Characteristics of Scalability and Their Impact on Performance

André B. Bondi
AT&T Labs
Network Design and Performance Analysis Department
200 Laurel Avenue, Room D5-3C01
Middletown, New Jersey 07748
abondi@att.com

ABSTRACT

Scalability is a desirable attribute of a network, system, or process. Poor scalability can result in poor system performance, necessitating the reengineering or duplication of systems. While scalability is valued, its characteristics and the characteristics that undermine it are usually only apparent from the context. Here, we attempt to define different aspects of scalability, such as structural scalability and load scalability. Structural scalability is the ability of a system to expand in a chosen dimension without major modifications to its architecture. Load scalability is the ability of a system to perform gracefully as the offered traffic increases. It is argued that systems with poor load scalability may exhibit it because they repeatedly engage in wasteful activity, because they are encumbered with poor scheduling algorithms, because they cannot fully take advantage of parallelism, or because they are algorithmically inefficient. We qualitatively illustrate these concepts with classical examples from the literature of operating systems and local area networks, as well as an example of our own. Some of these are accompanied by rudimentary delay analysis.

Keywords

Load scalability, performance, structural and space scalability.

1. INTRODUCTION

Scalability is a desirable attribute of a network, system, or process. The concept connotes the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement. When procuring or designing a system, we often require that it be scalable. The requirement may even be mentioned in a contract with a vendor.

When we say that a system is unscalable, we usually mean that the additional cost of coping with a given increase in traffic or size is excessive, or that it cannot cope at this increased level at all. Cost may be quantified in many ways, including but not limited to response time, processing overhead, space, memory, or even money. A system that does not scale well adds to labour costs or harms the quality of service. It can delay or deprive the user of revenue opportunities. Eventually, it must be replaced.

The scalability of a system subject to growing demand is crucial to its long-term success. At the same time, the concept of scalability and our understanding of the factors that improve or diminish it are vague and even subjective. Many systems designers and performance analysts have an intuitive feel for scalability, but the determining factors are not always clear. They may vary from one system to another.

In this paper, we attempt to define attributes that make a system scalable. This is a first step towards identifying those factors that typically impede scalability. Once that has been done, the factors may be recognised early in the design phase of a project. Then, bounds on the scalability of a proposed or existing system may be more easily understood.

An unidentified author recently placed a definition of scalability at the URL http://www.whatis.com/scalabil.htm. Two usages are cited: (1) the ability of a computer application or product (hardware of software) to function well as it (or its context) is changed in size or volume in order to meet a user need; and (2) the ability not only to function well in the rescaled situation, but to actually take full advantage of it, for example if it were moved from a smaller to a larger operating system or from a uniprocessor to a mulitprocessor environment. Jogalekar and Woodside define a metric to evaluate the scalability of a distributed system from one system to another, but do not attempt to classify attributes of or impediments to scalability [13]. Without explicitly attempting to define scalability, Hennessy cites an observation by Greg Papadopoulos of SUN that the amount of online storage on servers is currently expanding faster than Moore's law [11]. Hennessy also suggests that scalability is an aspect of research that should receive more emphasis in the future than performance. The work in this paper and in [13] suggests that performance and scalability need not be decoupled; indeed, they may be very closely intertwined.

The ability to scale up a system or system component may depend on the types of data structures and algorithms used to implement it or the mechanisms its components use to

WOSP 2000, Ontario, Canada
© ACM 2000 1-58113-195-X/00/09 ...$5.00

communicate with one another. The data structures support particular functions of a system. The algorithms may be used to search these structures, to schedule activities or access to resources, to coordinate interactions between processes, or to update multiple copies of data at different places. The data structures affect not only the amount of space required to perform a particular function, but also the time. These observations give rise to notions *of space scalability* and *space-time* scalability, which we shall formally define later. In addition, the fixed size of some data structures, such as arrays or address fields, may inherently limit the growth in the number of objects they track. We call the absence of this type of limitation *structural scalability*.

The scalability of a system may be impaired by inherent wastefulness in frequently repeated actions. It may also be impaired by the presence of access algorithms that lead to deadlock or that result in suboptimal scheduling of resources. Such systems may function well when the load is light, but suffer substantial performance degradation as the load increases. We call systems that do not suffer from such impairments *load scalable.* Classical examples of poor load scalability include Ethernet bus contention and busy waiting on locks in multiprocessor systems. We note in passing that systems with poor load scalability can be hard to model, because they may migrate from a state of graceful function to overload, and perhaps from there into deadlock.

The improvement of structural, space, and space-time scalability depends on the judicious choice of algorithms and data structures, and synchronization mechanisms. Since algorithmic analysis, programming techniques, and synchronization are well documented elsewhere [1, 19, 10], we shall not dwell on them further here. Nor shall we consider scalability aspects of parallel computation or connectivity. In the present paper, our focus shall be on load scalability.

In the next section, we elaborate on scalability concepts we described above. We then look at some examples. These examples will be used to illustrate the notion of unproductive cycles as well as instances in which scalability is undermined by one or more scheduling rules.

2. TYPES OF SCALABILITY

2.1 General Types of Scalability

We consider four types of scalability here: load scalability, space scalability, space-time scalability, and structural scalability. A system or system component may have more than one of these attributes. Moreover, two or more types of scalability may mutually interact.

- *Load scalability.* We say that a system has load scalability if it has the ability to function gracefully, i.e., without undue delay and without unproductive resource consumption or resource contention at light, moderate, or heavy loads while making good use of available resources. Some of the factors that can undermine load scalability include (1) the scheduling of a shared resource, (2) the scheduling of a class of resources in a manner that increases its own usage (self-expansion), and (3) inadequate exploitation of parallelism.

The Ethernet does not have load scalability, because the high collision rate at heavy loads prevents bandwidth from being used effectively. The token ring with nonexhaustive service does have load scalability, because every packet is served within a bounded amount of time.

A scheduling rule may or may not have load scalability, depending on its properties. For example, the Berkeley UNIX 4.2BSD operating system gives higher CPU priority to the first stage of processing inbound packets than to either the second stage or to the first stage of processing outbound packets. This in turn has higher priority than I/O, which in turn has higher priority than user activity. This means that sustained intense inbound traffic can starve the outbound traffic or prevent the processing of packets that have already arrived. This scenario is quite likely at a web server [16]. This situation can also lead to livelock, a form of blocking from which recovery is possible once the intense packet traffic abates. Inbound packets cannot be processed and therefore are unacknowledged. This eventually causes the TCP sliding window to shut, while triggering retransmissions. Network goodput then drops to zero. Even if acknowledgments could be generated for inbound packets, it would not be possible to transmit them, because of the starvation of outbound transmission. It is also worth noting that if I/O interrupts and interrupts triggered by inbound packets are handled at the same level of CPU priority, heavy inbound packet traffic will delay I/O handling as well. This delays information delivery from web servers.

A system may also have poor load scalability because one of the resources it contains has a performance measure that is *self-expanding*, i.e., its expectation is an increasing function of itself. This may occur in queueing systems in which a common FCFS work queue is used by processes wishing to acquire resources or wishing to return them to a free pool. This is because the holding time of a resource is increased by contention for a like resource, whose holding time is increased by the delay incurred by the customer wishing to free it. Self-expansion diminishes scalability by reducing the traffic volume at which saturation occurs. In some cases, it might be detected when performance models of the system in question based on fixed-point approximations predict that performance measures will increase without bound, rather than converging. In some cases, the presence self-expansion may make the performance of the system unpredictable when the system is heavily loaded. Despite this, the operating region in which self-expansion is likely to have the biggest impact may be readily identifiable: it is likely to be close to the point at which the loading of an active or passive resource begins to steeply increase delays.

Load scalability may be undermined by inadequate parallelism. A quantitative method for describing parallelism is given in [15]. Parallelism may be regarded as inadequate if system structure prevents the use of multiple processors for tasks that could be executed asynchronously. For example, a transaction processing (TP) monitor might handle multiple tasks that must all be executed within the context of a single process. These tasks can only be executed on one processor in a multiprocessor system, because the operating system only sees the registers for the TP monitor, not for the individual tasks. Such a system is said to be *single-threaded.*

- *Space scalability.* A system or application is regarded as having space scalability if its memory requirements do not grow to intolerable levels as the number of items it supports increases. Of course, *intolerable* is a relative term. We might say that a particular application or data structure is space scalable if its memory requirements increase at most sublinearly with the number of items in question. Various programming techniques might be used to achieve space scalability, such as sparse matrix methods or compression. Because compression takes time, it is possible that space scalability may only be achieved at the expense of load scalability.
- *Space-time scalability.* We regard a system as having space-time scalability if it continues to function gracefully as the number of objects it encompasses increases by orders of magnitude. A system may be space-time scalable if the data structures and algorithms used to implement it are conducive to smooth and speedy operation whether the system is of moderate size or large. For example, a search engine that is based on a linear search would not be space-time scalable, while one based on an indexed or sorted data structure such as a hash table or balanced tree could be. Notice that this may be a driver of load scalability for the following reasons:
    1. The presence of a large number of objects may lead to the presence of a heavier load.
    2. The ability to perform a quick search may be affected by the size of a data structure and how it is organised.
    3. A system or application that occupies a large amount of memory may incur considerable paging overhead.

Space scalability is a necessary condition for space-time scalability in most systems, because excessive storage requirements could lead to memory management problems and/or increased search times.

- *Structural scalability.* We think of a system as being structurally scalable if its implementation or standards do not impede the growth of the number of objects it encompasses, or at least will not do so within a chosen time frame. This is a relative term, because scalability depends on the number of objects of interest now relative to the number of objects later. Any system with a finite address space has limits on its scalability. The limits are inherent in the addressing scheme. For instance, a packet header field typically contains a fixed number of bits. If the field is an address field, the number of addressable nodes is limited. If the field is a window size, the amount of unacknowledged data is limited. A telephone numbering scheme with a fixed number of digits, such as the North American Numbering Plan, is scalable only to the extent that the maximum quantity of distinct numbers is significantly greater than the set of numbers to be assigned before the number of digits is expanded.

Load scalability may be improved by modifying scheduling rules, avoiding self-expansion, or exploiting parallelism. By contrast, the other forms of scalability we have described are inherent in architectural characteristics (such as word length or the choice of data structures) or standards (such as the number of bits in certain fields) that may be difficult or impossible to change.

2.2 Scalability over Long Distances

- *Distance Scalability.* An algorithm or protocol is distance scalable if it works well over long distances as well as short distances.
- *Speed/Distance Scalability.* An algorithm or protocol is speed/distance scalable if it works well over long distances as well as short distances at high and low speeds.

The motivation for these types of scalability is TCP/IP. Its sliding window protocol shows poor speed/distance scalability in its original form. Protocols in which bit status maps are periodically sent from the destination to the source, such as SSCOP [8] are intended to overcome this shortcoming. This is the subject of future work and will not be considered further here.

3. INDEPENDENCE AND OVERLAP BETWEEN SCALABILITY TYPES

When exploring taxonomy of defining characteristics, it is natural to ask whether they are independent of one another or whether they overlap. The examples presented here show that there are cases where load scalability is not undermined by poor space scalability or structural scalability. Systems with poor space scalability or space-time scalability might have poor load scalability because of the attendant memory management overhead or search costs. Systems with good space-time scalability because their data structures are well engineered might have poor load scalability because of poor decisions about scheduling or parallelism that have nothing to do with memory management.

Let us now consider the relationship between structural scalability and load scalability. Clearly, the latter is not a driver of the former, though the reverse could be true. For example, the inability to exploit parallelism and make use of such resources as multiple processors undermines load scalability, but could be attributed to a choice of implementation that is structurally unscalable.

The foregoing discussion shows that the types of scalability presented here are not entirely independent of one another, although many aspects of each type are. Therefore, though they provide a broad basis for a discussion of scalability, that basis is not orthogonal in the sense that a suitable set of base vectors could be. Nor is it clear that an attempt at orthogonalization, i.e., an attempt to provide a characterisation of scalability consisting only of independent components, would be useful to the software practitioner, because the areas of overlap between our aspects of scalability are a reflection of the sorts of design choices a practitioner might face.

4. QUALITATIVE ANALYSIS OF LOAD SCALABILITY AND EXAMPLES

In this section, we illustrate the analysis of load scalability. Our examples fall into two categories: systems with repeated unproductive cycling through finite state machines, and systems whose poor load scalability can be overcome with the judicious choice of a job scheduling rule. Of course, the use of finite state

machines to characterise system behaviour is not new [17]. In the context of software performance engineering, small finite state machines have been used to depict the behaviour of embedded components [18]. Concurrently interacting finite state machines have been studied by Kurshan and coworkers [14].

By an unproductive cycle, we mean a repeated sequence of states in which a process spends an undesirable amount of time using resources without actually accomplishing the goals of the user or programmer. Classical examples include busy waiting on locks in multiprocessor systems, Ethernet bus contention, and solutions to the dining philosophers problem that do not have controls for admission to the dining room [12, 7]. Other examples include systems whose performance does not degrade gracefully as the load on them increases beyond their rated capacity. Some systems or procedures that are perceived as scaling poorly use resources inefficiently. They may hold one or more resources while being in an idle or waiting state, or they may incur overheads or delays that are tolerable at low volumes of activity but not at high volumes.

We now turn to examples that suggest how load scalability might be improved by reducing the occurrence of unproductive cycles or by modifying a job scheduling rule.

4.1 Busy Waiting on Locks

In the early days of multiprogrammed computing, contention for shared objects was arbitrated solely with the aid of hardware locks in memory. The lock is set by the first process that finds it unset; this process then unsets it when it has finished with the object protected by the lock. Other processes must repeatedly check it to see if it is unset. This repeated checking steals processing and memory cycles that could otherwise be used for useful work. Repeated cycle stealing, especially in a multiprocessor environment, degrades performance.

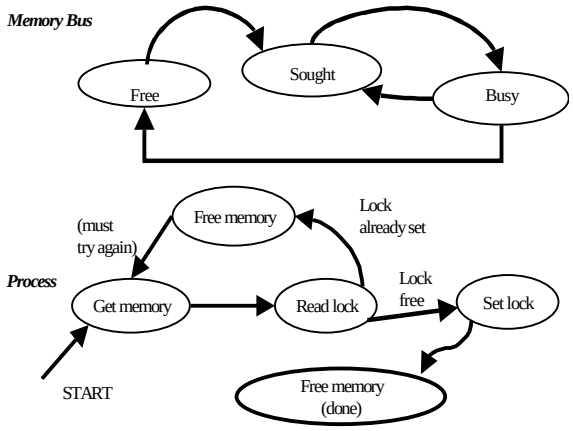

Figure 1: Busy waiting on locks.

Figure 1 shows a state transition diagram for this concurrency control mechanism. The first directed graph represents the states of the memory bus, the second the states of a process trying to access the lock. Each attempt to access a lock corresponds to a complete cycle through the memory bus state transition diagram, as well as through a loop consisting of the three states *Get memory, Read memory, Free memory* in that order. Each process trying to access the lock corresponds to an instance of the second directed graph. Although one cannot break these cycles, one can reduce the frequency with which they are traversed by implementing mutual exclusion with a semaphore [10, 7]. By putting a process to sleep until the object to which it seeks access becomes available, the semaphore mechanism simultaneously reduces lock contention, memory bus contention, and the unproductive consumption of CPU cycles. However, frequent context switching in a multiprocessor system with shared memory will still cause heavy memory bus usage, because of contention for the ready list lock [6]. Thus, the locking mechanism is not load scalable. However, minimizing the use of locks does ameliorate the problem by reducing the occurrence of unproductive cycles spent busy waiting.

4.2 Ethernet and Token Ring: a Comparison

The delay and throughput of the Ethernet and token ring with cyclic nonexhaustive service were compared in [5]. Here, we consider their performance with the aid of state transition diagrams. Figure 2 shows state transition diagrams for an Ethernet bus and for a single packet awaiting transmission. The bus repeatedly goes through the unproductive cycle *(Busy, Collision, Idle, Busy, Collision, …)* when more than one workstation attempts to transmit simultaneously. Similarly, a station attempting to transmit a packet could repeatedly go through the cycle *(Silent, Backoff, Silent, Backoff,…)* before finally being sent successfully. It is well known that the performance of the Ethernet does not degrade gracefully as it approaches saturation [2]. Repeated traversals of these cycles help explain why. Introducing a bus with higher bandwidth only defers the onset of the problem. Thus, the CSMA/CD protocol is not load scalable.

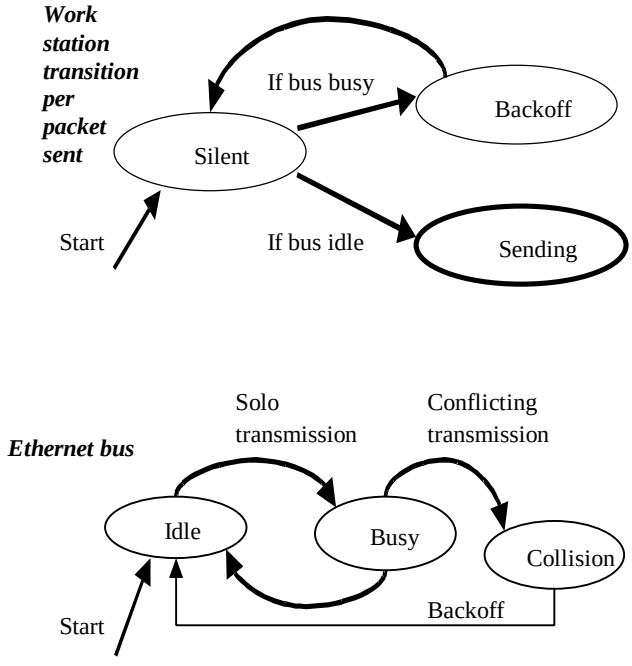

Figure 2: Transition diagram for Ethernet.

Let us now examine the token ring. The state transition diagrams for the token and for a workstation attempting transmission of a single packet are shown in Figure 3. We see that the token moves between available states cyclically, and that it suffers increased delay in moving from one station to the next only if it is doing useful work, i.e. if it is in use. A station attempting to transmit a packet is silent until the token becomes available to it. It then sends the packet without the need for backoff. Notice that the state transition graph for the LAN card contains no cycles. This helps to explain the graceful performance degradation of the token ring as it approaches saturation. The contrast with the Ethernet's cyclic behaviour is clear.

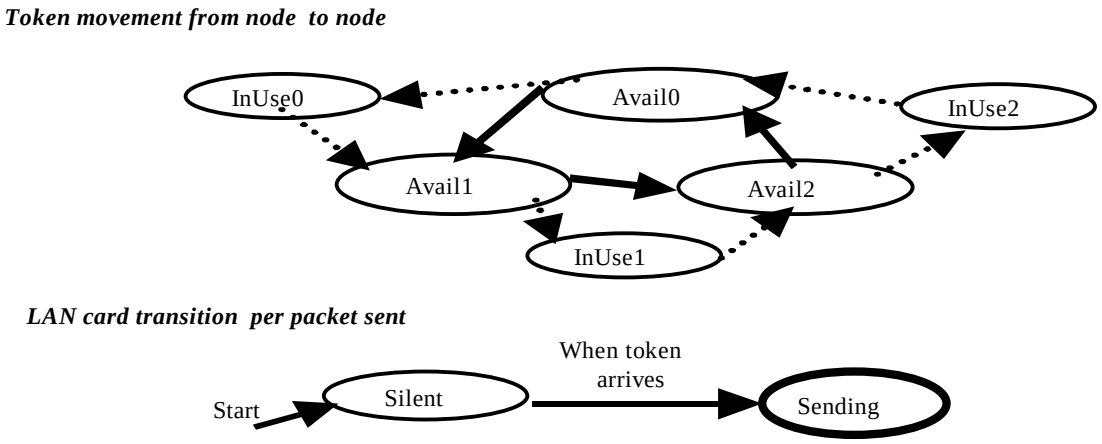

Figure 3: Transition diagrams for token ring.

4.3 Museum Checkroom

This is a simplification of a problem occurring in replicated database systems [4]. At a museum checkroom, visitors are required to join a common FCFS queue to deposit and collect their coats. The scarce resources are coat hangers (passive) and attendants (active). Coats are placed on hangers, which are hung on carousels, each of which is maintained by a dedicated attendant. An attendant taking a coat performs a linear search on the assigned carousel to find an empty hanger, or to find the visitor's coat. Our objective is to maximise the time the customer can spend looking at exhibits and spending money in the museum restaurant and gift shop.

The performance of this system degrades badly under heavy loads. First, the service time increases with the occupancy of the hangers, because it takes longer to find free ones. Second, the system is prone to deadlock at heavy loads, e.g., during the winter holiday season. In the morning most visitors are leaving their coats; in the evening they are all picking them up. The deadlock problem arises at midday or in the early afternoon, when many might be doing either. Deadlock occurs when the hangers have run out, and visitors wishing to collect coats are stuck in the queue behind those wishing to leave them. Attendants can break the deadlock by asking those who wish to collect coats to come forward. This amounts to the resolution of deadlock by timeout, which is inefficient.

With its original service rules, the museum checkroom will function adequately at light loads and with low hanger occupancy, but will almost certainly deadlock at some point otherwise. Regardless of the number of attendants and the number of coat hangers, the system is likely to go into deadlock if the volume of traffic is heavy enough. Increasing the number of hangers only defers the onset of deadlock; it does not eliminate the possibility. Load scalability is further undermined because the holding time of a currently occupied hanger is increased in two ways:

1. The search time for unused hangers increases as the hanger occupancy increases.
2. A customer arriving to collect a coat (and thus free a hanger) must wait behind all other waiting customers, including those wishing to leave coats. This increases the time to free a hanger.

Both of these factors make the hanger holding time *self-expanding*. If the holding time is self-expanding, the product of the customer arrival rate and the hanger holding time, i.e., the expected number of occupied hangers, will increase to exceed the total number of hangers even if the customer arrival rate does not increase. This is a sure sign of saturation.

Notice that the impediments to the scalability of this system vary with the time of day. In the morning, when almost all visitors are leaving coats, the impediments are the number of attendants

and the somewhat confined space in which they work. The same is true at the end of the day, when all visitors must pick up their coasts by closing time. At midday, the principal impediment is the FCFS queueing rule, which leads to deadlock.

For this system, load scalability can be increased with the following modifications.

1. There should be separate queues for those collecting and leaving coats, with (nonpreemptive) priority being given to the former so as to free the hangers as quickly as possible. This priority ordering reduces the tendency of the holding time to be self-expanding. Deadlock is avoided because the priority rule guarantees that a hanger will be freed if someone arrives to collect a coat.
2. To prevent attendants from jostling each other in peak periods, there should also be more than one place at which they can serve museum visitors, regardless of the carousel on which the coat is kept.
3. To prevent visitors from jostling one another, there must be a wide aisle between the queue and the counter.
4. A sorted list of free hangers could be maintained for each carousel, to reduce the time to search for one.

The first and second modifications are cheap, the third and fourth less so. The first alone would yield substantial benefits by eliminating the risk of deadlock and reducing hanger occupancy. The usefulness of the second depends on the patience of attendants and visitors alike, especially as closing time approaches. The cost of the fourth modification must be weighed against the reduced cost of keeping fewer attendants around.

Analogy of this Problem with Computer Systems. The visitors correspond to processes. The carousels and hangers respectively correspond to memory banks and memory partitions. The attendants correspond to processors. Finally, the corridor between the carousels and the counter at which visitors arrive collectively correspond to a shared memory bus via which all memory banks are accessed. This is an example in which the main (and most easily surmounted) impediment to scalability lies in the mechanism for scheduling service, rather than in the unproductive consumption of cycles. The corridor (memory bus) is a secondary impediment, because attendants bump into each other while hanging or fetching coats during peak periods, unless each attendant is assigned to one carousel only. In a computing environment, this would be analogous to having a single bus for each memory bank. Finally, scalability is impeded by the narrow doorway between the head of the queue and the counter. This is analogous to there being a unique path from the CPU to the entire set of memory banks, i.e., the bus.

For performance analysis of an open arrival system in which jobs queue for a memory partition before processing and I/O may begin, the reader is referred to [15, 3].

5. IMPROVING LOAD SCALABILITY

Our examples of poor load scalability show that it can have a variety of causes, ranging from access policies that are repetitively wasteful of active resources (e.g., busy waiting) to assignment policies that undermine the "common good" of the system by causing passive resources (e.g., coat hangers) to be held longer than is necessary to accomplish specific tasks.

One way to improve the scalability of a system is to reduce the amount of time it spends in unproductive cycles. This can be done by modifying the implementation so that the time spent cycling is reduced, or by eliminating the cycle altogether through a structural change or a scheduling change.

If a system is not structurally unscalable, e.g., if its scalability is not limited by its address space, its load scalability might be improved by mitigating the factors that prevent it from being load scalable, space-scalable, or space-time scalable. The first steps to improvement are:

- identification of unproductive execution cycles and their root causes, as well as means of breaking them.
- understanding the sojourn time in the cycles, and means of shortening them,
- understanding how and whether a system could migrate into an undesirable absorbing state (such as deadlock) as the load increases, and devising scheduling rules and/or access control mechanisms to prevent this from happening,
- identifying system characteristics that make performance measures self-expanding, and finding ways to eliminate or circumvent them,
- determining whether scalability is impeded by a scheduling rule, and altering the rule,
- understanding whether asynchronicity can be exploited to allow parallel execution, and modifying the system accordingly. Notice that when evaluating the increased benefit of parallelism, one must also account for the additional cost of controlling interprocess communication.

Not all of these steps are applicable to all situations. Nor can we be certain that this list is complete. But, once these steps have been taken, one may attempt to identify design changes and/or system improvements that either reduce the sojourn times in the cycles or break the cycles altogether. This must be done with care, because any design change could (a) result in the creation of a new set of cycles, and/or (b) result in a the creation of a new scheduling rule that induces anomalies of its own. Moreover, a modification might simply reveal the existence of another bottleneck that was concealed by the first one. On the other hand, modifications may well lead to unintended beneficial side effects such as reducing resource holding times and delays.

Let us reconsider the load scalability examples in the light of the foregoing.

Semaphores reduce the occurrence of unproductive busy waiting and hence memory cycle stealing in multiprocessor systems with shared memory. Thus, systems that use semaphores rather than locks are likely to perform better under heavy loads. However, the solution comes with a cost, the overhead of managing semaphores. The use of semaphores might also expose a previously hidden bottleneck, namely lock contention for the head and tail of the CPU run queue (ready list) in the presence of too many processors. This is not an argument against the introduction of semaphores, merely a warning about the next bottleneck that might arise.

Eliminating collisions in an unswitched Ethernet LAN clearly increases capacity for a given bandwidth. Unlike the Ethernet, the token ring provides an upper bound on packet transmission time, albeit at the cost of waiting ones turn as the token moves from one node to the next.

The museum checkroom example illustrates many facets of scalability. Giving priority to customers collecting coats in the museum checkroom reduces the average number of occupied hangers. This contributes to the reduction of delay by reducing the time to free a hanger, thus making it available to an arriving visitor. This in turn reduces the time the checkroom attendants spend on each visitor, and maybe even the number of attendants required to maintain a given level of service quality. It also prevents deadlock. Indexing the free and occupied hangers reduces search time, though not retrieval time, since the desired hanger is always brought to the front for access. Deploying attendants to the carousel where they are needed instead of assigning them to particular ones increases their utilization. Allowing them enough space to move around freely also reduces customer service time. Dedicating one attendant to coat retrievals reduces hanger occupancy while making scarce hangers available sooner. All of these modifications improve the ability of the system to function properly at increased loads, either by cutting down on active processing time or by reducing the holding times of passive resources.

6. SOME MATHEMATICAL ANALYSES

6.1 Comparison of Semaphores and Locks for Implementing Mutual Exclusion

In this section, we propose a framework for analysing the relative performance impacts of different mechanisms for implementing mutual exclusion. In particular, we shall compare locks and semaphores, as in our previous example. A semaphore does not eliminate lock contention entirely: it simply focuses lock contention on the shared data structures accessed by its primitives while preparing to put a process to sleep until a critical section becomes available. This mechanism could be applied to protect data in shared memory, or, as might be the case with a database record, on disc. Let *L* be the lock used to implement mutual exclusion without semaphores, and *S* be a lock that is used to impose mutual exclusion on the data structures used to implement a semaphore operation. In the database example, *L* could reside on disc before being loaded into memory as part of a record. The lock used by the semaphore might protect a ready list or queue: in any case it is only accessed by designated primitive operations that are part of the kernel of the host operating system.

Let p_i denote the probability that a lock of type i (=*L,S)* is accessed and set (or reset) successfully on the current attempt. Let a_i denote the number of successful attempts that must be made to acquire and relinquish control of a critical section. For a simple lock, a_L =2 since a test-and-set or a test-and-reset would be required for acquisition and release respectively. We make the simplifying assumption that the probabilities of success on each attempt to access the lock are identical, and that successive attempts are independent. The probability of success depends on how the lock is used, the number of processors, their loads, and on how much context switching is occurring at the time. The need to make at least one attempt, and the assumption that successes are i.i.d. give the number of attempts needed for success a displaced geometric distribution with mean $1/p_i$. If we assume that costs of accessing locks *S* and *L* are the same, and that some constant overhead *k* is associated with the semaphore mechanism, the ratio of the costs of the simple locking mechanism to that of the semaphore mechanism is given by

$$f(p_L, p_S) = (a_L / p_L) / [k + a_L / p_S]$$

We use semaphores because we expect p_S to be a good deal larger than p_L, but we do not know what values these probabilities take in practice. Therefore, we have evaluated *f* over a large subset of the unit square, (0,0.95]x(0,0.95], for $a_L = a_L = 2$ and *k*=0,5,10 as shown in Figure 4. The scale of probabilities on the long horizontal axis is repeated to allow the simultaneous display of three surfaces, one for each value of the semaphore overhead examined.

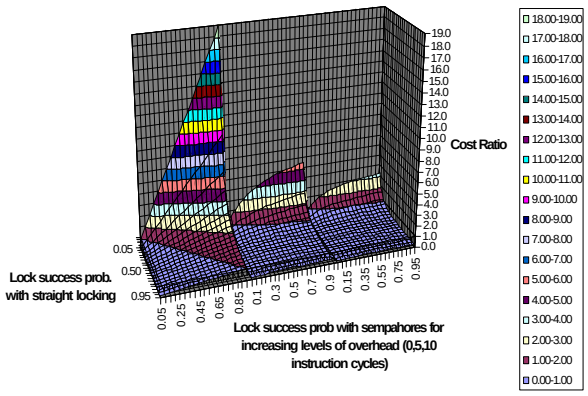

Figure 4: Ratios of the expected number of lock attempts with straight locking and semaphores

As one might expect, when the probability of successfully obtaining a lock is high for both locking mechanisms, there is not much to choose between them. As p_L tends to zero, the cost of the simple locking mechanism is many times that of the semaphore mechanism. This means that the onset of problems with load scalability is much less likely when a semaphore mechanism is used than when a straight locking mechanism is used, and that the performance of an implementation using the former is much more robust than that of a system using the latter. To enhance scalability, we would choose the mechanism whose performance is least sensitive to a change in the operating load.

In this instance, the semaphore mechanism is the better candidate, as prior literature has led us to expect [6].

6.2 Museum Checkroom

We have argued that the average delay in queueing for a coat hanger will be minimised if those collecting coats are given head of the line (HOL) priority over those leaving their coats, because this minimises the average hanger holding time. This is also the only discipline that avoids self-expansion of queueing delay with respect to hanger holding time.

Recall that our objective is to maximise the time available to look at exhibitions while avoiding deadlock on contention for hangers. The sojourn time in the museum may be decomposed into time spent waiting to deposit a coat *D*, exhibit viewing time *V*, and coat pickup time *T*. The hanger holding time *H* is given by

$$H = T + V$$

and the sojourn time in the museum is given by

$$S = D + T + V.$$

Since the museum is only open for a finite time each day, *M* say, we immediately have

$$S, H \pounds M.$$

Since we cannot control the behaviour or preferences of visitors, we cannot control *V*. Still, one should allow *V* to be as large as possible by reducing at least one or both of *D* and *T*, because this is what the visitor came to do, and because it allows more time to visit gift shops, coffee bars, etc. within the museum.

Because increased hanger holding time *H* increases the risk of deadlock while increasing both *D* and *T*, the queueing times to leave and collect coats respectively, we should first focus on reducing *T* in order to reduce *H*, and hence eliminate self-expansion. Under FCFS scheduling, customers collecting coats and leaving them delay each other. This increases *H* by increasing *T*. If *N* denotes the number of maximum number of hangers and λ the visitor throughput, we must have $\lambda H < N$ for the checkroom queue to be in equilibrium. Hence, increasing *T* tends to make the queue unstable by increasing *H*. Of course, *T* could also be reduced by speeding up service, but that would only increase the value of λ for which saturation might occur; it cannot prevent deadlock altogether.

Let *c* denote the customers in queue to collect a coat, and *a* the number of customers in queue to leave a coat at the time the next customer arrives to pick one up. Let *s* denote the service time. If the queue is FCFS and there is only one attendant, the latest customer must wait

$$T = (c + a + 1)s$$

to obtain a coat. This is an upper bound on *T* if more than one attendant is present. If those collecting coats are given HOL priority, we have $T \pounds (c+1)\, s$. This clearly reduces the upward pressure on *H*, and hence *D* and *T*. This shows that using HOL eliminates self-expansion and hence reduces the tendency to saturate the hangers. This policy is optimal with respect to the average waiting time of all customers, regardless of whether they are collecting or leaving coats, because it reduces to shortest hanger holding time next.

Not all customers may understand the optimality of this policy. If customers collecting coats temporarily outnumber customers leaving them or vice versa at certain times of the day, it may be worthwhile to convince the minority that they are not being ignored by ensuring that they are served one of every *k* times. This rule must be applied judiciously, because it allows self-expansion. Hence, *k* must be chosen with care. If too many customers leaving coats are served ahead of those collecting them, the resulting self-expansion in the hanger holding time could lead to deadlock, or at the very least, an increase in the average values of *D* and *T*, as well as of *H*. A policy for determining *k* in response to changing conditions is beyond the scope of this paper.

7. CONCLUSIONS

The foregoing represents a first attempt to classify the aspects of a system that fundamentally affect its scalability. By distinguishing between structural scalability and load scalability, we can distinguish between those aspects that limit growth because of space and/or structural considerations alone and those that affect performance.

8. ACKNOWLEDGMENTS

The author has benefited from discussions with Arthur Berger, David Hoeflin, Yaakov Kogan, Elaine Weyuker, and Murray Woodside.

9. REFERENCES

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[2] Almes, G. T., and Lazowska, E. D. The Behavior of Ethernet-Like Computer Communications Networks. Technical Report no. 79-05-01, Dept. of Computer Science, University of Washington, April 1979.

[3] Avi-Itzhak, B., and Heyman, D. P. Approximate queueing models for multiprogramming computer systems. *Operations Research* 21, 1, 1973, pp. 1212-1230.

[4] Bondi, A. B., and Jin, V. Y. A performance model of a design for a minimally replicated distributed database for database-driven telecommunications services. *Distributed and Parallel Databases* 4 (1996), 295-397.

[5] Bux, W. Local area subnetworks: a performance comparison. *IEEE Trans. Communications* COM-29,10 (1981), 1645-1673.

[6] Denning, P. J., Dennis, T., and Brumfield, J. A. Low contention semaphores and ready lists. *Communications of the ACM* 24 (1981), 687-699.

[7] Dijkstra, E. W. Solution of a problem in concurrent programming control. *Communications of the ACM* 8, 9 (1965), 569.

[8] Doshi, B. T., Johri, P. K., Netravali, A.N., and Sabnani, K.K. Error and Flow Control Performance of

High speed Protocols. *IEEE Trans. Commun.* COM 41, 5 (1993), 707-720.

[9] Eager, D., Zahorjan, J., and Lazowska, E. D. Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. on Computers* 38 (3), 1989, pp. 408-423.

[10] Habermann, A. N., *Introduction to Operating Systems Design*, SRA, Chicago, 1976.

[11] Hennessy, John. The Future of Systems Research. *IEEE Computer*, August 1999, pp. 27-33.

[12] Hoare, C. A. R. Monitors: an operating system structuring concept. *Comms. ACM* 17, 10 (1974), 549-557.

[13] Jogalekar, P. P., and Woodside, C. M.. A Scalability Metric for Distributed Computing Applications in Telecommunications. *Proc. Fifteenth International Teletraffic Congress (ITC-15) vol. 2a,* pp. 101-110, 1997.

[14] Katznelson, J., and Kurshan, R. S/R: A language for specifying protocols and other coordinating processes. *Fifth Annual International Phoenix Conference on Computers and Communications*, 1986.

[15] Latouche, G. Algorithmic Analysis of a Multiprogramming-Multiprocessing Computer System. *J. A.C.M.* 28, 4, 1981, pp. 662-679.

[16] Mogul, J. C., and Ramakrishnan, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems* 15, 3, 1997, 217-252.

[17] Shaw, A. C. *The Logical Design of Operating Systems*. Prentice Hall, 1974.

[18] Smith, C. U., and Williams, L. Performance Engineering Evaluation of CORBA-based Distributed Systems. *First International Workshop of Software and Performance (WOSP98)*, 1998.

[19] Wirth, N. *Algorithms + Data Structures = Programs.* Prentice Hall, 1976.