Proceedings of the First International EURO-PAR Conference, August 1995, pages 28-31, LNCS 966.

On the Scalability of Demand-Driven Parallel Systems

Ronald C. Unrau[1], Michael Stumm[2] and Orran Krieger[2]

[1] IBM Canada Ltd, North York, Canada, M3C 1V7
[2] Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, M5S 1A4

Abstract. Demand-driven systems follow the model where customers enter the system, request some service, and then depart. Examples are databases, transaction processing systems and operating systems, which form the system software layer between the applications and the hardware. Achieving scalability at the system software layer is critical for the scalability of the system as a whole, and yet this layer has largely been ignored.
In this paper, we characterize the scalability of the system software layer of demand-driven parallel systems based on fundamental metrics of quantitative system performance analysis. We develop a set of sufficient conditions so that if a system satisfies these conditions, then the system is scalable. We further argue that in practice these conditions are also necessary. In the remainder of the paper, we use the necessary and sufficient conditions to develop a set of practical design guidelines, to study the effect of application workloads, and to examine the scalability behavior of a system with only a limited number of processors.

1 Introduction

Demand-driven systems follow the model where customers enter the system, request some service, and then depart. Software-based demand-driven systems include databases, transaction processing systems, and operating systems. The design and implementation of scalable demand-driven parallel systems is difficult, yet important, if the performance potential of emerging large-scale parallel hardware bases is to be exploited.

Existing demand-driven parallel systems have typically been scaled to accommodate a large number of processors in an *ad hoc* manner, by repeatedly identifying and then removing the most contended bottlenecks. For example, on shared-memory multiprocessors, bottleneck contention is reduced either by splitting existing locks, or by replacing existing data structures with more elaborate, but concurrent ones. The process is long and tedious, and results in systems that 1) are fine-tuned for a specific architecture/workload and hence not easily portable to other environments with respect to scalability; 2) are not scalable in a generic sense, but only until the next bottleneck saturates; and 3) have a large number of locks that need to be held for common operations, with correspondingly large overhead.

Clearly, a more structured approach to designing scalable systems is needed. The key to finding an appropriate structuring is to first understand scalability itself. Although issues in scalability have been studied widely and many papers purporting scalable systems have been published [2, 4, 6, 9], little progress has been made towards a practical characterization of scalability. Most papers avoid defining scalability entirely, or simply state that their systems are targeted at some large number of processors [5].

An exception is Nussbaum and Agarwal's definition of scalability [8]: *The scalability of a machine for a given algorithm and problem size is the ratio of the asymptotic speedup on the real machine and the ideal realization of an EREW PRAM.* This definition is important because: 1) it recognizes that the applications and the architecture must work together if scalability is to be achieved; 2) it yields a quantitative measure of scalability, where a ratio of 1 indicates perfect scalability; and 3) it incorporates the notion of workload, in that the problem size is an explicit part of the definition.

However, the definition is not helpful for system designers for several reasons. First, the definition is based on speedup, while for demand-driven systems throughput is the more important performance metric[3]. Second, the definition of scalability is expressed in terms of inherent parallelism and asymptotic limits, and is therefore difficult to apply to real systems of fixed size and unknown inherent parallelism. Third, even if the definition could be used to evaluate the scalability of a real system, the quantitative number that is its result yields no insight into how to design or build a scalable demand-driven system.

In this paper, we develop an operational characterization of scalability that is applicable to demand-driven systems. Our development is based on an analysis of fundamental equations of quantitative performance evaluation, and is based on the definition that for a given workload, a system will scale if no resource saturates as the number of processors increases. From this definition, we argue that the following conditions are sufficient for a demand-driven system to scale:

1. The time at a particular resource devoted to servicing a particular request is bounded by a constant.
2. The number of resources available to service a particular class of request increases in proportion to p.
3. The system is balanced in its service capabilities.
4. The servicing of individual requests is localized and independent.

In addition, we believe that for practical systems, these conditions are necessary as well as sufficient for scalability.

The results we present are developed in a mathematical framework, which allows reasoning about asymptotic behavior, but the conditions are expressed in terms of physical quantities, so that they are of direct practical importance. In particular, Section 3 presents a set of design guidelines for building scalable

[3] It is interesting to note that Nussbaum and Agarwal specifically exclude the operating system from consideration in their definition and treat it as an extension of the hardware.

demand-driven systems that are derived from the scalability conditions. The mathematical derivation is based on broad assumptions of workload characteristics, and Section 4 refines these assumptions to see how real applications affect the scalability of a demand-driven system. Finally, Section 5 discusses issues in evaluating the scalability of existing systems, where only a limited number of processors are available.

While the following development is applicable to demand-driven systems in general, we are most familiar with parallel operating systems, and will therefore draw our examples from our experience in implementing an operating system for a scalable shared-memory multiprocessor [10].

2 Conditions for Scalability

In this section, we develop a set of conditions for the scalability of demand-driven systems. The development is based on an analysis of two fundamental performance metrics of computer systems: throughput and utilization. Section 2.1 reviews the definition of these metrics and discusses their properties in a scalable system. The assumptions upon which the scalability conditions are based and the conditions themselves are derived in Section 2.2.

2.1 Performance Metrics

The discussion examines throughput and utilization for servicing the requests of *customers* at *service centers*[4]. A particular class of customer is denoted as c, and a particular service center, or *resource*, as k. (In the following development, we use the terms resource and service center interchangeably.) Since a single service center can potentially support more than one class of customer, the subscripts ck are used to denote the customer class c at service center k.

In a demand-driven system, the notion of customers and service centers exists at multiple levels. At one level, applications are considered customers of the services provided by the system. For example, an operating system supports the services of creating a new process, or handling a page fault. The second level of customers and resources is within the system itself. At this internal level, the services are operations such as acquiring a lock, or allocating a descriptor for a physical page. The resources in these examples are the lock and the list of available page descriptors, respectively. The customers requesting service at the internal level are the servers and fault handlers of the operating system.

Throughput, X, is defined as the number of requests completed per unit time. An example of throughput is the number of page faults that are handled per second. Throughput can be measured at any service center in the system, where a service center is a resource as simple as a lock, or as complex as the system itself. The throughput for a customer class at resource k depends upon the arrival rate of requests, λ_c, and on the visit count at the service center, v_{ck}:

$$X_{ck} = v_{ck} \cdot \lambda_c \tag{1}$$

[4] The terminology and notation used is consistent with Lazowska *et al.* [7].

Formally, the visit count to resource k by customer class c is defined as the ratio of completions at resource k to the total number of system completions for class c. For a parallel demand-driven system, one intuitively expects the arrival rate of requests to the system to increase with the number of processors, either because there are more sequential applications, or because the parallel applications use more resources. However, we also expect the number of service centers to increase as more processors are added, so that the throughput of each resource need not necessarily be higher than for a sequential system. Hence, the visit count v_{ck} will typically decrease as the system size increases.

To illustrate this important point, consider a hash table with b bins that is used to access n elements; each bin and element in the table are locked separately. If b and n both increase with the size of the system, then the throughput required at each bin or element can remain relatively stable, assuming requests are distributed uniformly across the table resources.

Utilization, U, is defined as the percentage of time a resource spends serving requests. The utilization of a particular service center is related to throughput by the following equation:

$$U_{ck} \;=\; s_{ck} \cdot X_{ck} \tag{2}$$

where s_{ck} is the time required to service a request of type c at resource k. Over all customer classes, no resource can be busy more than 100% of the time, so that $1 \geq U_k = \sum_c U_{ck}$.

The resource with the highest utilization is designated the *primary bottleneck*; this is the resource that typically limits further increases in throughput, because of the queuing delay introduced by the resource. Performance can generally be improved by reducing the utilization of the primary bottleneck, either by reducing its service time or the number of requests it must service. These observations are the basis of the "lock-splitting" approach to scalability described earlier.

2.2 Development

Our development is based on the following two assumptions. First, we assume a workload with an arrival rate of requests for service, $\lambda_c(p)$, that increases in proportion to the number of processors in the system, p. Second, we assume that application service requests are spatially distributed across the system.

In a parallel environment, the utilization law of Equation 2 can be expressed as:

$$1 \;\geq\; U_k(p) \;=\; \sum_c U_{ck}(p) \;=\; \sum_c X_{ck}(p) \cdot s_{ck}(p) \tag{3}$$

where the components of the original equation are allowed to become general functions of the number of processors. If the system is to scale in p, then no resource k can saturate, or throughput will be limited. From these considerations, we define a scalable demand-driven system as one in which the utilization of any resource is bounded, such that saturation (ie. infinite queue lengths) never occurs.

From the definition and Equation 3 we conclude that the throughput and service times at resource k can only be functions of p if they are inversely proportional, so that their product is bounded for all p. In the remainder of the development, we use this observation to develop a set of properties sufficient for a demand-driven system to scale. We must consider the following three cases:

Case ***i*.** $s_{ck}(p)$ is independent of p,
Case ***ii*.** $s_{ck}(p)$ increases with p, and
Case ***iii*.** $s_{ck}(p)$ decreases with p.

We argue informally that case i, where service times are independent of p, is a reasonable base for structuring real systems, and derive the conditions sufficient to prevent saturation in this case. We further argue that case ii will make any system unscalable, and that case iii is unreasonable for real systems.

The discussion considers the class specific terms of Equation 3 ($s_{ck}(p)$ and $X_{ck}(p)$) because if any one of the class specific products is not bounded, it will dominate the sum and cause the resource to saturate. In addition, since all terms are functions of p, we will drop the explicit dependence in the notation for the remainder of the development.

Case *i*: Independent Service Time If service times are independent of the number of processors in the system, then operations such as creating a process or mapping a page, do not take longer to complete as the system grows. Keeping the service time bounded as the system grows seems an intuitive goal, albeit challenging to attain. Independent service time implies that the throughput for customer class c at resource k, X_{ck}, cannot increase as a function of p. From Equation 1, the throughput is proportional to both the system wide arrival rate, λ_c, and the visit count to the resource, v_{ck}. Since we expect λ_c to increase linearly in the number of processors, and since X_{ck} is to remain bounded, v_{ck} must decrease by at least $1/p$.

It is possible to have $v_{ck} \propto 1/p$ in practice if a) the number of instances of a resource grows proportional to the size of the system and b) requests are evenly distributed across the resource instances. This concept was illustrated by the hash table example of Section 2.1, where the number of hash bins grows with the size of the system, and is developed formally below. Define $\mathcal{R}_c$ as the set of identical resources available to service requests of class c, $K_c = \mid \mathcal{R}_c \mid$ as the number of resources in this set, and V_c as the total number of visits across the system that are needed to satisfy a request of class c:

$$V_c \; = \; \sum_{k \in \mathcal{R}_c} v_{ck} \tag{4}$$

If all v_{ck} within a set of identical resources are equal, (ie. $v_{ci} = v_{cj}, \forall i, j \in \mathcal{R}_c$) then the summation of Equation 4 can be replaced by a product:

$$V_c \; = \; K_c \cdot v_{ck} \quad \text{or} \quad v_{ck} \; = \; \frac{V_c}{K_c} \tag{5}$$

Since we need $v_{ck} \propto 1/p$, we must have $V_c/K_c \propto 1/p$. Below, we argue that in real systems, V_c should be independent of p, and K_c should increase proportional to p.

Equation 5 expresses v_{ck} as an *average*; it is likely not true at a given instant for a particular resource, but it must be true on average over time (and resources) if the system is to scale. In essence, Equation 5 requires a system to be balanced in its service capabilities to match the expected workload requirements. Note that to meet this requirement, the application load must be balanced across the system. Further, the servicing of individual requests must be independent of the servicing of other requests to different resources of the same class, otherwise concurrency will be compromised.

Case *ii*: Increasing Service Time We now consider the case where service times grow as an increasing function of p, $f(p)$, as processors are added to the system. This situation can occur, for example, if a list whose length increases with p must be searched linearly to service requests. However, we argue that systems with this property are unscalable.

If service time increases as $f(p)$, then the throughput at the resource must decrease as $1/f(p)$ if saturation is to be avoided. From the assumption of increasing λ_c and Equation 5, this requires $v_{ck} = V_c/K_c \propto 1/(pf(p))$. In real systems, it is not reasonable to expect V_c to decrease, because it implies that less work is required to satisfy requests of class c. Further, it is neither reasonable nor desirable to require K_c to grow faster than p. One reason this growth rate is unreasonable is that each resource has some space cost associated with it, so that if K_c grows faster than p, then the space costs will grow prohibitively as p becomes large. Note that the restriction that K_c grow no faster than p also restricts the growth of V_c, as given by Equation 5.

A second scenario that may appear to result in service times increasing with p are requests that require an identical operation simultaneously on many resources. We call this class of operations *compound* requests. Examples of compound requests include requests to destroy a parallel program containing many processes, or to invalidate data that has been replicated across several processors. From the point of view of the system, compound requests can be treated as multiple simultaneous but individual requests, each of which has service time independent of p.

Optimistically, one might hope to exploit concurrency in servicing compound requests, since multiple resources are involved. If the resources in the compound request are sufficiently independent, the service time for the n different resources of a compound request could even approach the service time for a single resource. However, in real systems the different resources are often coupled through shared resources such as locks. This coupling increases the demand on the shared resources, thus increasing the overall service time.

Case *iii*: Decreasing Service Time We now consider the case where service times decrease as a function of p, $1/f(p)$, and argue that this case is unrealistic.

In practical systems, service times can only decrease as $1/f(p)$ if the operation has speedup proportional to $f(p)$. To obtain this speedup, the service must be decomposed, or partitioned, into $f(p)$ independent operations that can be executed in parallel[5]. In practice, this partitioning is generally not possible. For example, operations such as creating a process or mapping a page act on a single process or page descriptor, so there is almost no potential for concurrency.

Having the service times decrease is also unrealistic because the assumption of increasing λ_c immediately precludes the possibility of applying multiple processors to the servicing of a particular request, because each processor must be available to service its share of the system-wide load. Even if all processors were involved in servicing each request, so that $s_{ck} \propto 1/p$, each processor would have to be involved in the handling of more requests, which requires the total visit count, V_c, to increase. This is only realistic under conditions of perfect speedup.

2.3 Conditions for Scalability

The previous discussion can be summarized as follows. We started with the definition that a demand-driven parallel system will scale if no resource saturates as the number of processors increases. We further assumed that the service request rate, λ_c, increases in proportion to the number of processors in the system, p, and that application requests are spatially distributed across the system. From the definition and these two assumptions, we have argued that the following conditions are sufficient for a demand-driven system to scale:

1. The time spent servicing request c at resource k, s_{ck}, is bounded by a constant independent of the number of processors in the system.
2. The number of resources available to service a request of class c, $K_c = \mid \mathcal{R}_c \mid$, increases in proportion to p.
3. The system is balanced in its service capabilities, so that $v_{ck} = V_c/K_c$ on average (Equation 5).
4. The servicing of individual requests is localized and independent, such that $V_c = \sum_{k \in \mathcal{R}_c} v_{ck}$ is independent of p.

As given, these properties form a set of sufficient conditions for a demand-driven system to scale. Further, we believe that: 1) any practical system must be based on the principle of bounded service times; 2) that the number of instances of a particular resource, K_c, can increase no faster than p; and 3) that the total system visit count for a particular class of operations, V_c is not a decreasing function of p. Given these assertions, the conditions above are both sufficient and necessary for scalability.

[5] It is important to note that parallelism in servicing a single request is different from the compound service requests discussed earlier, since compound requests are effectively identical operations performed on many similar resources.

3 Design Guidelines

The properties listed above identify conditions sufficient for a system to scale, but they do not directly specify how the requirements can be met. This section presents a set of design guidelines that are derived from the scalability criteria, and which can be used to design a scalable demand-driven system.

Preserving Parallelism *A demand-driven system must preserve the parallelism afforded by the applications.* Because we do not expect parallelism in servicing a single request, and because the system is primarily demand driven, parallelism within the system can only come from application demand. If several threads of a parallel application (or of simultaneously executing but independent applications) request independent services in parallel, then they must be serviced in parallel. This demand for parallel service can only be met if the number of service centers increases with the size of the system (scalability property 2), and if the concurrency available in accessing data structures also grows with the size of the system (scalability property 4).

Bounded Overhead *The overhead for each independent system service request must be bounded by a constant* [3]. This requirement follows directly from scalability property 1. If the overhead of each service call increases with the number of processors, the system will ultimately saturate, so the demand on any single resource cannot increase with the number of processors. For this reason, system-wide ordered queues cannot be used and objects must not be located by linear searches if the queue lengths or search lengths increase with the size of the system. Instead, structures that support search in order constant time, such as static positioning (for example, arrays), must be used.

The principle of bounded overhead is also applied to the space costs of the internal data structures. While the data structures are required (by scalability property 2) to grow at a rate proportional to the physical resources of the hardware, the principle of bounded space cost restricts growth to be no more than linear. For example, the size of memory management data structures should depend only on the amount of physical memory [1].

Preserving Locality *A demand-driven system must preserve the locality of the applications.* It is important to consider the memory access locality in large-scale systems, because, for example, many large-scale shared memory multiprocessors have non-uniform memory access (NUMA) times, where the cost of accessing memory is a function of the distance between accessing processor and the target memory, and because cache consistency incurs more overhead in large systems.

The mathematical model that we have developed is at the data structure level, and does not directly include effects due to memory placement or access latency. However, these effects are reflected through the service times, which will increase if remote accesses are made in a NUMA system. In particular, the latency to remote memory can be expected to increase as the system grows, and will increase even faster if the network is contended. Thus, the requirement that a demand-driven system preserves the locality of the applications is a direct consequence of scalability property 1.

Locality can be increased a) by properly choosing and placing data structures internal to the system, b) by directing requests from the application to nearby service points, and c) by enacting policies that increase locality in the applications' memory accesses and system requests. For example, policies should attempt to run the processes of a single application on processors close to each other, place memory pages in proximity to the processes accessing them, and direct file I/O to devices nearby. Within the operating system, descriptors of processes that interact frequently should lie close together, and memory mapping information should lie close to the processors that must access it to handle page faults.

4 Workload Effects

Previous sections have presented sufficient conditions for a demand-driven system to scale. However, a demand-driven system is only one part of the whole — the hardware, system, and applications must cooperate if scalability is to be achieved. Unfortunately, this means that no matter how well a system is designed, it is possible to construct an application that will saturate the system's resources. The mathematical model of Section 2.2 only considered independent, localized arrivals of requests for service. In practice, however, applications exhibit complex interactions and dependencies in their service requests. The application workload should therefore be considered in the design of a demand-driven system. This section examines how the workload affects the scalability of the system.

Consider a specific resource, k. As the number of processors is increased, we assume the arrival rate of requests for this resource will increase. To service the increasing number of requests, we require that the number of service centers, K_c, increase proportionally, so that the total demand placed on any service center is independent of the arrival rate, λ_c. However, an application program can undermine this strategy in two ways: (i) by creating an imbalance in its requests to the different instances of a particular resource, or (ii) by issuing requests that are not independent, and hence forcing resources to increase coordination as the arrival intensity increases. The effect of (i) is that some centers are idle while others saturate, which defeats the purpose of having multiple instances of a resource. Although the system can sometimes redistribute requests to idle centers, such load balancing does not come without cost, and is typically not effective for rapid, short-term changes in workload demand.

To illustrate (ii), consider a shared resource that is replicated across the processors of the system. Replicating objects is commonly used to improve access times by reducing contention and improving locality. Replicating the object increases the number of service centers because each replica is now available to service requests. However, performance will only improve if one can amortize both the cost of the replication and the cost of maintaining the consistency of the replicated object. The total demand[6] placed upon a service center, k, is thus the product of the percentage of requests which are directed to that replica (*i.e.*, the visit count, v_{1k}), and the time required to access the data (*i.e.*, the service requirement, s_{1k}) plus the product of the percentage of requests directed to center k due to maintaining consistency of the replicas (*i.e.*, v_{2k}), and the amount of time required to perform the consistency operations (*i.e.*, s_{2k}):

$$D_k = D_{1k} + D_{2k} = v_{1k} \cdot s_{1k} + v_{2k} \cdot s_{2k} \tag{6}$$

If the requests do not modify any of the replicas, then there is no cost in maintaining coherence, and D_{2k} of Equation 6 represents the one-time cost of replicating the object. If the requests modify the object, then the system must do work proportional to the p existing replicas to maintain coherence. Since this work is required for each modification, the system resources responsible for maintaining coherence will quickly saturate, and scalability is lost. Note also that if the object were not replicated it would saturate, since D_{1k} increases proportional to p.

For a workload with global writers then, the tradeoff between v and s (as well as their baseline costs) makes comparison between two systems with finite p possible. Balancing one against the other, even for small p, is reasonable. Notice that with a single instance of an object v grows linearly with p, and that when the object is replicated v can be made to grow logarithmically with p (if the auxiliary data structure is structured hierarchically). However, the baseline cost of replication (in this case s_{2k}), may be significantly higher than the other factors. Consequently, in mitigating the eventual saturation of the resource as p is increased the system designer may wish to facilitate a limited version of replication that balances time against frequency of access. This is the approach used in the design of the Hurricane operating system [10].

5 Evaluating Scalability

In principle, the properties that form our scalability criteria are observable quantities. However, their measurement in real systems can be non-trivial. For example, to prove a system is scalable, one must prove that s_{ck} and V_c are asymptotically bounded. For real systems, it is impossible to prove asymptotic independence through measurement alone, since there are only a finite number of processors. However, if a system is shown not to be scalable because of increasing service times, our set of conditions for scalability can yield more information than a simple yes or no answer to the question of scalability. This is because once one bases a system on bounded service times, a system that is found to be unscalable must violate one or more of the conditions for scalability.

Even if no resource saturates for a given workload and fixed p, the conditions for scalability can still be used to explore several interesting factors that affect

[6] Demand is related to utilization as $U_k = X \cdot D_k$.

the behavior of the system. For example, one can explore the rate of change of the system parameters as the size of the system is varied. Alternatively, one can vary the workload parameters for fixed p and examine the effects on the system. Finally, one can fix both the workload and system size, and tune the system for performance. The remainder of this section explores these options in detail.

Varying the Number of Processors For a fixed workload, it is possible to vary the number of processors and observe the effects on the system resources. In particular, if a resource saturates for some p, then the system has been proven unscalable for the given workload. If no resource saturates for a given range of processors, then the rate of increase in utilization, $\Delta U_k/\Delta p$, can be used as a first-order approximation to the range of processors for which the system resources do not saturate. An advantage of this approach is that $\Delta U_k/\Delta p$ can be determined empirically, typically by direct measurement of busy times or queue lengths. Ideally, if $\Delta U_k/\Delta p = 0$ then the utilization of resource k is independent of p. If this is true for all k, then the confidence that the system scales increases for the given workload. If $\Delta U/\Delta p > 0$, then the utilization of resource k is increasing as processors are added to the system and may eventually saturate. The rate of change of utilization, $\Delta U_k/\Delta p$, is not the only metric that can be used to estimate scalability. From the conditions for scalability, any of the metrics $\Delta s_{ck}/\Delta p$, $\Delta K_c/\Delta p$, or $\Delta V_c/\Delta p$ are useful. Ideally, $\Delta s_{ck}/\Delta p$ and $\Delta V_c/\Delta p$ should be 0, and $\Delta K_c/\Delta p$ should be p.

Varying the System The scalability criteria are also useful for comparisons between systems. From the perspective of scalability, the constants that bound service time and interaction need only be independent of the number of processors — no restriction is placed on the magnitude of the constants. However, if two systems are both proven to be scalable for a particular workload and number of processors, then the system with lower bounds will perform better, because its resources are less highly utilized.

To illustrate, consider a multiprocessor operating system targeted for a NUMA shared-memory architecture. At the level of abstraction used to develop the conditions for scalability, increased latency for accessing a remote memory is reflected in the service time, s_{ck}. Now consider two versions of an operating system data structure that are identical except for the way in which the K_c elements of the structure are distributed across the memories of the system. In one version, each element is placed local to the processor that has the highest probability of accessing it; in the other version the elements are distributed uniformly but randomly across the memories of the system. Since the data structures are identical at the programming level, we do not expect saturation for a given workload and system size (unless the interconnection network saturates because of the high number of remote accesses). However, we expect the highly localized version to perform better because the lower average access latencies result in lower average service times.

In practice, many systems will in fact trade scalability for performance. For example, a particular data structure may yield good performance for a small number of processors, but saturate quickly as the size of the system is increased. In a parallel operating system, this would be the case for, say, a single shared queue of processes that are ready to execute. This data structure provides good load-balancing and low overhead for a small number of processors, but will saturate as the number of processors increases if the queue of ready processes must be maintained in priority order. If the target system is small and load-balancing is deemed critical, some designers may trade-off the unscalability of the structure to obtain better performance.

Varying the Workload For a fixed number of processors and a given system configuration, one can vary the workload parameters and again observe the effect on system resources. This approach is useful for determining the range of workloads for which a system does not saturate.

To illustrate, consider a workload that shares a system resource among multiple processes. Clearly, this workload will eventually saturate the resource if the degree of sharing is allowed to increase arbitrarily. However, for a fixed number of processors, the resource may not saturate if the rate of requests for service (ie. the *granularity* of sharing) is low. As an example, a number of processes could share a single memory page of data in a NUMA multiprocessor environment. If the page is shared read-only, then it can be replicated as needed across the memories of the system to reduce access latency and network contention. If the page is subsequently modified, then the system must do work proportional to the degree of replication to maintain consistency (see Section 4). For a small degree of replication and/or a low rate of modifications to the page, the system resources can avoid saturation, but at some sharing granularity, the resources must saturate because the visit count, V_c, increases in proportion to the degree of replication. In effect, experiments that vary the workload are determining the granularity of sharing that can be tolerated by the system.

6 Conclusions

For a demand-driven parallel system to be scalable, all three layers of the system, the hardware, the system software, and the applications issuing requests, must be scalable. In this paper, we have focused on scalability issues at the system software layer. Our motivation for this study stems from the observation that this layer is critically important for scalability, but has typically been addressed only in a very *ad hoc* manner.

We presented an analytical model that characterizes the scalability of the system software layer, based on fundamental metrics of quantitative system performance analysis. In particular, a set of sufficient conditions was developed so that if a system satisfies these conditions then the system will be scalable. Further, it was argued that in practice these conditions are also necessary.

Analytical models typically have three limitations that prevent them from being practically useful. First, they yield no insight into how to build or design a scalable system. Second, they do not take into account the complex interactions exhibited by the workload of real applications. Finally, they are expressed in terms of asymptotic limits that make it difficult to evaluate the scalability of existing systems with only a limited number of processors.

We have addressed each of these limitations. First, the necessary and sufficient conditions we developed were translated into a specific set of practical design guidelines. Second, we explored how the application layer can impact the scalability of demand-driven systems. Finally, a number of techniques were presented that can be used to examine the scalability behavior of a system with only a limited number of processors.

References

1. Vadim Abrossimov, Marc Rozier, and Marc Shapiro. "Generic Virtual Memory Management for Operating System Kernels". In *Proc. 12th ACM Symposium on Operating System Principles*, pages 123–136, Litchfield Park, Arizona, Dec. 1989.
2. Ramesh Balan and Kurt Gollhardt. "A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machines". In *Summer '92 USENIX*, pages 107–115, San Antonio, TX, June 1992.
3. Amnon Barak and Yoram Kornatzky. Design principles of operating systems for large scale multicomputers. Computer Science RC 13220 (#59114), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, Oct. 1987.
4. Amnon Barak and On G. Paradise. "MOS - Scaling up UNIX". In *Proc. USENIX Conference*, pages 414–418, December 1986.
5. John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. "Scale and Performance in a Distributed File System". *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
6. David V. James, Anthony T Laudrie, Stein Gjessing, and Gurindar S. Sohi. "Distributed-Directory Scheme: Scalable Coherent Interface". *Computer*, 23(6):74–77, June 1990.
7. Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1984.
8. Daniel Nussbaum and Anant Agarwal. "Scalability of Parallel Machines". *Communications of the ACM*, 34(3):56–61, March 1991.
9. Mahadev Satyanarayanan. "Scalable, Secure, and Highly Available Distributed File Access". *Computer*, 23(5):9–21, May 1990.
10. Ron Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. "Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design". *Journal of Supercomputing*, To appear 1995.

This article was processed using the LaTeX macro package with LLNCS style