Coalesced QoS: A Pragmatic approach to a unified model for KLOS

Ashish Chawla
Sr. Software Consultant, Email: chawlaashish@acm.org

Ramesh Yerraballi[1], Amit Vasudevan
Department of Computer Science and Engineering, The University of Texas at Arlington
Arlington, TX 76019
Email: {ramesh,vasudeva}@cse.uta.edu

Abstract— **Advances in software and hardware technologies have given operating systems the ability to process data and handle various concurrent processes. The increased ability has been one of the driving forces which have led to the proliferation of mechanisms in operating systems to satisfy the performance requirements of applications with predictable resource allocation. As different classes of applications require different resource management policies one needs to look into ways to satisfy all classes of applications. Conventional general purpose operating systems have been developed for a single class of best-effort applications, hence, are inadequate to support multiple classes of applications. We present an abstract architecture for the support of Quality of Service (QoS) in Kernel-Less Operating System (KLOS). We propose new semantics for the QoS resource management paradigm, based on the notion of Quality of Service. By virtue of this new semantics, it is possible to provide the support required by KLOS to various components of the operating system such as memory manager, processor time, IO management etc. These mechanisms which are required within an operating system to support this paradigm are described, and the design and implementation of a prototypical kernel which implements them is presented. Various notions of negotiation rules between the application and the operating systems are discussed along-with a feature which allows the user to express its requirements and the fact is that this model assures the user in providing the selected parameters and returns feedback about the way it meets those requirements. This QoS model presents a design paradigm that allows the internal components to be rearranged dynamically, adapting the architecture to the high performance of KLOS.**
The benefits of our framework are demonstrated by building a simulation model to represent how the various modules of an operating system and the interface between the processes and the operating system can be tailored to provide Quality of Service guarantees.

Index Terms—**Quality of Service, Operating Systems, Resources, Utilization, High Performance, Kernel-Less Architecture.**

[1] Corresponding Author

I. Introduction

The rapid growth and coexistence of different application domains, such as multimedia and embedded systems, present a significant challenge to the provision of their Quality of Service (QoS). To solve this challenge, we need a unified QoS framework, which allows flexibility and re-configurability.

Techniques such as over-provisioning of resources work if there is a single application, but cannot easily be extended to realistic scenarios where we have multiple applications competing for the same resources. Therefore, the operating system must allow an application to request some QoS, which is a measure of the resources required by the application. In the following paper, we present an approach which is targeted towards providing quality of service for applications in terms of overall performance which includes both resource (Memory, I/O and Network) usage and timing.

The point to note is that the high performance applications have different QoS requirements and this poses some demands on the resources, like scheduling policies need to emphasize on meeting deadlines, memory management needs to ensure pages are pinned down with bounded access etc. Hence, an operating system which is meant to provide QoS support must have the entire system and each component to be aware of the QoS policies. We thus come up with a Coalesced QoS model which is a unification of the entire resources available in the system and made available to the applications.

KLOS is a Kernel-Less Operating System [1, 2] wherein all processes execute at the unprivileged level having their own private execution space and function independent of all other processes in the system. KLOS is able to achieve high performance due to its architecture and its service call mechanism. In traditional operating systems, a service call in almost all cases involves a ring-transition to the kernel where the required function is then performed. Traditionally this is achieved by using the trap/interrupt which transitions into the kernel to execute the required function. However, in case of KLOS, there is no ring-transition during service calls and this accounts for the High Performance attributed to KLOS. It is this feature of KLOS which makes us work on providing Quality of Service to further enhance the performance of the operating system.

Our resource management scheme builds on and significantly extends the established resource allocation model described in [7]. The precise resource requirements of the applications, as well as the number and timing constraints are unknown in advance. These details depend on many factors known only at runtime. The driving principle behind Coalesced QoS is to give applications enough control so as to specify the resources it requires, with the operating system accommodating the request through allocation and negotiation. Till date, research carried out in this domain has not dealt with the management of multiple resource types, concurrent access to different resources, explicit timeliness control, feedback about resource usage, control on resource overruns and management of interactions between resource users and considerations of portability and compatibility.

In our work, we therefore strive for practical and acceptable alternatives which can guarantee access to different resource types. We investigate various resource allocation techniques to provide a unified approach of providing Quality of Service (QoS) to applications. Our methodology takes advantage of the fact the quality of service being provided is not just limited to the processor time but also, all other system components do play a role in the negotiation. We propose integrated architecture for run-time application support, and runtime adaptation to application state variations. Our resource model captures the essential requirements of the applications in a simple, efficient yet general form. Our resource management work significantly extends established real-time scheduling theory, and reservation enforcement mechanisms.

Our approach is in developing a semantically rich model supporting the high-level design/representation of QoS adaptation strategies and concerns. We are designing and implementing mappings to derive runtime resource requirements and for incorporating the information gathered at runtime into refinement of the Quality of Service provided by the operating system.

II. Related Work

One of the fundamental problems with doing resource allocation in operating systems is that decisions as to which task should receive a given resource. These decisions are based on a measure (e.g. priority) which does not permit control over the actual quantity of a resource to be allocated over a given time period, particularly when this time period is small.

Managing Quality of Service (QoS) in an operating system can be done in a number of ways. At one extreme, one can make hard real time guarantees to applications, refusing them to run if the hard real time guarantees cannot be met at any stage. At the other extreme, one can provide a model which takes into account the fact that the resource reservation decisions are not made only at the beginning of the task but can be recurring to take fluctuations of resource availability and demand into account.

In general, the approach is to provide probabilistic guarantees and to expect applications to monitor their performance and adapt their behavior when resource allocation changes.

A. Environment

The growing popularity of providing QoS in operating systems to support multimedia applications has resulted in several research efforts that have focused on the design of predictable resource allocation mechanisms. Consequently, in the recent past many projects/techniques have attacked the problem of providing quality of service in operating systems in a number of ways [14]. Early work on reservation-based real-time systems focused on the CPU as the resource to be shared. But to make the design of the system more complete, a sufficient model should not only cover CPU but all resources of interest, such as disk bandwidth or network. Hence, systems such as “Resource Kernels” have extended the initial work. They use deterministic reservation times, which lead to over allocation of resources since the worst case must also be covered. The common approach adopted by all of these focused on CPU scheduling mechanisms.

Most of the systems described are intended for more coarse-grained resource allocations. The Q-RAM architecture [7] follows a centralized approach where tasks specify their resource requirements to a central entity responsible for resource management.

In the Eclipse OS [6, 8, 9], applications obtained a desired quality of service by initially acquiring a resource reservation for each required physical resource by incorporating a potential share scheduler for each resource.

Further research resulted in the prediction of response times of the applications before they are executed in order to meet the QoS requirement in the metric of timeliness [5].

Several extensions have been made in existing workstation operating systems, using QoS based Resource Management Model [4], to assist the execution of multimedia systems wherein applications arrive with a request for a certain amount of resources which is provided to the system via the resource demand function and describes the different QoS settings of the application.

Significant information regarding implementing QoS in operating systems could be gathered from the Synthesis Operating System [3] where the kernel combines several techniques to provide high performance, including kernel code synthesis, fine grain scheduling and optimistic synchronization.

Instead of priorities, Synthesis uses fine grain scheduling which assigns larger or smaller quanta to threads based on a “need to execute” criterion. Effective CPU time received by a thread is the determined by the quantum assigned to that thread divided by the sum of quanta assigned to all threads.

Furthermore, guarantees need to be dynamically renegotiated [7] to allow the users to agree upon QoS parameters that satisfy service user(s) and service provider.

The connection request is rejected if no common agreement upon QoS is found. The service provider or called service user should provide the reason for rejection and perhaps additional information to the calling service user so that the request can be modified and resubmitted (with a higher probability of success) or cancelled. Furthermore, a QoS contract might be changed (i.e., renegotiated) during a sessions lifetime.

Providing QoS through Reservation Domains makes use of hierarchical, proportional-share dynamically reconfigurable resource schedulers at the device driver level for the management of disk and network bandwidth and for CPU scheduling.

Reservation domains enable explicit control over the provisioning of system resources among applications in order to achieve the desired levels of predictable performance. In general each reservation domain is assigned a certain fraction of each resource. The basic idea behind reservation domains is to isolate the performance of reservation domains from each other. In particular, time-sensitive applications can coexist with batch processes on the same system. The importance of cumulative service guarantee is discussed which complements other QoS parameters like delay and fairness. Informally, guaranteed cumulative service means that the scheduling delays encountered by a process on various resources do no accumulate over the lifetime of the process.

Coalesced QoS addresses the shortcomings of the above mentioned models by accounting for other modules of the operating system also.

Jeffay et al. [15] employ hard real-time scheduling theory in a specialized micro-kernel to address timing issues related to different stages of live video and audio processing. Robin et al. [17] designed a system based on Chorus [16] micro-kernel that addresses both the network and end host QoS control.

B. Quality of Service in Operating Systems

The Quality of Service (QoS) refers to the level of commitment provided by the service and the integrity of that commitment.

Quality of Service is a general term for an abstraction covering aspects of the non-functional behavior of a system. In particular, it includes not only the specification of non-functional service characteristics, but also necessary data models, operational constraints, and information about data measurement, monitoring and maintenance.

QoS represents a user's expectations for the performance of an application. It can also be defined as the collective measure of the level of service as requested by the user. It has been realized that in order to guarantee any QoS, applications must receive predictable resource allocation from the operating system. Since QoS can change during the execution of the application, the resource allocation from the operating system should also provide flexibility to accommodate the dynamic QoS requirements. The effort of utilizing available resources to satisfy QoS requirements can be largely classified into two types: resource reservation provided by the operating systems, and adaptation from the applications to accommodate resource availability.

In the context of operating systems, on a typical system, multiple applications may contend for the same physical resources such as CPU, memory and disk or network bandwidth. An important goal of an operating system is therefore to schedule requests from different applications so that each application and the system as whole perform well. As an example, let us consider real-time applications, which must have their requests processed within certain performance bounds. To support real time applications correctly under arbitrary system load, the operating system must perform admission control and offer quality of service (QoS) guarantees. Many multimedia applications have timing requirements and other quality of service (QoS) parameters that represent the user's desires and expectations for the performance of the applications. The complexity of providing for these timing requirements at the system level is exacerbated by the fact that the user may change those timing requirements at any time during the execution of the applications; and of course the user may create and terminate multimedia applications at any time.

In dealing with QoS management it is important to realize that there are different types of QoS parameters for different levels of the system. Applications must interact with the user in terms of user-level QoS parameters. Once the user-level QoS parameters are determined, they have to be mapped into system-level QoS parameters that would be meaningful for system-level resource management mechanisms. These system-level QoS parameters would describe how much time is needed on various resources. They depend on the user-level QoS parameters and on detailed computations that the operating system performs on data elements.

To allow the user to specify the QoS parameters desired at the highest level, the application must be able to map from user-level QoS parameters to system-level QoS parameters. The system-level parameters are required for the application to be able to ask for the resources it will need to execute. If the resources are unavailable, the system-level resource management mechanism should be able to communicate the fact that those parameters cannot be guaranteed. It should then initiate a negotiation to arrive at a set of system-level parameters that can be supported by the system. The inverse mapping to user-level QoS parameters should yield a QoS specification that can be tolerated by the user. Thus, the inverse mapping from system-level to user-level QoS parameters is just as important as the forward mapping.

A **resource reservation approach** must allocate the reserve for its computation on behalf of an incoming application request; it must know what the reservation parameters should be. This approach requires the application and OS to enter into a dialogue to allow the application to explicitly request OS specific QoS level, meaning a certain pattern of OS operations to be called with certain timing constraints. The operating system must then map the requested QoS requirements to system resource requirements and decide whether it can acquire the reserves to support that activity. Further, the OS must have the machinery to map those QoS requirements to system resource requirements.

The operating system could store in a persistent preferences database some information about reservation levels used in previous instantiations of the applications. This information would be a good guess as to what reservation levels should be in case the incoming application is memory intensive or IO intensive, and it might be possible to maintain a small database to map prior experience with different QoS parameters to reservation parameters. This approach might get much more complicated as more QoS parameters, reservation parameters, and target system architectures are used. The initial reservation level could be set to zero or some other relatively small value that is known to be smaller than the actual reservation level. This approach requires the mechanisms for reservation level adaptation to quickly acquire the feedback on usage that is necessary to set a reasonable reservation level where desired quality of service parameters can be achieved. Hence, the operating system should support the process of self calibration. Namely the appropriate requirement is calculated from on-the-fly feedback of application performance, and is calibrated when the performance does not meet the QoS requirement. A system that adapts transparently to available platform resources should employ an adaptive QoS-mapping function.

Earlier in this paper, we effectively stated that QoS provides the ability to handle application traffic such that it meets the service needs of certain applications. We recall that, if operating resources were infinite, the service needs of all applications would be trivially met. **It follows that QoS is interesting to us because it enables us to meet the service needs of certain applications when resources are finite.**

A QoS enabled operating system should provide service guarantees appropriate for various application types while making efficient use of available resources. A performance-assured operating system must be able to provide each client or class of clients their desired QoS irrespective of the behavior of other clients. This implies protection of well-behaving clients from those violating their QoS specification.

III. Kernel-Less Operating System (KLOS)

A. Introduction

The purpose of operating system is to multiplex shared resources between applications. Operating Systems provide services that are accessed by processes via mechanisms that involve a ring-transition to transfer control to the kernel where the required function is performed. This has one significant drawback that every service call involves an overhead of a context-switch where processor state is saved and a protection domain transfer is performed. However, on processor architectures that support segmentation, it is possible to achieve a significant performance gain in accessing the services provided by the operating system by not performing a ring transition. KLOS is a Kernel-Less Operating System, acting as a proof-of-concept vehicle, wherein all processes execute at the unprivileged level having their own private execution space and function independent of all other processes in the system.

Operating systems define the interface between applications and physical resources. Unfortunately, this interface can significantly limit the performance and implementation freedom of applications.

Most, if not all production level operating systems have a dual mode of operation - (1) the privileged level where the kernel resides and, (2) the unprivileged level where application and system processes execute. A ring transition mechanism is used to move from one level to the other. The idea behind this separation has always been protection and stability. However, on processor architectures that support segmentation, it is possible to achieve a significant performance gain by eliminating the ring transition. Further, such gains can be achieved without compromising protection. This is made possible by the use of a subtle trick involving segmentation and Task State Segments (TSS).

To this end, we have designed new operating system architecture, in which –

- There is no kernel as perceived in current operating systems,
- Operating system services are accessed without a ring transition,
- All processes and the operating system execute at the unprivileged level and,
- Each process has its own private address space and virtual memory mappings and functions independent of all other processes in the system.

KLOS, a Kernel-Less Operating System is a realization of this design. Regular runaway processes do not pose a threat to the stability of the operating system built on this design. Processes that are specifically engineered to thwart the stability of the system are contained with a very high probability of success [1, 2]. KLOS is able to achieve high performance due to its architecture and its service call mechanism.

B. Design of KLOS

Traditional operating systems comprise of a kernel, that is responsible for providing the core services of the operating system and a shell or applications that reside on top of the kernel using its services and providing an interface to the user. In KLOS, there is no kernel per se. Instead, the entire operating system is made available to each application as a part of its execution space, running at the same privilege level.

The design of KLOS relies on the segmentation capability of the x86-based processors but is different in its approach with use of the TSS, doing away with the down-call to the kernel. The authors [1, 2] method is dynamic, which means there is no code pre-scanning or other procedures that need to be applied on application code before it is executed.

The architecture, at its heart consists of an event core that is responsible for acting upon external events (hardware interrupts and processor generated exceptions).

Events are the only means of vertical up-calls to the unprivileged level. There are no down-calls to the privileged level eliminating a protection domain transfer during normal program flow. All the components of a typical operating system like the memory manager, scheduler, process manager, device drivers etc. run in the unprivileged level and have a horizontal mode of interaction.

KLOS is able to achieve high performance due to its architecture and its service call mechanism. The service call mechanism of KLOS results in a general 4x improvement over the traditional trap (interrupt) and a 2x improvement over the Intel SYSENTER/SYSEXIT fast system call models.

IV. Coalesced QoS Framework

A. Design

One of the fundamental problems with conventional operating systems is the decision as to which task should receive a given resource. These decisions are based on a measure (e.g. priority), which does not permit control over actual quantity of resource to be allocated over a given time period, particularly when this time period is small.

By interacting with operating system kernels and application-level hooks, QoS aware operating system has been proved highly effective in supporting high performance applications via run-time probing, and adaptation of applications. Therefore, application configurations and performances can be tailored to different user behavior and characteristics of ubiquitous environments. To be more specific, the QoS aware framework is expected to provide the following critical functions:

Run-time probing. Applications are subject to run-time probing with respect to the instantaneous performances of their QoS parameters. Such QoS probing is the responsibility of operating system. Probing is useful in learning about application behavior, so that better run-time adaptation rules may be set accordingly.

Run-time adaptation. During application run-time, the operating system may assist the application to adapt to the changes of resource availability or user requirements. Such behavioral changes exist due to the sharing of resources among applications; or lack of resource reservation mechanisms. The operating system changes the application behavior correspondingly when significant variations in these triggering sources are detected.

Our goal is to find adaptation strategies that could be applied on-line during application runtime. We claim that solving sophisticated optimization problems will generate too much overhead and delay. Therefore, we have aimed at rather simple, but efficient heuristics. Therefore, this is the preferred approach for a dynamic "on the fly" adaptation of applications during runtime.

A key problem with the single resource scheduling algorithms is that they are not designed to make use of the information in the additional resource requirements. Our contribution is to come up with a new heuristic for scheduling, wherein our algorithms are *resource-aware*, meaning that they are able to use the additional resource information in the system to intelligently adjust the priority of the execution among the waiting jobs.

Our architecture is based on the notion of feedback so that the period assigned to threads change dynamically as the resource requirements of the processes changes.
Ideally resource allocation should ensure that every process maintains a sufficient rate of progress towards completing its tasks. The progress is determined by how much time does processes take to complete and this progress is relative to its priority in the system. The priority of a process is calculated based on the relative use of all the resources in the system and how much is each process utilizing a particular resource. This feedback mechanism periodically monitors the resource usage of each process and automatically calibrates its priority based on the current usage of the resources.

In our proposed architecture, we use the resource allocation information to come up with a scheduling heuristics so that processes can be sequenced in the queue as per their priority. The priority of the processes is based on a heuristic which takes into account the number of resources the process is using as well as the quantity of the resources. This is done by profiling the system calls for each process.

We modify the system call in KLOS, wherein it is built with QoS Support to include the resource requirements for each process so that we can profile the resource requirement of a process and calculate the priority based on this information.

Run time profiling of the processes is done in the system so that we can raise the priority for processes as time progresses. This is done on the basis of the age of a process and the relative use of all resources in the system by a particular process. Notion is being usable in practice and a process should be able to shed light itself at runtime regarding the resources it will be using.

B. Stochastic Scheduling

In order to satisfy our needs for scheduling policies that can leverage the performance variability of resources, we come up with a scheduling policy. We describe a methodology that allows our scheduling policy to take advantage of the information to improve application execution performance.

This work on resource-aware scheduling policy is being conducted in a larger context of real-time resource reservations in KLOS as to provide reasonably high Quality of Service. The resource allocation policies are controlled by the scheduler to meet the timeliness requirements, tailoring their behavior in response to the actual resources available. As opposed to systems, where all resource requirements are known well in advance, we have a model where the resource requirements are calculated at run-time and our heuristics support our claim for high performance in such an environment.

TABLE I
SUMMARY FOR EQUAL USAGE OF RESOURCES

Observed Parameters	Process Intensity = 4	Process Intensity = 7	Process Intensity = 10
CPU Utilization	Coalesced QoS (1.05)	Coalesced QoS (1.10)	Coalesced QoS (1.10)
Memory Utilization	Coalesced QoS (1.05)	Coalesced QoS (1.04)	Coalesced QoS (1.10)
System Throughput	OSP (1.03)	Coalesced QoS (1.10)	X
Avg. Waiting Time	X	X	Coalesced QoS (1.12)

Process Intensity refers to the Process Creation Intensity. The table tells us which model performs better and the value in bracket is an indicative of how much the performance is increased. An 'X' means that both the models perform equally.

Since our approach is primarily based on real-time scheduling of processes in the presence of multiple resource requirements, it needs to be addressed in a comprehensive manner. Successfully applied scheduling techniques have been based on preemptive fixed priority scheduling in most real-time systems developments since Liu and Layland introduced it in their paper [25]. However, dynamic priority schedulers can achieve higher schedulability than fixed ones, and non-preemptive schedulers incur less run-time overhead.

The fundamental desirable characteristic of a priority scheduler is that the system should always be executing at the highest priority. However, it is not always possible for a system to conform to this requirement. When circumstances within the system force a higher priority task to wait for a lower priority task, **priority inversion** occurs. In our framework, we have priority assigned to each process and this might lead to an uncontrolled priority inversion problem. It is important to note that every operating system experiences incidents of priority inversion; the issue is not the presence of priority inversion but rather the time duration of each source of priority inversion. The magnitude of priority inversion is thus very important in assessing whether an operating system will be suitable for a particular application or system.

We thus know that such problems occur because priorities alone are not expressive enough to capture all the relationships between the resources. Our approach here has an advantage here that the priority is calculated by taking into account the resources a process requires as well as the age of a process and this approach has some clear advantages which can be seen from the results.

V. PERFORMANCE EVALUATION

In this section, the simulation model designed to evaluate the performance of our resource allocation scheme is described. In order to compare the performance of our algorithm, we simulate a coalesced resource allocation scheme.

In the functional description of a technology, one can describe the different ways of performance, and measure one of the technology performance parameter (application, capability, performance, quality, and safety).

F. Zwicky (Zwicky, 1948) proposed that one can systematically explore technology sources for technical advance in a system by logically constructing all possible combinations of physical alternatives of the system. Thus our performance evaluation technique is based on this where we logically construct combinations of high and low resource utilization with low, medium or high process creation intensity to study the system behavior.

A. Performance Metrics

Once we have built a mathematical model, it must then be examined to see how it can be used to answer the questions of interest about the system it is supposed to represent. If the model is simple enough, it may be possible to work with its relationships and quantities to get an exact, *analytical* solution. If an analytical solution to a mathematical model is available and is computationally efficient, it is desirable to study the model via simulation. However, many systems are highly complex, so that valid mathematical models of them are themselves complex, precluding any possibility of an analytical solution. In this case, the model must be studied by means of *simulation*, i.e., numerically exercising the model for the inputs in question to see how they affect the output measures of performance.

We use simulation to analyze the performance of our proposed resource utilization scheme. Some of the main features in our simulation are:

- The call arrival process is a Poisson Process with rate lambda (λ). The interarrival times are exponentially distributed with mean interarrival time $1/\lambda$.
- The expected call duration times are exponentially distributed with a mean duration time.

Our algorithm was simulated on a suite of synthetic workload. This allowed us to vary different characteristics of the workload with respect to the resources. While the general shape of this distribution is exponential, the model captures all the characteristics as seen in a production environment.

B. Results and Analysis

In this section, we describe how the components described in the preceding sections can be implemented on a realistic platform. The experimental test-bed and implementation environment is based on OSP [26], an Operating System Project. OSP is a simulated system that provides the illusion of a computer system with a dynamically evolving collection of user processes to be multi-programmed. When simulation commences, OSP uses *simulation parameters* to decide whether the system should be I/O-bound or cpu-bound, how often to generate requests for various system resources, how long the simulation should last, etc.

The OSP simulator accumulates the statistics that reflect the amount of resources consumed by the various modules. The most significant are the following performance indicators:

- CPU Utilization

- Memory Utilization
- System Throughput
- Average Waiting Time per process

Output statistics generated by OSP are used for the performance measures in this simulation. For example, by altering the length of the cpu time quantum, one can see the effect on the overall system throughput and other statistics. Likewise, by varying the degree of prepaging, we can observe the effect of prepaging on the number of page faults, on the total number of pages swapped in or out, and on the average turnaround time.

C. Performance Graphs

As observed from the knee plots in Fig 1 through Fig 8, for the distribution of the various resources, we can say with 95% confidence that the Coalesced QoS based model performs better than a non-QoS based model. The graphs are a clear indication, that the system starts to take advantage of the Coalesced QoS based model when the resource usage in the system is high and also when all the resources are being equally consumed by the processes.

Table I presents the results of the comparison between the performances of a Coalesced QoS based system and a non-QoS based system when the resources are equally distributed in the system.

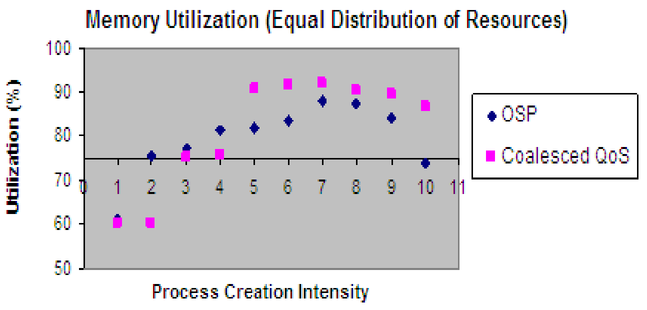

Fig. 1. Knee Plot for Memory Utilization with Equal Distribution of Resources

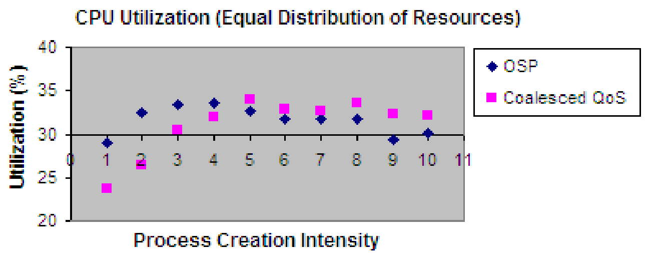

Fig. 2. Knee Plot for CPU Utilization with Equal Distribution of Resources

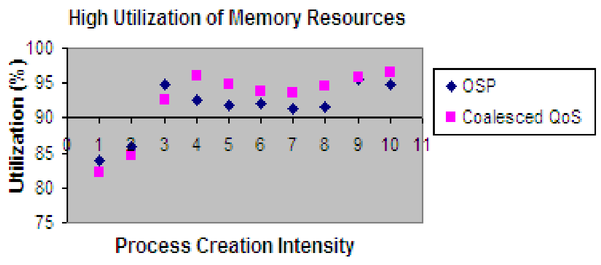

Fig. 3. Knee Plot for High Utilization of Memory Resource

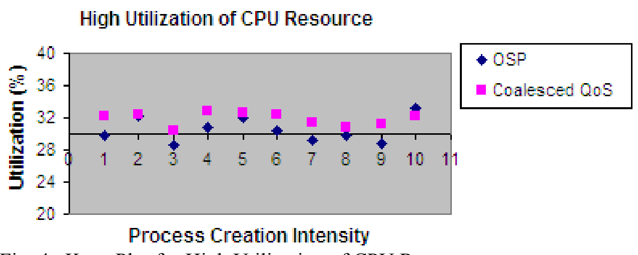

Fig. 4. Knee Plot for High Utilization of CPU Resource

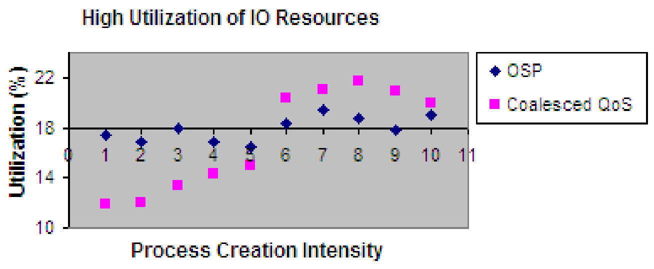

Fig. 5. Knee Plot for High Utilization of IO Resource

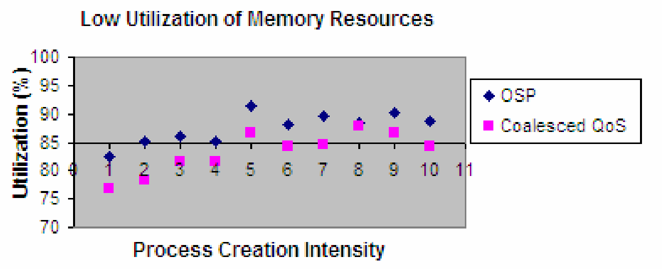

Fig. 6. Knee Plot for Low Utilization of Memory Resource

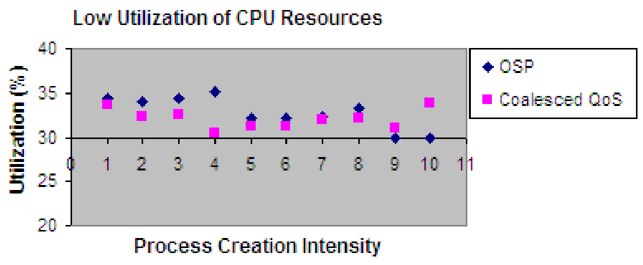

Fig. 7. Knee Plot for Low Utilization of CPU Resource

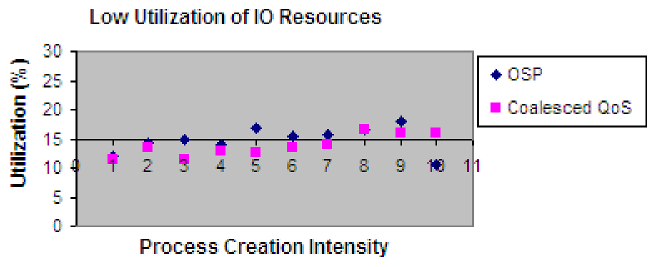

Fig. 8. Knee Plot for Low Utilization of IO Resource

After observing the performance from the graphs, we performed the paired t-test to confirm the authenticity of the above results which is summarized in Table I.

The paired t-test compares two paired groups so one can make inferences about the size of the average treatment effect (average difference between the paired measurements). The most important results are the P value and the confidence interval.

VI. Conclusion

In this paper, we are concerned with the requirements placed upon high performance operating systems, which are usually, but not necessarily, general-purpose operating systems with soft real-time extensions.

We have attempted to identify an uncontroversial set of requirements that the “ideal” high performance operating system would meet. Although it is unlikely that any single system or scheduling policy will be able to meet all of these requirements for all types of applications, the requirements are important because they describe the space within which high performance systems are designed. A particular set of prioritizations among the requirements will result in a specific set of tradeoffs, and these tradeoffs will constrain the design of the user interface and the application programming model of a system.

Particularly, the importance of all the resources is not reflected in current QoS approaches and we have made an effort to take into account all the resources that a process uses in order to calculate the priority for the efficient execution of applications.

Currently our work has some limitations that are being addressed. Our performance results are based on a simulation model, and hence are preliminary in many ways. We looked into what Kurtzweil (1999) calls the ‘knee of the curve’ in regard to the resource utilization in the system. We also need to look at the performance measures when this is implemented in the working model of KLOS and that should give us some realistic values as to where our idea stands. Currently, we have used a simulator for our performance evaluation, but it gives us a limited view of the process satisfaction and we need to come up with metrics which would give us this statistics to study process behavior.

References

[1] Vasudevan, A., Yerraballi, R. and Chawla, A. (2005). A High Performance Kernel-Less Operating System Architecture. In Proc. Twenty-Eighth Australasian Computer Science Conference (ACSC2005), Newcastle, Australia. CRPIT, 38. Estivill-Castro, V., Ed. ACS. 287-296.

[2] Amit Vasudevan, Ramesh Yerraballi, Ashish Chawla, “KLOS: High Performance Kernel Less Operating System”, Accepted for publication in the 24th IEEE Real-Time Systems Symposium, WIP Section, Cancun, 2003

[3] Henry Massalin, Calton Pu: “Threads and Input/Output in the Synthesis Kernel”, Proceedings of the twelfth ACM symposium on Operating systems principles, 1989

[4] Wonjun Lee, Jaideep Srivastava: “A Market Based Resource Management and QoS Support Framework for Distributed Multimedia Systems”, ACM conference on Information and Knowledge Management, USA, 2000

[5] E. Huh, L. R. Welch, B. A. Shirazi, B. Tjaden, and C. D. Cavanaugh, “Accommodating QoS Prediction in an Adaptive Resource Management Framework”, IPDPS, Mexico, 2000.

[6] J. Blanquer, J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz: Resource Management for QoS in Eclipse/BSD

[7] Thomas Plagemann, Knut A. Saethre and Vera Goebel: Application Requirements and QoS Negotiation in Multimedia Systems, Second Workshop on Protocols for Multimedia Systems, PROMS’95, Salzburg Austria, October 1995

[8] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz: The Eclipse Operating System: Providing Quality of Service via Reservation Domains.

[9] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz: Move-To-Rear List Scheduling: A New Scheduling Algorithm for Providing QoS Guarantees, ACM Multimedia 97 – Electronic Proceedings

[10] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek: A Resource Allocation Model for QoS Management. In Proc. of the 18th IEEE Real-Time Systems Symposium, San Francisco, USA, December 1997.

[11] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer., SPIN - an extensible microkernel for application specific operating system services. 1994 European SIGOPS Workshop, 1994.

[12] J. Blazewicz, W. Cellary, R. Slowinski and J. Weglarz. Scheduling under Resource Constraints – Deterministic Models. In Annals of Operations Research, Volume 7. Baltzer Science Publishers, 1986

[13] C. W. Mercer, S. Savage and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In Proceedings of the IEEE International Conference on Multimedia Computing and Systems. May 1994

[14] Raj Rajkumar, K. Juvva, A. Molano and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998.

[15] K. Jeffay, D. L. Stone, and F. D. Smith. “Kernel Support for Live Digital Audio and Video”. (Nov 1991), 10-21.

[16] Bricker, M. Gien, M. Guilllemont, J. Lipskis, D. Orr and M. Rozier. "Architectural Issues in Microkernel-base Operating Systems: the CHORUS Experience". Computer Communication 14, 6 (July 1991), 347-357.

[17] P. Robin, G. Coulson, A. Campbell, G. Blair and M. Papathomas. Implementing a QoS Controlled ATM Based Communications System in Chorus. Technical Report MPG-94-05, Dept. of Computing, Lancaster University, March, 1994.

[18] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In Proceedings of the 15th ACM Symposium on Operating System Principles, pages 251 - 266, 1995.

[19] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. The performance of μKernel-based systems. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 66 - 77, 1997.

[20] K. M. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: a small-memory real-time microkernel. In Proceedings of the 17th ACM Symposium on Operating System Principles, pages 277–291, Kiawah Island, SC, 1999.

[21] T. Shinagawa, K. Kono, T. Masuda. Fine-grained Protection Domain based on Segmentation Mechanism. Japan Society for Software Science and Technology, 2000.

[22] Tao Li, Lizy Kurian John, Anand Sivasubramaniam, N. Vijaykrishnan, Juan Rubio. Understanding and improving operating system effects in control flow prediction. In Proceedings of the 10th international conference on architectural support for programming languages and operating systems, pages 68-80, San Jose, CA, 2002.

[23] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, Daniel C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In Proceedings of the nineteenth ACM symposium on Operating systems principles, pages 15–28, Bolton Landing, NY, 2003.

[24] David Lie, Chandramohan A. Thekkath, Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In Proceedings of the nineteenth ACM symposium on Operating systems principles, pages 178-192, Bolton Landing, NY, 2003.

[25] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of the ACM, V20, N1, 1973, pp. 46-61.

[26] Michael Kifer, Scott A. Smolka: OSP: An Environment for Operating System Projects. *Operating Systems Review 26(4): 98-100 (1992)*