Simplifying Layered Queuing Network Models

Farhana Islam, Dorina Petriu, Murray Woodside

Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada

{fislam|petriu|cmw}@sce.carleton.ca

Abstract. The amount of detail to include in a performance model is usually regarded as a judgment to be made by an expert modeler and the question "how much detail is necessary?" is seldom asked and difficult to answer. However, if a simpler model gives essentially the same performance predictions, it may be more useful than a detailed model. It may solve more quickly, for instance. Or a model for a complex sub-system such as a database server may be usefully simplified so it can be included in larger system models. This paper describes an aggregation process for layered queuing models that reduces the number of queues (called tasks and processors, in layered models) while preserving the total execution demand and the bottleneck characteristics of the detailed model. It demonstrates that this process can greatly reduce the number of tasks and processors with a very small relative error.

Keywords: Performance Models, Layered Queuing Networks, Model simplification.

1 Introduction

A performance model may include a very large amount of detail about resources and operations, which makes it difficult to create, maintain and understand, and expensive to solve. This is often true of models created from a system design, for example, because the model includes every operation and component. Frequently many of the model entities have little impact on the performance, and can be aggregated or ignored.

This paper considers layered queuing (LQ) models of service systems with distributed and layered operations and resources. It examines a process for aggregating operations and entities in the model, and its impact on performance predictions. The ultimate goal is a process for automatically simplifying a model to an essential core level of detail governed by an accuracy requirement over a range of cases. The first step is to find operations that successfully simplify some details, and this is what is reported here.

The paper examines model-simplification operations that aggregate sub-operations (in LQ terms, activities), operations (in LQ terms, entries), software processes (in LQ terms, tasks), and physical resources (processors). The simplification operations are evaluated by their effect on the system response time or, equivalently, the system

throughput with a finite user population. Aggregation may be vertical (along a calling path) or horizontal (across multiple calling paths and classes of operation). Evidence is presented based on examples, that certain simplification operations introduce only very small errors. Restrictions on simplification that preserve the bottleneck characteristics of the model (which in turn determine its capacity) are investigated.

2 Related Work

In performance models which are product-form queuing networks, there is a powerful and much-used simplification result in the Norton Theorem for Queues [1], by which any subnetwork of queues can be replaced by a single server with a state-dependent service rate. The replacement is exact in the sense that the throughput and delay at the subnetwork interface is the same for the single server. The original result was for a single class of customer, and it was extended to multiple classes in [2]. The exact simplification for product form networks, and approximations that use the same construction technique for other models, can be referred to as flow-equivalent server (FES) methods [3]. When a submodel is replaced by a FES centre, the entire model is smaller and easier to solve, and parameter changes outside the submodel can be studied efficiently. However the FES construction method requires solving the subnetwork many times, once for every user population that it may experience, which does not scale well to large systems with thousands of customers.

Surrogate delay methods (e.g. [3]) replace a subsystem by a delay which is found by solving an auxiliary model. A surrogate delay is somewhat like a FES with a fixed delay rather than a state-dependent rate, but the construction method is different and requires an iterative solution which includes the auxiliary model. Surrogate delays are most useful to address problems of simultaneous resource possession, rather than particularly for model simplification.

When performance models are fitted by regression methods as in [4], a choice must be made for the level of detail in the model and the modeler can select a simple structure to fit (and test the goodness of fit afterwards). This approach therefore automatically includes the question of detail, and can answer it through tests of goodness of fit as discussed in [4] (this reference describes fitting ordinary queuing models but it applies equally to layered models). However this approach cannot be applied to models constructed from a design, before a system is built.

There does not appear to be any prior work on deriving a simplified layered queuing model directly from a detailed one. In particular there is a lack of simplification techniques that avoid the scalability problems of calibrating an FES. This work approximates the system by a model with ordinary multiservers with parameters derived during the aggregation.

This paper demonstrates a simplification process that can be applied on a detailed LQN model with a single class of end users, to obtain a simplified model containing only one non-bottleneck task and one bottleneck task or processor. A number of cases of various structures of LQN models are studied where the performance results are analyzed after applying the proposed simplification process.

Section 3 describes layered queuing network models, and Section 4 presents heuristic principles for simplification using two example LQN models. Section 5 presents application of simplification principles on a case study that presents and compares performance results among different levels of simplifications. Conclusions, limitations and future works are discussed in Section 6.

3 The Layered queuing network (LQN) model

Layered queuing networks (LQNs) are an elegant way to express simultaneous resource possession and are particularly intended to model layered software systems, in which a software server depends not just on its processor, but on other software servers as well [8]. The model represents software components, their interactions and their deployments. An LQN model basically presents software processes as tasks, one or more operations (or service classes) of a process as entries, interactions among different entries as calls or requests for service, and the host processors at which tasks are deployed. Tasks and processors are servers with queues. **Fig. 1** shows an example LQN model of a three tiered (three layered) architecture. For each task, the rightmost rectangle represents the task itself (labeled by the task's name and thread multiplicity m) and the other rectangles represent its entries (labeled by entry name and host demand s_e for one invocation of the entry e). Every task is deployed on a host drawn as an oval. A call from one entry to another is represented as an arrow labeled with the mean number of calls y_{de} from entry d to entry e. A task is a multiserver (the threads are the servers) with a single queue, usually FIFO, to hold all the calls to its entries, thus the calls are indicated to the entries but actually go first to the task queue.

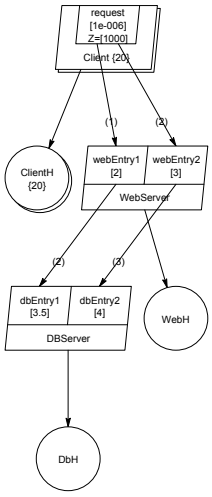


Fig. 1. LQN model of a three-tier architecture

In **Fig. 1**, the LQN model has three tasks - Client, WebServer and DBServer each of which is deployed on its own host - ClientH, WebH and DbH respectively. The 20 users each takes 1000 ms think time (Z) between requests. They are modeled as 20 tasks each running on its own processor ClientH. Both WebServer and DBServer are single threaded tasks and they each have two entries with host service demands indicated in braces (i.e. webEntry1 has service demand 2 ms). A single client operation includes one request to webEntry1 and two to webEntry2. Storage devices are not shown but they can be modeled by a task representing the storage logic (read, write operations for example) running on a host representing the device.

LQN models of real systems can be very large, if they describe systems with many servers, replicated servers, and storage devices. Models with a dozen layers and dozens of tasks are common, and hundreds of tasks may arise in complex cases or with large scale-out by replication. These large models are cumbersome and most of the detail does not impact the performance.

Some asymptotic (bottleneck) properties of the model can be deduced from its parameters and will be used to guide the simplification. Let:

- Y_e = the number of calls to entry e, per user Request. $Y_e = \Sigma_d \, Y_d \, y_{de}$, where the sum is over all the entries, with $Y_{request} = 1$.
- X_e = the service time of one request to entry *e*, including waiting for its host, and waiting for replies to calls it makes to other entries,
- U_h = utilization of each core in host *h*, per user response = $(1/\, m_h) \, \Sigma_{e(h)} \, Y_e \, s_e$, where the sum is over entries of tasks deployed on *h*,
- U_t = utilization of each thread of task *t* per user response = $(1/\, m_t) \, \Sigma_{e(t)} \, Y_e X_{e,}$, where the sum is over the entries of task *t.*

Then the most-saturated host is the one with the largest value of U_h and the most saturated task is the one with the largest value of U_t. The system bottleneck is the entity with the largest value of *S*, provided it is not a client of an entity that also has a large value of *S*. To identify the system bottleneck in a layered system we must consider the possibility of software bottlenecks as discussed in [9]. Considering any task, we say its "servers" are its processor and any tasks that it calls. The bottleneck strength of a task is the ratio of its utilization to the highest utilization among its servers. Then a task is a software bottleneck (and the system bottleneck) if it has the largest bottleneck strength (considerably greater than unity) and also a high utilization (say greater than 0.9). If no task qualifies, then the processor with the highest utilization is the system bottleneck. If there is a software bottleneck and a saturated processor or processors, then there are multiple system bottlenecks; this is uncommon but possible.

4 The Simplification process

An LQN model like **Fig. 1** is simplified by aggregating the activities, entries, tasks and processors, using the following four operations. The goal is to reduce the number of tasks and processors in the model while retaining the externally visible performance measures, in this case the mean throughput and response time seen by the users.

1. Substitute the activities of an entry by a total entry demand equal to the sum of the demands caused by executing the activities. Substitute the calls from these activities by calls from the entry, so for each destination entry the number of calls equals the sum of the calls from the activities.
2. Merge the entries of a task. Thus all calls to these entries are redirected to the merged entry, and all calls from these entries now originate from the merged entry. If this gives multiple call arcs between one pair of entries, they are merged also.
3. Merge a set of tasks deployed on a common processor into one task. The entries of each task are first merged separately, and then the merged entries are merged. The merged call rates are calculated based on the relative throughputs of the merged entries, as weights. The merged task's multiplicity is the summation of multiplicities of all the tasks that are being merged.
4. Merge a set of processors and all their tasks. The set of processors is replaced by a single processor whose multiplicity is the sum of the multiplicities in the set, and the merged task is assigned to the merged processor.

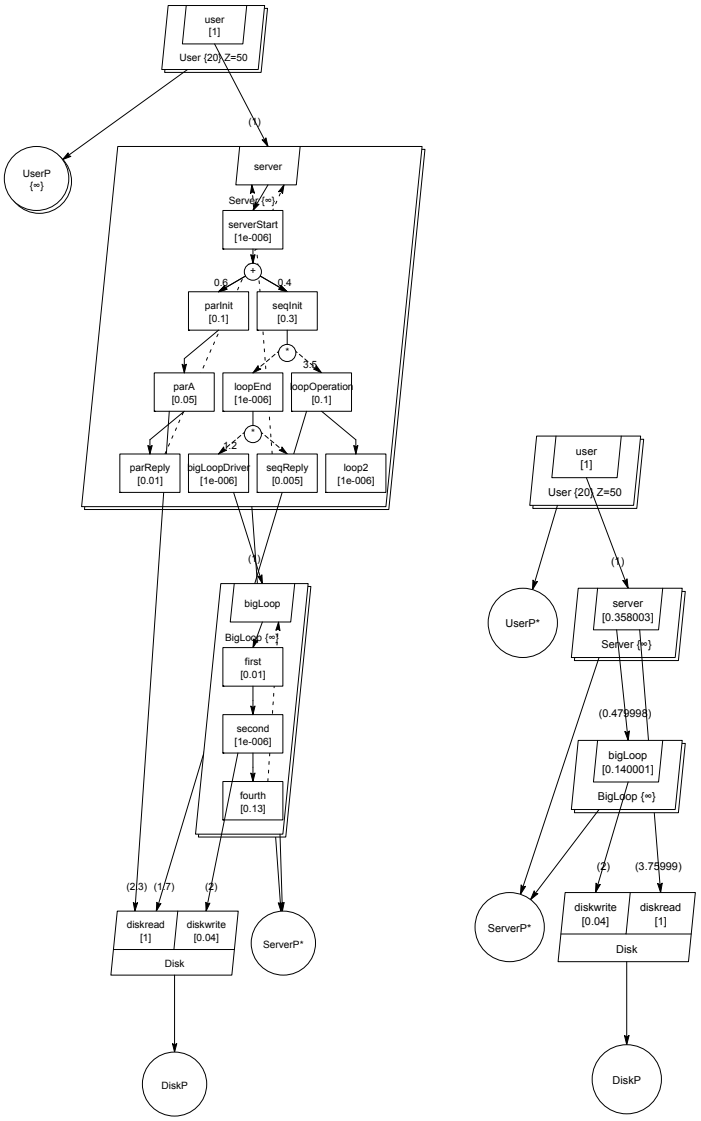
Simplification rules using these operations are applied with the goal of retaining the externally visible performance measures, in this case the mean throughput and response time seen by the users. The rules sequence the operations partly as indicated within the operation descriptions (activities, then entries, then tasks, then processors), and partly guided by the location of the system bottleneck.

The first principle of the simplification rules is to preserve the bottleneck task or processor, since the capacity limit of a system is a key property. Thus operations 1 and 2 are applied to all tasks, but operations 3 and 4 are not applied to a task or processor identified as a bottleneck.

A second principle is to preserve the total workload, so that the total throughput and host demand of a merged entry or task, per user request, is the same as for the entities that were merged. The third principle is to preserve concurrency, by which the total multiplicity of a merged task or processor is equal to the sum of multiplicities of the entities that were merged. These three principles are respected in the description of the operations, given above.

4.1 Details of the Operations: Example 1

The detailed application of the operations, including the parameter calculations, will be described with a running example defined by the LQN model from [6] presented in **Fig. 2Fig. 2**. **(a)**. Each of a number of users ($N = 20) make one visit to the Server task, which has one entry server with a number of activities. Some requests from different activities are delegated to the pseudo-task BigLoop and some are requested from the task Disk for diskread and diskwrite operations. Server and BigLoop are deployed on the same processor ServerP which has a processor-shared queuing discipline. Task Disk is deployed on DiskP with FIFO queuing discipline. From the initial experiments, Disk and its processor are found to be the bottleneck in this model. Thus, Disk and its processor are to be preserved in the simplification process.



(a) Original model from [6] (b) After aggregating the activities

Fig. 2. Aggregating activities in an example LQN model

The simplification operations are applied on this example and described as follows. Some calculations can take advantage of finding a single solution of the model being simplified, and this is assumed to be available.

Operation 1: Substituting activities. In each task *t*, for each entry *e* that has activities in its definition, the activities are aggregated. For activity *i*, let:

s_e, s_i = execution demand of entry *e* (to be found), and activity *i*, (given)

λ_e, λ_i = throughput of entry *e* and activity *i*, in any solution of the model.

w_i = executions of activity *i* per request to entry *e* (this may be calculated by examining the activity graph, or from a model solution as $w_i = \lambda_i/\lambda_e$)

y_{ib} = mean calls from activity *i* to another entry *b* of another task

y_{eb} = aggregated mean calls from entry *e* to entry *b* (to be found).

Then the aggregated execution demand is

$$s_e = \Sigma_i \, w_i \, s_i \tag{1}$$

and the aggregated number of calls from entry *e* to another entry *b* is

$$y_{eb} = \Sigma_i \, w_i \, y_{ib} \tag{2}$$

where the sum in both cases is over the activities of entry *e*.

In the example, in entry server for each activity, the values of (activity name, weight, execution demand) are (serverStart, 1, 1.e-6), (parinit, 0.6, 0.1), (parA, 0.6, 0.05), (parReply, 0.6, 0.01), (seqinit, 0.4, 0.3), (loopOperation, 1.4, 0.1), (loop2, 1.4, 1.e-6), (loopEnd, 0.4, 1.e-6), (bigLoopDriver, 0.48, 1.e-6), (seqReply, 0.4, 0.005). Applying Eq (1) we obtain $s_{server} = 0.358$. Applying Eq (2) for the call from bigLoopDriver to bigLoop, the entry has the aggregated calls $y_{server,bigLoopDriver} = 0.48$. **Fig. 2(b)** represents the model after aggregating all the activities from **Fig. 2(a)**.

Operation 2: Merging Entries. The second operation merges the entries of each task *t* having more than one entry. Let:

s_m, s_k = execution demand of the merged entry *m* (to be found), and of the original entry *k* of task *t*,

y_{kb}, y_{mb} = mean number of calls from entry *k* of task *t* to an entry *b* of another task, and from the merged entry *m* to entry *b*,

w_k = weight of original entry *k* = fraction of all calls to task *t*, that go to entry *k*. From any solution, w_k can be found as $\lambda_k/\Sigma_k \, \lambda_k$, where the sum is over the entries to be merged. Then the service demand of the merged entry is:

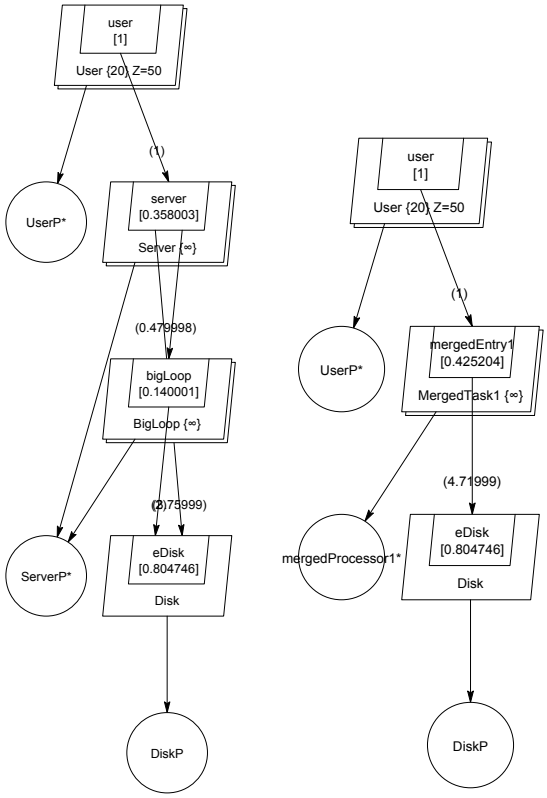
$$s_m = \Sigma_k \, w_k \, s_k \tag{3}$$

and the calls from entry *m* to another entry *b* are:

$$y_{mb} = \Sigma_k \, w_k \, y_{kb} \tag{4}$$

where the sums are over the entries to be merged in both equations.

In **Fig. 2(b)**, only task Disk has more than one entry. So, the values of (entry name, weight, execution demand) are (diskread, 0.797, 1), (diskwrite, 0.203, 0.04). Applying Eq (3), $s_m = 1 * 0.797 + 0.04 * 0.203 = 0.805$. There are no outgoing calls from Disk. The incoming calls are simply transferred to the merged entry (if this results in more than one call from a specific entry, the calls are merged and the numbers summed). **Fig. 3(a)** represents the model after merging entries.



(a) after merging entries (b) after merging the Server and BigLoop tasks

Fig. 3. More merging operations

Operation 3: Merging tasks on the same processor: We consider merging two tasks that share a host. Each task has a single entry (entries have been previously merged if necessary). If one task calls the other, we call it vertical merging, otherwise it is horizontal merging.

Vertical merging: Let

s_a, s_b, s_m = the service demands of the entries *a* and *b* of the two tasks, and the entry of the merged task, respectively.

y_{ab} = the number of calls from entry *a* to entry *b*

y_{ac}, y_{bc} , y_{mc} = the number of calls from entries *a* and *b* to a third entry *c*, and from the merged entry *m* to *c*, respectively.

Then the service demand and number of calls for the entry of the merged task are:

$$s_m = s_a + y_{ab}\, s_b \quad (5)$$

$$y_{mc} = y_{ac} + y_{ab}\, y_{bc} \quad (6)$$

The incoming calls in vertically merged tasks are calculated as for merged entries. In **Fig. 3(a)**, Server and BigLoop both are deployed on the same processor ServerP. They are merged in **Fig. 3(b)** as "MergedTask1" with an entry "mergedEntry1" with service demand of 0.3580003 + 0.479998 * 0.140001 = 0.425204 (following Eq (5)). The number of outgoing calls from mergedTask1 to eDisk is = 3.7599926 + 0.479998 *2 = 4.71999 (following Eq (6)).

Horizontal merging: We call it horizontal merging when there is no calling relationship between the tasks. Just as for merging two entries of the same task, the service demand and the calls of the merged task are computed by Eq (3) and (4), where the entry *k* designates the single entry of one of the tasks to be merged, and the sums are over this set of entries. As in merging entries, the calls into the separate entries are transferred to the merged entry *m* and if this results in multiple calls between a pair of entries, the calls are merged and the numbers summed. There are no additional sets of tasks sharing a processor in figure **Fig. 3(b)**, so this calculation is not applied. For this example the last step would be to possibly merge some of the processors, each having a single task. This step will be discussed in the second example in Section 4.2.

Table 1. Performance results of three simplification operations of Example1

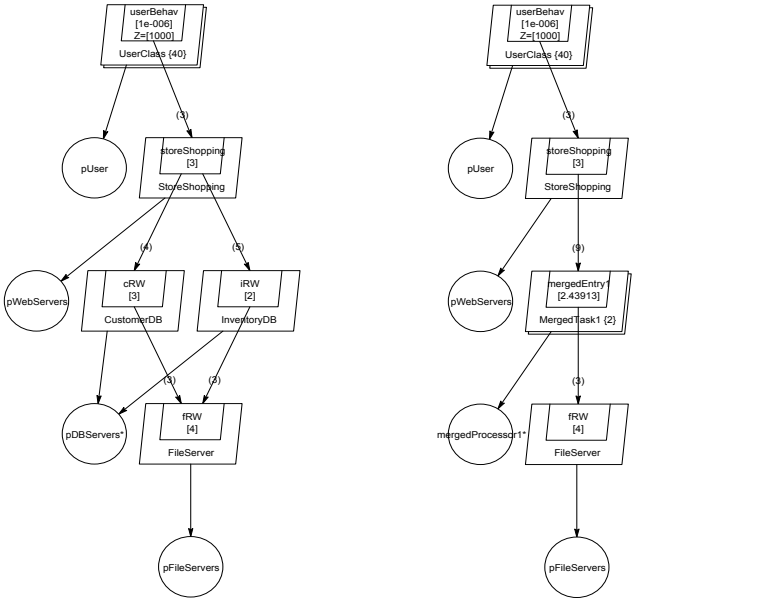
Model "SRVN"	Sys. Throughput	Sys. Response time	U_{Server}	$U_{BigLoop}$	U_{Disk}	$U_{ServerP}$	U_{DiskP}	Relative error (%) in Sys. Throughput	Relative error (%) in Sys. Response time
Original model	0.261	26.511	6.669	1.296	0.993	0.111	0.993		
Activity simplification	0.261	26.622	6.604	1.282	0.991	0.111	0.991	0.144	0.415
Entry simplification	0.263	25.962	6.486	1.407	1.000	0.112	1	0.723	2.073
Task Simplification	0.257	27.801	6.809		0.976	0.109	0.976	1.657	4.864

In Example1, the multiplicities of the tasks Server and BigLoop are infinite (i.e., no thread limit), whereas Disk and the Processors ServerP and DiskP are single servers.

The effect of the three levels of simplification on the model of Example1 can be seen in **Table 1**. On the first row of this table, the system throughput, system service time and resource utilizations of the original model are shown. In the subsequent rows, the same performance metrics are reported after activity, entry and task simplifications respectively. From the two rightmost columns of **Table 1**, it is observed that the amount of errors incurred by each simplification is relatively low comparing to the gain in the size of the models (discussed more in Section 5). Throughput error due to activity and entry simplifications are less than 1%, and to task simplification is less than 2%. The errors incurred by activity, entry and task simplifications on system response time is less than 1%, about 2% and almost 5% respectively. Moreover, along the simplifications steps, the utilizations of tasks and processors also remain almost same. The system bottleneck is DiskP (the disk hardware) for all cases. Although the Disk task is also saturated, its server DiskP is equally saturated, so Disk is not a software bottleneck (see [9] for techniques for identifying and mitigating software bottleneck).

4.2 Details of the Operations: Example 2

Fig. 4(a) represents another example of an LQN model called “eShop” where a number of users’ requests go through StoreApp, CustomerDB, InventoryDB and FileServer for read and write operations. This model has just one entry per task so it is ready for task-level simplification. Preliminary experiments show that the bottleneck is the task StoreShopping.



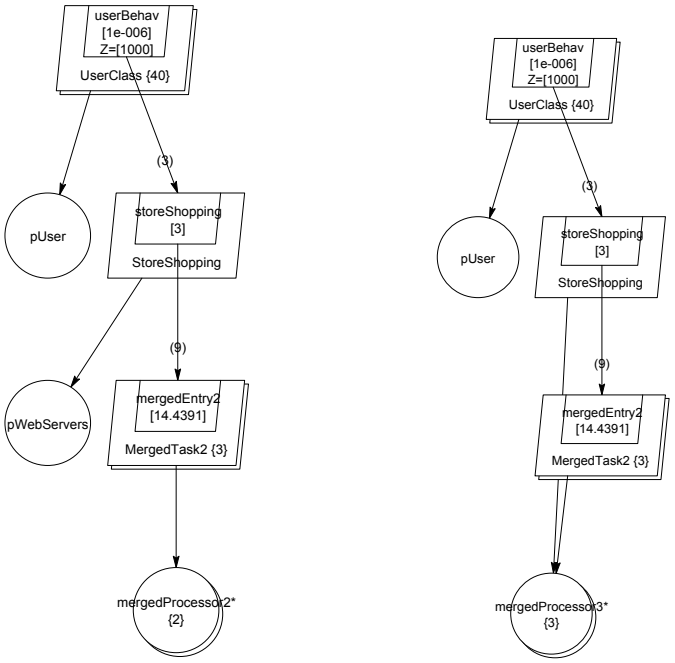
(a) Original model of eShop (b) After merging CustomerDB and InventoryDB

Fig. 4. An LQN model of eShop

In this model, tasks CustomerDB and InventoryDB are merged since they are deployed on the same processor. So, the values of (entry name, weight, execution demand) are (cRW, 0.439, 3), (iRW, 0.561, 2). So, Applying Eq (3) the service demand of the mergedTask we found, s_m = 3 * 0.439 + 2 * 0.561 = 2.439 (where the throughputs of cRW and iRW are 0.03001 and 0.03833 respectively). The number of incoming calls to the merged entry is 9 since the incoming calls from storeShopping should be directly summed up. For the number of outgoing calls, the values of (entry name, weight, number of calls from merging entry of task to fRW) are (cRW, 0.439, 3), (iRW, 0.561, 3). Thus, applying Eq (4), the number of calls from the merged entry to fRW is 3 * 0.439 + 3 * 0.561 = 3. **Fig. 4(b)** represents the model after merging CustomerDB and InventoryDB tasks.

Operation 4: Merging Processors and tasks

The next step of aggregation for this example will be merging different tasks that are deployed on different processors. In case of horizontal as well as vertical merging of such tasks, the service demands, incoming and outgoing calls and multiplicities of tasks are calculated as for horizontal and vertical merging of tasks on the same processor, as discussed in Operation 3 in Section 4.1. The merged processor's multiplicity is the aggregation of multiplicities of merging processors. In **Fig. 5(a)**, FileServer and MergedTask1 (originally deployed on different processors) are merged.



(a) Merging tasks deployed on different processors

(b) Merging processors only

Fig. 5. More simplification operations on eShop

Table 2. Effects of the simplification operations on system Response Time and Throughput for Example2

Model "eShop"	Sys. Throughput (jobs/ms)	Sys. Response time (ms)	Relative error in Sys.Throughput (%)	Relative error in Sys. Response time(%)
Original model	0.002	12236.2	N/A	N/A
Merging CustomerDB and InventoryDB	0.002	12360.6	0.820	1.017
Merging MergedTask1 and FileServer	0.003	11816.4	5.738	3.431
Merging non-bottleneck processors	0.003	10139.9	17.213	17.132

Table 3. Effects of simplification operations on Utilizations of resources of Example2

Model "eShop"	$U_{StoreShopping}$	$U_{CustomerDB}$	$U_{InventoryDB}$	$U_{FileServer}$	$U_{pWebServers}$	$U_{pDBServers}$	$U_{pFileServers}$
Original model	0.999	0.457	0.520	0.812	0.022	0.167	0.812
Merging CustomerDB and InventoryDB	1.000	0.977{2}		0.811	0.023	0.164	0.811
Merging MergedTask1 and FileServer	0.999	0.977{3}			0.023	0.984{2}	
Merging non-bottleneck processors	0.999	0.977{3}			0.992{3}		

Since pWebservers is not a bottleneck processor, it can be merged with the other non-bottleneck processor. In **Fig. 5(b)**, processors pWebServers (multiplicity 2) and mergedProcessor2 (multiplicity 1) are merged as mergedProcessor3 with multiplicity 3. **Table 2** and **Table 3** represent the performance results of the simplification process for Example2.

From **Table 2**, it is observed that merging tasks CustomerDB and InventoryDB incur only about 1% error in system throughput and system response time. Then, merging vertical tasks MergedTask1 and FileServer incur less than 6% and 4% errors in system throughput and system response time respectively. However processor merging incurred much higher errors (about 17% each). We see that the database processor utilization is only 0.16, compared to 0.81 for the file server processor. When the total capacity is shared the contention is significantly lower for the fileserver accesses, and this effect is even stronger after the very lightly loaded webserver processor is merged (note that the merged processor utilization of 0.992 is relative to a capacity of 3, so it is only 33% saturated). This effect would be much less pronounced if the original database processor were lower. At 81% saturation it is almost a bottleneck itself. So, merging near-bottleneck resources (tasks and processors) degrades accuracy.

This sugggests that merging processors might not be a good idea in many cases. But, it is also to be noted from **Table 3**, which shows the utilizations of tasks and processors after the simplification operations, the system bottleneck (i.e., StoreShopping task) remains the same througout the simplifcation process.

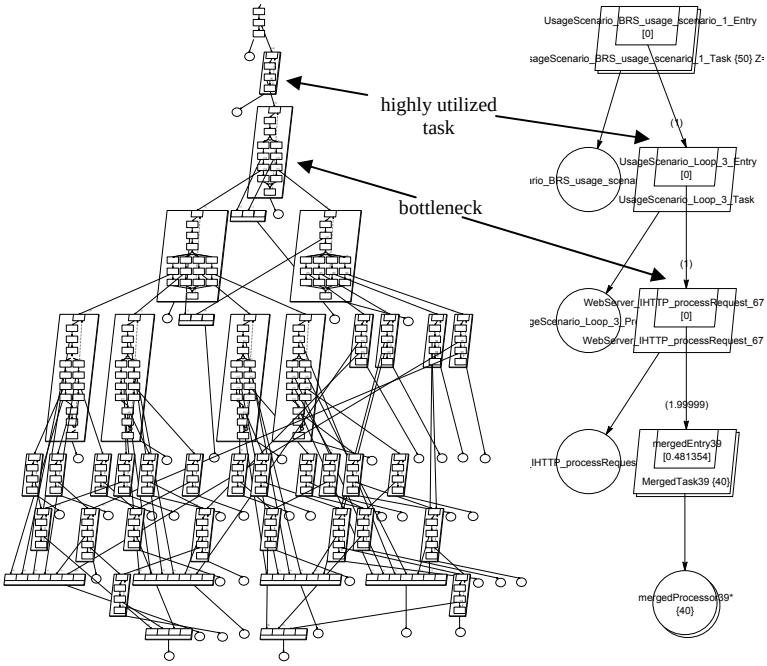
In Example2, all the tasks and processors are single-threaded initially. But, after the merging operations the number of threads of the merged resources are added to get new multiplicites. These new multipliticies are shown in curly braces in **Table 3**.

5 Case study

The performance results reported in this section were obtained by simulation with the lqsim solver [5] with a confidence interval of ±1% of the mean at 95% confidence level. A Java application is built which takes the original LQN model as input and generates a series of simplified models, which include models after merging activities, entries and tasks.

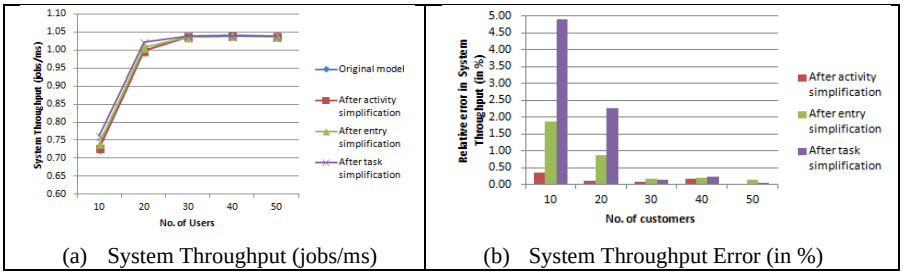
We consider a complex LQN model of a Business Reporting System generated from the Palladio Component Model (PCM) published in [7] as a case study to demonstrate the application of the proposed simplification process. Business Reporting

System lets users retrieve reports and statistical data about running business processes from a data base [7]. **Fig. 6(a)** shows the original model of our case study. This model contains a large number of tasks, entries and activities. **Fig. 6(b)** represents the simplified model from **Fig. 6(a)**. The original model has two highly utilized tasks, which are preserved in the simplified model: one is the software bottleneck of the system and the other is a direct caller of the bottleneck tasks. The other non-bottleneck tasks are simplified into a single non-bottleneck task. The performance results after different steps of the simplification process (i.e., activity, entry and task simplification) are comparable, as shown in the subsequent figures.



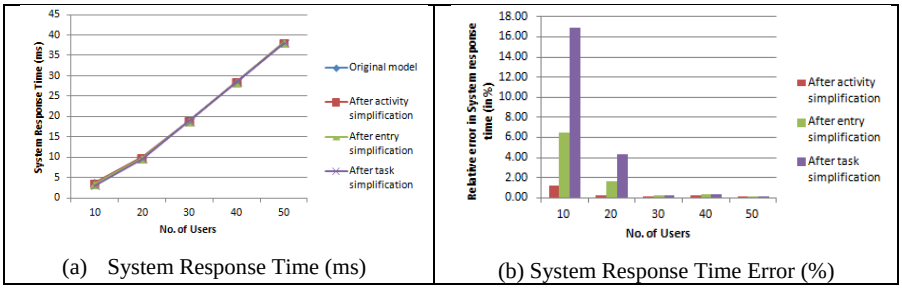
(a) Original model from [7] (b) Simplified model

Fig. 6. Layered Queueing Network of the Business Reporting System generated from PCM



(a) System Throughput (jobs/ms) (b) System Throughput Error (in %)

Fig. 7. System throughput after various simplification operations



(a) System Response Time (ms) (b) System Response Time Error (%)

Fig. 8. System response time after various simplification operations

From **Fig. 7(a)** it can be seen that the system throughput of the original model is very close to that of simplified models, obtained after activity, entry and task simplifications. **Fig. 7(b)** further shows that the error caused by task simplification is higher than that of entry simplification, which in turn is higher than that of activity simplification. This variation of error is expected, because the simplifications done for larger model elements (e.g., task) require more approximations than simplifications for smaller model elements (e.g., activity and entry). Also, the simplification process incurs more errors for small number of customers, because the error caused by the simplification process is spread over only few customers.

From **Fig. 8(a)**, it can be observed that the proposed simplification process maintains the response time behavior of the system as well. For a high number of customers (e.g., N>20), the simplification error in response time is less than 5% (see **Fig. 8(b)**). However, for small N (e.g., N=10), the error in system response time is higher after task simplification. On the contrary, what is important is that, throughout the simplification process, the bottlenecks of the system remain unchanged with similar utilizations. It was found that for N=10, the utilizations of both highly utilized tasks before any simplification and after all the simplifications are 70% and 73% respectively, which gives an error of less than 1%.

6 Conclusion

Large performance models are problematic for human and computer, as they are difficult to maintain and take a long time to solve. This paper proposes a model simplification process that compacts a given LQN model to its smallest possible size while preserving the result accuracy level, by reducing non-bottleneck resources to a single resource. The work shown in this paper can be extended in many ways. It needs to be investigated whether the simplifications are associative for a set of resources. If not, then further investigation can be done on finding the optimal order of simplification that incurs less error. Also, models can be classified into different patterns (e.g., sequential, tree-like etc.) and it can be studied whether they need different rules for finding the optimal order. The position (e.g., at the top, middle or bottom) of the bot-

tleneck resource as well as bottleneck intensity in a model may also affect the optimal rule. Furthermore, traceability models can be developed to keep track of the simplification steps so that the modeler can go back to an intermediate simplification step and modify performance parameters if needed. The proposed simplification has been applied so far to systems with a single class of users. Further investigation is needed to find the effect of the simplification process on performance results for multiple classes of users.

References

1. Chandy, K.M., Herzog, U., Woo, L.: Parametric analysis of queuing networks, IBM Journal of Research and Development, Vol.19, Issue 1, Pages 36-42 (1975)
2. Kritzinger, P.S., Wyk, S.V., Krzesinski, A.E.: A generalization of Norton's theorem for multiclass queueing networks, Performance Evaluation, Volume 2, Issue 2, Pages 98-107 (July 1982)
3. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C. Quantitative system performance - computer system analysis using queueing network models, Prentice Hall. ISBN: 978-0-13-746975-8 (1984)
4. Woodside, M.: The Relationship of Performance Models to Data, Proc SPEC Int Workshop on Performance Evaluation (SIPEW), Darmstadt, Lecture Notes In Computer Science, Vol. 5119, pp 9 - 28 (2008)
5. Layered Queuing Network homepage. http://www.sce.carleton.ca/rads/lqns/
6. Woodside, M.: Tutorial Introduction to Layered Modeling of Software Performance, Edition 4.0, RADS Lab: http://www.sce.carleton.ca/rads/lqns
7. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically Improve Software Architecture Models for Performance, Reliability, and Cost Using Evolutionary Algorithms, Proc. First Joint WOSP/SIPEW International Conference on Performance Engineering, Pages 105-116 (2010)
8. Franks, G., Al-Omari, T., Woodside, C.M., Das, O., Derisavi, S.: Enhanced Modeling and Solution of Layered Queueing Networks, IEEE Trans. on Software Eng. Vol. 35, No. 2 (2009)
9. Franks, G., Petriu, D., Woodside, M., Xu, J., Tregunno, P.: Layered bottlenecks and their mitigation, Proc of 3rd Int. Conference on Quantitative Evaluation of Systems QEST'2006, Pages 103-114, Riverside, CA, USA, Sept. 2006.