Layered Resources, Layered Queues and Software Bottlenecks
A tutorial to the Performance Tools 2003 conference

Murray Woodside
Department of Systems and Computer Engineering
Carleton University, Ottawa, Canada
cmw@sce.carleton.ca
Sept 2, 2003
We will discuss

- resource relationships in software systems
  - how software uses resources
    - simultaneous resources, nested usages
  - layered architecture from nested resource contexts
- layered queues
  - service time expansion
- bottlenecks in layered resources
  - identifying the root cause
  - mitigation
- applicability of layered modeling
Performance and Resource Behaviour

- Performance derives from the interaction of
  - execution scenarios (sequence of steps, parallelism)
  - and system resources (contention delays)

- Scalability and performance improvement requires understanding the limiting factors
  - scenario structure
  - resource bottlenecks

- Complex resource behaviour may confuse the diagnosis and mislead the improvement
  - a lot of behaviour complexity is due to layered relationships

- “Software Bottlenecks” are in middle layers.
There are many kinds of resource limitation:

..... A thread Bottleneck in a Web Server

- a *single-threaded* web server can only handle one request at a time
- it has to retrieve pages from disk, giving a long service time
- the processor is idle most of this time
- the web server thread is a resource that is easily saturated
- net delay makes it worse

**ANY SERVER TASK CAN BE A BOTTLENECK**
Many kinds of resource (2):
...... A Bottleneck at a Critical Section

- several processes share a critical section called CS
- CS is a resource with a queue
- the CS operation includes some long-latency operations with storage or a network, so it has a long holding time
Many kinds of resource (3)

..... A Bottleneck at a Buffer Pool

- Many user processes using the pool
- Buffer holding time until processing is complete
- Buffer may be released before storage of the results
To model these many kinds of resources

- How can we describe how they are used?
- Answer:
  - as in other kinds of performance analysis, we can begin from scenarios
  - a scenario shows the sequence of operations
  - we can add the resources used by each operation
    - its resource context
  - we will also pay attention to where resources are obtained and released
- Scenarios are a common approach... e.g. the “execution graphs” of Smith are a scenario notation.
Scenarios and Resource Use

- ...we will represent an execution path and the operations carried out along it as follows...

- ...each operation requires one or more resources

- *In the simplest case*, only one resource is required for each operation ("independent" resources)

resource: R1  R2  R3  R2
Independent “One-at-a-time” Resources

- lead to the basic queueing or schedulability analyses

- e.g. for a queueing model, the previous scenario can be shown as a transition path for a customer class:

  ![Transition Path Diagram]

  - the transition path can support deterministic or stochastic scenarios
Resource Contexts can be shown by a box around some operations showing where the resource is obtained and released.

- a resource may be held over several operations:

Program Execution Resource (Task Thread)
one scenario is a particular traversal of the graph, following the arcs down and then retracing upwards.

nested contexts give **acyclic** graphs
Different traversals give different scenarios with the same nesting heirarchy

- and thus the same graph. For example:

- there can also be stochastic traversals, with a probability assigned to competing descending paths.
Nested Resource Contexts: using a File Server

- a scenario that takes input, locks a file, processes it, and generates a report:
- Nested contexts means an *acyclic* graph
  - down to acquire, up to release
- deadlock-free
Layered Resource Architecture (LRA)

Resources are entities
Resource-operations define behaviour while holding the resource

Layered Resources, Layered Queues and Software Bottlenecks
TOOLS 03 © C. M. Woodside 2003
Different uses of a resource are encoded in the resource-operations

- two file server operations 
  read and write
- like different classes of 
  customers at the file server
- Describing a resource 
  operation:
  - detailed description as a 
    sequence of operations 
    and resource requests
  - OR, aggregated 
    description as total 
    average requests
Nested Resource Contexts give a Layered Resource Architecture

- **nested** means resources are released in reverse order to acquisition
  - guarantee against resource deadlock
  - this is a “clean” resource structure

- traversal path follows an **acyclic** resource graph

- **layers** are measured by hops downward (longest path) from the load-generating sources (top level) of the graph

- **top-level resource** is usually a user or a program, and is assumed to cycle, generating requests (closed system) or to receive a stream of requests (open)
Some resource context patterns...

A separate context for each activity

Resource pass-back:
... an interesting pattern that needs work...... the Buffer is released by the Agent
Examples of Resources

- physical
  - processor, disk, bus, network link, interface controller

- logical/physical
  - semaphore protecting a device
  - allocation of a memory area or buffer

- purely logical for concurrency
  - process thread
  - mutex on a critical section

- logical for data
  - semaphore for access
  - lock
Layered Resources in Web Service operations

...the server thread is held until the last ack is received
Resource Architecture for the Web Server

- **resources**
- **resource-operations**

```
User
Browsers

servePage

Web Server
Threads

(host)

Processor

host-ops

Disk

fetchPage

Net
delay
```
Layered Queueing notation for the Web Server

requestPage → User Browsers

servePage → Web Server Threads

(fetchPage, Disk) → (delay, Net)

Processor

(host)
Resources are servers with queues

Flow of user requests

Users... $N$ users

Web Server $M$ servers

servePage

fetchPage

Disk

Processor

one server

common queue of requests

competing requests for disk operations

delay Net

delay server (infinite server)

Disk

one server
Layered Resource Service Time

... the black path is all part of the service time
What is the service time for a layered server...

... it is not knowable without a full analysis

- because it includes lower service times and also the contention at lower servers
- the contention delay is affected by competing scenarios and applications
  - e.g. the competition at the disk
- this is the key difficulty in understanding performance in layered systems
  - for example, bottleneck location may be unstable
Layered resources in an Access Controller for a building

- operations to check access rights and log events use the database

```
| readRights | writeLogEvent | Database |
```

Diagram:
- CardReader
- Access Controller
- LockActuator
- User [deny] [OK]
- alert [fail]
- triggerLock
- log
- unlockDoor
- check rights
The service time of AccessController

- includes lower servers
- here *AccessControl* includes lower layers:
  - LockActuator,
  - Database, and
  - Disk
- service time of AccessControl can be found recursively
- it includes waiting time at the database
Layered queueing in AccessController

Flow of user requests

competing database requests for other operations

(common queue of requests)

Access Controller [local processing]

(delay but no queue, as there is one actuator per door)

multiclass server (possibly a multiserver also = multithreaded task)

unlock LockActuator

delay server (infinite server)

other readRights writeLogEvent Database

read write Disk
Layered Queueing gives an Extended Queueing Network

- simultaneous resources
- derived service times
- the advantage of using LQ over direct EQN construction is that it is a *canonical form,*
- much easier to derive than a general EQN from scratch, and
- solvers can be pre-constructed for the canonical form itself
Terminology for layered queueing networks

- by convention, call
  - all servers and resources "tasks",
  - all resource-operations "entries",
  - all requests for operations by a server "calls".

- all requests to a task enter a common queue, which can have any discipline,
  - entries define classes of service
  - many kinds of tasks cannot support pre-emptive disciplines

- synchronous calls (that block the caller) are distinguished from asynchronous (that do not).
  - sync calls always lead to a single reply
  - more complex request types can be built up
LQ Model (2)

- a sub-scenario defines *entry behaviour*, by a sequence of operations and requests.
  - can use a default stochastic model for entry behaviour:
    - random “slices”, with a given coefficient of variation
    - either random requests with given mean numbers, or deterministic numbers of requests, in random order
- there is a “*host*” *processor* server for every task, not always shown
  - instead of the task providing its “own” service, it is given a processor server, *which may be shared*
  - host servers have the same semantics as tasks, all requests are synchronous, every software entry generates an entry on its host,
Simplified notation for a LQ model

- with default entry behaviour
- parallelograms are optional....

Entries with host demand in sec.

entry E1
[y-host-E1]

entry E2
[y-host-E2]

Client Task

Synchronous call with mean number
(y-serv)

entry for a service
“serv”

Server Task

host attachment for processor

{25} multiple resource or multithreaded task (25 threads)

P1

P2
LQN notation for the Door Access Control

- “alarm” has no calls or load
- some parameters show “second phase” operations: (0,0.2) or [3.9, 0.2]
  - after the reply
- some host processors are left implicit
LQN for the Web Server, with parameters

- allows for $n$ round-trip delays (of $2T$ seconds) for acks, in window flow control
- allows for $v$ disk accesses per web request
Some additional aspects of LQ

- interaction semantics
  - usual asynchronous message
  - synchronous call-return or send and wait interaction, for layered RPC service
  - so-called “asynchronous RPC” can be built up with parallel operation at client and server
  - “forwarding” of a request by a load distributor or a pipeline segment

- second phase: server replies early, before all operations are completed (e.g. delayed writes)
  - performance optimization

- sequence detail can be specified within in an entry

- priorities at tasks and processors
**Bottleneck in a web server**

N Users with a thinking time of 5 sec.

<table>
<thead>
<tr>
<th>N users</th>
<th>500</th>
<th>500</th>
<th>500</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>M threads</td>
<td>10</td>
<td>30</td>
<td>100</td>
<td>inf</td>
</tr>
<tr>
<td>X server</td>
<td>.512</td>
<td>.52</td>
<td>.52</td>
<td>.52</td>
</tr>
<tr>
<td>f thruput</td>
<td>19.5</td>
<td>58.2</td>
<td>90.6</td>
<td>90.6</td>
</tr>
<tr>
<td>W user wait</td>
<td>20.6</td>
<td>3.6</td>
<td>0.51</td>
<td>0.5</td>
</tr>
<tr>
<td>U server</td>
<td>10</td>
<td>30</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>U net</td>
<td>9.7</td>
<td>29.1</td>
<td>45.3</td>
<td>45.3</td>
</tr>
<tr>
<td>U CPU</td>
<td>.097</td>
<td>.29</td>
<td>.45</td>
<td>.45</td>
</tr>
</tbody>
</table>
Graphs for moderate traffic (500 users)

results for 30 threads

- User throughput
- Util. of Server and Net
- User Cycle Time
- Server Service Time
- Server CPU Util.
Pattern around the bottleneck

- users are always “busy” (waiting or “thinking”)
  - saturated in a sense
- server is saturated
- devices are unsaturated

...with sufficient server threads, the server is unsaturated and the devices too... this is the ideal
General Pattern for a “Software Bottleneck”

- a saturated server
- but…. a saturated server *pushes back* on its clients
  - the long waiting time becomes part of the client service time!!
  - result is often a cluster of saturated tasks above the bottleneck
- thus: the “real” bottleneck is the “lowest” saturated task
  - its servers (including its processor) are not saturated
  - some or all of its clients are saturated
Hourglass pattern is typical

**above:** there must be sufficient clients to build a queue

**below:** there must be several servers to divide the effort of providing the services to the bottleneck task
**Hourglass pattern shows saturation behaviour**

**above:** tasks above the bottleneck are saturated because of pushback delays

**below:** if the bottleneck task has no servers, its host utilization is the same as the task (it only computes)

- so it must have at least one additional server, a device (e.g. disk), task, or other logical server

- tasks below are unsaturated because the bottleneck throttles the load
Recognizing the “real” bottleneck

- a saturated task with unsaturated servers and host
- look at resource utilizations
- look for a step downwards in utilization, in descending the hierarchy:
  - sat
  - sat
  - \textit{sat: bottleneck}
  - unsat
  - unsat
**Strength measure**

- \( \text{measure} = \frac{U_B}{\text{max } U_S} \)
  - \( U_B \) = utilization of candidate bottleneck \( T_1 \)
  - \( \text{max } U_S \) = largest utilization of a server of the candidate bottleneck
  - = utilization of \( T_2 \)
  - \( U \) is **mean busy servers** number of servers

- This measures the **height** of the step in \( U \)
- and identifies the next bottleneck \( T_2 \)
“Next bottleneck”

- the lower task T2 with the max Us is the next bottleneck if the capacity of T1 can be increased
- the potential throughput increase is bounded in ratio by the strength measure
  - the resulting throughput will raise $U_{T2}$ to unity and saturate T2
- in practice the utilization of T2 may increase more rapidly with throughput, and saturate a bit sooner
- IEEE TSE paper 1995
Some tasks may be outside the pattern...

- some tasks above the bottleneck may be unsaturated, if they make very few requests, or bypass it
- modified hourglass pattern:
Bottleneck patterns and *threads* or multiplicity

- a resource may be a multiserver (M servers)
  - multiprocessor
  - multithreaded task
- in identifying the “hourglass” pattern, the utilization is per server (max value is 1)
- I will call all multiple resources “threads”
- a (very rough) rule of thumb for threads, based on potential needs for concurrency at a task T1:
  \[ M = \min \{ (1 + \text{sum of threads of servers of } T1), \text{(sum of threads of clients of } T1) \} \]
Mitigation of a bottleneck

(1) provide additional resources at the bottleneck
   - for a software server, provide *multiple threads*
     - some “asynchronous server” designs provide unlimited threads
   - *replicated* servers can split the load and distribute it, but give them each a processor
   - for a processor, a *multiprocessor* (or faster CPU)

(2) reduce its service time to make it *faster*:
   - reduced host demand
   - reduced requests to its servers
   - less blocking time (phase 1 time) at its servers

(3) divert load away from it
Mitigation

- increase the capacity of the bottleneck resource
- throughput increases
  - lower resources see more load and also more waiting
  - their utilization increases

*move down:* usually the “next bottleneck” with max Us is the one that saturates next, but *not necessarily*
Bottleneck “support” at T2

- hardware bottleneck: at a processor
  - this limits the impact of multi-threading
- “processor supported” bottleneck: a task for which the next bottleneck T2 is the processor
  - this limits the impact of multi-threading
- server supported bottleneck: a task for which the next bottleneck T2 is a server task
  - this improves the prospects of increased capacity, if it is also enhanced
- latency-supported bottleneck: some of the holding time is a latency server T2 with no queueing
  - only latency reduction will help.
### Bottleneck: Results for a web server with net delay

N Users with a thinking time of 5 sec.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M threads</td>
<td>10</td>
<td>30</td>
<td>100</td>
<td>inf</td>
<td>10</td>
<td>30</td>
<td>100</td>
<td>inf</td>
</tr>
<tr>
<td>X server</td>
<td>.512</td>
<td>.52</td>
<td>.52</td>
<td>.52</td>
<td>.512</td>
<td>.515</td>
<td>.55</td>
<td>4.99</td>
</tr>
<tr>
<td>f thruput</td>
<td>19.5</td>
<td>58.2</td>
<td>90.6</td>
<td>90.6</td>
<td>19.5</td>
<td>56.7</td>
<td>180</td>
<td>200</td>
</tr>
<tr>
<td>W user wait</td>
<td>20.6</td>
<td>3.6</td>
<td>0.51</td>
<td>0.5</td>
<td>97.6</td>
<td>29.4</td>
<td>6.1</td>
<td>5</td>
</tr>
<tr>
<td>U server</td>
<td>10</td>
<td>30</td>
<td>47</td>
<td>47</td>
<td>10</td>
<td>30</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>U net</td>
<td>9.7</td>
<td>29.1</td>
<td>45.3</td>
<td>45.3</td>
<td>9.7</td>
<td>29.1</td>
<td>90.2</td>
<td>100</td>
</tr>
<tr>
<td>U CPU</td>
<td>.097</td>
<td>.29</td>
<td>.45</td>
<td>.45</td>
<td>.097</td>
<td>.29</td>
<td>.90</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Graphs for moderate traffic (500 users)

results for 30 threads

User throughput
Util. of Server and Net
User Cycle Time
Server Service Time
Server CPU Util.

Number of Web Server Threads
500 users and 30 threads: where is the delay?

Throughput 58/sec

Users... Cycle 8.6 sec

Cycle 0.515 sec, U = 1.0

Server

Think, type

5.0 s wait

Wait for Net Delay 0.5 s

Net delay

inf

CPU

29%

Strength Ratio = 3.45

Throughput = 58/sec

DBP

DB

Disk

D

17.4%

23%

DB

5.8

4.35: CPU is T2
500 users and 1000 threads: location of delay

Throughput 90.6/sec

Users... Cycle 5.5 sec

Server

CPU

Disk

DB

Net delay

wait for

Delay

wait

5.0 s

0.5 s

think, type

45%

D

inf

28%

DBP

38%

Cycle 11 sec, U= 0.047

Bottleneck has vanished (moved up to the users, which starve the system)
Graphs for heavier traffic (2000 users)

results for 30 threads

- Web Server Util.
- User throughput
- Util. of Net
- User Cycle Time
- Server Service Time
- Server CPU Util.
2000 users, 30 threads, location of delay...
... similar to 500 users inside, worse outside

Throughput 58/sec

Cycle 0.515 sec, U= 1.0

Users... Cycle 34 sec

think, type
wait

5.0 s

wait for Net Delay 0.5 s

Strength Ratio = 3.45
scalable to 1/3.45 = 200 users
...threads relieve the bottleneck
... with enough threads, it goes to the processor

software bottleneck       hardware bottleneck at CPU

User Cycle Time
Server Service Time
Server CPU Util.
2000 users, 300 threads, location of delay...
...activates “second bottleneck” at processor

Throughput 200/sec

think, type

wait

5.0 s

wait

Net 0.5

Users... Cycle 10 sec

Server

Cycle 1.5 sec, U= 1.0

CPU

100 %

Disk

DB

Net delay

inf

D

DBP

64.4%

85%

...mitigation by faster processor pushes bottleneck *sideways* to DB
Layered Resources, Layered Queues and Software Bottlenecks

**Tripled database operation demand,**

**.... now bottleneck is server-supported...**

Throughput 55.5/sec

Users... Cycle 9 sec

Cycle 0.54 sec, U = 1.0

Server

- **CPU**: 28%
- **Disk**: 16.6%
- **DB**: 66.5%
- **Net delay**: inf

think, type 5.0 s  
wait 4 sec

wait Net Delay 0.5 s

Strength now only 1.5, systems is scalable only to 100/sec.

T2 is now DB
A canonical “Tower” pattern for deeply layered software bottlenecks

- Users just make requests.
- All execution times are 1.0.
- All requests are 1.0.

User throughput \( f = \frac{1}{6} \)

Location of delay for single threads:
- Idle
- Waiting
- Execution

(all single processors, and single servers at the bottom)
Mitigation in the Tower pattern

- vary the multiplicity at each level, $m_i$ at level $i$
- zero thinking time
- 1 sec host demand at each server, one request to each lower task
- $U_i =$ task utilization at level $i$

<table>
<thead>
<tr>
<th>Threads</th>
<th>Thru</th>
<th>Utilizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(m_2, m_3, m_4)$... $f$</td>
<td>$(U_2, U_3, U_4, U_5)$</td>
<td>$(1, 0.83, 0.67, 0.167)$</td>
</tr>
<tr>
<td>$(1, 1, 1)$...</td>
<td>0.166,</td>
<td>$(1, 0.83, 0.67, 0.167)$</td>
</tr>
<tr>
<td>$(2, 1, 1)$...</td>
<td>0.200,</td>
<td>$(0.96, 1, 0.8, 0.2)$</td>
</tr>
<tr>
<td>$(3, 2, 1)$...</td>
<td>0.223,</td>
<td>$(2.9, 1.64, 0.89, 0.22)$</td>
</tr>
<tr>
<td>$(6, 5, 4)$...</td>
<td>0.475,</td>
<td>$(5.5, 3.9, 2.75, 0.475)$</td>
</tr>
<tr>
<td>$(10, 10, 10)$..</td>
<td>0.65,</td>
<td>$(9.3, 7.8, 6.2, 0.65)$</td>
</tr>
<tr>
<td>(all infinite)</td>
<td>0.666</td>
<td>$(10, 9.33, 8.67, 0.666)$</td>
</tr>
</tbody>
</table>

Layered Resources, Layered Queues and Software Bottlenecks
TOOLS 03 © C. M. Woodside 2003
Mitigation... plot

User throughput $f$, task $u_{ti}$, $u_{ni}$

Level

1

10 Users

m2

m3

m4

(single servers at the bottom)

(1 sec demand at each server, one request to each lower task)

User Thruput

10, 10, 10 threads

6, 5, 4

1, 1, 1

Users

10
Bottleneck can move up or down

\[ m = (1, 1, 1) \quad m = (1, 1, 4) \quad m = (1, 3, 5) \]

... it moves up if increased throughput saturates a resource above the old BN, even without the waiting at BN
Multiple independent bottlenecks

- there may be a web of servers and interactions

- perhaps there are multiple bottlenecks?
  - in a flat queueing network there can be as many independent bottlenecks as there are chains of customers
  - each is an independent limitation on chain throughputs

- in a layered queueing network there are only a few independent throughputs... e.g. the top-layer tasks
A Strategy for Dealing with Bottlenecks (Peter Tregunno)

- Design managers want:
  - Solutions to problems, not solutions of models
  - Early estimation of potential gains
  - Problem mitigation alternatives considered

- Software designers want:
  - To avoid mathematical formalisms
  - Concise problem solution methods
  - Easy conversion from numbers to problem solutions
Steps...

- Obtain utilization estimates for each task and processor in the system
- Locate each saturated task - a judgment call is required here - what is saturation? (U > 75%)
- For each non-pure server task, compute the bottleneck strength - largest bottleneck strength indicates the software bottleneck
- Strength $B_b$ at task $b$
  - Weak bottleneck: $1.0 < \text{strength} < 2.0$ (judgment call)
  - Strong bottleneck: $\text{strength} > 2.0$
Software Bottleneck Mitigation Strategies

Increased resources
1. Increase threading levels at the bottleneck task
2. Replicate the bottleneck task and its processor

Shorter service time of the resource
3. Reduce of the bottleneck task’s host demands
4. Decrease the number of interactions that the bottleneck task has with lower layers
1. Increasing Threading Levels

- Useful with a strong software bottleneck
- Potential throughput gain at task \( b \leq f_b * B_b \)
- Optimal threading level is usually found through experiment (rules of thumb exist)
- Cost is usually minimal (low overhead), unless software design is explicitly singlethreaded
2. Replication of Task & Processor

- AKA add more hardware…
- Useful with a weak processor supported software bottleneck (threading helps strong bottlenecks)
- Reduction in utilization of the bottleneck task proportional to $p/n$ (where $p$ is the percentage of total service time that a task spends blocked due to processor contention, and $n$ is the number of processors added)
- Only effective when processor contention is high
3. Reducing Processing Demands

- AKA write faster code…
- Only applicable for processor supported software bottlenecks
- The utilization gain is proportional to the reduction in processing demands (i.e. small)
- For a strong server supported software bottleneck, the underlying problem is blocking, not slow software at the bottleneck.
4. Decreasing Interactions

- AKA batching requests
- Assume that synchronous requests can be bundled together - server still has to be the same amount of work, but \( n \) times less waiting (waiting for rendezvous acceptance) required at the client
- Effective when bottleneck is weak (long rendezvous delays are a product of high server utilizations, high server utilization = weak bottleneck)
Bottleneck Study:
A Voice-over-Packet Class IV Switch

- a class IV switch connects trunks; the VoP system replaces it
- a call connection agent and trunking gateway present the same interface

(E.g., ATM) part to be modeled
The Call Connection Agent (simplified)
The Trunking Gateway Switch (simplified)

Interfaces to the ATM signaling network

- Control Card
- SVC Card
- Trunk Interface Cards

Interface to the voice transport network

- H248 Card
- Network Interface Cards

Interface to the packet (ATM) network

Interface to the call connection agent
A Layered Queueing Model for call setup

- Trunk to trunk…
- Synchronous calls represent the control flow

Diagram showing layers of network components, including
- Orig V
- SS7Inlp
- COCO
- H248lp
- DB
- SS7Outlp
- Term V

GWs and core switch components are interconnected, illustrating the flow of signals and control.
Message Flow between Major Components
Task Utilizations

- **Bottleneck at Coco**
- **Bottleneck strength** = 1 / 0.113 = 8.85, *server supported*

<table>
<thead>
<tr>
<th>Task</th>
<th>utilization</th>
<th>Task</th>
<th>utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>ss7IpIn</td>
<td>1.000</td>
<td>gw1Ilc</td>
<td>0.0254</td>
</tr>
<tr>
<td>ss7OutIp</td>
<td>0.0508</td>
<td>coreControl</td>
<td>0.214</td>
</tr>
<tr>
<td>h248Ip</td>
<td>0.113</td>
<td>coreSvc</td>
<td>0.0114</td>
</tr>
<tr>
<td>coco</td>
<td>1.000</td>
<td>coreNlc</td>
<td>0.034</td>
</tr>
<tr>
<td>dbServer</td>
<td>0.0191</td>
<td>gw2Control</td>
<td>0.177</td>
</tr>
<tr>
<td>gw1Control</td>
<td>0.172</td>
<td>gw2Svc</td>
<td>0.0338</td>
</tr>
<tr>
<td>gw1Svc</td>
<td>0.0979</td>
<td>gw2H248</td>
<td>0.0113</td>
</tr>
<tr>
<td>gw1H248</td>
<td>0.213</td>
<td>gw2Caco</td>
<td>0.0261</td>
</tr>
<tr>
<td>gw1Caco</td>
<td>0.509</td>
<td>gw2Nlc</td>
<td>0.0562</td>
</tr>
<tr>
<td>gw1Nlc</td>
<td>0.0508</td>
<td>gw2Ilc</td>
<td>0.0290</td>
</tr>
</tbody>
</table>
**Processor utilizations**

.... Are all very low

*(processor support of bottleneck is not far behind the server)*

<table>
<thead>
<tr>
<th>Processor</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>ss7Ip</td>
<td>0.00914</td>
</tr>
<tr>
<td>H248Ip</td>
<td>0.0141</td>
</tr>
<tr>
<td>dbServer</td>
<td>0.0191</td>
</tr>
<tr>
<td>coco</td>
<td>0.0838</td>
</tr>
<tr>
<td>gw1Control</td>
<td>0.00914</td>
</tr>
<tr>
<td>gw1Svc</td>
<td>0.107</td>
</tr>
<tr>
<td>gw1H248</td>
<td>0.0831</td>
</tr>
<tr>
<td>gw1Nlc</td>
<td>0.152</td>
</tr>
<tr>
<td>gw1Ilc</td>
<td>0.152</td>
</tr>
<tr>
<td>coreControl</td>
<td>0.00914</td>
</tr>
<tr>
<td>coreSvc</td>
<td>0.167</td>
</tr>
<tr>
<td>coreNlc</td>
<td>0.305</td>
</tr>
<tr>
<td>gw2Control</td>
<td>0.00914</td>
</tr>
<tr>
<td>gw2Svc</td>
<td>0.168</td>
</tr>
<tr>
<td>gw2H248</td>
<td>0.0915</td>
</tr>
<tr>
<td>gw2Nlc</td>
<td>0.152</td>
</tr>
<tr>
<td>gw2Ilc</td>
<td>0.152</td>
</tr>
</tbody>
</table>
Bottleneck in Layered Queueing Model
Mitigation: first iteration

Multithread task COCO times 8.85 (from 30 to 272)
New Bottleneck at gw1Caco, still server supported
Strength = 0.953 / 0.185 = 5.15

<table>
<thead>
<tr>
<th>Task</th>
<th>Utilization</th>
<th>Task</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coco</td>
<td>0.986</td>
<td>Gw1H248</td>
<td>0.124</td>
</tr>
<tr>
<td>H248Ip</td>
<td>0.998</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gw1Caco</td>
<td>0.953</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gw1Svc</td>
<td>0.185</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gw1Control</td>
<td>0.265</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(utilizations below the old bottleneck have gone up, but not enough to create a bottleneck at processor Coco)
Bottleneck after first iteration
Utilizations after second iteration

Add threads at gw1Caco (from 4 to 9)
Bottleneck is still at gw1Caco but is quite weak…
could add more threads yet?

<table>
<thead>
<tr>
<th>Task</th>
<th>Utilization</th>
<th>Processor</th>
<th>Utilization Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coco</td>
<td>0.182</td>
<td>Coco</td>
<td>0.150</td>
</tr>
<tr>
<td>H248Ip</td>
<td>0.310</td>
<td>Gw1H248</td>
<td>0.148</td>
</tr>
<tr>
<td>gw1Caco</td>
<td>0.635</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gw1Svc</td>
<td>0.278</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gw1Control</td>
<td>0.340</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

More capacity increase should be available
Final Question: Applicability of Layered Modeling

- **modeling power**: what is the range of systems that have layered models
  - most systems are *effectively layered*

  - either: resource contexts are *fully nested*,
  - or: the non-nested contexts are *second-phase nested*

- **practical techniques** exist for modeling software from specifications
  - from special scenario languages like Use Case Maps
  - from UML
Effectively layered systems
(from the resource architecture viewpoint)

With fully nested resource contexts:

- simple client-server systems with RPCs and no second phase, such as most CORBA systems or web services systems
- systems with asynchronous messaging in which requesting processes wait for replies, have effectively synchronous interactions and blocking
- asynchronous systems (nesting of depth 1 only)
- forwarding gives asynchronous lower layers, so OK
- logical resources are modeled by layered *pseudo-tasks*
Effectively layered systems (2)

- second-phase nested resource contexts
- example: R3 is NOT fully nested in R1... it has a second phase part shown as a heavier line
- in the second phase, outer resources of the first phase (e.g. R1) are not used
Second-phase nested contexts and the LQN

(second-phase calls are bold)
Reminder: Second Phase Service in software servers

- Idea: enhance performance
  - give a reply as early as possible
  - Do postponeable work after the reply, as “phase 2”

```
client → server
```

- e.g.: Database server update operation:
  - write to log file before returning, execute final writes later.

- Second-phase model may
  - place this work right after the return (approx), or
  - send a message to a clean-up process that does it later

- Queueing approximation paper in Performance 99
Modeling technique for a logical resource

- A buffer or critical section for a set of threads….
Resource pseudo-task when user tasks are distributed...

- Tasks A and B must use a resource (call it RES) for some work....
  - but they do different things in different places
- So:
  - Separate out the computation within RES into *Shadow Tasks* $A|RES$ and $B|RES$
  - to direct the call from A to $A|RES$, make RES a *pseudo-task* with two *pseudo-entries*
Model-building from software descriptions

- based on tracing scenarios, detecting resources and interpreting nesting and interaction types
  - from scenarios in *Use Case Maps*: TOOLS 2002 paper
  - from tracing ("angio traces"): MASCOTS 95 and TSE 2000 (Hrischuk)
    - ("TLC" = trace-based load characterization)
  - now analyzing *UML scenarios*, expressed with the UML Profile on Schedulability, Performance and Time (2002)
Proposed PUMA toolset architecture...... (Performance by Unified Model Analysis)

- **general** software model input via CSM (not only UML)
- **general** performance model types via CPM (not only layered queues)
- includes heavy element of model investigation, sensitivity tools, optimization
- proposal also for component libraries for completions
Example: layered modeling of a Building Security System BSS

.....for a hotel or university campus

- two functions:
  - video surveillance, by capturing webcam images from many sites, storing and displaying
  - door access control (already examined)

- video surveillance: poll all N cameras within 1 second (on 95% of polling cycles)

- door access: respond to an access card within 1 second, 95% of the time
Specification of the Building Security System (BSS): Use Cases

- User
- Access control
- Log entry/exit
- Video Camera
- Acquire/store video
- Manager
- Manage access rights
- Database

(not evaluated in this study)
Specification of BSS (2) Deployment gives physical resources and process allocation

SecurityCard Reader
DoorLock Actuator
Video Camera
Disk

ApplicCPU
Access Controller
Video Controller

DB CPU
Database

<<LAN>>
Spec. of BSS (3):

Annotated Sequence Diagram for the Video Scan scenario
(using the UML Performance Profile)
UML annotations

- resource acquisition and release for logical resources: a stereotype on a message
- resource multiplicity: a tagged value of the resource stereotype
- scenario steps are stereotypes on messages or on a focus of control
- host processing demand: a tagged value of a step stereotype
- workload intensity: tagged values for the “closedLoad” and “openLoad” stereotypes on the initial step of a scenario
Spec of BSS (4) Sequence Diagram for the Door Access Control scenario

<<PAcontext>>

CardReader | DoorLock | Alarm | Access Controller | Database | Disk

User

<<PAstep>> (PAextOp=(read, 1))
readCard

<<PAstep>>
(PAdemand=('asmd', 'mean', (1.2, 'ms'))), PAextOp = (network, 1))
admit (cardInfo)

enterBuilding

<<PAstep>>
(PAdemand=('asmd', 'mean', (1.2, 'ms')))
getRights()

<<PAstep>>
(PAdemand=('asmd', 'mean', (1.8, 'ms')))
readRights()

<<PAstep>>
(not_in_cache) readData()

<<PAopenLoad>>
(PAoccurrencePattern = ('poisson', 120, 's'), PArespTime =(({req'} per centile',95, (1, 's'))), ('pred', 'percentile', 95, $RT))

<<PAcontext>>

<<PAstep>>
(PAdemand=('asmd', 'mean', (1.5, 'ms')), PAprob = 0.4)
writeRec()
Modeling a logical resource: the Buffer pool for Video frames

- its service time includes parts of execution of various tasks
  - not identified with any particular process
- it will be modelled by a “task” which we can call a pseudo-task
  - runs on a pseudo-host
  - no execution time of its own
  - makes calls that define the operation executed with the resource
- sometimes the place of the pseudo-task can be taken by an actual “resource manager” task, if it exists
Path showing the buffer resource-operation and holding time
LQN fragment for the video buffer

- AcquireFrame has a fragment Acquire2 within the buffer resource context
- BufferManager also
- call to StoreImage is second phase
  - buffer is held, manager is not
Overall LQN $N$ cameras, $R$ buffers
Layered Resources, Layered Queues and Software Bottlenecks

BSS LQN...
follow the scenarios

Tools 03 © C. M. Woodside 2003
Typical results...

- one buffer: one camera at a time.
  - low rate of camera polling, low utilization of resources
- two buffers: “double buffering”, fill one while storing the other
- additional “acquire” threads: can start more buffers at once and each one addresses a camera;
  - can overlap packets from cameras on the network, if bandwidth is sufficient
- additional “store” threads can manage storage of multiple frames simultaneously,
  - with overlapped DB access
Applicability of layered modeling: preserving operation sequence

- within an entry the basic LQN just adds up and averages the demands
  - host processing
  - calls to services
- however it is equally possible to define the sequence by a scenario fragment
  - our tool calls this an “activity graph” for the entry
  - branching and looping
- AND forks too... define forking additional threads
Applicability: systems with parallel paths

- AND-forking within an entry describes fork-join parallelism within a scenario
- e.g. this can be used to describe asynchronous or “delayed” RPCs
  - one fork executes the main stream of the calling process,
  - the fork models a virtual or “agent” thread which makes a synchronous RPC and waits for the reply
  - eventually the task need the response and the two paths join
- also describes parallel access to servers, one path for each access
Significance of layered modeling

- is not in the technique itself, which is a version of extended queueing

- is in insight obtained about resource relationships, as in the bottleneck analysis described here.
Making it practical: tools

model building:
- PUMA project is one of several that start from UML
  - also can reduce other scenario languages to the “Core scenario model” CSM
  - converts to a LQN or QN or simulation

solvers: analytic and simulation
sensitivity: multiple run manager
optimization of allocations and priorities
- more information: www.layeredqueues.org
  - or email to cmw@sce.carleton.ca