Layered Bottlenecks and Their Mitigation

Greg Franks, Dorina Petriu, Murray Woodside, Jing Xu
Carleton University
{greg | petriu | cmw | xujing}@sce.carleton.ca

Peter Tregunno
Alcatel
peter.d.tregunno@alcatel.com

Abstract

Bottlenecks are a simple and well-understood phenomenon in service systems and queueing models. However in systems with layered resources bottlenecks are more complicated, because of simultaneous resource possession. Thus, the holding time of a higher-layer resource, such as a process thread, may include a small execution demand, but a large time to use other resources at a lower layer (such as a disk). A single saturation point may in fact saturate many other resources by push-back, making diagnosis of the problem difficult. This paper gives a new corrected definition of a layered bottleneck, and develops a framework for systematic detection of the source of a bottleneck, for applying improvements and for estimating their effectiveness. Many of the techniques are specific to layered bottlenecks.

1 Introduction

When a system is throughput-limited but none of the devices (processors, disks, bus, network) are saturated, the bottleneck is some other kind of resource. Here, these are called "layered bottlenecks", using a model which describes computer systems, and many other systems.

In simple service systems ("flat" resource systems) a job is using resources one at a time. The most heavily loaded server is the bottleneck, and if it is relieved by some means, the next most heavily loaded server takes over [7].

Layered bottlenecks arise from simultaneous resource possession. The holding time of a resource R may include waiting for and using other "lower" resources, one at a time. For example, while holding a process thread resource, a program may use the disk.

A layered bottleneck resource B has the following features:

1. B is a saturated resource, that is its units are all in use almost all the time.
2. Resources "below" it are unsaturated. As an example, a processor may have low utilization in a memory bottleneck.
3. It tends to spread saturation to resources which include it in their holding times. This is "pushback" of load away from the bottleneck.
4. Thus, there may be many saturated resources which are not themselves the bottleneck.

This makes the understanding of layered bottlenecks more difficult than flat resource system bottlenecks.

Although bottlenecks are often considered an asymptotic property of systems which are heavily loaded, the present discussion considers the limiting factors in any system, regardless of the workload intensity. If a closed system is lightly loaded, its bottleneck is defined to be at the source of workload.

Layered bottlenecks were described and named "software bottlenecks" in [11], but they were familiar to system and database programmers long before. The name layered bottlenecks recognizes that they are a feature of many kinds of resources, not only software.

Layered bottlenecks have been described by many authors under a variety of names. Thrashing in virtual memory systems is a well-known example [7]. Maly et al described a bus bottleneck in a switch, which was layered over the processor and memory resources [9]. Dilley et al. found a process thread bottleneck in a web server [2], and threads were also featured in [11]. Smith and Williams give a tutorial example with a global lock which forms a layered bottleneck, and limits an ATM system to a very low throughput [16]. Cechet et al measured a web-based application with a throughput limit due to database lock contention [1]. Petriu et al. , and Xu et al., describe a sequence of steps to mitigate bottlenecks involving process threads, a buffer pool, class interference, and excessive synchronization [12], [18]. Gerndt et al describe a cache which is a bottleneck [5], due to thrashing.

This paper describes a framework for understanding layered bottlenecks, gives an improved definition, and estimates the effect of the possible improvements (mitigations) in a given case.

2 Layered Resources

We may regard resources as servers with queues. Everything done during the holding time of a resource

T is part of its service time X_T. If another resource t is used during this time, its service time is incorporated into X_T, as is any time spent waiting to obtain t. Thus:

> ***Layered services*** *are services which include other services and their waiting; they are a nested form of simultaneous resource possession.*

Resource T above *depends* on resource t, because t is required (always or sometimes) by the holding-time operations of T. The dependency can be shown as a directed graph with nodes for resources, and an arc from resource T to each resource t that it depends on (see Figure 1). We will say that T is in a *higher layer* than t. Attention is restricted to acyclic graphs (to exclude systems with resource deadlocks), and to resources which are released in reverse order to acquisition (giving nested holding times). Many extended queueing networks are layered.

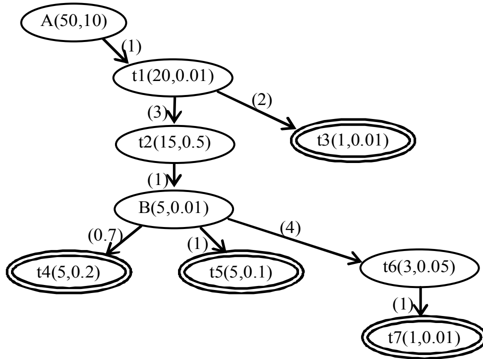


Figure 1. Resource Dependency Graph

In Figure 1,

- each node T is labeled by (m_T, Z_T) where:
  - m_T = the multiplicity of the resource T,
  - Z_T = the "local service" part of the holding time of T (that is due only to T itself, and not to nested resource use),
- each arc (T,t) is labeled by $y(T,t)$, the mean number of requests to another resource t during a holding time of T,
- the system has a closed workload driven by source node A, with 50 "users",
- the graph imposes a partial order on resources, ordered from top to bottom of Figure 1,
- requests can jump over layers (not shown here),
- the set of nodes connected by arcs directly from resource T will be called *RequestedBy*(T), those connected by arcs to T are the set *RequestsTo*(T).

The topmost node A is special, as it has no requests. Such resources model closed *sources* of workload to the system. A represents m_A entities which cycle forever, alternating between a think time of mean Z_A and requests for resources which are top-level servers.

The leaf nodes (with double outlines) are ordinary queueing servers with mean service time Z_T. The set of leaf nodes will be called *Processors*, as CPUs have this role in computer systems. They are not limited to CPUs, however.

The service time X_T of any resource T that is not a processor is defined recursively in terms of waiting times W_t and service times X_t of resources t, in the set *RequestedBy*(T) of nested requests:

$$X_T = Z_T + \Sigma_{t \text{ in } RequestedBy(T)}[y(T,t)\ (W_t + X_t)] \quad (1)$$

Real systems tend to have many classes of requests, but for simplicity we will first assume a single class of requests to each resource.

From the system or a model we can obtain performance measures for each resource:

- X_T = service time of T
- W_T = waiting time for requests to T
- R_T = response time = $W_T + X_T$
- f_T = throughput of resource T (acquisitions/sec)
- $U_T = f_T\ X_T$ = utilization of T (mean number of busy units of the resource)
- sat_T = the saturation level of $T = U_T/m_T$ (utilization relative to the number of units of resource T)

In a closed system each source of workload (such as A in Fig. 1) has throughput f_A and the system response time is its cycle time m_A / f_A, minus its think time Z_A:

Response Delay at $A = (m_A / f_A) - Z_A$

2.1 Examples

With these definitions we can show an example of performance measures for the system of Figure 1, with the parameters and result values in Table 1.

The throughput at A, as a function of the number of users m_A, follows a classic saturation curve shown in Figure 2. The fact that B is the one limiting the throughput is confirmed by the fact that an increase in m_B yields a higher throughput. We obtain the following values for (m_B, f_A) when m_B is varied:

(5, 3.06), (7, 3.62), (9, 3.82), (>15, 3.87)

On the other hand an increase in other resources (e.g. in m_{t1}) does not change the throughput at all.

Table 1. Example: some parameters and results

Res. T	Z_T (sec)	X_T (sec)	m_T	f_T /sec	U_T	sat_T
A	10.0	16.3	50	3.06	50	1
t1	0.01	6.02	20	3.06	18.4	0.92
t2	0.5	1.60	15	9.18	14.6	0.97
t3	.01	0.01	1	6.12	0.06	0.06

B	.01	0.543	5	9.18	4.98	0.996
t4	0.2	0.2	5	6.43	1.29	0.26
t5	0.1	0.1	5	9.18	0.92	0.18
t6	0.05	0.063	3	36.7	2.31	0.77
t7	0.01	0.01	1	36.7	0.37	0.37

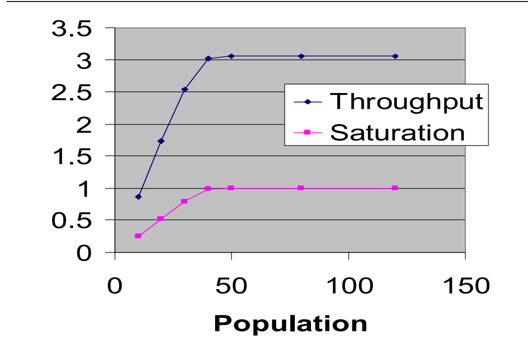
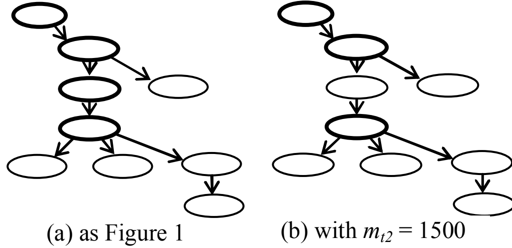


Figure 2. Saturation as Load Increases

A pattern emerges in Figure 3(a) which shows a bold outline for every task with $sat_T > 0.9$ (an ad hoc indicator of saturation of resource *T*). Upper layers are saturated, lower layers are not. The boundary resource *B* is the bottleneck which causes the saturation, and the set *Above(B)* is saturated by pushback.



(a) as Figure 1 (b) with $m_{t2} = 1500$

Figure 3. Resource Dependency Graph showing Saturated Resources in Bold

Based on this pattern, a "bottleneck strength" measure was defined in [11]:

$BStrength\text{-}old_T = sat_T / (\max_{t \text{ in } RequestedBy(T)} sat_t)$

The resource with the largest value was defined as the bottleneck. Table 2 shows the strength values for each task. Column 3 identifies *B*, with saturation over 0.9 and "old" strength measure 1.29.

Table 2. Bottleneck Strength Values

	Case with $m_{t2} = 15$			Case with m_{t2}=1500		
T	sat_T	*BStrength*		sat_T	*BStrength*	
		old	*new*		*old*	*new*
A	1	1/.92 =1.09	1/.996 =1.004	1	1/.92 =1.09	1/1 =1.0
t1	0.92	.92/.97 =.95	.92/.996 =.92	0.92	**.92/.012 =76.7**	**.92/1 =.92**
t2	0.97	.97/.996 =.97	.97/.996 =.97	0.012	**.012/1 =.012**	**.012/1 =.012**
t3	0.06	---	---	0.06	---	---
B	0.996	.996/.77 =1.29	.996/.77 =1.29	1	1/.77 =1.29	1/.77 =1.29
t4	0.26	---	---	0.26	---	---
t5	0.18	---	---	0.18	---	---
t6	0.77	.77/.37 =2.08	.77/.37 =2.08	0.77		
t7	0.37	---	---	0.37	---	---

However there is a defect in the measure *BStrength-old*, illustrated by modifying multiplicity m_2 to 1500, instead of 15. This gives Figure 3(b) and the saturation values on the left side of Table 2. The value of sat_{t2} becomes very small, and the largest value of *BStrength-old* is at *t1*, even though *B* is still the factor which limits the throughput. The pushback is transmitted through *t2* by its holding time, even though *t2* itself is not saturated.

This defect is corrected in the new definition:

$$BStrength_T = sat_T / sat_{Shadow(T)} \quad (2)$$

$$Shadow(T) = \arg\max_{t \text{ in } Below(T)} sat_t \quad (3)$$

where $\arg\max_t sat_t$ is the task *t* with the largest value of sat_t. Values of *BStrength* are shown in Table 2 for both cases, and correctly identify the bottleneck as task *B*. It has:

- saturation over 0.9 (this threshold may depend on the goals of the system)
- the largest value of *BStrength*.

The effect of the bottleneck is to limit the system throughput. The maximum possible throughput at B is $(5/X_B)$/sec = 9.2 requests/sec. The system throughput f_A is proportional to f_B:

Throughput proportionality (Forced Flow Law): *The rates of requests for all resources have fixed ratios.*

This follows from the mean number of requests made during a holding time of any resource *T*, to members of *RequestedBy(T)*. Thus we can write:

$$f_T = \Sigma_{t \text{ in } RequestedBy(T)}\, y(t,T)\, f_t$$

This homogeneous set of linear equations can be solved for every f_T in terms of f_A, with the constant of proportionality $y(A,T)$:

$$f_T = y(A, T)\, f_A \quad (4)$$

A ratio $y(T,t) = y(A,t)/y(A,T)$ is defined for request frequencies of any resources *T* and *t*. Using $y(T,t)$, Eq. (1) has the alternative form:

$$X_T = Z_T + \Sigma_t\, [y(T,t)\ (W_t + Z_t)] \quad (1a)$$

Also, using Eq. (2) we can write the system throughput in terms of the bottleneck throughput f_B:

$$f_B = m_B / X_B = y(A,B)\, f_A$$

$$f_A = m_B / (X_B\, y(A,B)) \quad (5)$$

Definition of a Layered Bottleneck

A layered bottleneck is defined as a saturated resource which actively limits the system throughput.

For bottleneck identification it is necessary to set an ad-hoc resource saturation threshold sat^*. Then

(a) if one or more resources in *Processors* has $sat_t > sat^*$, then:

$$B = \arg\max_{t \in Processors} sat_t$$

(b) else if one or more other resources has $sat_t > sat^*$, B is any resource which satisfies both of:

- $sat_B > sat^*$
- $B = \arg\max_{T \notin Processors} Bstrength_T$

2.2 Mitigation

Eq (5) dominates the end-to-end performance of the saturated system. As in ordinary queueing networks, the performance of a bottlenecked system is relatively insensitive to changes in parameters away from the bottleneck. To relieve (mitigate) the bottleneck at B requires changing one or more of

- $y(A,B)$, the mean requests for B per end-to-end response,
- X_B, the mean holding time of B,
- m_B, the units of resource at B.

Because Eq. (5) depends on contention delays via X_B, bottlenecks can be identified only after evaluating performance. This may use measurement, simulation, or solution with a layered queueing solver.

2.3 Analysis Tools

Layered resources can be analyzed as layered queueing networks (LQNs), which are a class of extended queueing networks defined for this situation (and for more general cases, including open arrivals and multiple classes of service). Solution methods for LQNs have been described in [3][4][10][13][14].

Fast Optimistic Bound Analysis: A simple calculation based only on service times, and ignoring the waiting term W_t in Eq (1a), is often effective. It calculates an "optimistic holding time" X_T^-, optimistic throughputs f_A^- and f_T^- and an "optimistic utilization" U_T^- all based on replacing Eq (1a) by:

$$X_T^- = Z_T + \Sigma_t [y(T,t)\, Z_t] \qquad (1b)$$

The optimistic system throughput f_A^- is then the largest feasible value, given the capacities of all the resources. Every resource utilization must satisfy (using optimistic values):

$$U_T^- = f_T^- X_T^- = f_A^- y(A,T)\, X_T^- =< m_T$$

so f_A^- is set to the largest value that satisfies this for every resource T in the system. This gives:

$$f_A^- = \min_T [m_T / [y(A,T)\, X_T^-]$$

$$sat_T^- = f_A^- y(A,T)\, X_T^- / m_T$$

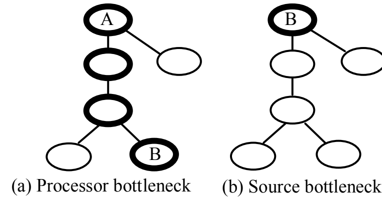
This "Optimistic Bounds Analysis" is elaborated for multiclass sytems in [17]. It works on the assumption that large queues occur with reduced relative service capacity, so the optimistic saturation will be largest where the actual saturation is largest. This assumption is more effective for resources below the bottleneck than above it, but that is sufficient for locating the bottleneck resource. It gives substantial errors in holding times and utilizations for resources above the bottleneck because the long wait at the bottleneck resource should be included, and is not. However, exact utilization values for resources above the bottleneck are not needed for locating the bottleneck, for recommending a mitigation strategy, or for estimating its probable effect.

2.4 Asymptotic Cases

Special asymptotic cases are sketched in Figure 4. In Figure 4(a) the bottleneck is at the bottom, at a processor, showing what is normally regarded as a bottleneck, at a saturated device. The design of layered resources may reasonably be oriented to getting the maximum out of the physical processor resources, and thus towards pushing the bottleneck down to the processor layer.

In Figure 4(b) it is at the top, at the load source. This is normally regarded as a non-saturated system as it does not have enough users to saturate it anywhere. The user "resources" are busy all the time in every closed system, since they perpetually cycle through their operations.

Thus there is guaranteed to be a bottleneck somewhere in a closed layered resource system. When one bottleneck is relieved, another one takes over. The characteristics of this "next bottleneck" determine the effectiveness of each step in increasing the capacity.



(a) Processor bottleneck (b) Source bottleneck

Figure 4. Asymptotic Cases

It is not correct to think of eliminating system bottlenecks, only of improving performance to a desired level. There will be a bottleneck somewhere, but the resulting capacity and response time will be acceptable.

2.5 Multiple Classes and Open Workloads

Multiple classes arise with multiple sources, or where a resource has classes of service. As in ordinary queueing, classes of service have different parameters and measures. For class C, Z_T is replaced by $Z_{T,C}$, X_T by $X_{T,C}$, U_T by $U_{T,C}$ and $y(T,t)$ by $(y(T,C1;t,C2)$. The holding time calculation in Eq (1) becomes:

$$X_{T,C1} = Z_{T,C1} + \Sigma_{t,C2 \text{ in } RequestedBy(T,C1)}[y(T,C1;t,C2)(W_t + X_t)] \quad (7)$$

Saturation is calculated independent of class, by summing the class utilizations $U_T = \Sigma_C U_{T,C}$, and proceeding as for a single class.

An open workload gives a stream of requests from outside the system to some class at some resource at a defined rate, balanced by departures. The request can be passed to other resources (with routing probabilities, including a probability of departing the system). Open requests form a distinct class, but during a holding time for an open request, nested requests can be made for other resources, including waiting for them to complete. Thus they can generate closed sub-behaviours.

The LQNS analysis tool [3][4] models multiclass and open workloads in layered resource models.

3 Patterns and Roles in Layered Bottlenecks

From the viewpoint of the bottleneck resource B we can divide the system into three parts:

- *Above(B)* = the set of resources that depend on *B*, directly or indirectly.
- *Below(B)* = the set of resources that *B* depends on, directly or indirectly,
- *Beside(B)* = the rest.

There are also:

- *Sources* = the set of load-generating resources,
- *RequestsTo(B)* = the subset of *Above(B)* that depends directly on *B*,
- *RequestedBy(B)* = the subset of *Below(B*) that is requested directly by *B*
- *Processors* = resources with no dependencies.

These sets are indicated in Figure 5, for the same system as Figure 1. Processors have double outlines. The Shadow bottleneck *Sh(B)* is described below.

The maximum number of concurrent requests to *B* is its available concurrency *AvConcur(B)*. It can be computed recursively using

$$AvConcur(A) = m_A \quad \text{for } A \text{ in } Sources \quad (6)$$

$$AvConcur\,(T) = \Sigma_{t \in RequestsTo(T)}\,(\min_t(AvConcur(t),\, m_t) \quad \text{for other resources } T.$$

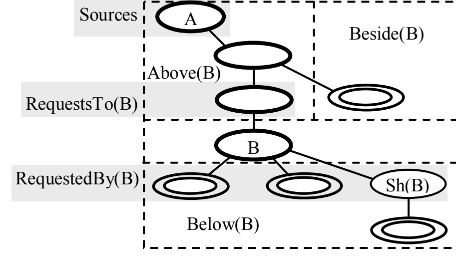


Figure 5. Resource roles relative to a bottleneck *B*

The resources in *Above(B)* are saturated not by their own workload but because ($W_B + X_B$) is large due to congestion at *B*. Increased workload increases the queue at *B* and service times in *Above(B)*.

On the other hand the resources in *Below(B)* are protected by the bottleneck, which prevents traffic from reaching them. The workload intensity in *Below(B)* is independent of the load on the system as a whole, if *B* is saturated. Many admission controls are bottlenecks which are deliberately introduced.

3.1 Taxonomy of Cases

The performance properties of the resources in their roles relative to *B* determines what will work in mitigating the effect of the bottleneck. Cases include:

- local bottleneck at *B* (Z_B is the only or major part of X_B): change is needed at *B,*
- resource-supported bottleneck (the *support* is *Shadow(B)*): reducing $y(B, Shadow(B)X_{Shadow(B)})$ is indicated,
- heavy bottleneck: if *AvConcur(B)* is much greater than m_B and sat_B is near unity, then the queue length at *B* is large and large improvements are possible by changes to m_B and X_B .
- light bottleneck: if *AvConcur(B)* is only slightly greater than m_B, there is limited potential for improvement.

3.2 Estimation of Effect

In examining various kinds of mitigation, we will estimate the potential improvement using the holding times calculated in the base case. These values naturally change under the mitigation, so the estimates are only approximate extrapolations of conditions in the base case. They may be optimistic or pessimistic.

4 Bottleneck Mitigation: Add Resources

An obvious way to relieve a bottleneck is to provide more resources, in the form of more resource units. If m_B is increased, Eq. (3) shows that throughput will increase in proportion to m_B at first. However this

change may make little difference, depending on the system context.

4.1 Max-Resources Analysis

A simple way to estimate suitable values for all multiplicities is to replace the system values by the largest feasible value, for all resources except sources, and resources with load-dependent Z_t, (the latter are simply too complex for this approach to be effective). Some resources by their nature exist in a single copy (an index page of a database and a critical section are two examples in software systems) or are constrained by economic factors. A derived *max-resource* performance model with these maximum values is solved, giving a bottleneck at resource B_{max}, and a throughput $f_{A,max}$. Then B_{max} is a fundamental limiting resource. The multiplicity of each other resource T may be set to a value somewhat greater than its utilization $U_{T,max}$, which is the mean number of busy resources in the max-resource case.

However this only considers resource multiplicity as a source of performance constraint, and it does not find the best combination of economical and effective design changes. Increasing the multiplicity of resources may be simple (as in changing the size of a buffer pool or thread pool) or difficult (as in introducing concurrency into sequential code).

4.2 Case of Leaf Node Bottleneck

If sufficient resources are added at a leaf node B they will reach the maximum number of concurrent requests that can be made by the dependent resources which we will call the available concurrency *AvConcur(B)*. If $AvConcur(B)>m_B$ then throughput at B increases, but is limited to a new value f^*_B given by:

$$f^*_B = AvConcur(B)/X_B = [AvConcur(B)/m_B]\, f_B$$

at which the resource B is starved of requests. This limits the system to a throughput estimated as:

$$f^*_{A,starve} = f^*_B \,/\, y(A,B) \qquad (8)$$

and the bottleneck migrates to another resource, as considered below.

4.3 Case of non-Leaf-Node Bottleneck

If m_B is increased B may become starved by limited available concurrency, or a resource below B may become saturated. A good candidate for this is the resource t in *Below(B)* with the largest value of sat_t, which we will call the Shadow Bottleneck:

$$Shadow(B) = \arg\max_{t \text{ in } Below(B)} (sat_t)$$

Assuming X_t stays the same and throughput increases due to larger m_B, resource *Shadow(B)* will be the new bottleneck, giving a new system throughput f^*_A:

$$\begin{aligned} f^*_{A,\text{shadow}} &= y(A, Shadow(B))\, f^*_{Shadow(B)} \\ &= y(A, Shadow(B))\, m_{Shadow(B)}/X_{Shadow(B)} \\ &= (sat_B/sat_{Shadow(B)})\, f_A \\ &= BStrength(B)\, f_A \qquad (9) \end{aligned}$$

In Figure 1, resource *t6* is the Shadow Bottleneck because its saturation value of 0.77 is the highest in *Below(B)* (which is {*t4, t5, t6*}); it is labeled *Sh(B)* in Figure 5. The limit on improvement is the smaller of $f^*_{A,starve}$ and $f^*_{A,shadow}$.

4.4 Migration of the Bottleneck

Predictions about the impact of a change are based on performance values at a nominal configuration, and must be checked. With that caveat,

- a bottleneck may migrate down to a Shadow bottleneck
- or up to a higher layer due to starvation of B.

In the latter case, a good candidate is the resource whose multiplicity is limiting in the min function in Eq (6), for *AvConcur(B)*.

4.5 Recommendation:

Increased resources are useful:

- for processor bottlenecks with a large value of the ratio $[AvConcur(B)/m_B]$
- for higher-level bottlenecks that satisfy both of
  - a large value of $sat_B/sat_{Shadow(B)}$
  - a large value of the ratio $[AvConcur(B)/m_B]$

If other factors cannot be changed, suitable resource levels can be found for all resources at once, by solving the derived max-resource model.

Examples: In computer systems: process pools, thread pools, buffer pools, and multiprocessing limits are software resources whose multiplicity can be increased. Multiprocessors and cluster sizes can be increased. In networks: window sizes for flow control, admission controls, links in parallel are all examples.

5 Bottleneck Mitigation: Reduce the Bottleneck Holding Time

The second factor that can give improvement is to reduce the holding time X_B of the bottleneck resource. From Eq (1a) there are three ways to do this:

- reduce the local service time Z_B, or any local service time Z_t that is included in X_B,
- reduce the requests to lower level resources.
- parallelize some local service, or some set of requests.

The impact of reducing any Z_t is given by Eq. (1a).

If a request parameter between any pair of tasks in B U *Below(B)* is reduced by an amount Δy, the reduction in $y(B,t)$ for any t has the linear form

$$\Delta y(B,t) = a + b\Delta y \qquad (10)$$

and the reduction in X_B :

$$\Delta X_B = \Sigma_t\, [\Delta y(B,t)(\, W_t + Z_t)].$$

Then Eq (4) gives

$$f^*_A = f_A / (1 - \Delta X_B / X_B)$$

(for a decrease in holding time, ΔX_B is positive). The limit to the improvement will come from rising throughputs which saturate some other resource. One possibility is *Shadow(B),* whose utilization rises with f_A; another is that a resource in *Beside(B)* may saturate and move the bottleneck there.

5.1 Recommendation

Look for a term with a large contribution in Eq (1a), and reduce it. For parallelization, the effect on reducing X_B depends on the relative delays of the parallel paths and the overhead introduced to launch them.

Examples: Batching of requests can be effective if the combined requests contribute less in Eq (1a); gains are made when the overhead of the combined operation is lower (communications and scheduling times). Smith and Williams describe principles that can be applied to reduce the workload of computer programs [16], with examples.

6 Mitigation: Reduce requests for B

A potent way to increase the saturation throughput is to reduce the value of $y(A,B)$, so the bottleneck is simply used less. This employs changes to the request parameters $y(t,t')$ in the *Above(B)* set. Eq. (10) makes $\Delta y(A,B) = a + b\Delta y$, and by Eq (5), the new system throughput bound can be predicted as roughly

$$f^*_A = f_A / [1 - \Delta y(A,B) / y(A,B)] \quad (11)$$

(for a decrease in requests, $\Delta y(A,B)$ is positive).

Improvement is limited by starvation, as for reduced holding time, but has more potential because other resources also have fewer requests. Resources in *Below(B)* see constant load at the increased throughput, and resources in *Above(B)* and *Independent(B)* may or may not see increased load, depending on the point where requests are reduced. To determine the limit:

- Apply Eq (10) find $\Delta y(A,t)$ for each t
- Apply Eq (11) with t in place of B, to give a virtual system throughput $f^{**}_{A,t}$ for a new bottleneck at t.
- as Δy increases, test to discover if some resource t causes a lower virtual throughput than B. If so, this t is the "next bottleneck" and $f^{**}_{A,t}$ is the resulting throughput.

6.1 Recommendation

Decreasing the requests to the bottleneck is recommended when it also reduces requests to other tasks above B. That is, the higher the resource where the change is made, the better.

Examples: batching of requests to B, or to a resource above B, is effective here also.

7 Asynchronous Resource Use

It is possible to reduce the bottleneck holding time by modifying not the entire holding time of its *RequestsTo* set, but just the part of the holding time that requires the simultaneous resource B. If part of the requested operation can be performed without B, it may increase performance. To describe this, the model of layered service must be extended to include:

Partly asynchronous service: a resource holding time is divided into two parts, which we will call phase 1 and phase 2. When resource T requests t, Phase 1 at t blocks the requesting resource T for only the phase 1 holding time X_{t1} . Phase 2 at t is executed either immediately after, or some time later, and is not included in any other resource holding time.

Each phase p of the holding time of resource T has a complete set of request parameters: a local service time Z_{Tp}, request rates $y(T,p;\, t)$ to other resources, and a holding time X_{Tp}. Then Eq (1) is modified to an equation for each phase at T, and nested holding of any other resource t only includes phase 1:

$$X_{Tp} = Z_{Tp} + \Sigma_{t \text{ in } RequestedBy(T)} [y(T,p;t)(Wt + X_{t1})] \quad (1b)$$

The resource utilization includes both phases:

$$U_T = f_T (X_{T1} + X_{T2})$$

The effect of asynchronous service at some T below B is to reduce the holding time of B, by propagating less delay upwards. If an amount Δx_T can be shifted to second phase, the reduction in holding time of B is $y(B,T)\Delta x_T$, and as long as the bottleneck remains at B the new system bottleneck bound is

$$f^*_A = [X_B/(X_B - y(B,T)\Delta x_T] f_A$$

However it is less effective than simple reduction of holding time, since the total holding time of T is still effective and T may saturate.

7.1 Recommendation

Apply asynchronous service wherever functionally permissible, as it is a no-lose option.

Examples: delayed writes in file systems and databases, operations which execute autonomously once initiated.

8 Load-Dependent Demands

Some resources have the additional feature that their demands depend on the intensity of the applied load.

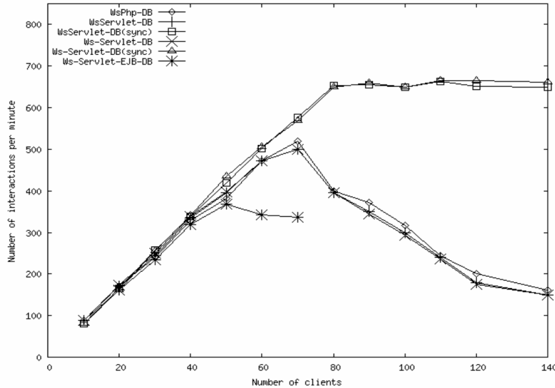


Figure 6. Throughput versus clients (f_A vs m_A) for a locking bottleneck (from [1])

For example, [1] describes an application with a lock management bottleneck in a MySQL database which causes throughput not just to saturate, but to drop sharply beyond a certain point. This is characteristic of cases where increasing congestion creates additional management overhead. Examples include optimistic locking (where high contention causes a high rate of transaction restarts), in thrashing in database buffers used as caches, and in virtual memory thrashing.

We shall assume that all the available customers of a bottleneck are in contention for it, which is approximately true; this number is $AvConcur(B)$ given by Eq. (6). Then the dependence makes Z_T, $y(T,t)$ and hence X_T to be functions of $AvConcur(B)$. Figure 6 compares implementations of a system that includes a database, in which throughputs show the effect of load-dependent bottleneck. In the cases with simple saturation (the highest and lowest curves, which level off) the authors [1] detected processor saturation; in the cases with declining throughput no processor was saturated and they described a "locking bottleneck". It shows the clear signs of load-dependent saturation with $AvConcur(B) = m_A$.

In general, if there is a load-dependent resource T then Eq. (3) for the limiting system throughput can be written as:

$$f_A = m_B / [y(A,B)\, X_B(AvConcur(B))]$$

Supposing that $AvConcur(B) = m_A$ throughout the range in Figure 6, then in the rising curve on the left of Figure 6 the bottleneck is the load source, whereas in the falling curve it is a load-dependent lock-related resource, with a holding time that rises with m_A.

9 Case Study

This section considers a distributed telephone switch, based loosely on an industrial project. Historically, the architecture of voice switches has been dominated by the need for increased capacity and performance. For instance, a description of Lucent's 5ESS architecture [15] emphasizes continual performance improvement, the tradeoffs between performance and other properties, call flows and delays, overload control, and software resource engineering. Standards govern acceptable delays to receive dial tone, and to obtain an indication of a connection.

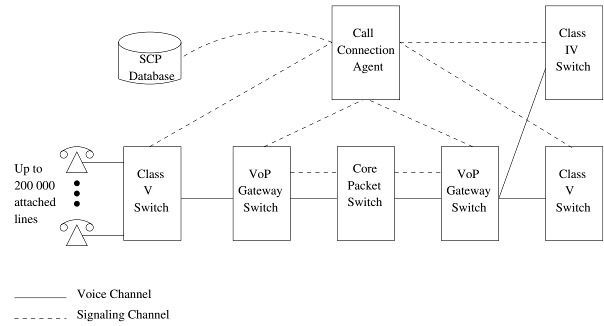


Figure 7. Abstract view of a telephone network

9.1 Architecture of a Class IV Voice-over-Packet (VoP) Switch

A Class IV switch connects Class V switches that actually have subscribers connected. This system distributes the functionality of a Class IV switch as shown in Figure 7, using a packet switch as a switching fabric. The Call Connection Agent includes:

- Line Information Database (DB)
- Call Processing Server (COCO): call routing decisions and coordination of connectivity.
- SS7 Interface (SS7): to the public network.
- H248 component: interface to the VoP network.

The gateway switch has a node controller (CTL) and connects to an intra-switch network using:

- H248 interface
- Call Control (CACO)
- A module SVC that terminates the connection requests and has an interface to the packet switch.
- line cards (interfaces) called NLC and ILC.

The packet switch has similar modules SVC, CTL, NLC and ILC.

9.2 Resources in the Distributed Switch

All of these modules are multithreaded concurrent processes which behave as resources in our model, and all have their own (single) processors. Figure 8 shows a resource graph for the system, with nodes shown as parallelograms. Each node represents a software

resource (many of them multithreaded) with a processor (not shown).

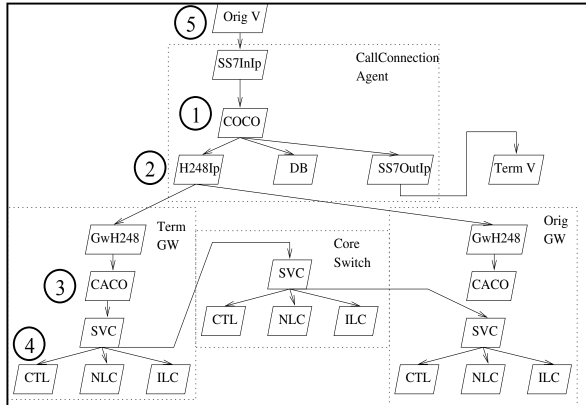


Figure 8. Resource Graph for the Switch. Each node shown has an associated processor node

There are four subsystems, with a source that represents an originating Class V switch, with m_{OrigV} = 1000 users and think time Z_A = 3.6 sec to represent a potential for 500,000 call requests per hour. An LQNS model based on Figure 8 was populated with parameters that roughly represent an actual prototype product with a loosely related resource structure, and solved. The initial model carried 280,000 calls/hour, and had a connection delay of 2.8 seconds with a bottleneck at the COCO node (Strength = 9.6, with *Shadow(COCO)* being node H248Ip).

9.3 Bottleneck Mitigation

Two steps of bottleneck mitigation were carried out. In the first, m_{COCO} was increased from 10 to 110. The new throughput was 463,000 calls/hour with a connection delay of 284 ms. The new bottleneck was CACO1 (the CACO node on the left) with Strength = 1.55. We notice that the shadow node did not become the bottleneck in this case, but it is in the saturated path from the top to CACO1. The new Shadow node is CTL1, the CTL node on the left (since SVC has a very high multiplicity).

In the second step m_{CACO1} was increased from 4 to 9. The final throughput was 490,320 calls/hour, with a connection delay of 68 ms. (this source can only reach 500,000/hour with zero response delay, so this is an excellent result). The final bottleneck is starvation at the source, which means that this configuration has capacity for additional traffic without saturating. The model-based recommendations were very similar to those developed by the project team. Table 3 summarizes some of the key results for the three configurations.

Table 3. Some Results for the Model in Figure 8.

Node	Saturation values, written as *U/m*		
	Case I	Case II	Case III
SS7In IP	150/150=1.0	150/150=1.0	19.3/150
COCO	10/10=1.0 (bottleneck)	73.6/110 =0.67	19.3/110 =0.175
H248Ip	2.1/20 (shadow)	19.97/20	5.5/20
COCO_Proc	0.086/1	0.141/1	0.150/1
CACO1	1.92/4	3.98/4=.995 (bottleneck)	5.2/9=0.61
CTL1	0.174/1	0.257/1 (shadow)	0.27/1
SVC1	1.84/20 =0.92	3.84/20 = 0.192	5.05/20 = 0.252

10 Conclusions

A framework for the systematic analysis and mitigation of layered bottlenecks has been described, including a taxonomy of cases and a detailed study of the effectiveness of the different possible changes. It is not surprising that many of the changes prescribed are related to Smith and Williams' principles for improving performance in general [16], but they take a specific form here. Space has precluded the inclusion of a great many examples, but the references help to supply this deficiency.

New definitions were given for the bottleneck strength measure first stated in [11], which deals with heavily provisioned resources in the bottleneck zone, and for a layered bottleneck. The latter depends on the value of a saturation threshold *sat** which must be chosen. If it is chosen too close to unity, no bottleneck will be found, and the analysis can be repeated with a smaller value. In a well balanced system there is no single bottleneck, and further improvement may require many simultaneous changes.

Not discussed here, but also relevant to removing bottlenecks in distributed computer systems, is re-allocation of processes to processors. An approach is described in [8] to optimize the allocations of layered resources.

Acknowledgements

This research was supported by Research Canada through its program of Discovery Grants. The authors are grateful to the referees for their suggestions, which improved the paper.

11 References

[1] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance Comparison of Middleware Architectures for Generating Dynamic Web Content," in *Proc. Middleware 2003*, LNCS 2672, Rio de Janeiro, June 2003, pp. 242-261.

[2] J. Dilley, R. Friedrich, T. Jin, and J. Rolia, "Web Server Performance measurement and modeling techniques," *Performance Evaluation*, vol. 33, pp. 5-26, 1998

[3] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside, "Performance Analysis of Distributed Server Systems", *Proc. Sixth Int. Conf. on Software Quality (6ICSQ)*, Ottawa, Canada, 1996, pp. 15-26.

[4] G. Franks, P. Maly, M. Woodside, D. C. Petriu, and A. Hubbard, *Layered Queueing Network Solver and Simulator User Manual,* Real-time and Distributed Systems Lab, Carleton University, Ottawa, 2005. Available at http://www.sce.carleton.ca/rads/lqn/lqn-documentation/

[5] M. Gerndt and A. Krumme, "A Rule-based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory Systems," *Proc. 1997 IEEE Workshop on High-Level Programming Models and Supportive Environments (HIPS '97)*, Genf, 1997, pp. 93-101.

[6] G. Hills, J.A. Rolia, G. Serazzi, "Performance Engineering of Distributed Software Process Architectures", *Proc Int. Conf. on Performance Tools (Tools 95)*, Heidelberg, LNCS 977, Springer, 1995, pp. 357-371.

[7] R. Jain, *The Art of Computer Systems Performance Analysis,* John Wiley & Sons Inc., 1991

[8] M. Litoiu, J.A. Rolia, "Object Allocation for Distributed Applications with Complex Workloads," *Proc. 11th Int. Conf. on Performance Tools (Tools 2000)*, LNCS 1786, Schaumberg, IL, Mar. 2000, pp. 25-39.

[9] P. Maly and C. M. Woodside, "Layered Modeling of Hardware and Software, with Application to a LAN Extension Router," *Proc 11th Int Conf on Performance Tools (Tools 2000)*, Chicago, Mar. 2000, pp. 10 - 24.

[10] D. Menasce, "Two-Level Iterative Queueing Model of Software Contention," *Proc. Modeling Analysis and Simulation of Computer and Telecom Systems (MASCOTS 2002)*, Fort Worth, 2002, pp. 267-280.

[11] J. E. Neilson, C. M. Woodside, D. C. Petriu, and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendez-vous Networks," *IEEE Trans. Software Engineering*, v. 21, n. 9 pp. 776-782, Sep. 1995

[12] D. Petriu and M. Woodside, "Analysing Software Requirements Specifications for Performance*," Proc. Int. Workshop on Software and Performance* (WOSP 02), Rome, July 2002, pp. 1 - 9.

[13] S. Ramesh and H. G. Perros, "A Multilayer Client-Server Queueing Network Model with Synchronous and Asynchronous Messages," *IEEE Trans on Software Engineering*, vol. 26, no. 11 pp. 1086-1100, 2000.

[14] J. A. Rolia and K. C. Sevcik, "The Method of Layers," *IEEE Trans. on Software Engineering*, v. 21, n. 8 pp. 689-700, August 1995.

[15] Richard Singer, "Overview of 5ESS R-2000 switch performance," in *Proc. Int. Workshop on Software and Performance*, WOSP'98, Santa Fe, New Mexico, USA, Oct. 1998, pp. 7 – 13

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

[17] P. Tregunno, Practical Analysis of Software Bottlenecks, MASc thesis, Carleton University, 2003

[18] J. Xu, M. Woodside, and D. Petriu, "Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time," *Proc. 13th Int. Conf. on Performance Tools (Tools '03)*, Urbana, USA, Sept. 2003.