Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time

Jing Xu, Murray Woodside, Dorina Petriu
Dept. of Systems and Computer Engineering,
Carleton University, Ottawa K1S 5B6, Canada
{xujing, cmw, petriu }@ sce.carleton.ca

Abstract: As software development cycles become shorter, it is more important to evaluate non-functional properties of a design, such as its performance (in the sense of response times, capacity and scalability). To assist users of UML (the Unified Modeling Language), a language extension called Profile for Schedulability, Performance and Time has been adopted by OMG. This paper demonstrates the use of the profile to describe performance aspects of design, and to evaluate and evolve the design to deal with performance issues, based on a performance model in the form of a layered queueing network. The focus is on addressing different kinds of performance concerns, and interpreting the results into modifications to the design and to the planned run-time configuration.

1. Introduction

The Unified Modeling Language (UML) [2] is the most widely used design notation for software at this time, unifying a number of popular approaches to specifying structure and behaviour. To enable users to capture time and performance requirements, and to evaluate those properties from early specifications, a language extension called the UML Profile for Schedulability, Performance and Time has been defined and adopted (the SPT Profile)[7]. In [18], the process of specifying a system with the SPT Profile was described, together with a layered queueing model created from it. The example was a building security system called BSS. This paper considers how to use the same model to study several performance questions, and to improve the design. The goal of the study is to provide a blueprint to users of the SPT Profile for exploring how performance issues are related to features of a software design, and to gain experience with use of the Profile. This is the first step towards a methodology for guiding design changes and explorations, based on UML and layered modeling, and previous work such as view navigation [19], optimal configuration [4], and performance patterns and anti-patterns [16].

The use of the SPTProfile can be envisaged as in Figure 1, with a process to interpret performance estimates made by a model, and to suggest changes to the design or to the configuration in the intended environment. If there is a performance shortfall, the process could iteratively improve the design until is satisfactory.

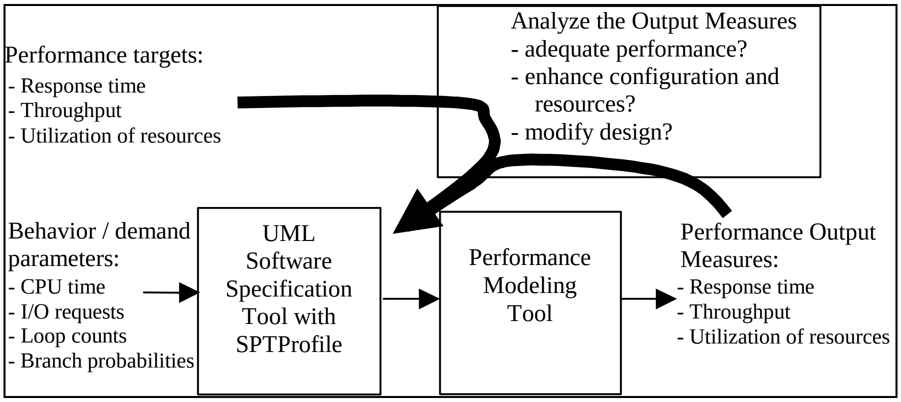

Analyze the Output Measures
- adequate performance?
- enhance configuration and resources?
- modify design?
Performance targets:
- Response time
- Throughput
- Utilization of resources
Behavior / demand parameters:
- CPU time
- I/O requests
- Loop counts
- Branch probabilities
UML Software Specification Tool with SPTProfile
Performance Modeling Tool
Performance Output Measures:
- Response time
- Throughput
- Utilization of resources

Figure 1. Performance measures: targets, input and output, and improvement process

The SPT profile extends UML by providing stereotypes and tagged values to represent performance requirements, the resources used by the system and some behaviour parameters, to be applied to certain UML behaviour models. The selected behaviour models describe scenarios (e.g. sequence diagrams and activity diagrams), because performance is usually specified and analyzed relative to selected scenarios (which in turn represent system responses). Some examples of the Profile stereotypes are shown below, for the example system. The performance model used in this work is a layered queueing network (LQN) model, just one of several possible target formalisms. LQNs are particularly well suited to analyzing software performance because they model layered resources and logical resources in a natural way, and they scale up well for large systems [6]. The concepts and notation for LQNs will be briefly introduced for the example, below.

The process for improving designs will be explored using a Building Security System (BSS), which is intended to control access and to monitor activity in a building like a hotel or a university laboratory. Scenarios derived from two Use Cases will be considered, related to control of door locks by access cards, and to video surveillance. In the Access Control scenario a card is inserted into a door-side reader, read and transmitted to a server, which checks the access rights associated with the card in a data base of access rights, and then either triggers the lock to open the door, or denies access. In the Aquire/Store Video scenario, video frames are captured periodically from a number of web cameras located around the building, and stored in the database. The system must implement other Use Cases as well, such as operations for administration of the access rights, for sending an alarm after multiple access failures, or for viewing the video frames, but for simplicity we assume that the main performance concerns relate to the two Use Cases described above.

Both scenarios have delay requirements. The access control scenario has a target completion time of one second, and the surveillance cycle has a target of one second or less between consecutive polls of a given camera. In both cases we will suppose that 95% of responses, or of polling cycles, should meet the target delay. Further, it is desired to initially handle access requests at about 1 per 2 second on average, and to deploy about 50 cameras. Additional camera capacity would be desirable, and a practical plan for scaling up the system to larger buildings and higher loads is to be created.

2. Behaviour Specification of BSS and its Performance Annotations

The BSS has the planned deployment shown in Figure 2, with one application processor, a separate database processor, and peripheral devices accessed over a LAN.

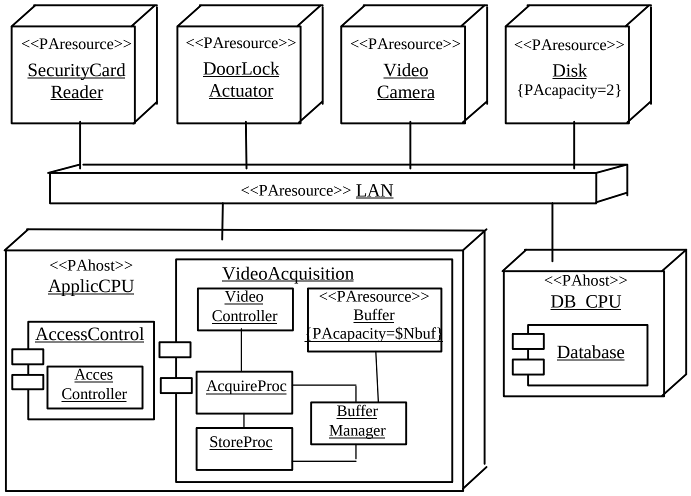


Figure 2. Deployment of the Building Security System

The access and surveillance scenarios will be described through sequence diagrams, using stereotypes and tagged values defined in the SPT Profile [7]. Some of the key stereotypes seen in these diagrams are a *performance context* defining a scenario made up of *steps* and driven by a *workload*, and a *resource*, with a special *host* resource for a processor. These stereotypes are, respectively, <<`PAcontext`>>, <<`PAstep`>>, <<`PAopenLoad`>> and <<`PAclosedLoad`>> for workloads, <<`PAresource`>> and <<`PAhost`>>.

Figure 3 shows the scenario for access control. The User provides an open workload, meaning a given arrival process. The tagged values define it as a Poisson process with a mean interarrival time of 0.5 seconds, and state a percentile requirement on the response time (95% of responses under 1 second). They also define a variable name `$UserR` for the resulting 95th percentile value, to be estimated. Each step is defined as a focus of control for some component, and the stereotype can be applied to the focus of control or to the message that initiates it; it can also be defined in a note. The steps are tagged with a demand value for processing time (tag `PAdemand`) which is the CPU demand for the step. The request goes from the card reader to the `Access Controller` software task, to the database and its disk, and then back to execute the check logic and either allow the entry or not. `openDoor` is a conditional step which can be tagged with a probability (`PAprob`) which here is set to unity.

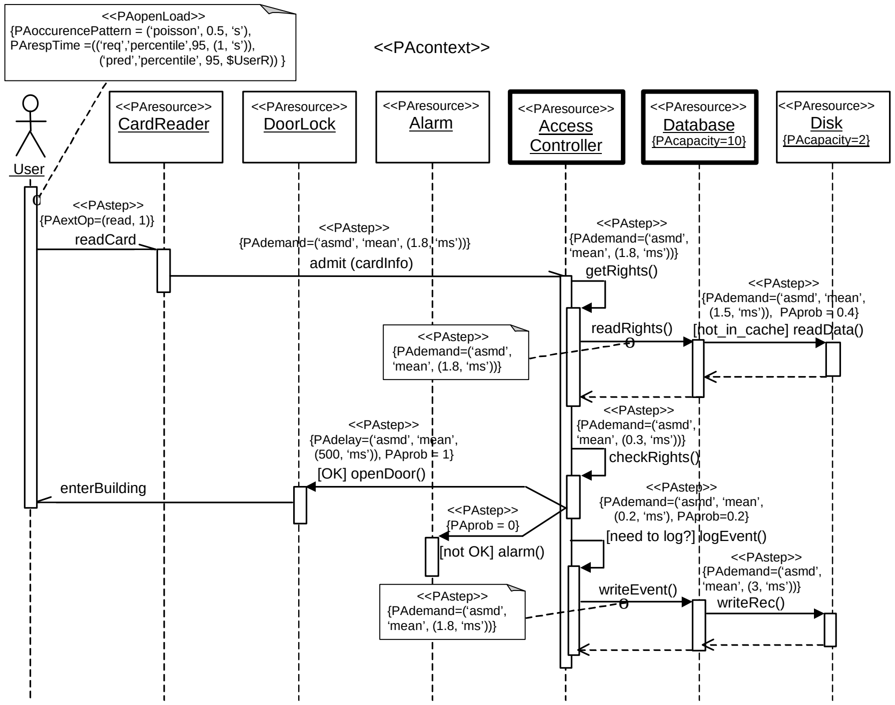

<<PAopenLoad>>
{PAoccurencePattern = ('poisson', 0.5, 's'),
PArespTime =(('req','percentile',95, (1, 's')),
('pred','percentile', 95, $UserR)) }
<<PAcontext>>
User
<<PAresource>>
CardReader
<<PAresource>>
DoorLock
<<PAresource>>
Alarm
<<PAresource>>
Access Controller
<<PAresource>>
Database
{PAcapacity=10}
<<PAresource>>
Disk
{PAcapacity=2}
<<PAstep>>
{PAextOp=(read, 1)}
readCard
<<PAstep>>
{PAdemand=('asmd', 'mean', (1.8, 'ms'))}
admit (cardInfo)
<<PAstep>>
{PAdemand=('asmd', 'mean', (1.8, 'ms'))}
getRights()
<<PAstep>>
{PAdemand=('asmd', 'mean', (1.5, 'ms')), PAprob = 0.4}
readRights()
[not_in_cache] readData()
<<PAstep>>
{PAdemand=('asmd', 'mean', (1.8, 'ms'))}
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.3, 'ms'))}
checkRights()
<<PAstep>>
{PAdelay=('asmd', 'mean', (500, 'ms')), PAprob = 1}
[OK] openDoor()
enterBuilding
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.2, 'ms'), PAprob=0.2}
<<PAstep>>
{PAprob = 0}
[need to log?] logEvent()
[not OK] alarm()
<<PAstep>>
{PAdemand=('asmd', 'mean', (3, 'ms'))}
writeEvent()
writeRec()
<<PAstep>>
{PAdemand=('asmd', 'mean', (1.8, 'ms'))}

Figure 3. Annotated Sequence Diagram for the Access Control Scenario

The devices are stereotyped as `<<PAresource>>`, as in the deployment diagram, and so are the software tasks `AccessController` and `Database`; this is because a task has a queue and acts as a server to its messages. A resource can be tagged as having multiple copies, as in a multiprocessor or a multithreaded task. The Database process is tagged with 10 threads, by `{PAcapacity = 10}`, and its disk subsystem is tagged as having two disks.

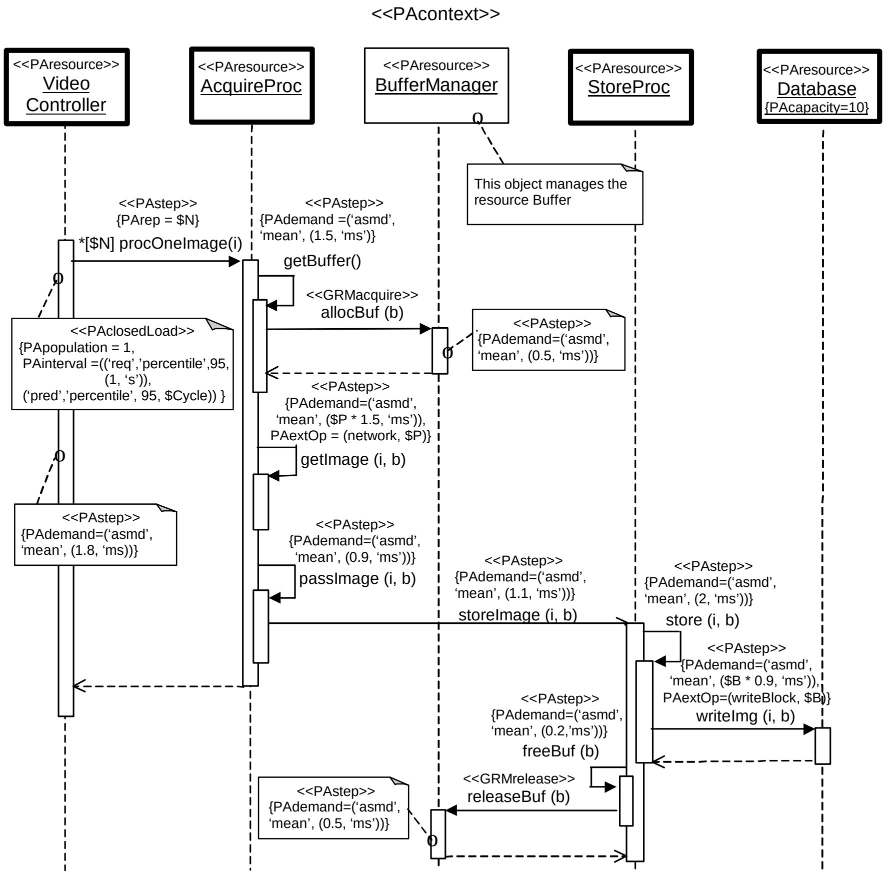

<<PAcontext>>
<<PAresource>>
Video Controller
<<PAresource>>
AcquireProc
<<PAresource>>
BufferManager
<<PAresource>>
StoreProc
<<PAresource>>
Database
{PAcapacity=10}
This object manages the resource Buffer
<<PAstep>>
{PArep = $N}
*[$N] procOneImage(i)
<<PAstep>>
{PAdemand =('asmd', 'mean', (1.5, 'ms')}
getBuffer()
<<GRMacquire>>
allocBuf (b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.5, 'ms'))}
<<PAclosedLoad>>
{PApopulation = 1,
PAinterval =(('req','percentile',95, (1, 's')),
('pred','percentile', 95, $Cycle)) }
<<PAstep>>
{PAdemand=('asmd', 'mean', ($P * 1.5, 'ms')),
PAextOp = (network, $P)}
getImage (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (1.8, 'ms))}
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.9, 'ms'))}
passImage (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (1.1, 'ms'))}
storeImage (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (2, 'ms'))}
store (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', ($B * 0.9, 'ms')),
PAextOp=(writeBlock, $B)}
writeImg (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.2,'ms'))}
freeBuf (b)
<<GRMrelease>>
releaseBuf (b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.5, 'ms'))}

Figure 4. Annotated Sequence Diagram for the Acquire/Store Video Scenario

The scenario for the video surveillance is shown in Figure 4. There is a single `VideoController` task which commands the acquisition of video frames from `$N` cameras in turn, by a process `AcquireProc`. The initial step is the focus of control of `VideoController` which is stereotyped as a closed workload source with one instance, with a required cycle time having 95% of cycles below 1 second, and a predicted value `$Cycle` to represent the model result. `AcquireProc` is a concurrent process (<<`PAresource`>>). It acquires a `Buffer` resource by a step `allocBuf` which is also stereotyped as <<`GRMacquire`>>, indicating a resource acquisition. Buffer is a passive resource shown in the deployment diagram with a multiplicity $Nbuf, managed by BufManager. In the sequence diagram the use of Buffer is indicated by a note and by the stereotype <<GRMacquire>>. In the base case, $Nbuf is set to 1. Once a buffer is acquired, AcquireProc requests the image from the camera, receives it and passes the full buffer to a separate process StoreProc, which stores the frame in the database and releases the buffer. The writeImg operation on the Database has a tag PAextOp to indicate that it calls ($B times) for an operation writeBlock, which is not defined in the diagram. This operation can be filled in, in the performance model, by a suitable operation to write one block of data to disk.

3. LQN Model

A layered queueing model was derived from the concurrent processes and their interactions, using the principles of scenario traversal described in [11]. The resulting model is shown in Figure 5. Each process is represented by a "task" rectangle with one or more "entry" rectangles attached to its left. A "task" models an active object, process, thread or any other logical resource that requires mutual exclusion (such as the buffer pool described below). An "entry" models the operation which processes a distinct class of messages received by the task, For example, if a "task" models an object, an "entry" models a method. Arrows to other entries indicate requests made by an operation to other components. A solid arrowhead shows a synchronous call (where the caller expects a reply, and is blocked until receiving it), as from the User to the CardReader in Figure 5; it may be shown as a call and its corresponding return in the sequence diagram. An open arrowhead shows an asynchronous message, and a dashed arrow shows a synchronous request which is forwarded to another task.

A server task can carry out part of its work after replying to its client; this is termed a "second phase" of service, and may have its own workload. For each entry the host demand is represented by [s1, s2] for first and second phase CPU demand in time units. For each request arc the mean number of calls in the two phases are represented by (y1, y2); the second phase value is optional. For example, the entry admit of the task AccessControl logs a message to the database and has some execution in second phase.

A request arc in the model can have a mean number of calls per entry invocation, or a deterministic integer number. Here all the calls are mostly given as averages, however $N is the exact number of calls in a polling cycle, and each one leads to exactly one buffer request, one getImage, one passImage, one storeImage, and one writeImage operation. Similarly one User leads to one readRights and one unlock operation.

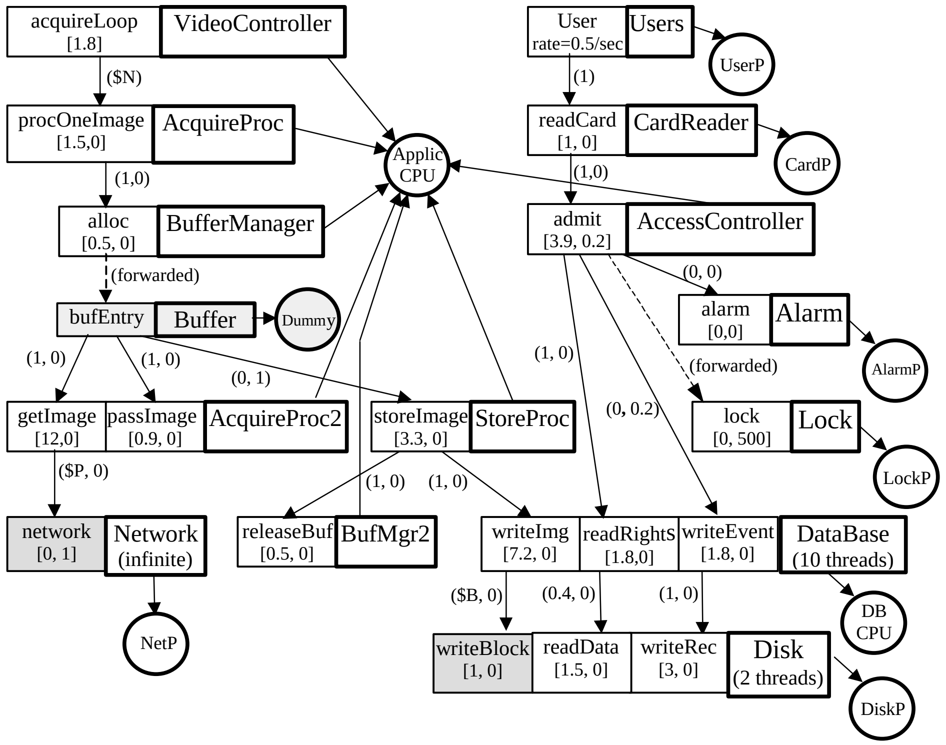


Figure 5. Layered Queueing Network model for the Building Security System

Since logical resources are represented by “tasks” in an LQN, the buffer pool is modeled by a task which is shaded in Figure 5 (we can think of it as a virtual task). It has an “entry” `bufEntry` which makes synchronous virtual calls to invoke the operations which are carried out holding the buffer (in [20] these operations were identified with the *resource context* of the buffer). Although the Sequence Diagram shows that these operations are in the same `AcquireProc` task in the software, they are separated in the model into a nested pseudo task `AcquireProc2`, which executes while `AcquireProc` is blocked. This only breaks a calling cycle which would otherwise appear around `Buffer`, and does not affect the behaviour of the model. Passing the buffer to `Store` is also modeled as a call from the `Buffer` virtual task to `Store`. It is a second phase call because the reference task `VideoController,` the originator of the chain of requests, is not supposed to wait for the storing of the frame in the database, only `Buffer` must wait for it. `Store` finally calls the BufferManager task to release the buffer; however, to avoid another calling cycle in the model, the release is again modeled as an entry of a pseudo task BufMgr2.

Task multiplicities represent the number of identical replicated processes (or threads) that work in parallel serving requests from the same queue, or the number of logical resources of the same type (e.g., buffers). The parameter values for $P, packets per video frame and $B, disk operations to store a video frame, are both set to 8. The number of buffers $NBuf for the buffer pool was set to 1.

4. Performance Evaluation and Improvement

The model was solved by simulation to obtain the percentile values for delays, giving results for the user response times, throughputs, service times of entries and tasks, utilizations and waiting times for software or hardware resources, and probabilities of missing the deadlines. As mentioned above, the performance requirements are to meet a 1-second-deadline for both the Access Control scenario and the Acquire/Store Video scenario, with a 95% confidence level. In the LQN model, these requirements are translated to requiring the service time of the Video Controller task (also called its cycle time) and the response time of the User task to be less than 1 second with probability of 95%.

4.1. Base case for performance evaluation

At the very beginning we did not know if the performance of the system could be satisfied and if there were some bottleneck or design pitfalls in the system. Therefore, we started the evaluation with a base case, which is using a single copy for all software and hardware resources, except for the network, database and disk (whose multiplicities were set according to the system design).

Table 1 shows the LQN results for the base case. It lists the cycle time for polling all cameras, the response time for a human user accessing the door, the normalized utilizations of the software and hardware resources and the probabilities of missing deadlines. Here we list only the normalized utilizations of the most heavily loaded resources. The *normalized utilization* is the ratio of the mean number of busy resources to the total number of the corresponding resources. A resource with a normalized utilization of 100% is fully saturated. By using normalized utilization, we can assess at a glance the actual usage of a resource without worrying about the total number of resources.

Checking the simulation results for the base case, we can see that the internal throughputs and utilizations are constant, and the cycle time to poll all cameras grows linearly, as the number of cameras is increased. This follows from the design polling the cameras one at a time. No new polling request is generated before the AcquireProc completes the polling of a camera and returns.

Table 1. Simulation results for the base case

Ncam	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
	Cycle (sec)	User (sec)	AcqProc	Buffer	StoreProc	AppCPU	Cycle	RUser
10	0.327	0.127	0.960	0.9998	0.582	0.549	0	0.031
20	0.655	0.138	0.963	0.9999	0.582	0.545	0.0007	0.036
30	0.983	0.133	0.964	0.9999	0.582	0.544	0.4196	0.038
40	1.310	0.129	0.965	0.9999	0.582	0.544	0.9962	0.034

The results show the performance requirement for the Access Control scenario can be achieved in all cases, with about 3%-4% probability of missing the deadline. However, the other requirement for the Video Acquire scenario cannot be fulfilled for 50 cameras, or even for 30. The probability of missing the deadline jumps from 0.07% for 20 cameras to 42.96% for 30 cameras, and to 99.62% for 40 cameras. This is clearly unsatisfactory.

In this paper, we use the term *capacity* to indicate the maximum number of cameras the system can support while still meeting the 5% deadline miss requirement. From the simulation results, we learn the capacity for the base case is just above 20, which is far from satisfactory. Therefore, we have to analyze more deeply the LQN performance results, in order to identify bottlenecks and to eliminate design pitfalls.

We can see that in the base case two tasks are nearly fully saturated, `AcquireProc` and `Buffer`. This is a typical example of the bottleneck push-back phenomenon described in [8]. Here `Buffer` can be deemed as a server, which provides services to `AcquireProc`. In spite of being saturated, `AcquireProc` is not the bottleneck, because its underlying server `Buffer` is also saturated. On the other hand, `Buffer` is the real bottleneck, because it is saturated while its direct or indirect servers are not saturated.

As suggested in [8], a standard inexpensive way of relieving a software bottleneck is by cloning (i.e., making multiple identical copies of the constrained server that share the same incoming request queue). In the case of the buffer pool, clones take the form of additional buffers. We also expect that by relieving one bottleneck, another bottleneck may appear, and that we can repeat the process until the bottleneck is either pushed down to the hardware resources (hardware saturation), or up to the client end (adequate capacity for the offered load). Hardware bottlenecks can also be solved by cloning in the form of multiple devices such as multiprocessors.

There are other ways of solving bottlenecks, such as changing the scenario design, using more efficient strategies for scheduling, modifying the deployment, etc.

Furthermore, when all bottlenecks are eventually solved or have been pushed to client end, we have to depend on other methods for further improving the performance, as discussed in the next section.

4.2. Strategy for improving the performance

Our strategy for improving the system performance is sketched in Figure 6. We start with the base case of the performance model, which is translated directly from the design. Solving the model by simulation, we can get the performance result data from which we can identify the performance problems. If the performance requirements were satisfied, that means the current design is fine. Otherwise, we further explore the performance results, looking for bottlenecks. If a bottleneck is found, we can solve it by cloning the bottleneck resource, such as using multiple buffers, multi-threading software processes or using multiple processors. We can achieve this by modifying the performance model, then solving it with new parameters and repeating the same procedure.

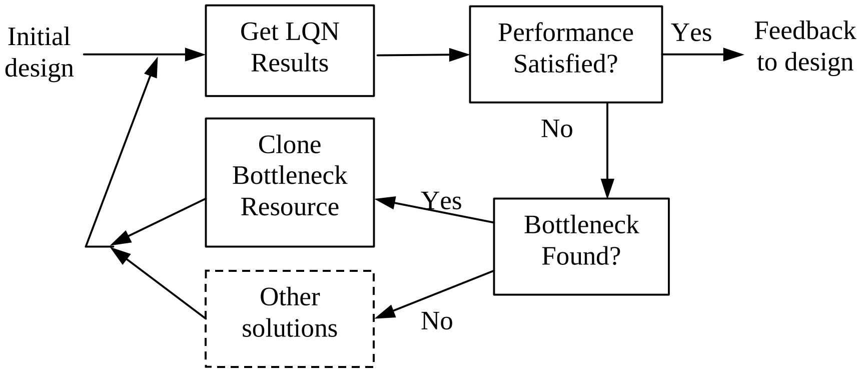


Figure 6. Strategy of performance improving for BSS

In [8] utilization measures are used to locate the bottleneck in client-server systems and rendezvous networks. In this paper, we use the normalized utilization, which has been defined in section 4.2, as one of the indicators. The most saturated resource has the greatest potential to be the bottleneck. However, to decide whether it is the real bottleneck, the system architecture should also be considered, because a client resource may be blocked by a server resource, which is in fact the real bottleneck. (Please note that a resource that requests a service is called a client resource, whereas one that provides the service is called a server resource.) Usually, a resource with a large number of outgoing (fan-outs) calls, second phase service, or incoming asynchronous calls can become easily the bottleneck.

Using multiple identical copies of resources is a straightforward way to solve the bottleneck. Usually, resolving one bottleneck in this way will push the bottleneck to another resource, either to a lower layer or to a higher one. Repeatedly, by adjusting the number of copies for different resources within the system, the bottleneck will move around, until is finally eliminated. In this paper, we call this procedure *system configuration tuning*. At that point, in open systems there is no saturated resource, and in closed systems the saturation is pushed back to the external client end.

There is a second path in the strategy given in Figure 6, which is seeking other solutions for performance improvement when no bottleneck could be identified. If the performance requirements still cannot be satisfied after tuning the system configuration, it is not because of limited resources. The cause may be heavy execution demand, long scenario paths, or lack of concurrency in the system. In this case, we take the second path of the strategy, for which there is no standard approach. The solution is usually project specific. Typical solutions include changing the scenario design, shortening long scenarios, decomposing large components, using more efficient scheduling strategies, and modifying the deployment.

After applying these solutions, bottlenecks usually appear again in the system, because such solutions lead to a more efficient, and therefore more intense, usage of the existing resources. Thus we are back on the main path of the strategy. By repeating the strategy, we will eventually reach a point where the performance requirements can be met (assuming that the requirements are reasonable). Then we translate the changes that were applied to the performance model in terms of system configuration information and software design description, and give feedback to the designer.

4.3. Using multiple copies or clones of resources

This section describes efforts to solve the software and hardware bottlenecks by using multiple copies of resources (i.e., by cloning). As discussed in the base case evaluation (section 4.1), the first bottleneck is the `Buffer`. Therefore, our first solution step is to use multiple buffers. Many cases were solved for the system under different configurations, i.e. with different numbers of cameras and buffers. Table 2 lists the data for 40 cameras and different numbers of buffers.

As seen in Table 2, the performance improvement due to multiple buffers is obvious. The probability of missing the cycle time deadline drops greatly, from 99% for 1 buffer to 9.35% for 10 buffers, but the requirement of a 5% probability for missing the deadline is still not achieved. We can see that now there is a newly saturated resource, namely `StoreProc`, which is the real bottleneck in the case with 10 buffers. We notice that the normalized utilization of `Buffer` drops at first as `NBuf` grows from 1 to 4, then raises slightly afterwards. However, the normalized utilization of `Buffer` is only high (over 84%) in the case with 10 buffers, because it is blocked by its server resource `StoreProc`. The bottleneck is pushed to a lower layer in the model.

Table 2. LQN Results for using multiple Buffers (40 cameras)

NBuf	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
	Cycle (sec)	User (sec)	AcqProc	Buffer	StoreProc	AppCPU	Cycle	RUser
1	1.309	0.137	0.965	0.9999	0.583	0.544	0.9961	0.034
2	1.016	0.132	0.975	0.8762	0.800	0.702	0.5503	0.032
3	0.941	0.132	0.980	0.8235	0.893	0.756	0.2506	0.036
4	0.911	0.131	0.983	0.8042	0.936	0.782	0.1597	0.032
7	0.879	0.132	0.986	0.8136	0.984	0.810	0.0948	0.033
10	0.872	0.129	0.987	0.8437	0.995	0.817	0.0935	0.034

Therefore, the second solution step is to clone the `StoreProc` task. Table 3 shows the results for the case of 40 cameras with 4 buffers. We can see that with 2 `StoreProc` threads, the probability of missing the deadline has dropped to a satisfying level. The system capacity is now above 40 cameras, about double that for the base case.

Table 3. LQN Results for multi-threading StoreProc (40 cameras, 4 Buffers)

No. of Store Proc	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
	Cycle (sec)	User (sec)	AcqProc	Buffer	StoreProc	AppCPU	Cycle	RUser
1	0.911	0.131	0.983	0.8042	0.936	0.782	0.1597	0.032
2	0.756	0.137	0.946	0.5805	0.616	0.940	0.0022	0.035
3	0.743	0.139	0.932	0.5484	0.441	0.956	0.0015	0.039

According to the same reasoning, the new bottleneck is `ApplicationCPU`. The bottleneck was pushed from software resources to hardware resources. This bottleneck can be relieved by using a multi-processor, giving the results in Table 4.

The simulation results show that 2 ApplicationCPUs are enough for solving the hardware bottleneck here. Using a double-processor is a typical configuration strategy. The system capacity is now 50 cameras, with 4 `Buffers`, 2 `StoreProc` threads and 2 `ApplicationCPUs`. This is 2.5 times higher than the base case and achieves our initial goal of system capacity.

The point has been reached where, except for the reference task `VideoController`, there is only one saturated resource in the system, namely

AcquireProc. We may consider it as the bottleneck. However, LQN results show that cloning it gives no improvement. In fact, its queue contains at most one request at any time and never grows. The performance is not limited by a lack of resources now, but by design limitations.

Table 4. LQN Results for using multiple processors for ApplicationCPU
(40 cameras, 4 Buffers, 2 StoreProc threads)

No. of CPU	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
	Cycle (sec)	User (sec)	AcqProc	Buffer	StoreProc	AppCPU	Cycle	RUser
1	0.756	0.137	0.946	0.5805	0.616	0.94	0.0022	0.035
2	0.648	0.127	0.995	0.6111	0.653	0.549	0	0.035
3	0.644	0.128	0.997	0.6105	0.652	0.368	0	0.033

Now we take the second path of our performance-improving strategy.

4.4. Changing the Scenario Design to introduce more concurrency

As mentioned before, there are two saturated tasks in the model, the reference task VideoController, and AcquireProc. A reference task in a closed model drives the system by generating workload, and usually represents the behaviour of an external client. Its normalized utilization is always 1, because it is always blocked by all of the services in the scenario.

Here VideoController is similar to an external client, although it is a part of the system. The VideoController has to wait for the message returned from AcquireProc before generating the next polling call. The call from the VideoController to AcquireProc is synchronous, and all of the work of AcquireProc is finished in its first phase. Therefore, only one instance of AcquireProc can be activated at any time. The system suffers from too much serialization, and the system capacity is limited by the duration of the scenario which polls one video camera.

To solve this problem, a change in the system design is required. A key point is to enable concurrent activations of the AcquireProc task by multi-threading the process. A solution is to move the calls made by AcquireProc for allocating and using the buffer into its second phase, and making an early reply to VideoController. Then VideoController can generate its next polling call earlier.

After introducing more concurrency into the system, the software and hardware bottlenecks appear again. We come to the main path in our strategy again, tuning the system configuration. During the tuning, the bottleneck moves around within the system. For example, the bottleneck may move to task `AcquireProc` or `StoreProc` as well as `ApplicationCPU`. There are different configuration strategies to address these problems. By repeatedly tuning the system configuration on software and hardware, the system performance can be improved dramatically.

Table 5 shows some system performance results under different configurations. Here we aim for a capacity of 100 cameras and increase the number of threads for `Buffer`, `AcquireProc`, `StoreProc` and the number of `ApplicationCPU` step by step. Finally, with 3 `AcquireProc`, 6 `StoreProc` threads and 3 `ApplicationCPU`, we manage to achieve the 1-second-deadline for the cycle time for the case with 100 cameras with a probability of 99.95%. This capacity is 5 times higher than the base case, and twice the capacity before changing the design.

Table 5. LQN results for with more concurrency case (100 cameras, 10 Buffers)

Multiplicity (Acquire, Buffer, Store, App. CPU)	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
	Cycle (sec)	User (ms)	Acquire Proc	Buffer	Store Proc	App CPU	Cycle	RUser
2, 4, 2, 2	1.250	0.133	0.988	0.923	0.886	0.710	0.9995	0.0332
2, 10, 6, 3	0.837	0.132	0.988	0.689	0.751	0.707	0.0057	0.0307
3, 10, 6, 3	0.768	0.134	0.983	0.895	0.910	0.769	0.0005	0.0352

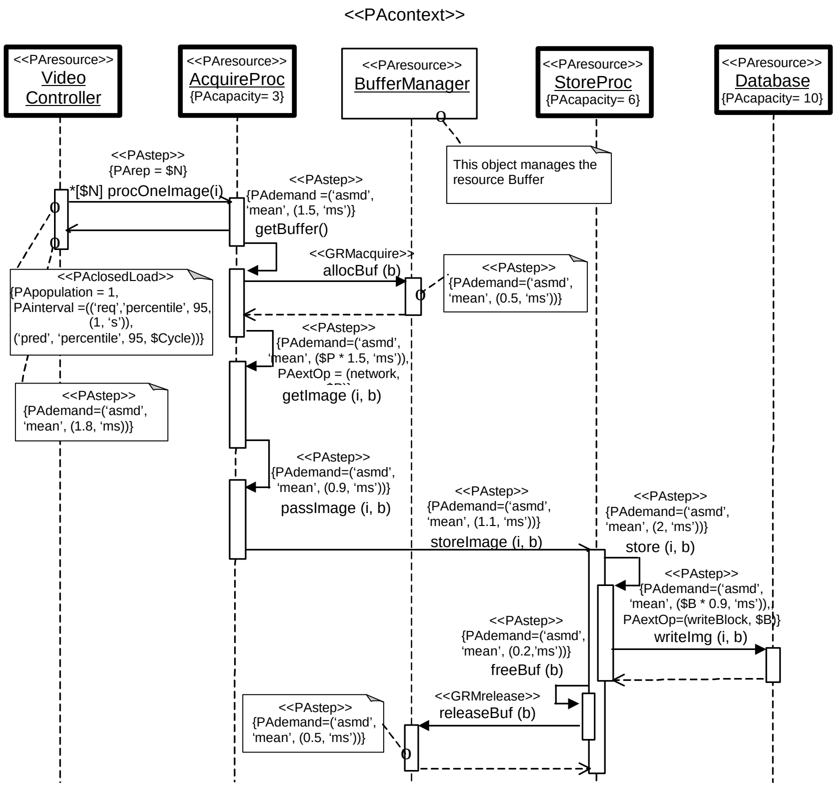
The results also show that `AcquireProc` and `StoreProc` tasks are saturated again. Therefore, we expect that there is more room for improving the capacity by further tuning the system configuration.

4.5. Feedback into the Software Design

The exploration described above is carried out in the space of LQN models, but the final result must be transferred back into the software design. The modified sequence diagram in Figure 7 shows two kinds of change:

?? suggestions on multithreading of active objects are represented by the tag `PAcapacity`, as in the object `AcquireProc` and `StoreProc`.

?? the design change to `AcquireProc`, to put all the buffer processing into a second phase. The second phase is incorporated into the specification by changing the synchronous message from `VideoController` to `AcquireProc` into two asynchronous messages, for the `VideoController`'s request and the corresponding reply. After sending the reply, `AcquireProc` invokes its own `getBuffer` operation and everything that follows, as the second phase.



<<PAcontext>>
<<PAresource>>
Video Controller
<<PAresource>>
AcquireProc
{PAcapacity= 3}
<<PAresource>>
BufferManager
<<PAresource>>
StoreProc
{PAcapacity= 6}
<<PAresource>>
Database
{PAcapacity= 10}
This object manages the resource Buffer
<<PAstep>>
{PArep = $N}
*[$N] procOneImage(i)
<<PAstep>>
{PAdemand =('asmd', 'mean', (1.5, 'ms')}
getBuffer()
<<GRMacquire>>
allocBuf (b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.5, 'ms'))}
<<PAclosedLoad>>
{PApopulation = 1,
PAinterval =(('req','percentile', 95, (1, 's')),
('pred', 'percentile', 95, $Cycle))}
<<PAstep>>
{PAdemand=('asmd', 'mean', ($P * 1.5, 'ms')),
PAextOp = (network,
getImage (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (1.8, 'ms))}
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.9, 'ms'))}
passImage (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (1.1, 'ms'))}
storeImage (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (2, 'ms'))}
store (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', ($B * 0.9, 'ms')),
PAextOp=(writeBlock, $B)}
writeImg (i, b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.2,'ms'))}
freeBuf (b)
<<GRMrelease>>
releaseBuf (b)
<<PAstep>>
{PAdemand=('asmd', 'mean', (0.5, 'ms'))}

Figure 7. Modified Sequence Diagram for Acquire/Store scenario

As this example shows, some kinds of feedback can be presented in the design model just by using tag values defined in SPT Profile. However, others require deeper changes to be made by the designers. For instance, multithreading may require changes to synchronize threads or to maintain consistency in data shared by the threads; partitioning an object into two concurrent objects would require new classes.

Techniques to support these changes, possibly based on patterns to solve typical problems that arise, will be needed.

5. Related Work

Other researchers have developed approaches to convert UML software specifications into different kinds of performance models, including Queueing networks (Smith and Williams [17]), Petri Net models (Merseguer et al [9], and work surveyed by Pooley

[14]), and Stochastic Process Algebras (Canevet et. al. [3]). Layered queueing models were produced from UML activity diagrams by Petriu and Shen in [13]. Other kinds of specifications have also been converted, for instance LQN models were produced by automated transformation of a non-UML scenario specification in [11]. Most of these papers focus on the conversion process itself.

The use of a model to improve a software design is the focus of the "performance principles" in Smith's book [17] and of her work with Williams on antipatterns [16]. In [10], Menasce and Gomaa modeled an information system and redesigned database transactions for performance. In [9] Merseguer et.al. evolved the design of a wireless application. Improvements related to software bottlenecks were defined by Neilson et. al. in [8], including their systematic removal by introduction of task threads. In [12], Petriu and Woodside showed how part of an e-commerce system could be redesigned to achieve performance goals.

The question of how to navigate through the results of a software model, to identify the source of performance problems, was discussed in a general way in [19]. In a broader context of computer and network systems (not just software designs) various kinds of reasoning aids for performance diagnosis have been described. An example for distributed computing (with references to other work) is described by Hellerstein [5] in the form of a QPD (Quantitative Performance Diagnosis) algorithm. It estimates how much certain system attributes such as traffic levels and hardware capacities contribute to performance problems, to assist in hardware improvement. QPD includes navigation of the measurements, guided by a model-like view of the system.

6. Conclusion

A layered performance model has been used to expose performance problems in a software design, and to evaluate design changes, in a case study which combines client-server aspects with real-time deadlines including video frame transfer.

Three kinds of performance issue emerged in this study. First, there was a question of video buffers; it is essential to provide multiple buffers, to overlap video acquisition from the cameras with the storage of the frames. For a capacity of 50 cameras, double buffering (with two buffers alternating) is not as good as four buffers used in rotation. For a higher capacity of 100 cameras, 10 buffers are needed. Second, there are deployment and configuration issues, such as providing multithreaded tasks for adequate concurrency. Finally a change in the software execution sequence within the video acquisition task was found beneficial. By providing an early reply to the control task that manages the acquisition loop, greater concurrency in video acquisition could be achieved and a much higher capacity was obtained.

A notable result here is that the software change could only be identified as beneficial, after the thread configuration and buffer questions had been resolved. Without the buffers and the threads, the early reply from `AcquireProc` would not help (the results for that case are not given here, but they are identical to the base case). So we have evidence for a general principle for software design improvement:

Holistic Improvement Principle: *Software design improvements can only be evaluated in the context of the best possible deployment and configuration alternatives.*

The case study shows how drastic changes in the design can be inserted into a performance model and evaluated very quickly and inexpensively. Many possible changes can be assessed, the best ones are selected, and finally the design is updated to incorporate the beneficial changes.

A question raised and not resolved here is the systematic navigation of the model results to identify and rank the potential design changes at each step. This is one subject of the PUMA project [15] for integration of UML design and performance engineering. An outline of the complete PUMA project is shown in Figure 1, including UML model transformation, performance model experimentation, and feedback of results.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Simona Bernardi, Susanna Donatelli, Jose Merseguer, “From UML sequence diagrams and statecharts to analysable petri net models”, Proc. 3rd international workshop on Software and performance 2002, Rome, Italy, pp35-45.

[2] Grady Booch, Ivar Jacobson, and James Rumbaugh, The Unified Modeling Language User Guide, Reading Mass.: Addison-Wesley, 1999.

[3] C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens, “Performance modelling with UML and stochastic process algebras”, Proc IEE on Computers and Digital Techniques, October 2002.

[4] H. E. El-Sayed, Don Cameron, C. M. Woodside, “Automation Support for Software Performance Engineering”, Proc Joint Int. Conf on Measurement and Modeling of Computer Systems (Sigmetrics 2001/Performance 2001), Cambridge, MA, June 16 - 20, 2001, ACM order no. 488010, pp 301-311.

[5] Joseph L. Hellerstein. “A General-Purpose Algorithm for Quantitative Diagnosis of Performance Problems”, Journal of Network and Systems Management, 2001.

[6] Prasad Jogalekar, Murray Woodside, “Evaluating the Scalability of Distributed Systems”, IEEE Trans. on Parallel and Distributed Systems, v 11 n 6 pp 589-603, June 2000.

[7] Object Management Group, “UML Profile for Schedulability, Performance, and Time Specification”, OMG Adopted Specification ptc/02-03-02, July 1, 2002.

[8] J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, “Software Bottlenecking in Client-Server Systems and Rendez-vous Networks”, IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 776-782, September 1995.

[9] Jose Merseguer, Javier Campos, Eduardo Mena, "Performance analysis of Internet based software retrieval systems using Petri Nets", Proceedings of the 4th ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems 2001 , Rome, Italy, pp 47 – 56.

[10] D. Menasce and H. Gomaa, " A Method for Design and Performance Modeling of Client/Server Systems," IEEE Transactions on Software Engineering, vol. 26, no. 11 pp. 1066-1085, 2000.

[11] Dorin Petriu, Murray Woodside, "Software Performance Models from System Scenarios in Use Case Maps", Proc. 12 Int. Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (Performance TOOLS 2002), London, April 2002.

[12] Dorin Petriu, Murray Woodside, "Analysing Software Requirements Specifications for Performance", Proc. 3rd Int. Workshop on Software and Performance, Rome, pp 1 – 9, July 2002.

[13] D.C.Petriu, H.Shen, " Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications", in Computer Performance Evaluation - Modelling Techniques and Tools, (Tony Fields, Peter Harrison, Jeremy Bradley, Uli Harder, Eds.) Lecture Notes in Computer Science 2324, pp.159-177, Springer Verlag, 2002.

[14] R. Pooley, "Software Engineering and Performance: a Roadmap", in The Future of Software Engineering, part of the 22nd Int. Conf. on Software Engineering (ICSE2000), Limerick, Ireland, June 2000, pp. 189-200.

[15] PUMA (Performance from Unified Model Analysis), www.sce.carleton.ca/rads/puma/.

[16] C. Smith and L. Williams, "Software Performance Antipatterns", in Proceedings of the Second International Workshop on Software and Performance (WOSP2000), Ottawa, Canada, September 17-20, 2000, pp. 127-136.

[17] C. U. Smith and L. G. Williams, Performance Solutions. Addison-Wesley, 2002.

[18] C.M.Woodside, D. Petriu, "Performance Analysis with UML", Chapter 11 in " UML for Real: Design of Embedded Real-Time Systems", Editors: Luciano Lavagno, Grant Martin, and Bran Selic, Kluwer Academic Publisher, New York, to be published in 2003.

[19] C.M. Woodside, " A Three-View Model for Performance Engineering of Concurrent Software", IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 754-767, Sept. 1995

[20] Murray Woodside, "Software Resource Architecture", Int. Journal on Software Engineering and Knowledge Engineering (IJSEKE), v 11, pp 407-429, 2001.