Choice of Aggregation Groups for Layered Performance Model Simplification*

Farhana Islam
Dept. of Systems and Computer Engineering
Carleton University
Ottawa, Canada
fislam@sce.carleton.ca

Dorina Petriu
Dept. of Systems and Computer Engineering
Carleton University
Ottawa, Canada
petriu@sce.carleton.ca

Murray Woodside
Dept. of Systems and Computer Engineering
Carleton University
Ottawa, Canada
cmw@sce.carleton.ca

ABSTRACT

The authors [1] previously showed that a complex layered performance model could be simplified by aggregating the contributions of subsystems, following a few simple principles which give good accuracy in many cases. The question of which subsystems to merge in layered performance models is further examined here, leading to identifying groups of subsystems (corresponding to "tasks" in layered queuing models) which can be safely aggregated. The grouping begins by identifying tasks which should be preserved, not aggregated, including those which are (or might become) bottlenecks. Then the groups are defined by their relationship to these preserved tasks. Aggregation by groups provides adequate accuracy in the vast majority of cases examined.

CCS CONCEPTS

• **Software and its engineering** → Software performance.

KEYWORDS

Performance Models, Layered Queuing Network Models, Model simplification.

ACM Reference format:

F. Islam, D. C. Petriu, C.M. Woodside. 2018. Choice of Aggregation Groups for Layered Performance Model Simplification. In *Proceedings of 9th ACM/SPEC International Conference on Performance Engineering, Berlin, Germany, April 2018 (ICPE 2018)*, 12 pages.
https://doi.org/10.1145/3184407.3184411

1 INTRODUCTION

Analytic performance models are powerful tools to predict the performance and scalability of a system before it is completed and deployed, and methods have been devised to create such models from software specifications or architectural designs [1], [2]. However these models are often more detailed than really needed, because they include all the design detail when only some details are significant for performance. A simplified model would be more useful, provided it retains the detail of the significant subsystems. This research is focussed on Layered Queueing Network (LQN) models of software systems with distributed and layered operations and resources. LQN models have the useful property that they retain much of the structure of the system, and measures predicted by the model can be traced to software entities. The goal of the research 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 authors previously described a process for aggregating the model elements, which in some cases could reduce a model to just two or three elements with little loss of accuracy [3]. This paper describes a more flexible and robust process.

The paper considers layered queuing (LQ) models of service systems with a single class of users, and with distributed and layered operations and resources.

2 Layered Queueing Network (LQN) Performance Models

The role of performance models is to make predictions for the performance of systems that do not yet exist, either using a model derived from a specification [1], or by studying modifications of an existing system and model [2]. Queueing models are used because they account for the effect of contention for resources, which is important in systems under load, and layered queueing network models (LQNs) are adapted to layered software systems. Fig. 1 shows an example LQN of a three-tiered web application, discussed further below.

An LQN model describes the interaction of system-elements (which may be any kind of software or hardware entity) via requests for service from one entity to another. The entities in the model are called *tasks* (analogous to objects) which accept service requests as *calls* made to *entries* (analogous to methods). Calls may be procedure calls, RPCs, or synchronous or asynchronous messages over a network. Each task is executed by its host (processor) which may be multi-core or a multiprocessor. Tasks may be multi-threaded, with threads sharing a queue, and the threads are scheduled on the host by a host queueing discipline.

*Produces the permission block, and copyright information

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184411

Tasks may be used to model software processes and also any system features which generate contention, such as mutexes, buffer pools, or locks.

Entry E is part of a task T(E) and has a host demand parameter d_E (= mean host demand in time units per invocation of E, as determined for a "nominal" host type). The call (E_1, E_2) has a call parameter $y_{E1,E2}$ (= mean calls to E_2 per invocation of E_1). We will only consider calls which block the caller (the calling thread waits for the reply after the entry execution is finished), but LQN can also model asynchronous calls and calls which are forwarded along a chain of tasks for service.

Task T has a multiplicity m_T and a host H(T). Host H has a multiplicity m_H and a speed factor S_H (speed relative to the "nominal" host type for which the demand values are found). The actual demand of entry E on host H is d_E/S_H sec. The capacity of the host is the product $c_H = m_H S_H$, with units of seconds of "nominal" execution per second; thus a single-core nominal host has a capacity of 1.0. In general, entries accept calls and also make calls to other tasks, usually at lower layers in a layered system.

A special User task represents the system's user population, with a multiplicity m_{User} equal to the number of users. The User task has a single entry, which may include a thinking time (a pure delay Z_{User}) and one or more calls into the system. One User entry execution corresponds to one user response from the system.

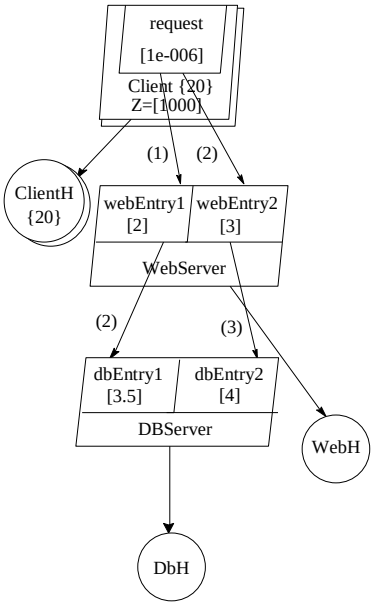

Figure 1: LQN model of a three-tier architecture [3]

Fig. 1 will serve as an example of the LQN notation. In this figure, the LQN model has three tasks - *Client*, *WebServer* and *DBServer*, each of which is deployed on its own host - *ClientH*, *WebH* and *DbH*, respectively. There are 20 users modeled as the 20-multiplicity task C*lient* running on the 20-multiplicity host *ClientH*. Each user takes a 1000 ms think time (Z_{User} = 1000) between requests. Both *WebServer* and *DBServer* are single threaded tasks and each has two entries with host service demands indicated in braces (i.e. *webEntry1* has service demand $D_{webEntry1} = 2$ *ms*). A single user 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.

This paper focuses its attention on LQNs with:

- a single User task (there can in principle be more, representing different classes of user),
- calling patterns with no cycles in the call graph,
- no internal parallelism on execution paths,
- entries that complete execution before replying (there is no "phase-2 execution" in LQN terms).

2.1 Performance Measures

The service time X_E of entry E is the total mean time to complete execution of the entry, including waiting and execution at its host H(T(E)), and waiting for a reply for each call. Each call delay in turn includes waiting for the called task to accept the request, and the entry execution time. Thus X_E is not known before the queueing delays are found; it is this property that makes layered system performance difficult to predict, and drives the use of LQN models.

Some other performance measures predicted by LQN models, that we will use, are

λ_E, λ_T = throughput of entry E and task T in invocations/s.

U_E, U_T, U_H = utilization of entry E, task T, host H,

where in this work utilization is defined as

$U_E = \lambda_E X_E$,

$U_T = \Sigma_E U_E$,

$U_H = \Sigma_{T \text{ deployed on } H} \Sigma_{E \text{ in } T} \lambda_E d_E/S_H$.

With this definition the utilization of a task or host ranges from zero to its multiplicity value. We also use its saturation level which ranges from zero to one, defined as

Saturation level of resource H = U_H / m_H.

Under heavy load the saturation of a bottleneck task or processor approaches one. Under lighter load the most saturated resource can be identified as the bottleneck.

2.2 Expectations for Aggregation

In aggregating LQN tasks and processors, we are aggregating two kinds of things: queueing resources (the tasks and processors) and customer classes (the entries of the tasks that are aggregated).

For queueing resources there is a well-established expectation that overall queueing delays are reduced when resources are aggregated together, which is well-known, for example, if a set of identical servers with separate queues are merged into a multiserver. However there are no comparable results available for layered servers.

For class aggregation the expected tendency is in the opposite direction. An LQN model includes many customer classes, for

example, each different operation (entry) of a subsystem (task) is a different class of service. In order to simplify the model it is necessary to aggregate some of these classes, which introduces errors. For certain multi-class queueing models, Dowdy et al showed in [4] that a single-class aggregated model always has lower overall throughput (and thus longer delay). Although their result does not apply to the present models, this result suggests that class aggregation at a server may give worse performance.
Thus the two kinds of aggregation may be expected to produce opposite effects in the performance measures of the simplified LQN.

3 Direct Task and Host Aggregation

We consider an original model M, and methods to aggregate some tasks to produce a final model M'. The previous report [3] described a process we will call Direct Aggregation, which operates directly on the tasks of M. It stated three principles:

- *Capacity limit principle*: preserve the capacity limit of the model by preserving the bottleneck element(s);
- *Total workload principle*: to preserve the total workload (CPU operations) per user response; and
- *Concurrency principle*: to preserve the total concurrency available in software and hardware.

If the host processors are homogeneous, the third principle also preserves the total computing capacity of the system.
In [3] a 4-step procedure was defined to aggregate all the tasks except the User, and any tasks identified as bottlenecks. It was assumed that all operations completed before replying (no second phase operations, in LQN terms) and this assumption is retained here. For our purposes the first two steps of that procedure will be called *Stage 1*, producing a *Stage-1 aggregated model*, and tasks which are not aggregated will be called *preserved tasks*.
Stage 1 aggregates all activities and entries of a task T_i into a single entry as described in [3]. Since each task now has only one entry, we can without confusion label the entry parameters d_E and calls (E_1, E2) by the task names, as demand d_T and calls (T_1, T_2). The aggregated host-demand and call values at Stage 1 are exact; the queueing times in this *Stage-1 model* have some degree of approximation error due to merging of classes, which was however found in [3] to often be small.
Step 3 and 4 in [3] aggregate all the tasks except the bottleneck and the User into a single task. Instead, this work identifies additional possible preserved tasks, and one or more groups of other tasks. It creates an aggregated task for each group, based on the following additional principle:

- *Dependency principle*: the dependency of a preserved task on each original task T will be preserved in M' as a dependency on the aggregated task that includes T.

This work also introduces a different process for aggregating a group of tasks. In [3] this was done incrementally by adding one task at a time; here each group is created in a single step.

4 Defining Groups of Tasks for Aggregation

The tasks that will not be aggregated (the preserved tasks) may include:

1) *Bottleneck tasks:* It was shown in our previous work that aggregating a bottleneck task together with others sometimes gives poor accuracy, and this seems to be generally true. Also the bottleneck is important in defining the saturation properties of a system, so preserving it should preserve those properties.
2) *Other highly saturated tasks:* If we intend to improve the system by mitigating the bottleneck then another task may emerge as a candidate "second bottleneck" and it would then (for the same reasons) be desirable not to have merged that task [3]. In general the candidates for bottlenecks are the highest tasks in an ordering based on task saturation level.
3) *Tasks subject to change:* We may also choose to keep a task out of aggregation if we want to study the effect of major changes in that task,
4) *Tasks with key measures:* We would like to observe the performance measures gathered for that task. In particular the measures for the User task define the user-related performance measures of the system, and it will always be a preserved task.
5) *Tasks deployed on bottleneck processors or highly saturated processor:* Tasks deployed on bottleneck processors also need to be preserved since merging the task (deployed on bottleneck processor) with other non-bottleneck tasks requires merging the bottleneck processor with other non-bottleneck processors. For the same reason as mentioned in item (1) above for tasks, bottleneck processors are preserved in this aggregation. In [3], it has been shown that merging a highly utilized/saturated processor can degrade the accuracy. So, we may preserve a second bottleneck processor as well as the task(s) deployed on it. If there is more than one task deployed on a bottleneck processor and none of them is a bottleneck task, those tasks can be merged into one task and preserved.

Before defining groups for aggregation we define a set **TP** = {TP_1, TP_2, ...} of tasks to be preserved. We wish the performance measures of these tasks to be well approximated in the aggregated model. Therefore we define the groups to preserve the dependency of the performance of the preserved tasks, on the tasks that are grouped, according to the Dependency Principle above.

4.1 Tasks Grouped by their Dependencies

A task T may affect the performance of a preserved task TP through delay dependency or by processor contention dependency. Delay dependence arises if TP makes a blocking call to T directly, or calls intermediary tasks with delay dependency on T (that is, if there is a path of blocking calls from TP to T). Processor contention dependency arises if T shares a processor with TP. Here we focus on the effect of delay dependency by restricting the original system to provide a separate host, (possibly a virtual machine) for each task, to eliminate processor contention dependency.

A blocking dependency of task T_1 on task T_2 in an LQ model is created by existence of a call path from T_1 to T_2 (that is, there is a direct call from T_1 to T_2 or indirect calls via one or more other tasks). We will define this dependency relative to a set of preserved tasks, as:

Definition: Task T_1 is *Preserved-Task Dependent* (PT-Dependent) on task T_2 if there is a call path from T_1 to T_2 that does not pass through a preserved task.

We will express this PT-dependency as $T_1 \prec T_2$. We assume that the system (and the layered model) does not have cyclical dependencies. If task T_1 has no dependency relation with T_2 we can write $T_1 \| T_2$.

For each task T_i its PT-dependency set $\mathbf{P}_i$ is then defined as

$$\mathbf{P}_i = \{P \mid P \prec T_i\}$$

$\mathbf{P}_i$ is non-empty for every non-preserved task T_i since every task except the User may be called in executing the application, therefore it always contains at least the User task.

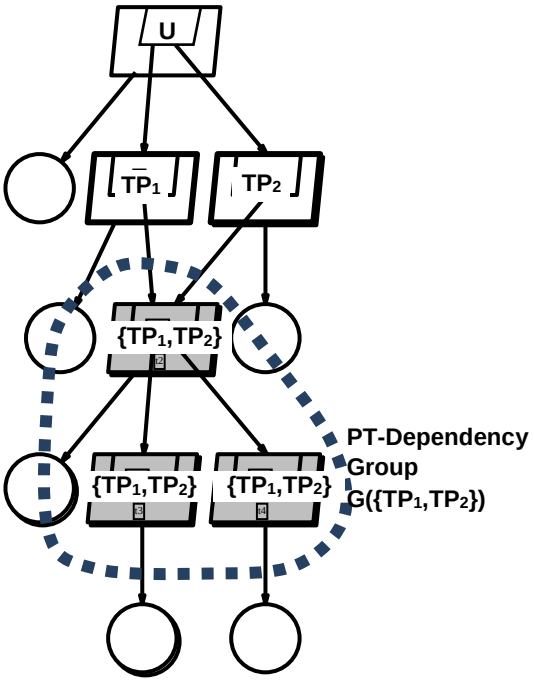

Figure 2: LQN model showing tasks with PT-dependency set {P1, P2}

Fig. 2 shows a model with three preserved tasks in the set **TP** = {U, TP_1, TP_2}. The three shaded tasks all have the same PT-dependency set **P** = {TP_1, TP_2}. The PT-dependency sets **P**, one for each non-preserved task, partition the non-preserved tasks into subsets **G**(**P**) which will be a basis for the groups for aggregation. All the tasks in **G**(**P**) have the same PT-dependency set, and all other tasks have different PT-dependency sets. The tasks in a particular **G**(**P**) have an impact (through blocking calls) on just those preserved tasks in subset **P**, and no others, and in M' all those blocking delay effects are captured approximately by blocking calls to a single task TA(**G(P)**) created by aggregating the tasks in **G**(**P**).

The groups and the aggregated tasks are illustrated in Fig. 3 by an example "case-A" with three preserved tasks U, TP_1 and TP_2 (shown as parallelograms with bold borders). The tasks have four different dependency sets: {U}, shown as tasks with no shading, {TP_1} as tasks with diagonal stripes, {TP_2} as tasks with grey shading, and {U, TP_2} as the one task with diamond shading. Tasks with the same shading form a group.

To simplify the model, each group is aggregated into a single task running on its own processor, following the method in [3] as modified in the following section. This gives the simplified model in Fig. 4.

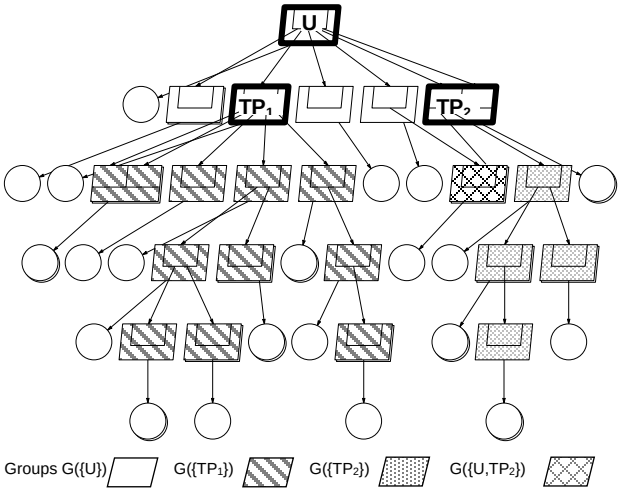

Figure 3: "case-A" showing groups defined by four distinct PT-Dependency sets P = {U}, {TP_1}, {TP_2}, {U, TP_2}

4.2 Desirable properties of the Aggregation Groups

The model simplification described here is motivated by a desire to preserve the essential components of the system, and to make the results traceable back to these components, while merging the elements that contribute less to the performance result. Direct Aggregation as in [3] preserves for each group

- its total workload
- the total processing capacity available to it
- the maximum concurrency of execution threads and cores

What is new here is:

- there can be an arbitrary number of preserved tasks, giving the modeller flexibility
- the dependencies of the preserved tasks are also preserved, in the following sense. For every preserved task P and unpreserved task T:
  - If P depends on T, then in M' P depends on a merged task derived from a group containing T
  - If T depends on P, then a merged task derived from a group containing T depends on P.

Thus, if we can think of each task as contributing a shadow of itself to its merged task, then there is a shadow of every non-persistent task and the dependencies between the persistent tasks and the shadows are preserved in M'.

5 Aggregation of a Group of Tasks

We consider a group or subset **G** of tasks T, each of which has one entry (as produced by Stage 1 aggregation) and its own processor (as assumed for this paper). T has demand d_T and makes an average of $y_{T,Ti}$ calls to each other task T_i, each time it is invoked.

An aggregated task TA(**G**) is substituted for **G**, with CPU demand DA(**G**), and $y_{Ti,TA(G)}$ calls coming to TA(**G**) from each task T_i not in **G.** The calculation of DA and y begins by finding Y_i for each task T_i in M:

Y_i = mean invocations of T_i per user response, which will be called the "total calls" to T_i.

Total calls Y_i for each task T_i is found by setting $Y_{User} = 1$ (for one user response) and solving these equations for all tasks T_i in M:

$$Y_i = \Sigma_{i,j} (Y_j * y_{Tj,Ti}) \text{ for all tasks } T_j \text{ calls } T_i.$$

From this the invocations of **G** per user response (or "total calls" to **G**) is YA(**G**):

$$YA(\mathbf{G}) = \Sigma_{Ti \notin G} \Sigma_{Tj \in G} Y_i * y_{Ti,Tj}$$

Then the demand DA(**G**) is the total demand of **G** per user response, divided by the number of calls to **G**:

$$DA(\mathbf{G}) = \Sigma_{Tj \in G} (Y_j * d_{Tj}) / YA(\mathbf{G})$$

The number of calls $y_{TA(G),Tk}$ from TA(**G**) to target task $T_k \notin$ **G**, per call into group **G**, is defined by a weighted average of calls from tasks T_j in **G**, weighted by Y_j:

$$y_{TA(G),Tk} = (\text{total calls from } \mathbf{G} \text{ to } T_k)/\text{total calls to } \mathbf{G}$$
$$= \Sigma_{Tj \in G} (Y_j * y_{Tj,Tk})/YA(\mathbf{G})$$

If the target tasks are also members of other groups, the number of calls between groups is calculated directly as follows. The number of calls from TA(**G1**) to a task representing another group, say TA(**G2**), is $y_{TA(G1),TA(G2)}$, which is the sum of the calls from TA(**G1**) over tasks T_k in **G2**, thus:

$$y_{TA(G1),TA(G2)} = \Sigma_{Tk \in G2} \Sigma_{Tj \in G1} [(Y_j * y_{Tj,Tk})/YA(\mathbf{G1})]$$

An aggregated host HA(TA(**G**)) is created for the aggregated task TA(**G**), with processing capacity c and multiplicity m, equal to the total capacity and multiplicity. In M' the host HA(TA(**G**)) has the properties:

$$c_{HA(TA(G))} = \Sigma_{Tj \in G} c(H(T_j))$$
$$m_{HA(TA(G))} = \Sigma_{\{H(T) \mid T \in G\}} m_{H(T)}$$
$$S_{HA(TA(G))} = c_{HA(TA(G))} / m_{HA(TA(G))}$$

Fig. 4 represents an aggregated model of the original model "case-A" in Fig. 3. The preserved tasks from the original model are shown with bold borders. The aggregated model has 7 tasks and 7 processors whereas the original model in Fig. 3 has 21 tasks and 21 processors.

6 Dependency Grouping vs a Single Group

In [3] a single group of tasks was formed from all the non-bottleneck tasks. This section shows examples which demonstrate the improved accuracy obtained when using the dependency groups proposed in this paper.

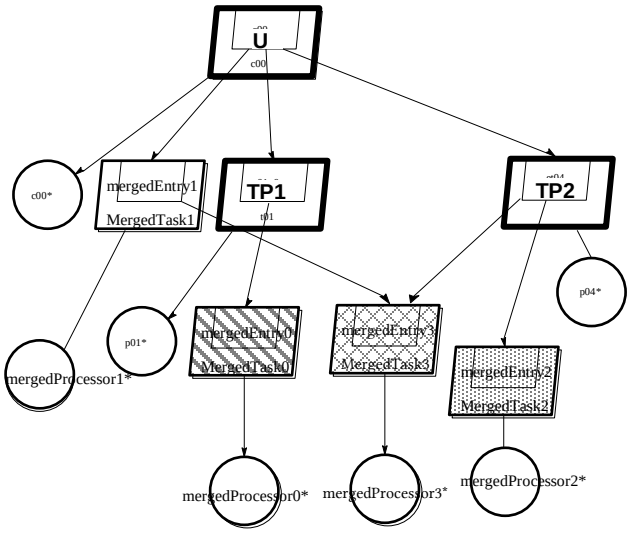

Figure 4: Aggregated model for "case-A" using the dependency groups shown in Fig. 3

First example: Cycle introduced by aggregation

This example shows the value of the dependency groups proposed here. Fig. 5 presents an LQN model called "case-11" with 11 tasks and 11 processors. The bottleneck task is t3 with 89.81% saturation level, shown with a bold outline.

Applying the previous aggregation algorithm [3], we get the model in Fig. 6. In this model, the bottleneck task t3 and its processor p3 are preserved and all other tasks except the user task c0 are merged into one task (with one processor). This produces a cycle in the call graph as seen in Fig. 6.

In comparing with the original model, the aggregated model generates 100% System throughput error and 3.78E+10% System response time error. In fact the calculation did not converge, the solver just stopped. The model is structurally different and the cycle creates a call explosion and an explosive increase in delay, and drop in throughput.

The simplification algorithm of this paper gives the second aggregated model shown in Fig. 7. The bottleneck task t3 is preserved, and there are two groups of tasks in the model in Fig. 5 that are identified by the proposed aggregation algorithm. One group is "below" the bottleneck (and produces the MergedTask0 in Fig. 7) and the other group is "above" the bottleneck (and produces MergedTask1 in Fig. 7).

To summarize the results:

Accuracy: Relative absolute error for "case-11"

Single-group: 100% in throughput,
3.78E+10% in response time

Dependency groups based on one bottleneck task:
23.55% in throughput,
19.06% in response time

Task aggregation based on groups provides a much better aggregated model, although the error is still large.

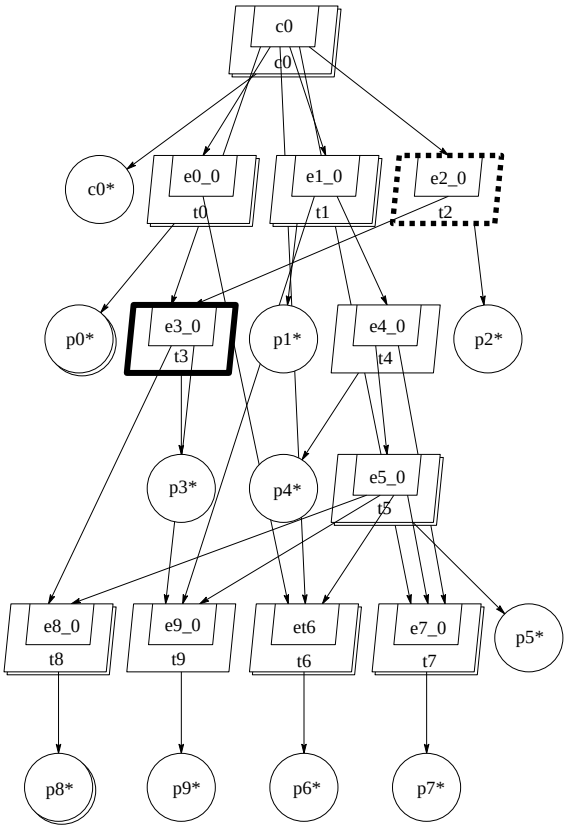

Figure 5: LQN model of "case-11"

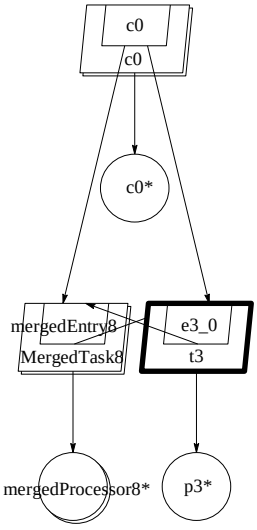

Figure 6: First aggregated model for "case-11" shown in Fig. 5, following the single-group algorithm of [3]

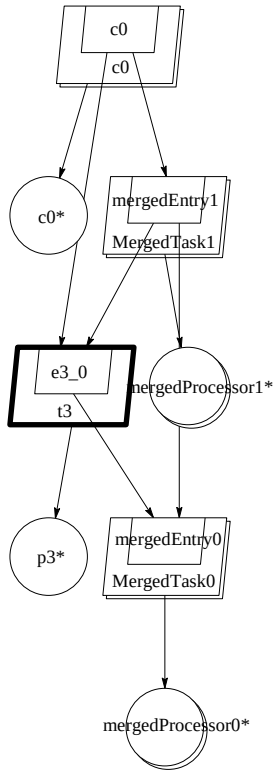

Figure 7: Second aggregated model for "case-11" (shown in Fig. 5) by generating two groups preserving one bottleneck task

Further examination revealed a second highly saturated task t2 (97.26% saturation level) in "case-11" in Fig. 5 shown in dashed outline. Task t2 is a direct caller of the bottleneck task, saturated due to pushback (waiting for service that is delayed by congestion). Preserving both t2 and t3 gives the third aggregated model shown in Fig. 8.

Accuracy: Relative absolute error for "case-11" (continued)
Dependency groups based on two highly saturated tasks:

6.91% in throughput,
6.47% in response time

The saturation level of both of the preserved tasks remain similar to the original model. Task t2's and t3's level of saturation are 97.26% (same as original model) and 87.43% (changed by 2%) respectively.
We can draw two lessons from this example, first that it may be important to preserve more than one highly saturated task, and second that the grouping should avoid aggregations that introduce cyclical calls between aggregated entries. Cyclical calling changes the structure of the system and totally distorts the predictions.

Second Example: Bottleneck Processor
In the following, we discuss another LQN model "case-41" where the model has a bottleneck processor.

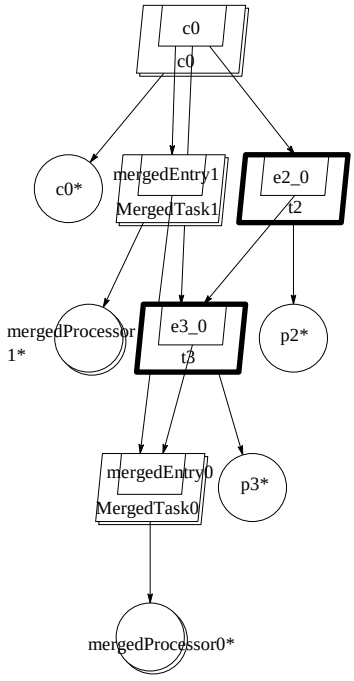

Figure 8: Third aggregated model for "case-11" by generating groups preserving two heavily saturated tasks

We compare the performance results of single group aggregation with dependency grouping. As shown in Fig. 9, the model contains 25 tasks and 25 processors along with a reference task and its processor. In this model, the bottleneck is the processor p09 with 78.35% saturation (shown in bold outline). The task deployed on p09 is t09 (also shown in bold outline). Both t09 and p09 are preserved in the aggregated model.

Applying the previous single-group aggregation algorithm [3], we get the model presented in Fig. 10, in which the bottleneck processor p09 along with its task t09 are preserved and all other tasks except the user task c00 are merged into one task (and their corresponding processors are merged into one processor).

In the model of Fig. 10 the System throughput error is 27.57% and System response time error is 21.61%, which are substantial. Analysis of "case-41" shows that there is a second bottleneck task t05 (shown in dashed outline) having 98% saturation which is a direct caller of the preserved task t09. Applying the simplification algorithm using task dependency groups which also preserves the second bottleneck, we get the aggregated model as presented in Fig. 11, with 5 tasks and 5 processors along with the reference task and its processor.

The results for the aggregation based on dependency group are much better than those for single-group aggregation:

Accuracy: Relative absolute error for "case-41"

Single-group: 27.57% in throughput,
21.61% in response time
The saturation level of p09 is changed by 27%

Dependency groups based on two preserved tasks and one bottleneck processor: 1.78% in throughput,
1.77% in response time

The saturation of t05 and p09 are 98.33% (changed by 0.34%) and 79.76% (changed by 1.8%) respectively.

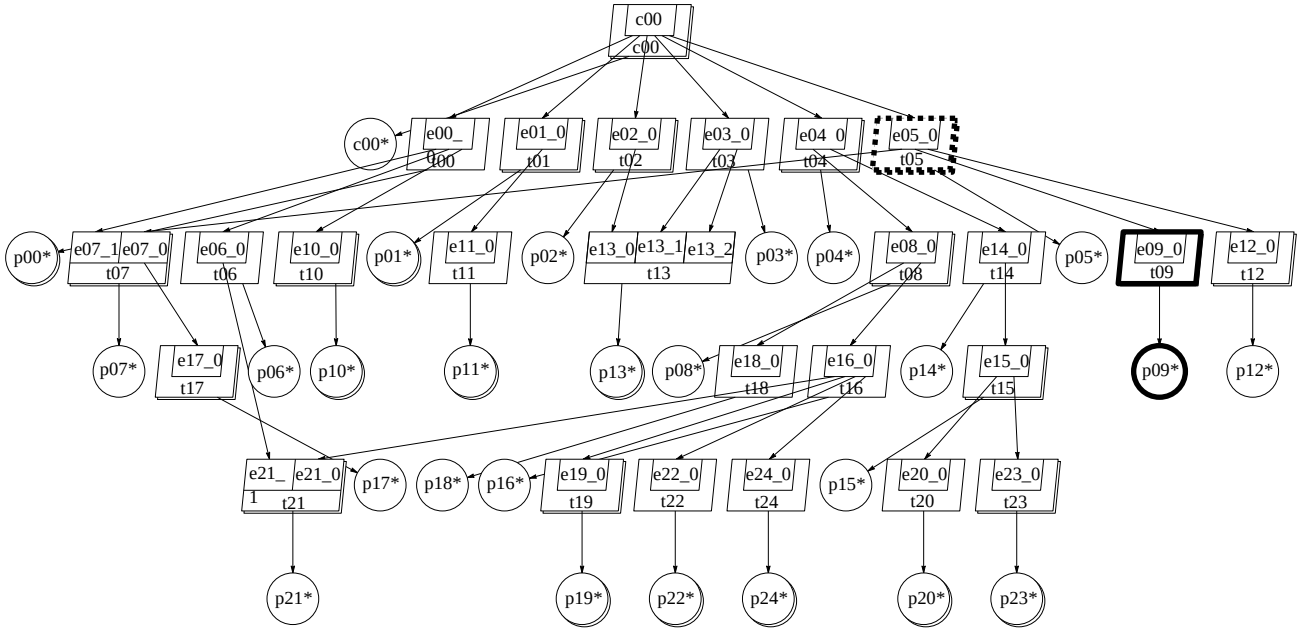

Figure 9: LQN model "case-41" with bottleneck processor p09 and its deployed task t09 in bold outline and second bottleneck task t05 in dashed outline

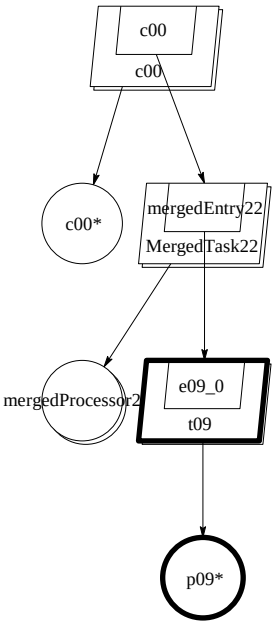

Figure 10: Aggregated LQN model for "case-41" (shown in Fig. 9) following the single-group algorithm of [3]

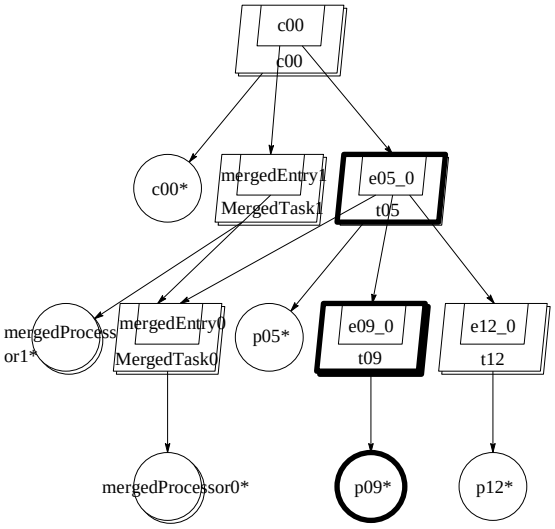

Figure 11: Aggregated model from "case-41" of Fig. 9 by generating groups preserving bottleneck processor and a second bottleneck task

7 Empirical Evaluation of Simplification Accuracy with Groups

This section describes experiments on groups of randomly generated models, to explore the accuracy of the dependency grouping strategy and to identify factors which may degrade the accuracy. Models of different total sizes with random structure and parameters were generated using the tool lqngen [5]. The grouping algorithm was implemented to automatically simplify the models, which were solved with the analytic solver LQNS [5].

Table 1 shows the average absolute errors in system throughput and response time for five groups of models of ten models each. We run all 50 models with random number of users (call it X users).

To investigate how the reduced model could be used for sensitivity studies, the accuracy of predictions for a system with twice as many users was also found, and is reported in the table. During the generation of these models one additional case was created that is treated separately below.

From the table we see that the average throughput error ranges from 3.5% to almost 6% and average response time error ranges from 3.4% to almost 5.5% for X users. There is no apparent relationship between the model sizes and the average error.

For 2X users, the errors are mostly slightly larger, but sometimes smaller, and in every case very little changed.

Table 1: Experiments with average throughput and response time error

Random LQN models	Throughput error (%)		Response time error (%)	
	X users	2X users	X users	2X users
10 models with 10 tasks	4.5	5.47	4.8	5.25
10 models with 15 tasks	3.5	3.45	3.4	3.47
10 models with 20 tasks	5.82	5.95	5.37	5.59
10 models with 25 tasks	3.52	3.56	3.48	3.53
10 models with 30 tasks	5.01	4.33	4.75	4.15

Fig. 12 and 13 show the frequency of absolute System throughput error (%) and absolute System response time error (%) for 50 models and X users, as in Table 1. From the histograms, we see that the frequency of System throughput error ranges mostly from 2% to 4% and System response time ranges mostly from 0 to 3%. They show that in general smaller errors are more frequent than larger errors. From experiments on these models, the maximum throughput error we found is 12.73% and response time error is 11.3%.

The results given so far understate the errors we found because they ignore one outlier. This outlier gives unsatisfactory errors in a case which occurred only rarely; this rare case must be treated by an extended process which is discussed next.

One Difficult Case: Case X

One randomly generated case, which we shall call "case-X", had a much larger error when aggregated following the grouping strategy: over 49% in response time and over 97% in throughput. The initial model and simplification are shown in Fig. 14 and 15. Inspection of the details suggests that the error arises partly through aggregation of very different classes, both moderately

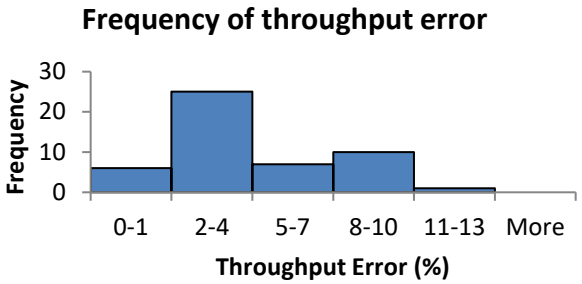

Figure 12: Frequency of absolute System throughput error (%) on the 50 models in Table 1

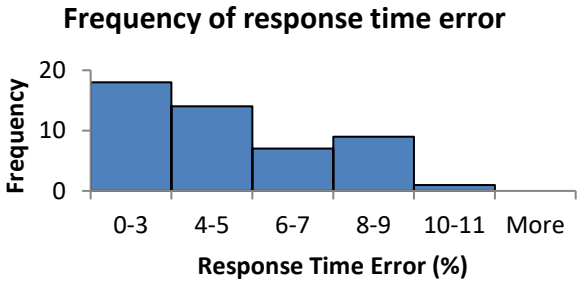

Figure 13: Frequency of absolute System response time error (%) on the 50 models in Table 1

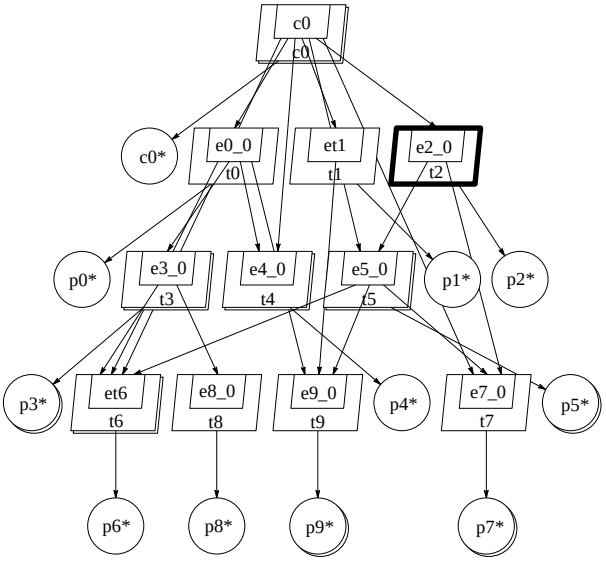

Figure 14: Original LQN model "case-X" with 10 tasks and 10 processors

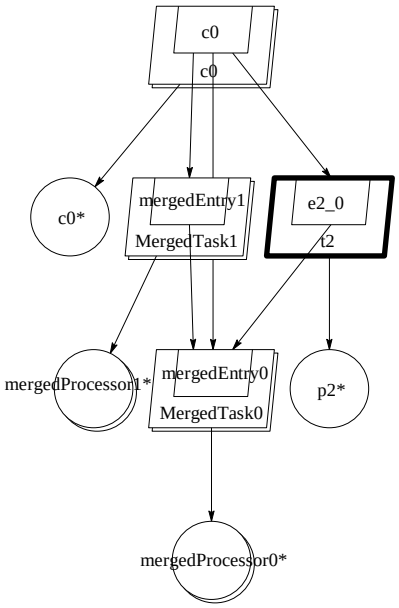

Figure 15: First Aggregated model of "case-X" from Fig. 14 preserving task t2

heavily loaded, in one of the groups. This is an example of the known threat of errors due to class aggregation. To deal with this threat the method must preserve additional tasks. To show that preserving additional tasks is effective, "case-X" was aggregated repeatedly with one task added at each step to the list of preserved tasks, with the results shown in Table 2. At each step, the most saturated non-preserved task or processor was chosen to be preserved. We see that we can obtain as small an error as we wish.

Table 2: Errors of Different Aggregations of "case-X"

Preserved tasks	Throughput error (%)	Response time error (%)
{t2}	97.14	49.25
{t2,t0}	14.29	13.12
{t2,t0,t5}	8.57	6.8
{t2,t0,t5,t6}	0	0.12

8 Scalability

Since our goal is to simplify large models, a much larger and more complex case is included here. Fig. 16 shows a model called "case-50" with 50 tasks and 50 processors that was generated randomly.

Table 3: Errors of Different Aggregations of "case-50"

Figure	Tasks and Processors	Throughput error %	Response time error %
Figure 17	7	12.73	11.3
Figure 18	9	4.85	4.63

The tasks outlined in bold were preserved based on high saturation level. Fig. 17 shows a first aggregated model of "case-50" with 7 tasks and 7 processors based on preserving only the bottleneck task and other highly saturated tasks. If we preserve the next most highly saturated resource which is processor p31 (47.92% saturated) and its deployed task t31 as shown in Fig. 18, the error is reduced as shown in Table 3.

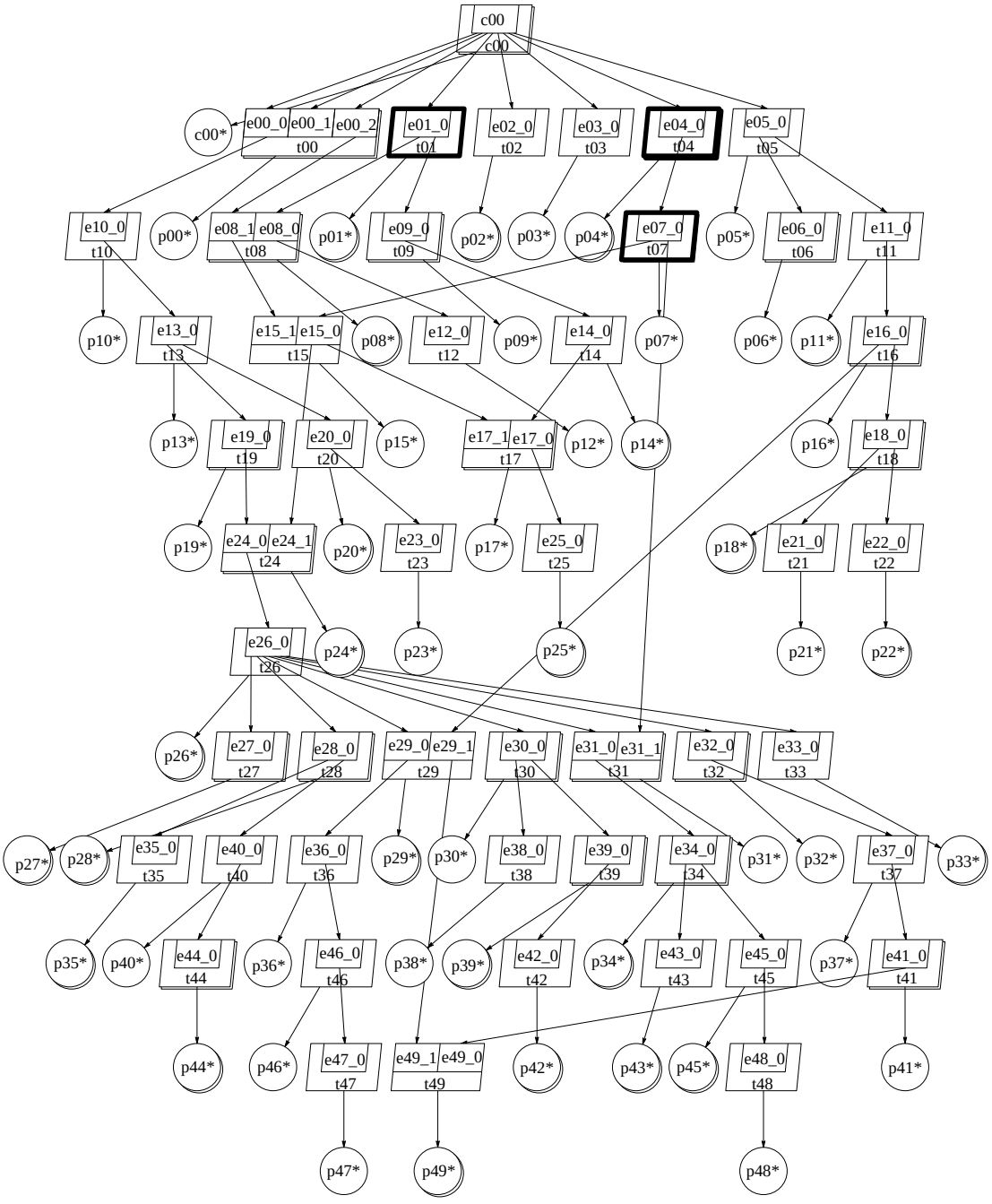

Figure 16: LQN model of "case-50" with 50 tasks and 50 processors

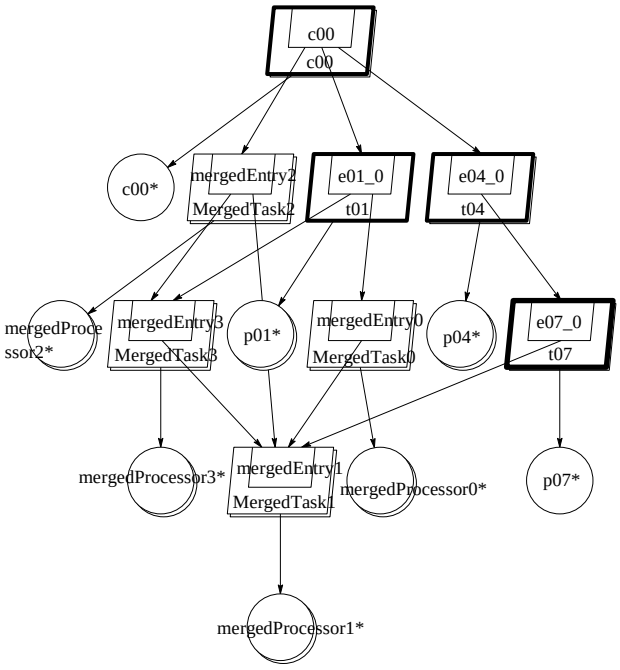

Figure 17: First aggregated model of "case-50" with 7 tasks and 7 processors

The errors under the simplest task-preservation in the first case are barely satisfactory; one improvement step gives a very satisfactory accuracy.
The execution time of the aggregation algorithm was found to be 1.2 second on average, on a commodity PC.

9 Related Work

Various kinds of simplification methods have been used for performance models, particularly for queueing models. There is a powerful and much-used simplification result in the Norton Theorem for Queues [6] which applies to product-form queuing networks. By this theorem 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 [3]. The original result was for a single class of customers, and it was extended to multiple classes in [7].
A flow-equivalent server (FES) [8] is a generalization of this. When any submodel is replaced by a FES the entire model is smaller and easier to solve, and parameter changes outside the submodel can be studied efficiently. Outside of product-form queueing networks the simplification is approximate. The FES construction method isolates the subnetwork and drives it with a fixed number of customers, cycling endlessly; the mean delay of a customer in the subnetwork is taken as the service time of the FES for that number of customers. This is repeated for every user population that it may experience, which does not scale well to large systems with thousands of customers [3].

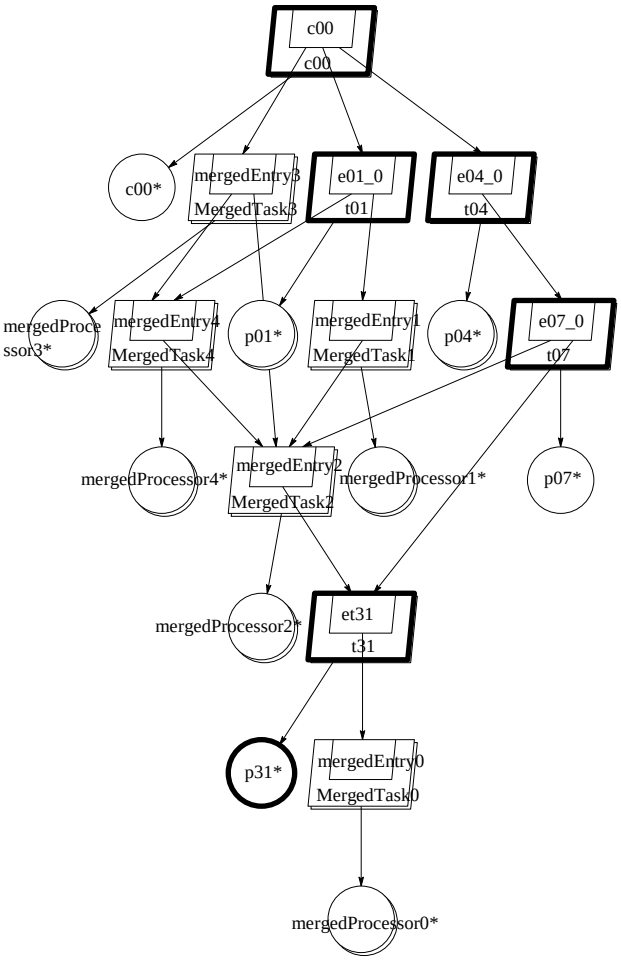

Figure 18: Second aggregated model of "case-50" with 9 tasks and 9 processors

Hierarchical decomposition as described in [8] applies this systematically. In hierarchical decomposition, a large model is partitioned into a number of submodels. Each submodel is then evaluated and individual solutions are combined to get the solution of the original model [8]. In this technique, the system is modeled using multiple levels of models. The highest level (level 0) of the models consists of a number of FESs, each of which represents some portion of the system being modelled. The following level contains a number of models that are more detailed representation of a subsystem represented in the first level as an FES. In general, every level in the hierarchy contains more detailed representation of the submodels from previous level until the final level (Level L) where all models are fully detailed and do not have any FESs. The models in hierarchical decomposition should be evaluated from level L to level 0 so that the performance projections for the system being modeled are obtained from its solution.
Surrogate delay methods (e.g. [8]) replace a subsystem by a delay which is found by solving an auxiliary model. A surrogate delay

is somewhat like a FES, but with a fixed delay rather than a state-dependent rate. However 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, but they can also be used for model simplification.

When performance models are fitted by regression methods as in [9], a choice must be made for the model structure including the level of detail in the model. The modeler can compare the goodness of fit of models with more or less detail. Regression thus automatically raises the question of detail, and can answer it through tests of goodness of fit as discussed in [9]. However this approach cannot be applied to models constructed from a design before a system is built, because it requires operational data for the regression.

In the Shadow Server method, one service node that violates conditions required for efficient, exact analytic solution in queuing network model is replaced by two or more servers that enable efficient analytic solution, such that the performance represents the original server [10]. As an example, a CPU server with a priority queue-scheduling discipline can be replaced with a shadow CPU server for each priority class, with jobs of different priorities being routed to different servers. However, this technique does not generate a simpler and smaller model than the original one.

The authors in [11] proposed an estimation technique for performance parameters in web based software systems. For web based applications, they use a combination of clustering algorithm and tracking filter for effective grouping of classes of services in layered queuing models. Clustering uses the K-means algorithm. The target application is autonomic control of web clusters. They considered the application URLs as first class entities and each URL request as a class. Their proposed tracking approach identifies performance parameters of groups of URLs instead of individual URLs. They proposed an algorithm that finds the appropriate number of clusters with a pre-defined clustering accuracy. For example, if one can accept 17% error, the number of needed clusters for estimation would be dropped from 14 to 9 on average.

Overall, we are unaware of any prior work on deriving a simplified layered queuing model directly from a detailed one, apart from our own paper [3]. In particular, there is a lack of simplification techniques that avoid the scalability problems of calibrating a FES.

10 Conclusion

In this paper, a simplification method for LQN model is presented which is an improved version of previous work of the authors [3]. In this new method, groups of tasks to be aggregated are determined based on the dependency relationships between the tasks in the groups, and a set of "preserved tasks" which should include at least the users and a bottleneck task. The paper defines grouping criteria and shows by experiments on randomly generated models of various sizes that the throughput and response time errors are less than 10% in the vast majority of cases. In every case the error can be reduced by adding preserved tasks, based on their relative saturation, and can be made as small as desired (at the cost of a larger simplified model and more complex simplification). The best strategy for adding preserved tasks is the subject of current additional research.

This work has considered only systems with a single class of users, and system modules ("tasks") that do not share a host processor. This latter is in line with the practice in cloud deployments of giving each module its own virtual machine. However current work is considering shared hosts. The grouping has also only considered delay dependencies; current work is also considering host dependencies.

The grouping strategies described here address a fundamental problem of the required level of detail in modeling, and could be applied far beyond the domain of LQN performance models, to adjust the detail level of an analysis in real time.

The LQN models used in the cases of this paper can be found at https://github.com/FarhanaIslam/lqnmodels.

ACKNOWLEDGMENTS

This research was supported by grants from NSERC, the Natural Sciences and Engineering Research Council of Canada, through its Discovery Grants program.

REFERENCES

[1] Murray Woodside, Dorina C. Petriu, José Merseguer, Dorin B. Petriu, Mohammad Alhaj. 2014. Transformation challenges: from software models to performance models. *Software and Systems Modeling*, 13, 4 (Oct 2014), 1529-1552. Published online Oct 2013. DOI: https://doi.org/10.1007/s10270-013-0385-x

[2] Steffen Becker, Heiko Koziolek, Ralf Reussner. 2009. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82, 1 (January, 2009), 3–22. DOI: https://doi.org/10.1016/j.jss.2008.03.066

[3] Farhana Islam, Dorina Petriu, Murray Woodside. 2015. Simplifying Layered Queuing Network Models. In *Computer Performance Engineering: 12th European Performance Engineering Workshop (EPEW 2015)*, Springer LNCS 9272, 65–79. DOI: https://doi.org/10.1007/978-3-319-23267-6_5

[4] Lawrence W. Dowdy, Brian M. Carlson, Alan T. Krantz, Satish K. Tripathi. 1992. Single-Class Bounds of Multi-Class Queuing Networks, *J.A.C.M*, 39, 1 (Jan 1992), 188-213. DOI: 10.1145/147508.147530

[5] Layered Queuing Network homepage. Retrieved from http://www.sce.carleton.ca/rads/lqns/.

[6] K. M. Chandy, U. Herzog, L. Woo. 1975. Parametric analysis of queuing networks. *IBM Journal of Research and Development*, 19, 1 (January 1975), 36-42. DOI: 10.1147/rd.191.0036

[7] P. S. Kritzinger, S. V. Wyk, A. E. Krzesinski. 1982. A generalization of Norton's theorem for multiclass queueing networks. *Performance Evaluation*, 2, 2 (July, 1982), 98-107. DOI: https://doi.org/10.1016/0166-5316(82)90002-5

[8] Edward D. Lazowska, John Zahorjan, G. S. Graham, Kenneth C. Sevcik. 1984. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall.

[9] Murray Woodside. 2008. The Relationship of Performance Models to Data. In *Proc. International Performance Evaluation Workshop (SIPEW)*, Springer, Lecture Notes In Computer Science, 5119, 9 – 28. DOI: https://doi.org/10.1007/978-3-540-69814-2_3

[10] Connie U. Smith. 1990. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA,.

[11] Hamoun Ghanbari, Cornel Barna, Marin Litoiu, Murray Woodside, Tao Zheng, Johnny Wong, Gabriel Iszlai. 2011. Tracking adaptive performance models using dynamic clustering of user classes. In *Proc. Int. Conf. on Performance Engineering (ICPE '11)*, Karlsruhe, 179-188. DOI: 10.1145/2160803.2160823