A Performance Tuning Framework

Roland Kaschek
UBS AG,
Zürich
roland.kaschek@ubs.com

Abstract: Design and implementation of applications comprise an anticipation of what increased performance requirements might occur in future. However this is not a simple thing to do and resources are limited. Thus in practice it happens, that after some time of operation an application no more meets the increased performance requirements. Sometimes it turns out that the application has to be retired and a new one needs to be installed and used. In other cases performance tuning activities can make the application again meet the requirements. This paper based on a simple distributed applications performance model gives hints on how to proceed during performance tuning. It further states several performance tuning requirements on system architecture and design respectively.

1. Introduction

It is well known that activities going on in the area of distributed systems strongly are influenced by e-business and web based systems. According to [Ma98] there are three fundamental internet commerce infrastructure elements which "have major impact on user perception of … business -to-business or business-to-consumer site: scalability, availability, and performance." Scalability here is understood as an applications' capability of being adapted to increased performance requirements while being operative. Typical areas of increased requirements are: throughput, capacity, response time or resource consumption. Practical activities using an applications scalability to meet increased requirements collectively are denoted as performance tuning, see e.g. [SS00]. It is reasonable to design and implement new applications for performance. It is however not sufficient to only care for applications to be constructed and forget about operating ones. Lots of operating applications are suffered from increased performance requirements. Not all of them can or should be ruled out due to actual performance problems. The present paper proposes a performance tuning framework for distributed applications as well as a couple of tuning hints. It attempts to overcome performance troubles concerning operating applications.

Software performance engineering aims at construction of applications meeting their performance requirements. For material on this topic see, e.g. [P*00], especially

[DK00, KW00] within the report mentioned first. It is not covered in this paper. However since not all of them might be trivial, obvious or useless some performance tuning requirements to system architecture and design are included in the paper.

Concerning the economy of internet applications [Ma98] states that even of the satisfied customers 50% switch to a competitor. This means that in fact a customers market has established and companies must care for customers whishes. "Convenience -rather than getting the best price- is the overriding factor in mailing purchases over the net." One of the essentials in internet applications' economy is "…the customer self service capability, which reduces the need for human intervention in the customer service infrastructure even as the customer base grows. Self service lets … serve more customers without additional significant investment in customer support research." Together this implies that performance tuning not only is important due to its ability to protect investments. Additionally it directly may impact an enterprises market position.

Performance within this papers addresses performance of distributed applications and not to performance of one node, i.e. host or PC applications. Increased performance requirements are caused by either increased number of service requests (which are to be served for in an unchanged period of time) or increased resource consumption of individual (or average) service requests. A simple idea to prevent applications from being ruled out by increased performance requirements is to employ redundancy, i.e. redundancy of processing, storage, or transportation capacity respectively. The respective redundant equipment is invoked in case more processing capacity is needed. Software performance engineering (and in worst case ordinary system construction) has to cause this invocation to be done easily. To employ it in practice however is the job of performance tuning.

In what follows, within section 2 the tuning framework is introduced. Then in section 3 performance tuning hints are listed. Section 4 states performance tuning requirements on application architecture. Section 5 states performance tuning requirements on application design. Section 6 contains acknowledgements and section 7 the references.

2. The tuning framework

Application performance tuning affords to have a distributed applications' model allowing to propose performance tuning operations as well as assess the respective probable consequences. The model employed within this paper is a systems model, i.e. it is assumed that an application is a system. Thus applications are assumed to consist of communicating components. These are associated to nodes, i.e. processors. Communication taking place among components associated to two nodes belonging to a given application is significantly more time consuming than communication taking place among components associated to only one of the nodes. At any given node three different actions may take place: Data may be processed, stored or retrieved. It is assumed that storage and retrieval of data are about equal time and resource consuming. The time needed for data storage is assumed to be much bigger than the one for data

processing. Finally communication among two applications may occur. The amount of his inter system communication time however is significantly larger than the amount of time consumed by intra system communication, i.e. communication among components associated to two nodes belonging to the same application, see e.g. [Ra00]. Data processing capacity, data storage and retrieval capacity, intra system communication capacity as well as inter system communication capacity collectively are denoted system resources.

Referring to this model application tuning can be discussed in terms of the following task areas: *data processing* (*application logic*), *data storage* or *retrieval* (*data access*), *intra system communication* (*infrastructure*) and *inter system communication* (*input / output*). System operation further is discussed in terms of resource consumption. According to the model given above resource consumption takes place in exactly the mentioned task areas. The amount of resources is assumed to be constant for each given system (but may be enlarged). Service requests occupy resources. Some resources may be occupied by several service requests simultaneously. In case the respective resources are available service requests will be processed. Finishing them in general will free the occupied resources. Data storage capacity however can only be freed by deleting data. If required resources are not available the respective service request has to wait until the resources become available. Resource consumption in terms of a coarse grained model can be said to occur frequently or to last for long.

If resource utilization frequency or average individual resource usage time increases to a relatively high score then the respective task areas capacity in tendency might become exhausted. Thus good performance tuning capabilities imply that the respective capacity can be enlarged easily or the respective load can be reduced easily. Performance of an application is determined by its worst performing sequential component. Up to some degree sequential order is given for each application with respect to the task areas mentioned. At first some input / output takes place then intra system communication takes place invoke the required resources. Then the data has to be processed. Finally it has be put out. Though this sequencing of actions takes place in any application some of the respective actions may take place in parallel.

Average individual resource usage time	**Utilization frequency**							
	Application logic		**Infrastructure**		**Input / Output**		**Data access**	
	high	*low*	*high*	*low*	*high*	*low*	*high*	*low*
high	**D**	C	**D**	C	**D**	C	**D**	C
low	C	E	C	E	C	E	C	E

Table 1 : Classyfication schema for applications

Based on **Table 1** one can rank applications according to the following scale:

Dangerous, at least one resource shows high resource usage frequency as well as high average individual resource usage time.

Critical, at least one resource shows high resource usage frequency but low average individual resource usage time or vice versa.

Easy, all resources show low usage frequency as well as low average individual usage time.

Clearly, to yield a meaningful classification of an application affords to monitor it in terms of usage frequency and average individual usage time and assess the found results. This equals pretty much the situation found in database tuning: Without statistics data available to the tuner only very rarely effective actions can be proposed. For a text on this topic see, e.g. [Sh92].

The proposed method for performance tuning consists in attempting to drive dangerous components into critical ones. Further critical components are to be turned into easy ones. A further prioritization of possible actions can be derived from the amount of time necessary for individual actions at the task areas, i.e. first work on task areas which require longer periods of time. During performance tuning activities one must keep in mind that improvements at one component may cause another component deteriorating. At least some of the operating applications which are targeted by performance tuning activities are of significant complexity. Further the respective documentation might be worse or inaccessible. Since time in practice often is short a complete or at least good understanding of the whole application as well as its environment cannot always be reached. Thus up to a certain degree performance tuning is and will stay an art which heavily draws on intelligent and systematically organized guesses.

3. Performance tuning hints

The hints that will follow by no means are intended to be complete or at least the most important ones. They just are a collection that shall help in performance tuning activities. More important than the individual hints is the schema to group them. It can be used in generating hypothesis concerning practical performance improvement actions to take.

Before starting to modify an application one should do a lot of performance monitoring, measuring and modeling to assure the important issues are really understood. Investigate the system behavior based on simulation studies and prototypes. To assess performance tuning capabilities of applications one can conduct simulation studies as was indicated by [DR92].

Referring to Table 1 the hints are grouped according to what observation they are intended to impact: high average individual resource usage time or high resource usage frequency respectively.

A. High average individual resource usage time

I. *Data processing*, i.e. application logic
consider introduction of redundant data to simplify and by that shorten the processing necessary to fulfill required functionality. Note however that the introduction of redundant data causes a maintenance load as well as a storage

load. The respective hints in [B*92] can help to judge on the pros and cons of the respective redundancy.

consider re-implementation of central algorithms with a more efficient language or using the same language but utilizing more efficient algorithms. A CORBA environment, e.g. gives you something like a separation of a physical level from a logical one. Replacement of a software unit within such environment will not affect service calls as long as the service signatures remain unchanged since the respective calls are coded at the logical level while the actual invocation takes place at the physical level.

consider to use compiler based technology instead of interpreter based technology, i.e. use C++ instead of interpreted Java or compiled Java instead of interpreted Java. For a recent paper on Java performance issues see [SD01].

consider to use servers that can be multi-instantiated or multi threaded servers or server pools in order to have improved server performance.

consider to introduce more concurrent processing with respect to. application logic by putting some of it onto other machines. Clearly this needs an application level concurrency analysis as well as an architecture that is aware of this improvement attempt.

consider to refine the services. It might be reasonable to have finer grained services which then take less time in execution, can be re-used more often and thus far better performance optimization can be done on them.

II. *Intra system communication*, i.e. infrastructure

use as less as possible connections between software units. Especially inheritance can cause considerable load on the infrastructure.

use as much as possible connection caching such that connections between software units can be re-used.

consider that initiated communication attempts might last for long. Three major causes for this can be pointed out:

- *inaccessible data*. Data may be locked or the database may be overloaded due to a large number of requests. Also data storage capacity may be exhausted.
- *insufficient amount of communication capacity*. The infrastructure might be overloaded. Other applications might cause infrastructure load.
- *insufficient amount of data processing capacity*. Servers may be overloaded, service processing might be waiting for another task to complete.

In all cases a cure could be to start communication attempts not earlier than all inputs necessary for it are actually available. Of course the waiting time has to be used for some reasonable purpose. Assure that time out mechanisms which are offered by utilized infrastructure actually are used. Another often more efficient cure is to use asynchronous communication mode to prepare for the input of communication attempts.

consider the infrastructure as possibly being overloaded. If this is the case think about temporal partition, i.e. can some of your tasks also run at another time?

Do you cause load by joining big remote tables? To reduce infrastructure load one can use stored procedure calls instead of query shipping to reduce the amount of data transferred. This also has the advantage that decoupling is improved. Further avoid gateway passages since the respective format translation overhead might be significant. Of course also a re-configuration of infrastructure might be an issue as well as introducing more efficient or larger hardware. Note however that in bad designs hardware can't surpass the bottleneck.

consider distribution of data and functionality. Is it possible to re-arrange it such that less load is imposed on the infrastructure?

consider horizontal as well as vertical partition of data, i.e. consider to pose questions like:

- Do I really need to submit all this instances of the same data type? Can I put them into several chunks to be submitted more or less independently from each other?
- Do I need to submit all the data describing this particular data item type at the same time? It often is the case that to continue processing in specific situations not all data describing an object is necessary. In such case based on the respective probabilities one can decide to submit only partial data and to recharge the omitted data on demand.

III. *Inter system communication*, i.e. input / output

Consider external systems such as other software systems to be co-operated with or users which can input data or commit having received date or the like. In case synchronous communication modus is used this can cause termination of input or output operations to be deferred. More or less things apply that were mentioned concerning infrastructure. Note however that human error probability and indetermination of action is significantly higher than those of machines.

IV. *Data storage*

Two reasonable aspects have to be considered. The first is that the data to be retrieved is accessible but that the respective accesses take long. Then besides tuning of database and queries which might include modification of indices or clusters, introduction of redundant data and re-writing of queries as well as granularity concerns as discussed earlier have to be taken into account. Further stored procedure calls should be issued instead of query shipping since parsing and optimization efforts take a significant amount of time. The second aspect is that the data is not accessible since it is locked. If the locking application is different from the application under concern then again temporal distribution may become an issue as well as in rare cases a re-design of that application. If the respective locks however are held by the application under consideration then its locking strategy should be (re-) investigated. Clearly to improve performance it is reasonable to request for locks late and to release them early. Granularity of locking has to be considered carefully.

Surprisingly not only too coarse grained locking but also too fine grained locking may cause data to be unavailable. This is due to the fact that in the first case a transaction locks data it actually at a certain point in time does not need

and thus prevents other transactions to access it. In the second case the locks itself as a resource can become rare and thus since no more locks can be allocated to transactions these cannot make exclusive use of this data.

In case multiple transactions run on the same database also the dependencies among them have to be considered. If, e.g. transactions are prioritized then priority inversion may occur. It leads to the fact that transaction with higher priority are blocked from locking data by transactions of lower priority because these are blocked to release the locks since they cannot terminate due to being blocked by another transaction of higher priority.

B. High resource utilization frequency

I. Application logic

consider the introduction of caches which might hold intermediate results which could be used in processing service requests

consider the granularity of services with respect to 3 questions:

- are all the computations performed really necessary to meet the required functionality?
- is the required functionality consistent with the actual needs?
- should certain computations be coalesced? I.e. is the granularity of service design too fine grained?

consider the introduction of temporal separation such that non urgent service requests could be postponed. Note that sometimes this interrelates with how accurate a result must be. If, i.e. only the magnitude of the result is of interest then actual values don't need to be computed every time the value is needed.

II. Infrastructure

consider to coalesce data items that are submitted via infrastructure. Note that each transfer causes a certain basic load to the infrastructure which is independent to the amount of data transferred.

consider granularity of objects and interfaces. Often it is more efficient to have a coarse grained design which so to say automatically gives results in the appropriate chunk size.

consider to modify the scenario implemented. It might be the case that you can reduce the number of service execution requests by just implementing other scenarios.

consider the separation of data from the functionality required to process it. Though up to some degree this could be necessary to improve de-coupling and thus scalability it might cause unnecessary data traffic. This can cause the traffic to exceed the infrastructures' capacity

III. Input / Output

As was the case with respect to infrastructure and to application logic aspects of granularity apply.

IV. Data access

consider the granularity of objects or services and decide whether this granularity is too fine grained.

In case that access rates are high even if the load caused by individual accesses is low this high rates might flaw performance. Each access implies a certain system load which more or less is independent from its specific cause.
consider to enlarge the set of cached data to reduce disk accesses. In general it is a good advice to have that much memory that all the data for static web pages can be held. This situation given processing power may be increased.

In case resources are used efficiently it might be a good idea to think about limitations of hardware such as in terms of processor speed, data caches at the various levels, main memory, hard disk size or communication capacity. Of course the system architecture must be such that one can make use of better performing hardware. Note however, that though hardware performance will continue to increase fast as well as hardware cost to reduce additional hardware might not be sufficient to solve performance and scalability problems. It often is the case that intelligent system design by far outperforms brute force performance gains of modern hardware.

4. Requirements on system architecture

An applications' architecture impacts its performance tuning possibilities. Maximum performance tuning capability can be characterized such that adaptation to increased service request number, service request time or resource consumption requirements can be done easily, fast and while the application is operating. This adaptation should consist in modifying configuration files, exchanging hardware or replacing overloaded components or subsystems by more efficient ones. New components thus must be both downward compatible with respect to imported and exported interfaces respectively.

Application performance only can be tuned effectively for certain regions within the problem space. Outside this regions they can't. Thus it is important to specify and implement an overload policy. In case an application cannot be managed to be situated within a problem space region of good performance tuning ability the overload policy causes the respective application to enter an overload mode. Thus minimum service quality may be guaranteed and dedicated action targeting the very application may be taken by operating staff.

Network Architectures. For a recent paper see, e.g. [N*00]. In a 4-tier architecture, e.g. one typically would have a net client, a net server, an application server and a database server and all this servers would run on dedicated machines. In a 3-tier architecture however one would have a client, an application server and a database server. In the first architecture the association of task areas to tiers is obvious. In the second architecture there is some latitude. Clearly infrastructure load strongly depends on this kind of layering. Recommendations concerning the association of architecture components to nodes can be found in [Cr99].

Work load profile, i.e. figure out what kinds of operation requests in what frequency with how high throughput are to be served by the application. For example, does the application make heavy use of data accesses or is it more processing oriented. In case data accesses are important, what about the distribution of updates and reads? Decisions on data replication need this input since efficiency

of replication protocols in a specific case depends on this type of information. For more detail on this subject see e.g. [Li98]. For applications with processing focus: Can the respective computations be carried out concurrently? Those processes being conducted in parallel should be mapped onto different nodes. This mapping should be based on an analysis of the universe of discourse. Such analysis can be carried out with help of Petri net models, Harel state chart models or even scenario techniques may be used.

Expected evolution of the work load profile all over the intended life of the application. Of course in general only rough estimations are possible and useful. Planned or expected modifications or substitutions of hardware or software have to be considered. What variety of performance requirements has to be considered?

Avoid 2PC for distributed transactions. For a recent introductory paper on distributed transactions see, e.g. [Er01]. The effects of traditional transaction management applied to multi tier architectures is investigated in [R*99]. Sometimes distributed transactions can be re-arranged such that they don't need 2PC. Use instead of 2PC optimistic locking if you expect only a small number of conflicts to occur. For time stamp based concurrency control see [Ul88], pp.524-535.

For internet applications *provide for a reserve capacity of 100%* since such application often have strong peak performance requirements.

Take care that the application conforms to architectural constraints and boundary conditions in power, such as:

- *architectural ideas*, e.g. fat clients, are prohibited, or 4-tier architecture are strongly recommended,
- *concepts*, e.g. distributed transactions, are illegal,
- *protocols*, e.g. replication protocols or communication protocols are to be used,
- *products*, e.g. MQSeries as a messaging software, CICS as a transaction monitor, or UDB, i.e. as database management system are prescribed to be used for the IBM platform.

Take care for comparable load imposed on the hardware and software units in your application. The sequential unit with worst performance might limit and determine the performance of the application. Provide for some easy to use means for load balancing.

Use extensible hardware, i.e. hardware such that additional memory, processors or hard disk may be plugged in or just be activated. Note that often extension of memory or addition of processors may have stronger impact than extension of hard disk storage.

5. Requirements on system design

To prepare the design of a suitable architecture consider the following hints:

All processing related to the areas of resource consumption mentioned in section 2 are treated by separate software modules, i.e. a 4-tier architecture with (thin) web client, web server, application server and database server is used . Even more layers could be desirable in case intermediate clients shall be employed to further increase the degree of concurrency.

Client server communication only is done in asynchronous mode and the coupling between client and server is weak due to the introduction of service request queues as well as result supply queues. This, e.g. results from the usage of message queueing software such as MQSeries.

Stateless servers and no sessions are used. The client state completely is managed by the client (with help of respective infrastructure). Thus backups can be done during operation and by the one of the redundant servers with least work load.

Concerning critical resources centralized management is avoided where it is reasonable to do so. The respective management is service driven, i.e. managed by the very service instance enacted in response to a specific service request. Note that centralized resource management easily can turn into a performance bottleneck. Central counters used to generate unique identification numbers are a well known example in this respect.

The application has a low level of resource consumption.

The application is composed from software which is available on the platforms that will be used throughout the applications' life time.

The application grounds on standard infrastructure, i.e. middle ware and operating system such that underlying hardware may be exchanged easily.

The most important performance related configuration parameters are factored out of the software and held in configuration files.

Avoid using technology you are not familiar with.

6. Acknowledgments

To produce an early draft version of this paper I was funded from UBS AG. While working on that draft I was a member of the software architecture team within the department of architectures and standards of UBS AG. Comments of Felix Stehli, Erwin Schwarz and Jirka Hoppe, at that time all with UBS AG, helped me improve the draft version. Initial input to this draft was given to me by Stefan Zumbrunn, also UBS AG. I'm however the only one responsible for the contents of this paper. The hints provided for by Stefan Zumbrunn were a result of his experience as a distributed systems expert at UBS AG.

7. References

[B*92] Batini, Carlo and Ceri, Stefano and Navathe, Shamkant B.; Conceptual Database Design; The Benjamin/Cummings Publishing Company, Inc.; Redwood City, Cal.; 1992.

[Cr99] Van Cruyningen, Ike; E-Business Everlasting; Intelligent Enterprise; Vol. 2, No. 15, Oct. 26, 1999.

[DK00] Dumke, Rainer and Koeppe, Reinhard; Conception of a Web Based SPE Development Infrastructure, (In German); In [P*00], pp. 1 – 16.

[DR92] Delis, Alexis and Roussopoulos, Nick; Performance and Scalability of Client Server Database Architectures; Proceedings of the 18th VLDB Conference; Vancouver, British Columbia; Canada; 1992.

[Er01] Transaction Processing With Java; Java Report; January 2001; pp. 30 – 35.

[KW00] Kraiss, Achim and Weikum, Gerhard; Target Based Performance Engineering Based on Message Oriented Middleware, (In German); In [P*00], pp. 63 – 71.

[Li98] Linnhoff – Popien, Claudia; CORBA: Kommunikation und Management; Springer Verlag; Berlin et al.; 1998.

[Ma98] Makmuri, Sukan; Best Practices for Internet Commerce; Intelligent Enterprise; Vol. 1, No. 3, Dec. 1998.

[N*00] Noack, Jörg and Mehmanesh, Hamarz and Mehmaneche, Homayoun and Zendler, Andreas; Architectures for Network Computing, (In German); Wirtschaftsinformatik, Heft 1, February 2000; pp. 5 – 14.

[P*00] Proceedings of the 1. Workshop: Performance Engineering in Software Development, (In German); Darmstadt, Germany; March 15th. 2000; http://www-wi.cs.uni-magdeburg.de/pe2000.

[Ra00] Ramaswamy, Ramkumar; Latency in Distributed Sequential Application Designs; SIGSOFT Software Engineering Notes; Vol. 25, No. 2; pp. 51 – 55.

[R*99] Ram, Prabhu et al; Distributed Transactions in Practice; SIGMOD Record; Vol. 28, No. 3, Sept. 1999;

[Sh92] Shasha, Dennis E.; Database Tuning: A Principled Approach; Prentice Hall; Englewood Cliffs, New Jersey; 1992.

[SD01] Schatzman, James and Donehower, James; High – Performance Java Software Development; Java Report; February 2001; pp. 24 – 42.

[SS00] Scholz, AndrÈ and Schmietendorf, Andreas; Aspects of Performance Engineering – Tasks and Content, (In German); In [P*00], pp. 33 – 52.

[UL88] Ullman, Jeffrey D.; Principles of Database And Knowledge-Base Systems; Computer Science Press; Rockville, Maryland; 1988.

[Wi99] Winter, Richard; Lexicology of Scale; Intelligent Enterprise; Feb. 1999.