Determining the Number of CPUs for Query Processing

Fatemah Panahi   Elizabeth Soechting
CS747 Advanced Computer Systems Analysis Techniques
The University of Wisconsin-Madison
fatemeh@cs.wisc.edu, eas@cs.wisc.edu

Abstract

The purpose of the project is to determine the number of CPUs needed to satisfy a target query response time in a query processing system. Several iterations of modeling were performed to validate against emulation measurements. Ultimately, we modeled the system as a single class mixed model which very closely validated against the measurements.

1. Introduction

Companies that provide query processing services must be able to accurately determine what each customer's demand would be on their system. This serves to make the provided service customizable, which is desirable for customers who will be paying for the service. It is also important for the customer to understand the requirements inherent in the query processing service in order to make the right choice in purchasing such a system. The purpose of this model is to help customers identify the number of CPUs to meet their processing needs based on the specific queries run on their system. This is especially important for the customer since they are charged on a per CPU basis.

In order to help the customer predict the number of CPUs needed, we use an analytic model of the system. The advantage of an analytical model is that it can be easily applied to different scenarios/customers with low cost. Moreover, it will help the customers to understand and measure the parameters of their system that are critical for determining the performance they expect from the system.

We were presented with an initial model the company had developed which under predicted the number of CPUs needed and they had been told was incorrect. We were asked to model the system so that it accurately predicted the number of CPUs. We went through many iterations of the model, from an open model to a closed and finally a mixed model. Validating the emulation data provided by the customer before starting the project was very valuable as it gave us insight into the system. We were immediately aware that something was missing or not modeled in the system.

For us to complete this project, it was very important to communicate well with the company and ask the right questions that would help us refine our model. A good background in Databases helped us to rule out many issues that might have been the cause of inaccuracies in the model. This enabled us to find a possible explanation for the discrepancy between the analytic model and the measurements.

We found that the cause of the under prediction in the analytic model was that we had not modeled the Java Garbage Collection (GC) tasks that periodically run and increase the CPU utilization. This lead us to a mixed model where the garbage collection tasks were modeled as an open class of customers that entered the system, used some resources, and left the system.

The resulting model validated very well with the emulation results which lead us to conclude that the Java Garbage Collection was the missing component in the analytic model. This model was tried at different customer sites later by the company and proved to be an accurate representation of the system.

The rest of the paper is organized as follows. In Section 2 we discuss the architecture of the system. In Section 3 we discuss the parameters to the model and in Section 5 we discuss the calculations made from the standalone measurements provided by the customer. Section 6 describes our methods employed for data validation. We discuss the implementation of

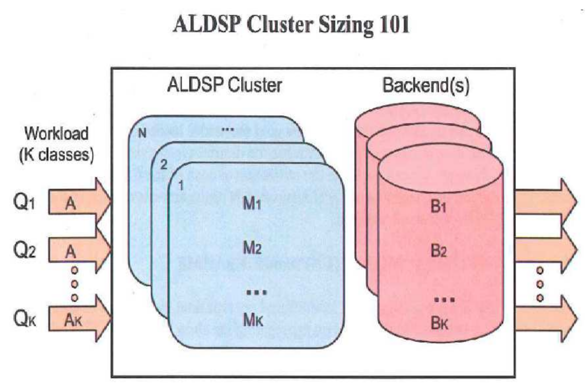

Figure 1. Initial open class model

our model in Section 7 and we present our results from these models in Section 8. We then discuss related work in Section 9 and future work in Section 10. Finally we conclude in Section 11.

2. System Architecture

While we do not model the entire system (shown in Figure 1), it is important to understand the system as a whole. Queries come into the system, they are serviced by the CPUs in the ALDSP cluster and the backend, and they leave the system. We assume perfect load balancing of the queries on the CPUs. The backend retrieves the required data for the queries to complete and interacts with the server. We do not model the backend configuration but rather treat it as a black box and simply know the time a query requires for backend processing. It is reasonable for us to do this because the company assured us that the backend is not the bottleneck and the backend is a system developed by a third party and the company cannot provide us with enough detail to model its internal functionality.

The company had already built a simple open class model of the system. The model is based on Figure 1. However, this model has continuously under predicted the number of CPUs that the customer needs to purchase. Moreover, they have previously shown this model to a consultant who had mentioned to them that this model is incorrect. Thus, our task in this project is to help them identify a model that will accurately validate against the system measurements.

3. Model Parameters

The parameters that were used in the initial model are presented below. The customer is required to group their queries based on the different requirements that each query type might have. For each query type i, they should accurately measure and provide the input parameters for use in the model.

Input

- The percentage of total queries that this query stands for (f_i)
- Expected arrival rate (A_i)
- Desired average response time (RG_i)
- CPU time that the query needs (S_{DSP_i})
- Backend time that the query needs (B_i)

Output

- Number of CPUs that satisfies the requirements specified by the customer.

4. Data

We were presented with two sets of data: standalone measurements and emulation data. All of these measurements were gathered using 2 client CPUs and 2 server CPUs during 5 minute experiments. The client think time (Z) was reported as 1 second.

4.1. Standalone Measurements

In these measurements, the queries were run in a stand alone fashion in the system. The client issues a request, waits for the response, and issues another request. These measurements provided us with the following information about the system:

- Average response time for each query (R_i)
- Throughput (queries/second) (X)
- Client CPU utilization when idle (U_{Client_idle})
- Client CPU utilization when issuing queries (U_{Client_busy})
- Server CPU utilization when idle (U_{DSP_idle})
- Server CPU utilization when serving queries (U_{DSP_busy})
- Number of database rows returned ($dbrow_i$)

It is important to note that in order to get accurate measures, R_i is calculated based on the 5 minute runs with first minute numbers excluded. This is to make sure that the system has stabilized. Experiments are repeated many times and their consistency across runs is validated.

4.2. Emulation Data

In order to validate the analytic model, the company put significant effort into accurately emulating the customer site in house. Different experiments were performed in the emulated system, and the results were given to us. In these measurements, different number of client threads were run in the system that issued queries.

It is important to note that the client utilization was measured, and in all of these experiments, even with 25 threads, it is as low as 6%. This ensures that the emulation system behaves similarly to the case that each of the clients have their own dedicated CPU, which will be the case in the real system.

The system was tested with 1, 5, 10, 15, 20, and 25 client threads in the system. 25 threads was the point at which the system saturated and the throughput started to decrease. We were presented with the same measurements that were listed in section 4.1 for each of the experiments.

4.3. Cyclesoak

In order to accurately measure utilization the company used a tool called *Cyclesoak*. Cyclesoak runs low priority jobs in the system which soaks all the CPU idle time. This allows for an accurate measure of the CPU utilization by subtracting Cyclesoak cycles from the total CPU cycles in a specific duration.

Special care was was taken to run Cyclesoak at times where the system has stabilized to get accurate measures. For example, for measuring U_{DSP_busy}, we start Cyclesoak about 30 seconds after we start running the queries, and stop Cyclesoak about 30 seconds before we stop issuing queries.

5. Initial Calculations

Using the standalone measurements, we were able to calculate measures that we found critical for validating the emulation data and would later be used as input to our model. These measures were S_{DSP}, S_{Client}, and $R_{Backend}$ for each of the query types.

We calculated the measures as follows:

$$S_{DSP} = \frac{2 \times (U_{DSP\_busy} - U_{DSP\_idle})}{X} \quad (1)$$

$$S_{Client} = \frac{2 \times (U_{Client\_busy} - U_{Client\_idle})}{X} \quad (2)$$

$$R_{Backend} = \frac{1 - [X \times (S_{Client} + S_{DSP})]}{X} \quad (3)$$

Note that for S_{DSP} and S_{Client} we use half the total throughput because both the client and the server have 2 CPUs. For calculating $R_{Backend}$, we do not simply subtract S_{DSP} and S_{Client} from R_i because the client spends a small amount of time getting the query response from the network and submitting the query to the network and we do not want this overhead time to affect our measure.

6. Data Validation

Before modeling the system, we validated the data that the company provided to us in different ways. This has two advantages:

- It will give us more confidence on the data that we have
- There might be some inconsistencies in the measurements that will be revealed

The inconsistencies in the measurements could be either due to measurement error, or due to the fact that some aspects of the system are not represented in the measured data. In the second case, we need to find out the missing component and incorporate it in our model if necessary. This practice is valuable since it will give us and the company insight on how the system works and on the aspects of the system that are important to measure.

In order to validate the data, we performed general validations and also applied Little's Law on different boundaries in the system which are explained below.

6.1. General Validations

First we looked for apparent inconsistencies in the data. We observed that the emulated average response time with one client thread in the system is lower than the standalone measurements. These two values need to be the same. Thus, the company validated that the number from the standalone measurements is the accurate measure.

We also calculated the throughput for the emulation measurements with different threads in the system as follows:

$$X = \frac{Total\ queries\ run}{Total\ measurement\ time} \tag{4}$$

We observed that the throughput increases as the number of threads increase, but finally the system saturates and at 25 threads in the system the throughput starts to decrease. This trend appeared reasonable.

It is also desirable to calculate the throughput in different ways and to compare the resulting values with each other. One approach for calculating throughput is using Little's Law:

$$X = \frac{Total\ number\ of\ client\ threads}{Think\ time + average\ response\ time} \tag{5}$$

Based on the data, we could also use Little's Law to validate that the average think time of 1 second that was given to us is accurate using the following formula:

$$Z = \frac{N}{X} - R \tag{6}$$

6.2. Applying Little's Law

We applied Little's Law to different boundaries in the system which can be seen in Figure 2. Knowing that Little's Law must be exact in all system boundaries gave us insight on the aspects of the system that might be missing in the measurements.

6.2.1. Client CPU Utilization The Client utilization was calculated as shown below:

$$U_{Client} = \frac{X}{2} \times S_{Client} \tag{7}$$

In the equation above, total system throughput is divided by number of client CPUs. We multiple the client throughput by the average amount of time that each query spends in the client.

We noticed that the utilization that we calculated using the above formula is constantly lower that the measured utilization. This means that there is an aspect of the system that increases the client CPU utilization that we are not aware of.

The company is investigating possible explanations for this discrepancy. However, since client utilization is not used in our model, we felt comfortable that this aspect of our model is accurate. Client utilization is only measured to make sure that it is low enough to give us an accurate representation of client threads running on independent CPUs.

6.2.2. Server CPU Utilization We calculated the server CPU utilization in a similar manner to the client utilization as follows:

$$U_{DSP} = \frac{X}{2} \times S_{DSP} \tag{8}$$

We observed that the utilization that we calculate is consistently lower than the measured utilization. The discrepancy gets worse as the number of client threads increases. Two possible explanations for this discrepancy are:

- There is a constraint in the system that we are missing
- There are other jobs running on the CPU that we are not aware of

The company was not sure at this point what the reason for this discrepancy is. However, using a lower than actual CPU utilization in the analytical model could be the reason why the initial model always under predicted the number of CPUs that were needed.

6.2.3. Queuing Effect We also calculated total number of clients that are present in the boundary containing the CPU and backend. We applied Little's Law as follows:

$$Q = X \times R \tag{9}$$

We observed that for more than 10 threads there are 3-15 customers in the system, which indicated that customers are queued in the system for resources.

7. Model

After the data validation and some discussion with the company we decided to move from the open model we had initially been presented with to a closed model. We decided we could make this transition because the think time (Z) could be accurately measured in the system and the number of clients in the system at any given time does not change.

We also decided to use a single class of customers but with different types, very similar to the model described in [1]. This approach is valid since the customers (which are effectively the queries) in the queueing system share the same queue and use similar resources. The only factor that we need to consider for each customer type is the probability that a customer (query) is of this type (f_i).

7.1. Closed Model

The closed version of our model, as seen in Figure 2 is represented as follows:

$$X = \frac{N}{Z + R_k(N)} \tag{10}$$

Where N is the number of client threads and Z is the think time.

$$R_i(N) = \frac{1}{CPU} \times S_{DSP_i} \times (1 + Q_k(N))) \tag{11}$$

Where CPU is the number of CPUs in the ALDSP cluster.

$$R_k(N) = R_{backend} + \sum_i R_i(N) \times f_i(N) \tag{12}$$

$$Q_k(N) = X \times R_k(N) \tag{13}$$

We solved the queueing system using the exact MVA approach. This decision was based on two factors:

- The exact solution can be implemented easily for a single class of customers, specially since the maximum number of client threads in our system is 25

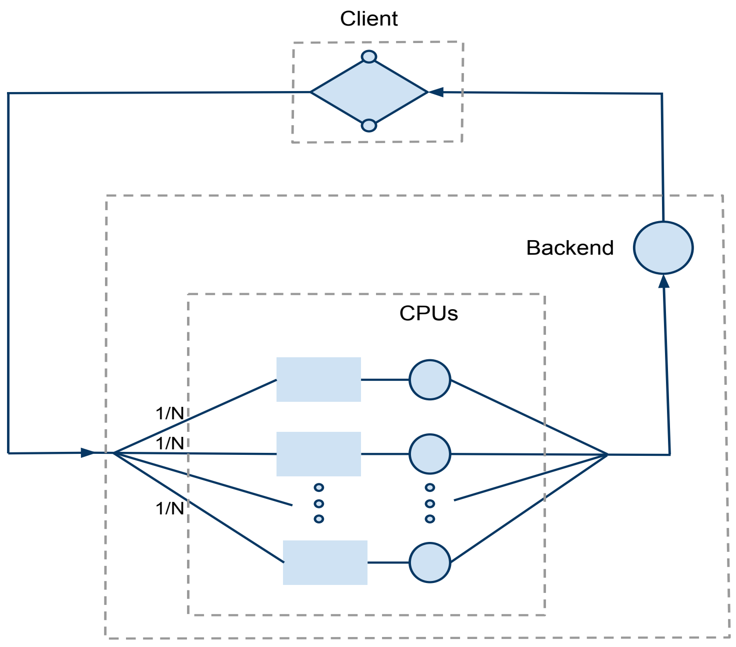

Figure 2. Closed model of our system.

- We had been told there was a problem with the model, and we wanted did not want any error we found to be due to the Schweitzer approximation

Implementation of the above model shows high and increasing percent error in average response time as the number of client threads increases. We constantly get lower average response time and lower CPU utilization.

The above model is a classic MVA model which is fairly simple. Therefore, we were sure that it is exact for the model that it represents. Thus, we concluded that we are missing some aspects of the system in our model.

7.2. Model Discrepancies

We asked the company representative about potential aspects of the system that were not currently being captured in the model. Specifically, we were looking for issues that would increase the server CPU utilization since our model under predicted that measure. We discussed many potential problems including:

- Thread management overhead
- Scheduling overhead
- Cache coherence or overhead
- Context switching overhead
- Thrashing due to high CPU utilization
- Imperfect load balancing
- Main memory conflicts
- Disk I/O bottleneck
- Conflicting transactions in the database system

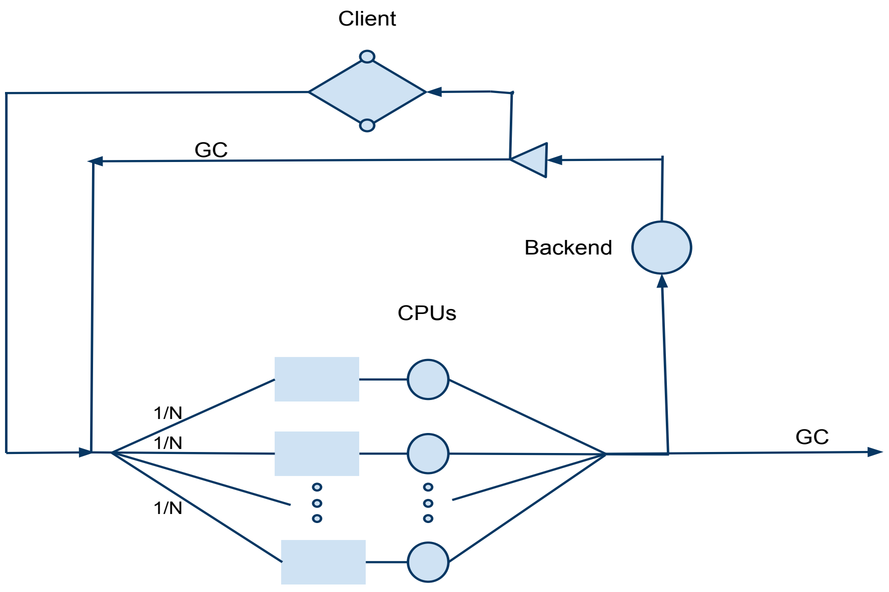

Figure 3. Model of our system with the closed and open classes of customers.

The company assured us with high confidence that none of the above could explain the discrepancy between the analytic model and emulation. They were not sure what could be the cause of discrepancy. However, they informed us that the queries went through a third party middleware when the backend and the server communicate. This lead us to more possibilities to consider. One such possibility is that the middleware could force additional constraints on the system. Also the middleware is written in Java which has implications because of the garbage collection policy.

We were assured that the middleware does not impose any constraints on the system, such as limiting the number of queries that can be submitted from the server to the backend. However, the company felt that Java Garbage Collection (GC) could be causing the increase in the server CPU utilization. This is because Java needs to issue garbage collection tasks to the CPU once memory usage reaches a certain threshold. In addition, the Java garbage collection policy is to stop the execution of the Java program until the garbage collection is complete. Our new model which includes garbage collection is described in the next section.

7.3. Mixed Model

To add the garbage collection overhead to the closed model, we design a mixed model solution. In this model, the garbage collection requests are an open class of customers that enter the system, use some resources (CPU), and leave the system. The modified model is shown in Figure 3.

We modified our equation for calculating $R_i(N)$ by inflating the service time by the utilization of the garbage collection customers in the following manner:

$$R_i(N) = \frac{1}{CPU} \times (\frac{S_{DSP_i}}{1 - U_{GC}} \times (1 + Q_k(N))) \tag{14}$$

To calculate U_{GC} we take into consideration the number of rows that are returned for each query class and the time it takes to garbage collect a single row (μ). We then calculate the amount of time used for garbage collection for each query class based on the number of rows they return.

$$U_{GC} = X \times \sum_i f_i \times \mu \times dbrow_i \tag{15}$$

We solve this model by fixing μ and iterating until U_{GC} converges.

It is important to note that the amount of garbage collection overhead is not proportional to the number of threads. It is proportional to the number of rows that are returned from the backend for each query type and the total throughput. We do not consider U_{GC_i}, only consider a single U_{GC} for all query types because the garbage collection tasks that are resulted from one query can affect all the queries in the system.

8. Results

The customer provided us with emulation data with which to validate our model. From this data we were able to measure the average response time of the whole system and of individual query types, utilization, and throughput. We were also able to apply Little's Result to the data to validate it before ever building our model to ensure the data the customer had given us made sense and there were no glaring measurement errors.

8.1. Average Response Time

We were able to compare both the overall average response time of the system and the individual average response times of each query time since the customer provided us with that data. After changing the model the customer gave us initially from an open model a closed model, we got results that did not compare very well with the measured values. However, after changing the model again to account for garbage collection in the system, we were able to very closely match the measured data. Since the middleware is a third party software and we do not have significant knowledge about its functionality we can never be exactly sure if the garbage collection is the true cause of the discrepancy.

The total amount of garbage collection time per row is very low, 1.94×10^{-5} seconds. We believe this result to be reasonable because we would expect the garbage collection time per row to be very low. However, since this is a value that the customer would normally determine, we would like the company to measure this in their emulation system to see how closely the value matches the value we have determined with our model.

The results from both models and the measured data can be seen in Figure 4. We find that we are much closer to the measured data once we add the overhead caused by the garbage collection into the model. We believe this model accurately represents the real system and could be used to predict the number of desired CPUs for future systems.

We also look at the percent error of the individual $R_i(N)$s which were measured for each (see Figure 5). This is important for two reasons: First, we can validate our model on a finer granularity since we have emulation measurements for each query type. And second, the customer provides us target response times for each of the query types and thus our model should also validate on the individual $R_i(N)$s to be accurate.

For most query types the percent error is low. However for *defaultview1* the percent error is consistently high. This query type has a much smaller number of rows returned from the database than all the other query types, so that could be the reason for such a high percent error.

8.2. Throughput

We are also able to examine how our measure of throughput compares to the value we calculate from the measured data we are given. We determine throughput both for our closed class model and our mixed model to compare to the measured data and report our findings in Figure 6.

We find that our measure of throughput with our closed model is generally too high which was expected since our $R_k(N)$ was generally very low. However, once we change to a mixed model which includes garbage collection, our curves are almost exact. We are able to show the system saturating although not to the degree that the measured values indicate since our throughput does not actually begin to decrease for the number of threads shown here.

9. Related Work

The closest work to our consulting project is [1]. In this paper, the authors use customized Mean Value Analysis (MVA) to model bus contention due to multiple processors using the same bus.

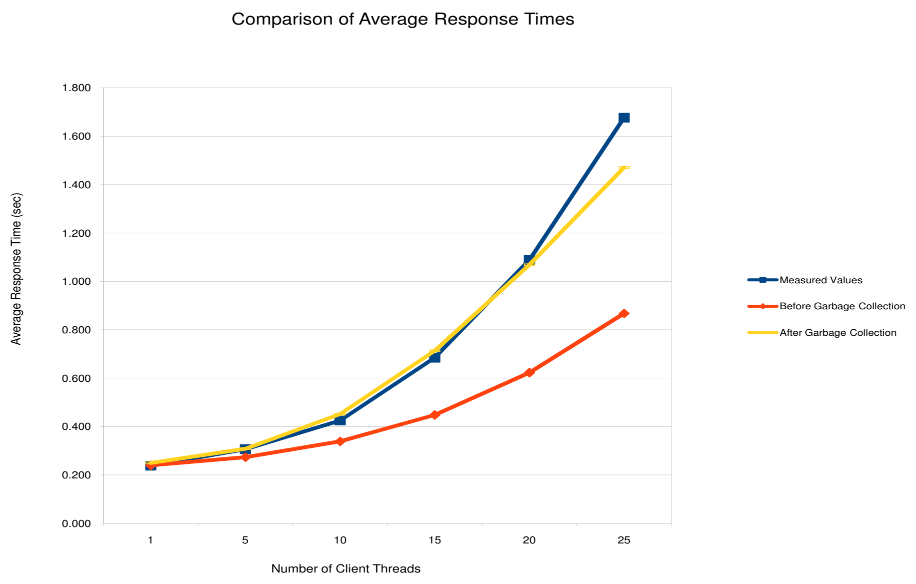

Figure 4. The average response time of the measured system, the initial closed model without garbage collection, and the final mixed model with garbage collection.

	Number of Client Threads					
Query Type	1	5	10	15	20	25
defaultview1	0.282	0.484	0.165	0.292	0.372	0.382
defaultview5	0.071	0.143	0.026	0.083	0.011	0.009
orderbyyear100	0.012	0.092	0.001	0.050	0.050	0.091
orderbyyear300	0.065	0.057	0.040	0.122	0.132	0.207
demograph100	0.094	0.013	0.029	0.040	0.109	0.266
demograph200	0.135	0.070	0.075	0.054	0.696	0.213

Figure 5. Percent error for individual query types.

Similar to our system where clients issue queries independently, processors execute independent code. The requests that need the bus are read, invalidate, and write-back. Read and invalidate requests are analogous to our different query type requests. Since each of the request types have similar behavior in the system, and use the same queue, they modeled the closed system as a single class but with multiple types. The types are differentiated using the fraction of requests that is of that type (analogous to f_i in our system).

Write-backs are modeled as open class that are generated in the system, share the same queue, and leave the system. Write-backs are similar to Java Garbage Collection jobs in our system. The ultimate model in this paper is a mixed model which is very similar to the final version of our model.

10. Future Work

We believe that implementing a Markov Chain solution to the model would be helpful. Having the probability distributions of the queue lengths would provide additional insight to the customers of the company about the demand at the server which can better help them predict the number of CPUs to purchase.

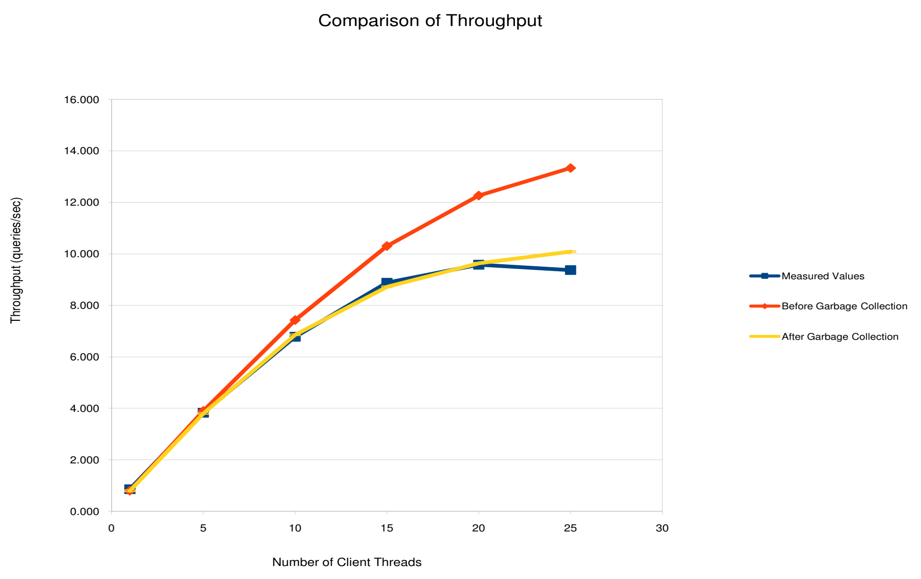

Figure 6. **The throughput of the measured system, the initial closed model without garbage collection, and the final mixed model with garbage collection.**

If they know the maximum number of clients which will be in the queue at any given time and the probability that this situation will occur, they can better predict how necessary it is to prepare for this type of situation. If they will have their highest number of clients in the queue with a high probability, then they need to have enough CPUs to handle this load; but, if they do not, then they can decide whether this situation will occur often enough to merit the purchase of additional CPUs which may not be used frequently but may result in the loss of some customers who would otherwise have used their service.

11. Conclusions

In this project, we built an analytic model to determine the number of CPUs that customers need to purchase in order to achieve a target query response time. We were presented with an initial open class model, which we changed to a closed, and finally a mixed model.

Our initial model did not validate well against the emulation measurements. We found the cause of the discrepancy to be Java Garbage Collection tasks that periodically run and used server CPU time. We modeled the garbage collection tasks as an open class of customers, which lead to our final mixed model. This model validates very well with the emulation.

We believe our results indicate that the addition of the open class representing garbage collection into the model explains the initial discrepancy between the model and the measured data. We also believe that this model can be used to predict the number of CPUs a new customer system requires based on the different query types.

References

[1] M.-C. Chiang and G. S. Sohi. Evaluating design choices for shared bus multiprocessors in a throughput-oriented environment. *IEEE Trans. Comput.*, 41:297–317, March 1992.