Database Economic Cost Optimization for Cloud Computing

Adam Conrad
Department of Computer Science
Brown University
Providence, RI

adam@cs.brown.edu

ABSTRACT

As the demand for cheaper commodity machines and large-scale databases rise, organizations have turned to cloud computing, or simply "the cloud," as the solution. One such service, Amazon Web Services, aims to provide large-scale computing and storage services in a virtual fashion by creating a $2 billion infrastructure. Over the course of the past year, our research group has tested and deployed this technology utilizing Amazon's Simple Storage Service (S3), Elastic Block Storage (EBS) and Elastic Compute cloud (EC2) products. This paper aims to provide a road map for deploying MySQL with sample data on EC2, as well as initial experimental results and findings on how to efficiently optimize the selection of Amazon's Web Services products for monetary cost. Finally, we provide a mechanism for achieving money cost performance improvements over a traditional query optimizer.

1.INTRODUCTION

The goal of this project is to modify the MySQL optimizer to define a cost model for deploying a database on the Amazon cloud. We aimed to achieve the following:

- Deploying MySQL on Amazon's EC2 and EBS services.
- Add a statistics collection mechanism to MySQL to trace and monitor the interaction of MySQL with Amazon Web Services (i.e., EC2 and EBS).
- Use the above statistics collection process to gather data for a given workload, namely the TPC-C Benchmark.
- Test various optimizations that could potentially improve the operational cost of MySQL.
- Create a meta-optimizer based on these trials to accurately depict cost for transactions and create plans designed to pick Amazon cloud instances that minimize cost.

Further, we have conceived the following specific optimizations in considering the final step of the project:

- IO
  - Database files - Accessing rows in a table.
  - Index files - Add, read, write, remove a new key from the index file.
  - Transaction logs - These files include info on the transactions (e.g. updates) performed on the RDBMS. They are used to recover the RDBMS after a failure of the server.
  - Temporary tables - Temporary storage of the intermediate query results/table. This occurs when a query requires access to a large number of rows (or a sub-query created large intermediate tables) that cannot be stored in memory.

- CPU usage - Understanding the percentage of CPU usage for a given workload, and how it affects the query response latency.
- Memory usage - Monitoring usage of the buffer pool and other caches and the distribution across different types of data.
- Disk usage - What is the required disk space for our workload? This includes the space

required by indexes, materialized views, intermediate query results, or temp tables.
- Compression - What compression levels minimize storage on the cloud and simultaneously maximize efficacy for IO.
- Query response latency - Determining the average response latency of our workload for various query types.

Overall, we explore all of these goals and their results are outlined in more detail below. We explore a hands-on understanding of the Amazon Web Services infrastructure, as well as the inner workings of the popular MySQL database and the TPC-C benchmarking suite. We successfully deployed 32 and 64-bit EC2 instances with EBS from a blank Cent-OS Linux image at a very low cost. Total charges for the entirety of the experiments were surprisingly cheap, at under $1000 for the entire academic year. For IO, we created a lightweight statistics collection mechanism on top of the monitors built into the TPC-C benchmark. We tested basic effects of IO and memory modifications to increase performance and decrease the cost of operation on the cloud. We tackled more advanced database constructs such as materialized views and compression to provide a robust solution to cost reduction. Finally, we began construction of a meta-optimizer designed to create query plans that minimize monetary cost given a minimum quality of service and a sample budget.

2.DEPLOYING MYSQL

The first step in the process was to choose the best database to monitor over the cloud. We wanted a lightweight, yet robust open-source database with ample features that enterprise-level databases demand, such as materialized views. A database that performs well for transactions and has readable code was a must to enable on-the-fly modifications. These criteria brought us down to two databases, MySQL and H2. While H2 was simple and well-documented, it was clear through previous experimentation that MySQL was our choice because of its popularity, and would provide a more appealing experiment to real-world consumers.

As of the writing of this paper, MySQL 5.1 Source is the latest stable version used in this experiment. Our first step was getting to know the inner workings of this database to begin to understand how it should be modified for use on the cloud. Since IO is among the greatest concern in reducing cost for databases on Amazon, the first task was to understand the various storage engines of MySQL. By default, MySQL uses the InnoDB storage engine for transaction processing, and MyISAM for analytical data mining. We analyzed the documentation to determine that InnoDB was the storage engine of choice. After understanding and modifying the source code, we pushed the database onto the blank Cent-OS images to begin testing the database on Amazon.

3.BENCHMARKING

Deploying the TPC-C benchmark was the next step. The Transaction Processing Council produces four benchmarking tools designed to stress test relational databases. The default benchmarking solution by the Council is TPC-C which we finally chose as our sample workload. Due to new constraints imposed by the Transaction Processing Council, TPC-C is not available, but has an open-source equivalent known as DBT2. DBT2 provides a similar sample schema and data scalable from megabytes to several terabytes.

The most difficult challenged faced was deploying TPC-C/DBT2. Unfortunately, the documentation on how to successfully deploy DBT2 is non-existent. It wasn't until some research that we found independent documentation[1] outlining the steps to successfully deploy the data. From significant Linux distribution-based issues to the delicate

nature of cloud computing, DBT2 refused to deploy on any sample image we gave it. Subsequently, we drop our original configuration on Fedora 9 for Cent-OS. Much of the time was spent simply repeating the steps outlined in [1] to attempt a working run of TPC-C. Next we integrated the benchmark with our custom source build. After weeks of trial and error, we finally managed to execute a workload on the schema and produce transaction results. We then created new images of our changes, and had 32-bit and 64-bit versions of Cent-OS successfully running MySQL with TPC-C on the cloud. Over the course of the year, the team added more images to accommodate certain facets of MySQL, such as compression and materialized views.

4.OPTIMIZATIONS AND RESULTS

4.1Buffer Pool and Indexing

With MySQL deployed on the cloud with data from TPC-C, we start with simple optimizations of the database. Since the majority of costs on EC2 are attributed to puts and gets, reducing IO is paramount in reducing overall operational cost. Our test plan is initially limited to the 32-bit instances of EC2, M1.Small and C1.Medium. M1.Small is the smallest machine available from Amazon and is designed to be used for memory-intensive workloads. C1.Medium is the next-smallest machine and is designed for computation-intensive workloads. Exact specifications for these machines and their 64-bit counterparts are available at <http://aws.amazon.com/ec2/#instance>. For both sizes, each runs two separate instances: one with an EBS volume attached, and one without. Besides the obvious benefits of persistent storage on EC2, evidence has shown that, when utilizing EBS, "IO rates can be multiple times faster than ephemeral storage and even local disk IO"[2]. For each of these four instances, four tests were executed:

1. Naive - Tests the out-of-the-box version of MySQL without IO optimizations. This test is designed to see if the location of data, log and temporary directories on local storage will require more IO than mounted on an EBS volume. In addition, it tests to see if this reduced IO is more significant than the cost of the additional overhead of a persistent storage medium.
2. Augmented Buffer Pool Size - Tests to see if an increased buffer pool size decreases total read/writes across transactions. According to MySQL documentation, the buffer pool should be set to 75% of main memory, along with a log file size of 25% of the buffer pool size. M1.Small and C1.Medium have 1.7GB of memory, so a buffer pool size of 1.275GB and a log file size of 318MB were used.
3. Data Indexing - Indexing data is a rudimentary procedure for improving the speed of operations on a database table by creating a copy of part of a table. Though creating and storing the index requires more space and IO, the hope is to reduce total IO over the life of an instance by providing faster lookups, thereby reducing the IO count of transactions. A simple index was used by creating a key on the new_order table .
4. Data Indexing with an Augmented Buffer Pool Size - Combining steps 2 and 3.

After personally experimenting with the benchmark to find the parameters to properly stress the system, we ran these tests with the DBT2 dataset of 20 warehouses with 1.4GB of data, stressed under 20 threads each running 20 concurrent transactions with a benchmark processing time of 300 seconds. Results are based on the performance of the speed of the queries, indicated by the number of New-Order Transactions Per Minute (NOTPM) and the average transaction response times across the tables. Only the EBS instances include further IO statistics from the `iostat` system command. EBS instances are charged for their IO because of their data located on persistent storage, so this extra information is included.

The initial results coincide with our hypothesis: correct buffer pool modifications and indexes

provide increased IO performance when coupled together. Modifying the buffer pool improves performance by two whole orders of magnitude alone, while indexes only double performance. We believe the reasons for these results are that default buffer pool size is more than 150 times smaller than the optimal size for the small and medium instances. There is a great deal of room for improvement by increasing buffer pools, however creating an index is two steps forward and one step back. Indexes clearly make lookups quicker, but require more space as well as more IO to create.

Our results also indicate that the Medium instance requires 66% of the time on average to complete. We expected better results from the Medium instance because with the same amount of main memory, these units have 5 times the number of CPUs. These tests appear to indicate that main memory is a more significant factor in IO performance than raw CPU power.

It is questionable whether or not Amazon's claims that mounting an EBS volume for persistent storage truly increases IO performance. When the buffer pool size is properly increased, the EBS instances perform worse by an entire order of magnitude. In the other cases, EBS performs 5.5 times better on average. Unfortunately, our EBS mounts refused to run TPC-C on the medium instances, further proving that the relationship between MySQL and the DBT2 dataset is a fragile and unpredictable one.

From our results, we can conclude that appropriate database optimization prior to uploading to the cloud will decrease IO by several orders of magnitude and potentially remove thousands of PUTS and GETS from running (and charging) on the cloud. Our preliminary data indicates that the most important factor in choosing an instance should be the amount of available main memory. If consumers can achieve an acceptable quality of service response time at a given quantity of memory, the cost of running a higher-performing machine does not match the increase in price, and should be avoided when possible.

4.2Tuning Workload Parameters

Our next task was to tune DBT2s workload parameters to properly stress all instances. We learned that all instances were returning the same level of throughput for each instance, which meant we were under-utilizing resources for even the smallest instance. We began navigating through DBT2 documentation to determine a new set of parameters, and through rigorous experimentation over the following weeks, determined a new standard for testing. To properly stress the entire cloud and show different results for each instance, we must now run 100 threads concurrently and extend the processing time to 3600 seconds per workload run[3].

4.3Materialized Views and Compression

The third optimization was to determine the efficacy of advanced database techniques such as compression and materialized views. We investigated how to deploy materialized views for read-intensive transactions in MySQL. We looked into two avenues for materialized view implementation in MySQL. The first option is the `VIEW` command, only available in later editions of MySQL. This allows for a more traditional materialized view, defined by MySQL. Unfortunately, after implementing a sample view to test on the system, we determined it is only a temporary table and gets deleted every time we stop the MySQL daemon. Views have very limited support for updates, making it hard for us to enforce ACID properties. Finally, they have limited support for joins and do not understand aggregate functions (e.g. `MIN, MAX, SUM`)[4].

Our second option is to simulate materialized views. We accomplish this by creating a table pre-computing our desired query and creating triggers in MySQL to issue updates whenever the tables involved change. Instead of triggers, we optionally create procedures that update the table every *n* periods. This is an acceptable approach

because TPC-C has relaxed consistency requirements for its read-intensive operations.

We ran our simulated materialized view stored procedures and triggers with a modified TPC-C workload. Throughout our experimentation it became apparent that the write-intensive TPC-C test does not accurately depict the typical workload of web applications, which are much more read-heavy. Therefore, in continuing with the experiments, we included a modified version of the TPC-C workload that balanced queries in favor of the read-intensive. In the read-intensive workload we have an order mix dominated by read-heavy transactions (40% each). Ordinarily they make up about 8% of the workload.

Our results showed reduced query latency by an average of 45% for the queries we modified. This is due to omitting many expensive joins. It decreased reads by about 1%, but increased writes by 23% because it spent a considerable amount of IO issuing updates.

In parallel with investigating materialized views was compression for InnoDB tables. Compression in MySQL involves compacting data at the table-level. We increase CPU utilization to compress and decompress the data into user-specified page sizes for the benefit of smaller databases and reduced IO to improve throughput, our greatest cost on the cloud. Lower IO is achieved through fewer reads and writes needed to access the user data[5].

Experimentation for compression included testing both read-intensive and write-intensive TPC-C workloads across all tables of the schema. Each run was duplicated for each available compression level available in InnoDB, ranging from 16k all the way down to 1k page sizes. The results can be summarized in the following diagrams:

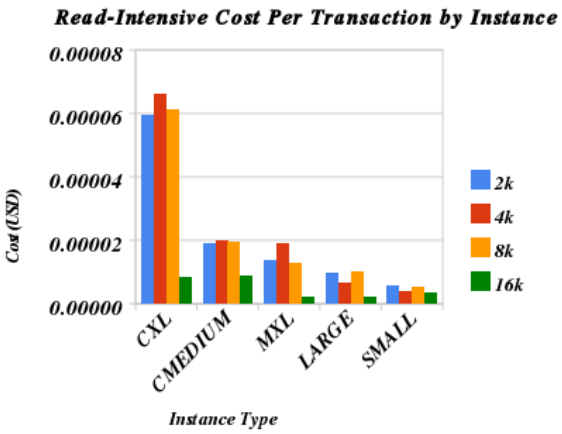

Figure 1: Read-Intensive Compression Cost

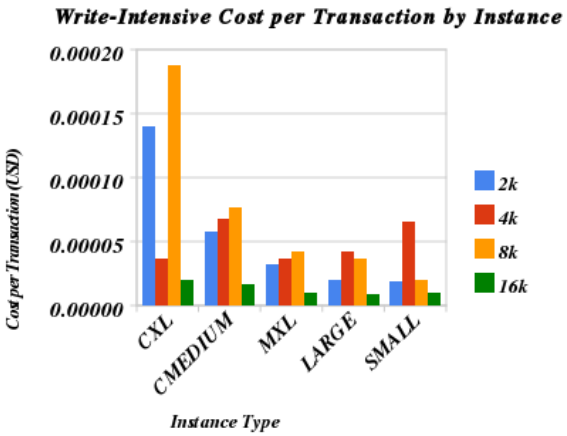

Figure 2: Write-Intensive Compression Cost

As we can see from Figures 1 and 2, the least-aggressive compression provides the best cost savings. Stronger levels of compression create a significant burden on the system due to the added expense of having to decompress and re-compress even tighter page files. The trend shows that beyond 16k, as we increase the level of compression, cost per transaction decreases, but never comes close to matching the cost savings of the mildest 16k compression setting. Preliminary results also show that even the modest 16k compression size decreases the storage size by 66%. With compression, we get the benefits of reduced storage size, reduced IO, and overall reduced cost if we use compression in moderation.

4.4Modifying the MySQL Query Optimizer

Finally, we observe how a modified MySQL query optimizer can reflect money cost to help determine the best instances to run on the cloud. The query optimizer regards everything as a join, even when it contains only one table. It has a few ways of searching for the best query execution plan. All of these searches revolve around a cost function called `best_access_path`. This function, located in the SQL source code of MySQL, decides what access methods to use by looking at a table's index(es), partitioning and allowed cache space. It looks at a proposed table to add to the end of a partial query execution plan for a join. It determines the best way to access the proposed addition using a series of complex calculations. At the highest level it is deciding whether to use an index or a table scan and how to access it. It compares the projected cost of using one of these optimizations to different types of page scans, such as range scans. It then picks the best access method for a given table, based on the context of what is passed in, such as how many records or select statements it thinks it will execute. It may accept an empty set as input if a table is being proposed as the first one in the query execution plan.

Within this function, there are two parameters that we want to tweak to reflect our costs accurately. The first are the IO reads which are the base unit of optimization in the system. We can modify this by either putting a constant in front of all of the optimizer's read assessments in `best_access_path` or just modifying the other constants accordingly. The second is the CPU cost, contained in the variable `TIME_FOR_COMPARE`. The query optimizer estimates that each comparison will take about 20% of the time of one read. Thus, in order to accurately estimate this we need to know both how many comparisons per second the instance can accomplish and what its read speeds are from the first parameter. These two parameters can collectively help us decide the performance of each instance. Costs from our modified optimizer can be calculated using constants determined by the performance of these parameters on each instance.

To simulate these variables on the instances, we obtained a benchmarking tool called UNIXBench. UNIXBench is a universal benchmarking service designed to stress the hardware components of a UNIX-based computer. We chose two tests from UNIXBench that mimicked the above-mentioned parameters. One was an arithmetic test for double comparison similar to the comparison operator. The other was a file system buffer reader operating with a 256-byte buffer, exactly the same buffer size used for IO operations in MySQL.

Each test was run 50 times on each of the 5 instances, as well as their EBS counterparts. Units were converted accordingly to provide comparison between CPU data and IO data. The importance of the ratio is to illustrate the number of CPU operations per bytes read. The basis of all computations in the MySQL cost estimator is bytes read. Therefore, this ratio provides a unit for deciding the cost performance (in bytes read) for each instance.

Our results from the UNIXBench runs, summarized in Figure 3, show on most IO types, it is beneficial to go off EBS for cost. In the CPU case, CPU is unaffected by CPU on virtually all instances. For the most part, EBS is not beneficial to performance, and is very similar in all cases but C1.XLarge. However, EBS does play a significant role in monetary cost, and is severely detrimental to cost performance.

Given all of the above, we can begin to construct our meta-optimizer to additional compute monetary cost. We do this by forking and optimizing for each instance based on manipulating the `TIME_FOR_COMPARE` variable and evaluating how much it will cost. All of these variables are derived from the *x* cents per hour we are using to run a virtual machine on AWS. Since latency is the prime decider for the standard optimizer, these results will be very comparable to traditional query optimization with one difference: we need to multiply the projected

latency by each hourly dollar cost to determine the overall value of a query (i.e. for each instance $i \rightarrow C_i$ = latency(workload) * $hourly\_rate_i$). The cost variables are as follows:

- CPU latency – We are charged per hour per instance. This drives our latency and is an important cost. CPU latency varies in terms of compute units on AWS, but we have quantified them using double comparison benchmarks.
- IO latency – This also is reflected in the amount of time we spend on the cloud using EBS or S3. Our latency impacts our query latency.
- Buffer Pool Cost – Buffer pool size varies by instance. This is used in the latency estimates of the traditional optimizer.
- IO (EBS) – Here we are charged $0.10 per million IO requests. An IO request occurs for every block we read/write. We are using XFS for our EBS volumes, which default to 4096 bytes per block. We calculate this is to presume we are byte-aligned and for each set of reads we can calculate $Cost_{IO}$ = tuples_read * size_per_tuple / 4096. For future work, there may be additional reading overhead we need to consider such as index reading.
- S3 Storage – We pay a certain amount per month to store our images and more if we snapshot them for backup. We trade off between paying for S3 storage or EBS, but some of the benefits are harder to quantify such as the persistence of EBS.
- EBS Storage – We are charged $0.10 / GB / month. It is minuscule in comparison to our instance costs, but still bears consideration. It also makes a quantitative comparison for S3 storage at various levels of snapshotting.
- Cost to transfer data in/out of the cloud – This is a relatively fixed cost for us, but one we should consider nonetheless.

.

With our variables we can determine the cost for `TIME_FOR_COMPARE`. We do this for each instance and have best_access_path return the cost of access for each instance after having the `TIME_FOR_COMPARE` ratio passed in. `Best_access_path` makes available the number of records and IO that it projects the query will use.

A modified version of the greedy search calculates the projected cost with the following formulas:

EBS-Cost = `0.1*(IO_in_bytes/bytes per IO operation)/1,000,000 + rate_per_hour*((IO_in_bytes*(IO_rate) + records * CPU_comparisons / sec)/3600 ).`

Non-EBS-Cost = $rate\_per\_hour_{instance}$*((*IO_in_bytes*/ $(IO\_rate_{instance})$ + *records* / (CPU_comparisons / sec)instance)/3600).

We then compare our results to those of the traditional optimizer.

We check to see how often our query plan is equivalent to traditional optimizer. We have considered the following optimizations explored throughout the year to minimize costs:

- Indexes - This tool is reflected in terms of space paid for. On both S3 and EBS this is negligible in comparison to the cost of renting an instance or EBS accesses. Finding the utility of each index and selecting the best ones is the greatest challenge. Adding an index does not modify our query execution plan in many cases. Still, it will save money in latency reductions, usually in a manner that greatly outweighs the cost of storing it.
- Materialized Views - Materialized views help pre-compute the most common read-intensive queries. Well-designed materialized views save enough in IO to amortize the cost of maintaining a view. Consequently this is most beneficial in a read-intensive environment. In order to

determine whether a materialized view is a good choice for a given workload we calculate a break-even point for the projected cost of our query. Materialized views may be useful in cases where the select statement and predicates only access a small percentage of the columns in the original tables, saving IO.

- Compression - Compression helps save IO at the cost of more latency. This trade-off benefits the clouds, but not traditional optimizers where latency is the primary concern. More intuitively, it saves space and thus money on the cloud.

Implemented together for a given dataset, our preliminary results show that our meta-optimizer can predict the same plan or better than the traditional optimizer for 60% of the queries. This means that our meta-optimizer can choose the same or cheaper instance to run a given query 60% of the time more than the standard MySQL optimizer. Of course instance selection is workload dependent, so for the future we must abstract our meta-optimizer on a higher level to evaluate whole workloads for instance allocation rather than individual queries.

5.CONCLUSION

Cloud computing is a relatively new and highly desirable infrastructure for both businesses and researchers. Numerous papers have previously explored the cloud, specifically the Amazon Web Services, to determine the efficacy of such an architecture. As of the writing of this paper, we are the first group to not only deploy and fully stress a database system on the cloud, but also to analyze how to minimize the costs of deploying an entire database system, in our case MySQL, on Amazon.

We utilized a standard database benchmarking tool to stress the various instances available on Amazon. Gathering information on the bottlenecks of performance, such as IO and CPU, we were able to generate a meta-optimizer that additionally evaluates monetary cost based on Amazon's cloud services pricing structure. The uniqueness of our approach lies in our priority for cost savings over maximizing performance. Given a budget and a minimum quality of service, this optimizer has already proven to choose the same, if not better, option for instance allocation than the standard query optimizer for MySQL for more than half of all query plans, even at the expense of sacrificing performance for cost.

Going forward, we can continue to make strides in tuning the optimizer to continuously provide accurate and cost-saving choices for instance selection on the Amazon cloud. This contribution has a truly monumental impact, with the potential of saving businesses hundreds upon thousands of dollars over conventional instance acquisition and allocation.

6.FUTURE WORK

This project aims to be a stepping-stone for future work on cost reduction of any cloud computing architecture, such as Windows Azure and Google App Engine. We encourage continued research to further explore database optimizations for CPU, disk and query response latency in an aim to achieve optimal configurations of databases on all cloud interfaces. Specifically, as the next steps we would like to find an interface to generalize our optimizer to any DBMS. Further, we want to explore read-intensive OLAP architectures and data warehousing, which more accurately depict the workloads of web applications. In addition, we hope to explore an intelligent advisor/optimizer that can dynamically allocate instances to the database as the data changes from user input over time.

7.REFERENCES

[1] "MySQL DBT2 Benchmark on EC2" http://blog.dbadojo.com/2007/08/mysql-dbt2-benchmark-on-ec2-part-1_31.html

[2] "Running MySQL on Amazon EC2 with Elastic Block Store" http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1663

[3] "EC2 Results for Stressing DBT2 " http://spreadsheets.google.com/ccc?key=p106rdN5ml86j58V8wFMOGA

[4] "Updateable and Insertable Views" http://dev.mysql.com/doc/refman/5.0/en/view-updatability.html

[5] "InnoDB Data Compression" http://www.innodb.com/doc/innodb_plugin-1.0/innodb-compression.html

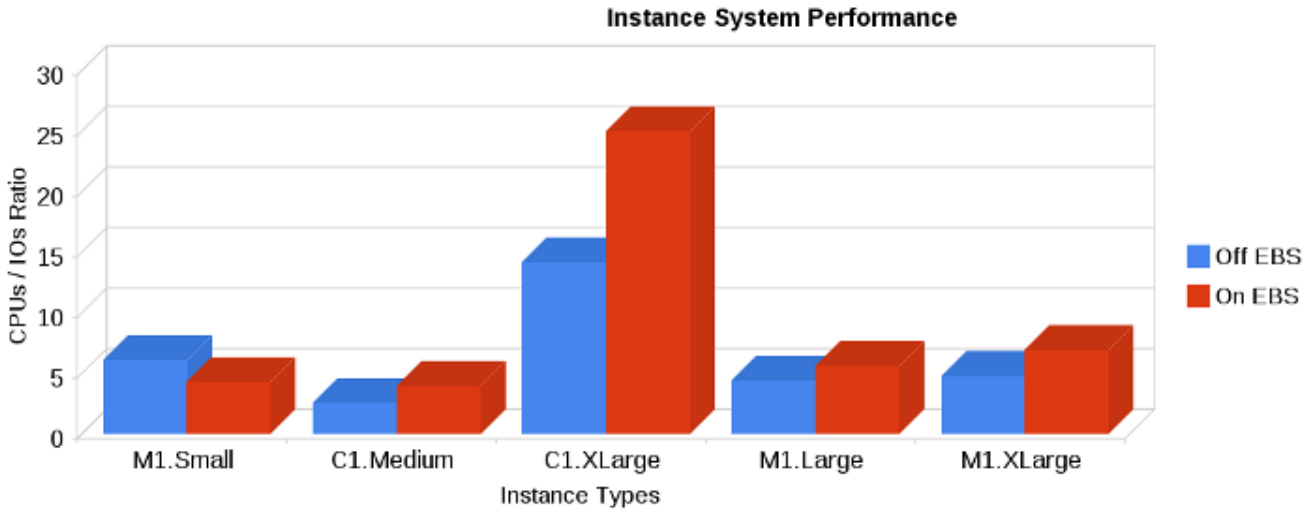

Figure 3: UNIXBench Total System Performance