              /* Insert the buffer into the regular lists */
              __insert_into_queues(bh);
       out:
              write_unlock(&hash_table_lock);
              spin_unlock(&lru_list_lock);
              touch_buffer(bh);
              return bh;
       }
       /*
        * If we block while refilling the free list, somebody may
        * create the buffer first ... search the hashes again.
        */
       write_unlock(&hash_table_lock);
       spin_unlock(&lru_list_lock);
       refill_freelist(size);
       goto repeat;
}
```

The getblk() function takes in as parameters a device number, block number and block size. It checks to see if the function is in the hash table of blocks currently in the buffer cache. If the block with the specified size is in the buffer cache, the function returns the buffer_head structure for the block, which contains information about the block, the buffer state, and the location in RAM. If it is not in the hash table, it takes a buffer_head from the list of free buffer_heads. If a buffer_head is acquired, it fills in the device number, block, and state of the buffer_head and returns it. Otherwise it refills the list of free buffers and repeats the process.

The code in bold font is the code added for this project to measure buffer cache hits and misses and the checks to make sure only hits and misses to the disks the database files are located on are recorded.

Chapter IV. Results and Analysis

This chapter describes the data that was gathered from the numerous tests runs on the databases. This data is then analyzed, and some conclusions are drawn based upon the trends in the results. Each subsection displays graphs broken down by database size and are followed by a detailed analysis of each graph.

4.1 Database Size

This section describes the effect on performance of using different database sizes. In order to evaluate each database, 4 different Oracle cache sizes were used for tests. The tests utilized the micro-benchmarks that are described in the Approach chapter. Each test consisted of 100 transactions, chosen based upon the frequency of each transaction as specified by TPC-C. The average number of queries per transaction is approximately 2.74 – this accounts for the probability and number of queries for each specific transaction.

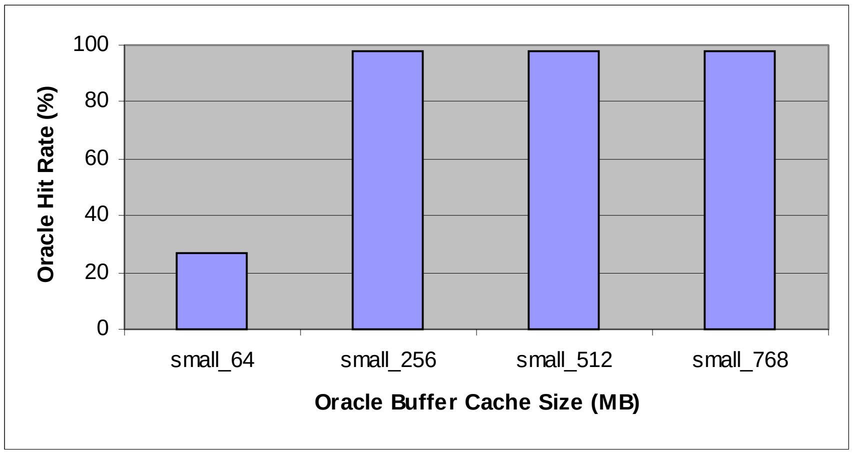

Figure 4.1. Oracle Buffer Cache Hit Ratio for Small Database

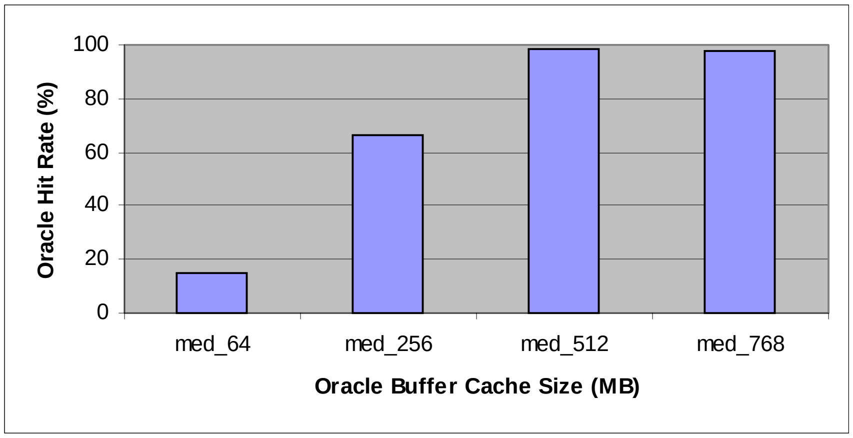

Figure 4.2. Oracle Buffer Cache Hit Ratio for Medium Database

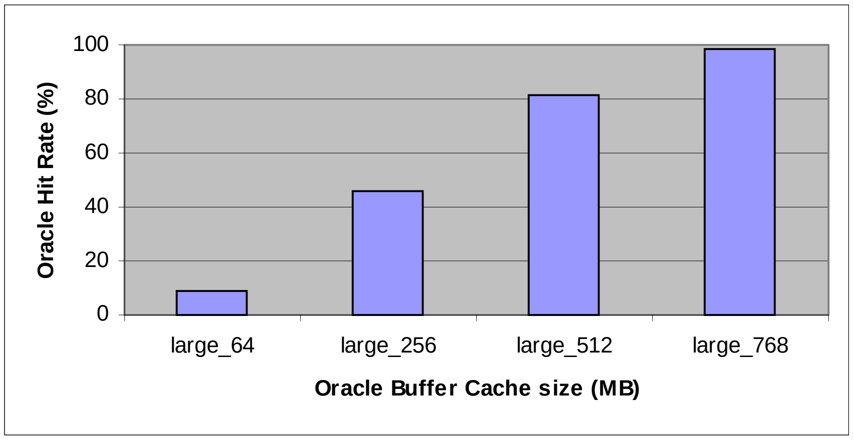

Figure 4.3. Oracle Buffer Cache Hit Ratio for Large Database

Tests run using the small database shown in Figure 4.1 result in the same Oracle hit rate for all Oracle cache sizes except the smallest (64M). This is because the database is small enough to be able to have almost all of the data loaded into the Oracle cache at the same time. The medium size database (Figure 4.2) also can be loaded almost entirely into the Oracle cache for the 512M and 768M cache sizes, and thus the Oracle hit rates do not change for any higher Oracle cache sizes. Using the large database, as shown in Figure 4.3, it is not possible to entirely cache the database into RAM. Because of this, the test results using the large database more clearly show a trend of higher Oracle hit rates for larger Oracle cache sizes. If, on the other hand, the database were small enough to fit entirely into the Oracle buffer cache, then all of the data could be loaded into it and the Oracle buffer cache would have very few cache misses. This would not allow useful data to be extracted from the tests. For this reason, all of the subsequent tests were conducted using the large database.

4.2 Oracle Buffer Cache

Figure 4.4 shows the Oracle buffer cache hit ratio, as reported by Oracle, for tests that were run using the large database and cache sizes ranging from 64MB to 1536MB (compared to the physical memory size of 768MB and approximately 1536MB of swap space). The graph clearly shows a trend of increasing hit ratio as the cache size used increased. Oracle recommends using an Oracle buffer cache size that results in a hit ratio of 90 percent or greater [Bobrowski 2000]. The results in Figure 4.4 show that the hit ratio exceeds 90 percent for every Oracle cache size that is greater than or equal to 672MB.

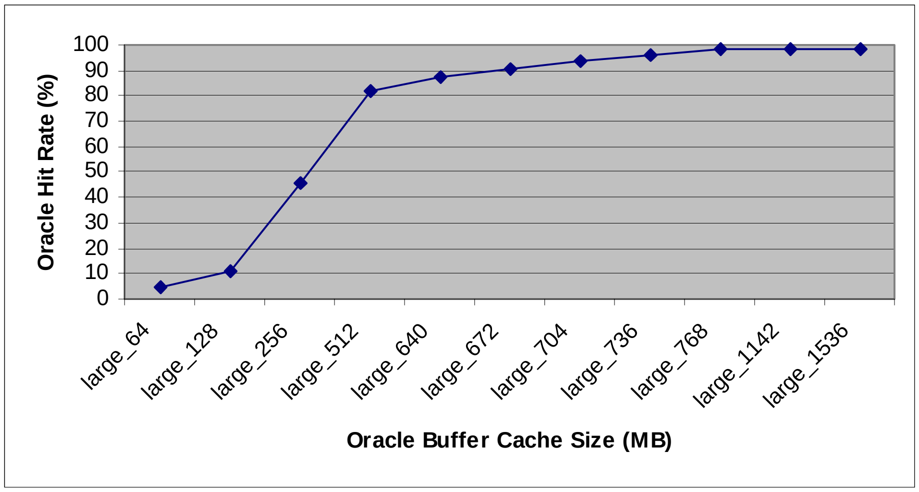

Figure 4.4. Oracle Buffer Cache Hit Ratio

The average number of Oracle buffer cache misses occurring per query is displayed in Figure 4.5. These numbers were obtained by dividing the total number of cache misses by the average number of queries in a test with 100 transactions. As the size of the buffer cache increased there was more space allotted to store the data. This resulted in a lower number of cache misses for tests that used a larger buffer cache size.

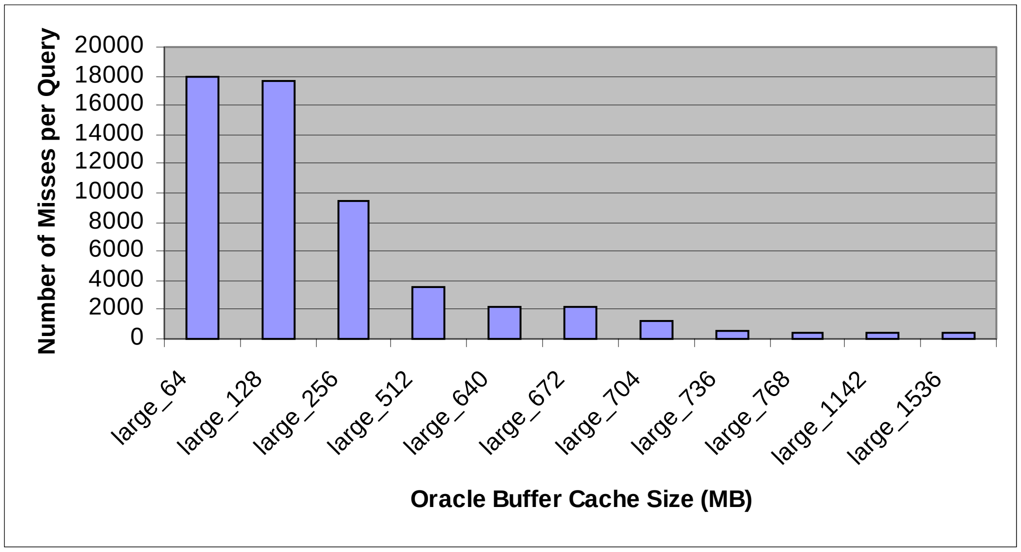

Figure 4.5. Oracle Buffer Cache Misses Occurring per Query

4.3 Linux Buffer Cache

The Linux buffer cache hit ratio as measured by the kernel hooks added to buffer.c, shown in Figure 4.6, exceeds 99% for every test that was run regardless of the size of the Oracle buffer cache. This fact shows that there is little performance change in Linux based upon this variance in size. However, it is important to note that there is a slight decrease in the Linux hit ratio if the Oracle buffer cache size is near or exceeds the size of the system RAM (ie. For all cache sizes greater than or equal to 736MB). This is most likely due to the fact that the Oracle cache could not entirely fit into RAM and must therefore be swapped in and out.

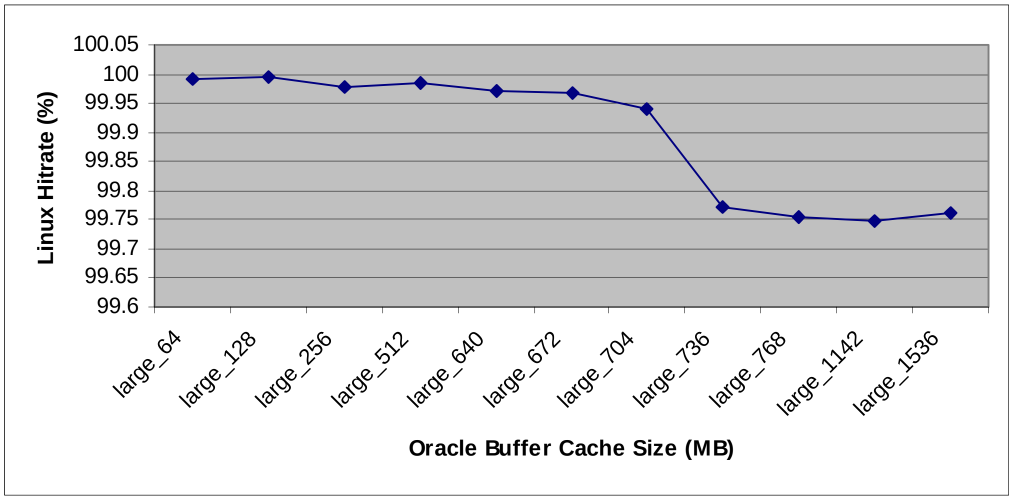

Figure 4.6. Linux Buffer Cache Hit Ratio

Figure 4.7 shows that the number of cache misses per transaction in the Linux buffer cache is mostly consistent regardless of the size of the Oracle buffer cache. The Oracle buffer cache size does not have a large effect on the number of misses in the Linux buffer cache and there is a very low number of misses per query. These could both be a result of the usage of the Linux page cache and the swap space. The entire database could be stored in swap space and then swapped in from disk once it is detected that it is not in memory. Any subsequent requests for data would result in buffer cache hits rather than misses. This is also the reason that the Linux buffer cache hit ratio is so high.

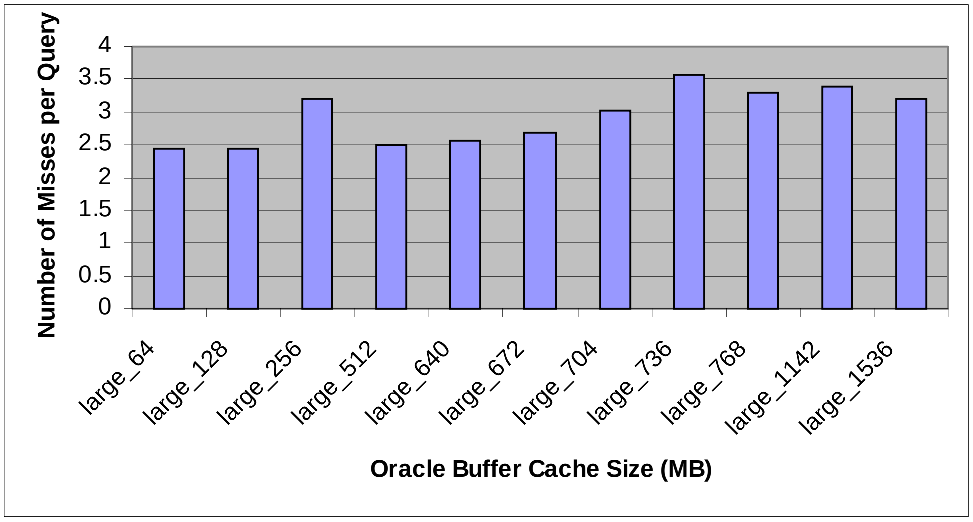

Figure 4.7. Linux Buffer Cache Misses Occurring per Query

4.4 Balancing the Linux and Oracle Buffer Caches

Figure 4.8 shows the number of Oracle buffer caches misses that occur for every Linux buffer cache miss. A possible ideal situation would be for an approximately equal number of misses to occur in each buffer cache rather than having one or the other thrashing to retrieve all of the data from disk. It is also important to minimize the number of Oracle and Linux buffer cache misses as shown in Figures 4.5 and 4.7 respectively. This graph shows that small Oracle cache sizes cause too many cache misses, whereas large Oracle cache sizes perform about equally well. However, to obtain the best performance, it is also important to consider the run time as described in 4.5.

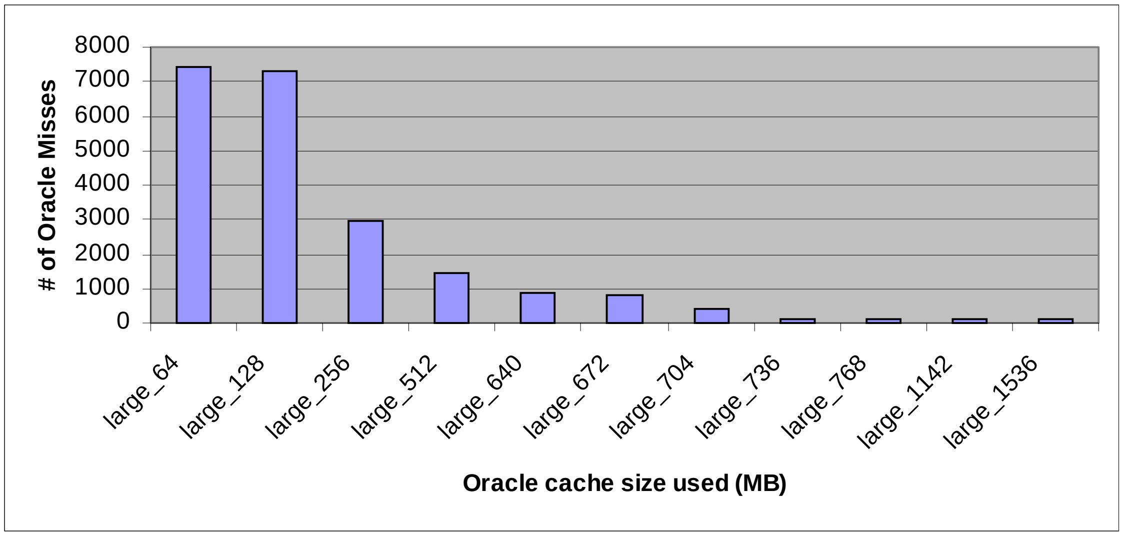

Figure 4.8. Oracle Misses Occurring per Linux Miss

4.5 Test Run Time

The actual run time for completing an equal number of transactions using different sizes for the Oracle buffer cache can be see in Figure 4.9. In some cases, this information is ultimately the most important result because database system users are only concerned with the speed with which their transactions are processed. The run times that are shown in this graph are an average of three test runs for each size buffer cache, and the standard deviation for these was approximately 200 seconds. Because of the random selection of transactions – some of which take longer to complete – these results would be impossible to reproduce exactly, however, the intent is to show the trend that results from using different size Oracle caches.

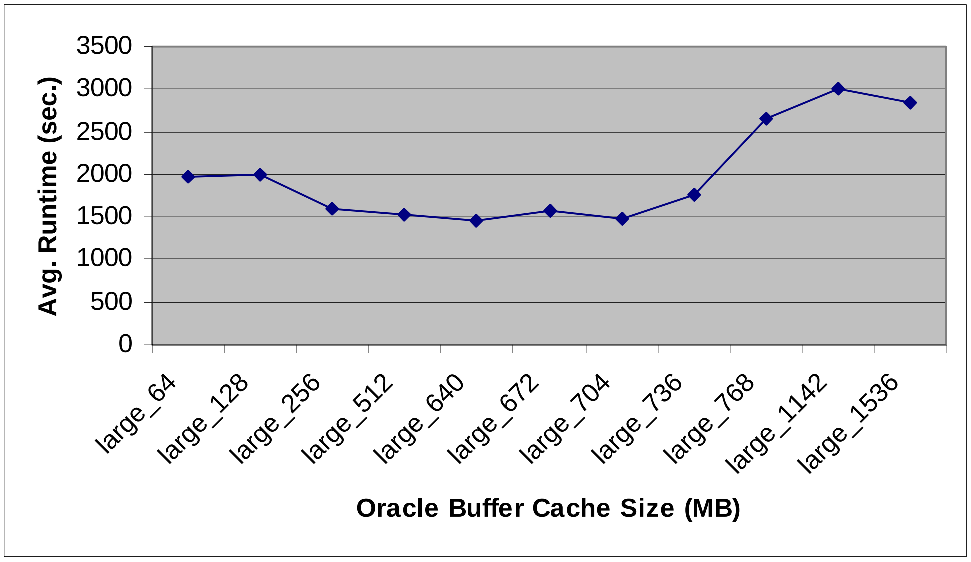

Figure 4.9. Test Run Time

As can be seen in the graph, if the Oracle buffer cache is too small or too large then it will take more time to complete the tests. The reason that more time is required for a very large Oracle cache (larger than memory) is because the entire buffer cache will not fit into memory at once. Thus, more data more must be copied into and out of swap space. Also, too small of an Oracle buffer cache takes more time because data must be shuffled in and out of Oracle's buffer cache because it fills up very quickly. The fastest test time occurs when the Oracle buffer cache is small enough to fit into RAM, but not so small that it cannot hold a significant portion of the database.

4.6 Linux Page Cache and Swap Space

The Linux page cache and swap space were found to be the cause of low Linux buffer cache hit rates in Section 4.3 Linux Buffer Cache. /proc/meminfo was used to monitor the memory and swap usage during several tests. It became evident that Linux does not keep the Oracle buffer cache in memory – it swaps part or all of it out to the swap space onto disk. In an attempt to prevent Linux from swapping out the Oracle cache, the Linux swap space was turned off. It was determined that the Oracle cache size must be less than or equal to the size of the Linux swap space. Therefore, if you turn off the Linux swap space then you cannot run Oracle. Linux must be able to swap the entire Oracle buffer cache out disk. The user is also unable to make the Oracle buffer cache larger than the size of the swap space and run Oracle. Swapping the Oracle buffer cache decreases system performance because Oracle buffer cache data that is supposedly loaded into RAM was actually swapped out to disk and requires additional overhead to swap it back into RAM.

4.7 Buffer Cache Algorithm Modifications

In this project, several modifications to the Linux buffer cache algorithm, as mentioned in Chapter III section 3.4, resulted in tests taking as much as 15 minutes longer to complete. The modifications adjusted the conditions required to run bdflush() and kupdate() which write out dirty buffers to disk. Do to time constraints an exhaustive study of the effects of changing these variables could not be completed, however this shows that there is a performance impact and leads to the belief that these variables can be tuned to specific server applications.

Chapter V. Conclusions

The goal for this project was to test the performance of an OLTP system on the Linux operating system. OLTP has become a very important function of database systems and the Linux operating system has gained popularity as a server operating system because of its open source policy. Because of the time involved with reading and writing data to and from disk, understanding the Linux buffer cache presents an ideal opportunity to better the performance of the entire system for OLTP applications.

Performance analysis of the system was performed using a micro-benchmark suite that generated a large OLTP workload on the Oracle database. The Oracle buffer cache hit ratio was obtained from Oracle while the Linux buffer cache hit ratio was gathered using kernel variables and a kernel module that recorded the number of cache hits and misses. Tests were run on three different sized databases using several different Oracle buffer cache sizes.

In order to achieve optimal Oracle performance, a large enough Oracle buffer cache must be used. As the size of the oracle buffer cache approaches the size of physical memory, the Oracle buffer cache hit ratio exceeds 90 percent, which is considered optimal by Oracle. Once the Oracle buffer cache size reaches or exceeds the size of physical memory, the hit ratio levels off however the time to run each test increased because Linux had to use swap space to store parts of the Oracle buffer cache and other kernel data in memory. This swapping in and out of swap space required more reads and writes to disk by the kernel, which slowed down system performance and forced Oracle to wait until data was back into physical memory before it could access it. It was also discovered that the Oracle buffer cache size cannot exceed the size of the Linux swap space.

With any sized Oracle buffer cache, the Linux buffer cache hit ratio was found to be over 99 percent but drops off 0.25 percent when the Oracle buffer cache approaches and exceeds the size of the physical memory. This suggests that the data for the database is being completely loaded into RAM and swap space and then accessed by Oracle. The slight drop in hit ratio around the size of physical memory suggests that more misses occur in the Linux buffer cache when data is in swap space and not in physical memory because it has to be brought back into physical memory.

The Linux buffer cache can be adjusted and tuned to the application of the Linux server. The modified kernel code for this project offers a way to gather statistics to help tune a system for optimal performance under any application. Micro-benchmarks such as those developed for this project are also useful to further understand how the system is behaving.

Chapter VI. Future Work

This chapter discuses further research and testing that can be done to help better understand the inner workings of the Linux kernel and how to tune it to optimal performance for large-scale servers.

Further testing is required to determine all of the effects of adjusting the parameters of bdflush_param. In database applications where far more reads occur to the database than writes, the bdflush_param parameters may be able to be tuned for better performance. Likewise these parameters may be tuned to a database system with far more writes than reads or and equal number of both. For example, a system with more reads may be tuned to have the flushes out to disk occur less often, or even only if the buffer cache contains too many dirty buffers.

Testing of the Linux page cache and Linux swap space may also discover further system tuning. As shown in this report, the Linux page cache and swap space greatly affect the performance of the OLTP transactions. Modifying parts of the Linux kernel might lead to a more efficient buffer cache algorithm or reduce the need for the buffer cache all together. There are many possible ways to alter Linux to be more efficient for large-scale servers.

References

[Bobrowski 2000]	Bobrowski, Steve. Oracle 8i for Linux Starter Kit. Berkeley, CA: McGraw-Hill 2000.
[Bovet 2001]	Bovet, Daniel P. and Marco Cesati. Understanding the Linux Kernel. Sebastopol, CA: O'Reilly 2001.
[EMC 2001]	www.emc.com. Dec. 10, 2001.
[Keeton 1999]	Keeton, Kimberly K. "Computer Architecture Support for Database Applications," University of California at Berkeley: 1999.
[Morle 2000]	Morle, James. Scaling Oracle 8i. Reading, MA: Addison-Wesley 2000.
[TPC 2001]	"Overview of the TPC Benchmark C: The Order-Entry Benchmark" www.tpc.org/tpcc/detail.asp. Oct. 14, 2001.

Appendix A. TPC-C Results

Database	Hardware Vendor	System	tpmC	Price/ tpmC	System availability	OS	TP Monitor	Date Submitted
Oracle 8i Enterprise Edition v. 8.1.7	Bull	Bull Escala Epc 810 c/s	66,750	37.57 US $	05/28/01	IBM AIX 4.3.3	Webshpere App. Server Ent. Edition V.3.0	05/28/01
Oracle 8i Enterprise Edition v. 8.1.7	Bull	Bull Escala EPC2450 c/s	220,807	34.67 US $	05/28/01	IBM AIX 4.3.3	Webshpere App. Server Ent. Edition V.3.0	05/28/01
Oracle 8i Enterprise Edition v. 8.1.7	Compaq	Compaq AlphaServer GS320 Model 6/731	155,179	52.88 US $	02/02/01	Compaq Tru64 UNIX V5.1	Compaq DB Web Connector	04/03/01
Oracle 9i Database Enterprise Ed. 9.0.1	Bull	Bull Escala PL800R	105,025	25.41 US $	09/26/01	IBM AIX 4.3.3	Websphere App. Server Ent. Edition V 3.0	09/26/01
Oracle 9i Database Enterprise Ed. 9.0.1	IBM	IBM eServer pSeries 660 Model 6M1	105,025	25.33 US $	09/21/01	IBM AIX 4.3.3	Websphere App. Server Ent. Edition V 3.0	09/10/01
Oracle 9i Database Enterprise Edition	HP	HP 9000 Superdome Enterprise Server	389,434	21.24 US $	05/15/02	HP UX 11.i 64-bit	BEA Tuxedo 6.4	11/19/01
Oracle 9i Database Enterprise Edition	Compaq	Compaq AlphaServer GS320	230,533	44.62 US $	07/30/01	Compaq Tru64 UNIX V5.1	Compaq DB Web Connector V1.1	06/18/01

Source: www.tpc.org (November, 2001)

Appendix B. Sample code

This section contains sample code of the files created for this project.

Kernel module emcModule.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <stdio.h>
#include "/usr/include/linux/emcCache.h"
#include <linux/modversions.h>

/*EMCMQP cach hit count variable */
extern int emcmqp_hits;
extern int emcmqp_misses;

int init_module()
{
        printk("----------------TEST STARTED-------------------\n");
        printk("Cache hits %d \n",emcmqp_hits);
        printk("Cache misses %d \n",emcmqp_misses);
        emcmqp_hits = 0;
        emcmqp_misses = 0;
        return 0;
}

void cleanup_module()
{
        printk("----------------TEST DONE----------------------\n");
        printk("Cache hits %d \n",emcmqp_hits);
        printk("Cache misses %d \n",emcmqp_misses);

}
```

emcCache.h

```
/*EMCMQP cach hit count variable */
int emcmqp_hits;
int emcmqp_misses;
```

Sample transaction script: payment

```
#!/usr/bin/perl -w

use DBI;
use strict;
use DBI qw(:sql_types);

my $dbh = DBI->connect("dbi:Oracle:mqp1.emcmqp.wpi.edu", <oracle user>, <password>, {
RaiseError => 1, PrintError => 1 } ) or die $DBI::errstr;

my $numcustomers = 2500000; # 2.5 Million customers

#---------- insert payment into table
my $account = rand($numcustomers) % $numcustomers;
my $amount = rand(5000) % 5000;
my $paydate = rand(10000) % 10000;

my $sql = qq{ INSERT INTO payment_small VALUES ($account,$amount,$paydate)};
$dbh->do($sql);

$dbh->disconnect;
```

Control.pl

```
# File: control.pl
# Authors: Michael Narris, Joshua Obal
# Description:  This program randomly selects which microbenchmarks to run.  Neworder and
# payment occur 10 times as often as the other microbenchmarks.
# Run Instructions:     "> ./control.pl"

print "Number of repetitions:";
my $repetitions = <STDIN>;
print "Number of sets:";
my $numsets = <STDIN>;

  open (FILE, "more /proc/meminfo|grep Cached |");
  while ($line = <FILE>){
    print "$line";
  }
  close FILE;

  open (FILE, "more /proc/meminfo|grep SwapFree |");
  while ($line = <FILE>){
    print "$line";
  }
  close FILE;

  open (FILE, "more /proc/meminfo|grep MemFree |");
  while ($line = <FILE>){
    print "$line";
  }
  close FILE;


for ($j=0; $j<$numsets; $j++){

`insmod emcModule.o`;
open (FILE, "tail -15 /var/log/messages |");
while ($line = <FILE>){
   print "$line";
}
close FILE;

my $queries = 0;
for ($i=0;$i< $repetitions; $i++){
    $transaction = rand(24) % 24;

    if ($transaction < 10){
        $queries = $queries + 5;
        $returnval = fork;
        if ($returnval == 0){ # child
            `./neworder.pl`;
            exit;
        }

    }
    elsif ($transaction < 20){
        $queries++;
        $returnval = fork;
        if ($returnval == 0){ # child
            `./payment.pl`;
            exit;
        }

    }
    elsif ($transaction == 20){
        $queries++;
        $returnval = fork;
        if ($returnval == 0){ # child
            `./checkorder.pl`;
            exit;
        }
    }

    elsif ($transaction == 21){
        $queries++;
        $returnval = fork;
        if ($returnval == 0){ # child
            `./deliveries.pl`;
            exit;
        }

    }
    elsif ($transaction == 22){
        $queries++;
        $returnval = fork;
        if ($returnval == 0){ # child
            `./stock.pl`;
            exit;
        }

    }

if ($i % 100 == 0){
  open (FILE, "more /proc/meminfo|grep Cached |");
  while ($line = <FILE>){
    print "$line";
  }
  close FILE;

  open (FILE, "more /proc/meminfo|grep SwapFree |");
  while ($line = <FILE>){
    print "$line";
  }
  close FILE;

  open (FILE, "more /proc/meminfo|grep MemFree |");
  while ($line = <FILE>){
    print "$line";
  }
  close FILE;
  }

  wait();

}

print "========================================\n";
print "Finished transactions.\n";
`rmmod emcModule`;
print "Number of total transactions = $repetitions\n";
print "Number of total queries = $queries\n";

sleep(2);
open (FILE, "tail -20 /var/log/messages |");
while ($line = <FILE>){
   print "$line";
}

 if ($j < $numsets-1){
    sleep $repetitions * 2;
 }
}


close FILE;
```