Memory Management for Quality of Service Provisioning

Michael J. Katchabaw, Hanan L. Lutfiyya, and Michael A. Bauer
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A 5B7

Abstract

Applications used in today's computing environments are becoming more complicated and are being used in more mission critical roles in the enterprise. At the same time, users' demands for performance, reliability, and availability are increasing rapidly. To meet these needs, a high level of quality of service must be delivered to the user. This, in turn, entails effective management of the scarce supply of computing resources that are available.

While great strides have been made towards resource management for quality of service provisioning, memory is one resource that has been consistently left out of the picture. This is unfortunate, since applications that have unlimited supplies of CPU cycles, network bandwidth, and other such resources, will still fail to meet quality of service expectations if memory is not available to contain the necessary application code and data.

In this paper, we present an approach to memory management for quality of service provisioning to address this problem. We present our general strategy, the design of a solution realizing this approach, and a prototype implementation based on this architecture. We describe experimentation and experience to date, and evaluate our work and its effectiveness based on these preliminary results. Finally, we conclude with a summary of our work and outline our plans to evolve it in the future.

[**Keywords:** Quality of service provisioning, memory management, resource management]

1 Introduction

To meet increasing user needs for performance, reliability, and availability in mission critical applications, a high level of quality of service must be delivered. Given an infinite supply of all computing resources (including CPU cycles, memory, disk bandwidth, network bandwidth, and so on), we would be able to satisfy the quality expectations of all users on all applications. Since this will never be the case, providing users with expected quality of service entails effective management of the limited supply of resources that are actually available.

To that end, as noted in [3], considerable research effort has been expended towards CPU and network resource scheduling and allocation to support quality of service provisioning, producing some stunning and effective results. At the same time, unfortunately, memory management for quality of service provisioning has not received much serious consideration.

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

In order for processes to make effective use of their CPU, network, and other resources, they must have a sufficient and reliable supply of memory available for their code and data. Otherwise, memory shortages result and heavy page faulting can occur, preventing quality of service expectations from being met.

Applications that are time sensitive, for instance, can suffer greatly under memory shortages and page faulting. The penalty incurred for a page fault is in the range of 700,000–6,000,000 clock cycles (on the order of 10 milliseconds) [21]. For time sensitive applications, a few page faults could cause a noticeable degradation in performance. Distributed multimedia programs, such as video conferencing, video-on-demand, and so on, are good examples of these applications. If a video application was expected to deliver video at a rate of 30 frames per second (or 33 ms per frame), quality of service expectations could be violated when only a few page faults are encountered [3].

Furthermore, data intensive applications, such as databases, web servers, and electronic commerce applications, can also be susceptible to problems caused by memory shortages or heavy page faulting, as buffers, caches, and primary storage space become scarce and access becomes unpredictable. The authors in [20], for example, identify memory issues (in particular, buffer pools), to be a leading factor in the performance of a database system. Consequently, memory management is a critical component to resource management supporting quality of service provisioning.

For a quality of service memory management solution to be successful, several requirements need to be met. The solution must:

- Be able to minimize page faulting in processes that need reliable access to certain memory regions; this must be done in an efficient manner with low overhead so that as many resources as possible are available to applications that need them.
- Manage memory in a dynamic fashion on a process by process basis – each process can have different quality of service expectations (or none at all), and these expectations can change over time. This results in changes in memory needs, which must be addressed by the memory management system. The memory management process itself should also be dynamic, allowing its behaviour to adapt over time to changing environment conditions and administrator and user needs.
- Be implementable on currently available commercial operating systems and should ideally not require kernel modifications. As a result, the approach needs to be generic and support a variety of underlying memory management strategies.
- Integrate well with support for managing other resources such as CPU cycles, network bandwidth, and so on, to provide a comprehensive resource management solution.
- Be transparent to both users and programmers such that neither need to know or specify memory needs directly – the solution should do this for them. This makes quality of service enabled applications easier to use, and facilitates their development.

Our current work takes steps towards addressing the above problems. It focusses on the development of an approach to memory management to support quality of service provisioning. Users specify quality of service expectations for their applications in terms of

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

application-level metrics, as opposed to low-level memory needs. In response to application behaviour and requests, the underlying memory management system adjusts resources given to individual applications until the users' quality of service expectations are met. The system is dynamic in that it can adapt to meet changing demands and user needs, while at the same time minimizing unnecessary memory allocations. The approach is generic in that it supports the management of a variety of resources in addition to memory, and it makes no assumptions on the underlying resource management mechanisms in place. Finally, our approach is structured to assist application developers in building quality of service enabled applications.

In this paper, we begin with a discussion of related work in this area. We then outline the general approach taken in our work, and introduce an architectural framework that realizes this approach. We describe a preliminary prototype system based on this design work capable of adaptive allocation of memory resources to applications, and present experimental results and evaluations demonstrating the suitability of our system to its task. The paper concludes with a project status report and our plans for future work.

2 Related Work

The leading solution to the problems outlined in the previous section today is to simply add additional memory to systems in an attempt to make the problems go away. Unfortunately, this tends to only delay the inevitable shortage, and can be quite costly over the long term. Without proper resource management, it has been shown that this approach wastes approximately 20% of resource capacity, and fails to support quality of service needs in 90% of cases [8]. As a result, to fully address these problems, research into memory management is needed. In this section, we take a look at some of the research work to date in this area and assess its effectiveness and suitability to task.

2.1 Memory Management Algorithms and Policies

Over the years since the concept of virtual memory was first introduced, several approaches to memory management have been developed. These include first-in-first-out (FIFO) selection [10], least recently used (LRU) selection [4], the loop detection method [19], the access-graph algorithm [9], the working set approach [5], clock selection [6], and several others. Of these, the working set approach and clock selection are most commonly examined in detail; the working set approach because of its optimality, and clock selection because of its simplicity, efficiency, and ease of implementation.

Unfortunately, all of the above approaches have their drawbacks. FIFO selection can make very poor decisions as it pages based on time of page in rather than page referencing, which can lead to problems, particularly in overloaded conditions. LRU selection works well in principle, but maintaining LRU page ordering tends to be inefficient and incurs substantial overhead in practice. The loop detection method has been proven to work well only for scientific programs and not in general, and is costly to implement. The access-graph algorithm tends be complex and expensive, working best when memory access information is known in advance, rather than being obtained at run-time. While the working set approach does well in minimizing page faulting, it has high overhead and is costly to do in practice, often requiring

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

special hardware support. Clock selection, through gaining simplicity and efficiency, does not fare as well in reducing page faulting. Innovative approaches such as WSClock [2] are aimed at combining the benefits of the working set approach and clock selection. However, none of these approaches have been designed with quality of service in mind, and are by themselves inadequate for meeting the expectations of individual applications as there are no ways to tailor memory management to the needs of particular processes.

2.2 Commercial Operating Systems

Memory management in most commercial operating systems is based heavily on the algorithms and policies listed above. Some operating systems, however, have provided additional support to improve memory management.

Virtual memory management in the Solaris operating system is largely based on a two-handed clock selection approach [25], as are many flavours of the Unix operating system. To provide additional support to programmers to reduce page faulting, Solaris also provides a memory advisement facility [22]. Programmers provide intended memory access patterns for address ranges to the kernel, which in turn attempts to optimize paging according to this advice. Since this advice can be ignored and requires specific a-priori programmer knowledge, the effectiveness of this is somewhat limited.

Memory management in Windows NT attempts to approximate the working set approach, without many of the overhead costs associated with this approach [17]. Programs start with a default working set size, which grows as memory demand increases. Periodically, the working set manager steals pages from each process. If a page fault occurs on a stolen page in short order, it is returned to the process. Otherwise, the working set size for the process is reduced and the page is freed for allocation elsewhere. While this approach is relatively efficient, it does induce unnecessary page faults, and is not necessarily as optimal as the traditional working set approach, as unneeded pages may be retained longer than necessary before they are stolen.

Surprisingly enough, as reported in [7], many Linux variants adopt a similar page stealing approach in virtual memory management, with similar benefits and problems.

2.3 Quality of Service Research

As mentioned in the previous section, very little research effort has been recently directed towards memory management to support quality of service. Examples of the work that has been done are described below.

The work in [3] is an interesting approach, whereby processes needing guaranteed access to memory can share locked memory from a memory server. Programs can then store data into this shared memory, guaranteed that it will never be paged out and will always be available for accessing. Unfortunately, this approach requires a-priori knowledge of what data needs to remain in memory, putting additional burden on the programmer. Furthermore, code and stack segments for processes remain pageable, so memory problems can still occur.

A different approach is taken in [7]. In this work, processes can specify minimum and maximum memory bounds, providing guaranteed levels of memory access. Like the approach above, this requires the programmer to have knowledge of a program's memory needs in advance, which tends to be difficult to predict under a wide variety of workloads. Furthermore,

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

while each process can have a guaranteed amount of memory available, the operating system still decides which pages are kept in this memory and which are not, which could lead to unexpected and undesirable results.

In our current work, we will apply the experience and lessons learned from the related work described in this section to the development of an approach to memory management taking significant steps towards solving many of the problems discussed above. In the next section, we examine this approach in detail.

3 An Approach to Quality of Service Resource Management

To meet the requirements of Section 1, we will be making use of a generic approach to resource management to support quality of service provisioning. This approach is capable of managing a wide variety of resources, including memory, which is the focus of this paper. This approach is based on our earlier work in dynamic application-driven quality of service management [15].

3.1 General Strategy

The general strategy in our approach, shown in Figure 1, is to dynamically adjust the resource allocations given to an application until the quality of service it delivers meets expectations. As the application executes, the quality delivered is measured according to whatever metric was used in expressing user expectations (for example, frame rates, responsiveness, data loss, jitter, and so on). If the delivered quality of service exceeds expectations by a large margin, the application could do with a less generous resource allocation. On the other hand, if the delivered quality of service is below expectations, the application requires a better resource allocation, either through the allocation of additional resources, or a better distribution of previously allocated resources. By making adjustments in this way, this strategy ensures suitable resource allocations are made to satisfy quality of service expectations, while minimizing resource wastage due to overallocations.

In our current work we are focussing on memory as the resource to be managed. To provide a process with a better resource allocation in this scenario, we ensure that more of its memory pages are resident in physical memory; alternatively, to provide a process with a less generous allocation, we reduce the number of memory pages it has resident. In either case, to deliver the best quality of service possible, we should ensure that the pages that are resident are those that provide the most benefit to the process. Because users need only specify *what* their expectations are and not *how* to achieve them, these underlying memory management details are effectively hidden from the user and application developers.

In the following section, we examine an architectural design that realizes this general strategy.

3.2 Architectural Design

To support our strategic approach, we have developed the architecture shown in Figure 2. This architecture is derived from our earlier work in application instrumentation [13]. This

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

```
Given: Application p.
       Initial resource allocation a.
       QOS metric m.
       QOS expectations e.
---------------------------------------------
p is started with resource allocation a.
while (p not done) do:
    Execute code in p.
    q ← QOS measured with m.
    if (q > e) then
        Determine reduction r.
        a ← a - r.
    else if (q < e) then
        Determine improvement i.
        a ← a + i.
    endif
endwhile
```

Figure 1: General Resource Management Strategy

approach requires the insertion of code into applications at strategic locations to facilitate the collection of quality of service metrics and exertion of control over the applications.

The resource management architecture we have developed is object-oriented: Classes of standard components can be developed for general tasks and new components can be specialized easily from existing standard classes. This allows application developers to reuse components easily to reduce development effort, while allowing them to build application-specific components tuned to their particular needs. The components of this architecture are discussed in the sections below.

3.2.1 Instrumented Process

An instrumented process is an application process with embedded instrumentation code. It is this instrumentation code that enables the management of the application process. The architecture components that comprise the instrumentation code are discussed below.

- **Coordinator:** The coordinator facilitates interactions between a resource manager and the instrumented process. Its role includes message routing, managing sensors and actuators, and handling initialization and termination activities within the process. All knowledge of the resource manager is confined to this component, effectively hiding it from the remaining instrumentation components.

- **Sensors:** Sensors are used to collect, maintain, and (perhaps) process a wide variety of metric information within the instrumented processes. Sensors get their input data from probes inserted at strategic points in the application code or by reading other sensors. Sensors provide their information to the coordinator in the form of periodic

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

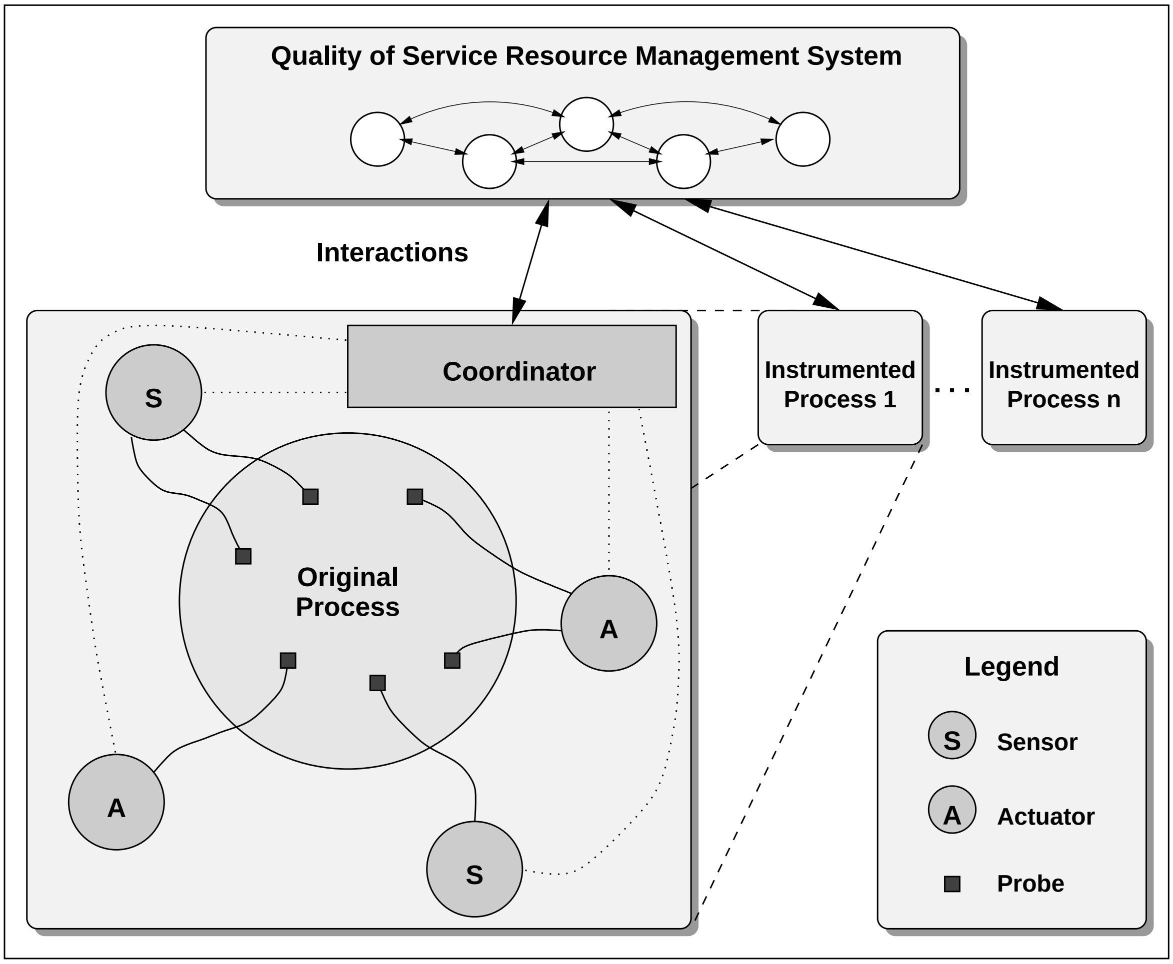

Figure 2: General Resource Management Architecture

Technical Report #546, Department of Computer Science, The University of Western Ontario,
London, Canada, November 1999

reports, alarm reports (when exceptional or critical circumstances arise), or in response to explicit requests. Sensors are dynamic components – they can be enabled or disabled, reporting intervals can be adjusted, thresholds can be modified, and sensor processing algorithms can be changed, all at run-time while the application process is executing.

- **Actuators:** Actuators are used to encapsulate functions that can exert control over the instrumented process to change its operation or behaviour. Actuators, like sensors, function through the use of probes or other actuators, and possess similar dynamic properties. In this current work, they are not used extensively, but can be used to support quality of service negotiation, adaptation, and other functions in the future.

- **Probes:** Probes are embedded in process code to facilitate interactions with sensors and actuators. Probes allow sensors to collect metric information and allow actuators to exert control over process behaviour. Probes are the only instrumentation component that must be injected into the original process source code – all other components need only reside in the same address space as the process, and do not require code modifications.

3.2.2 Quality of Service Resource Management System

A quality of service resource management system, or simply a resource management system, is a collection of applications responsible for managing a set of resources, such as CPU cycles, memory, network and disk bandwidth, and so on. This includes allocating and scheduling the resources, as well as accounting for their use. A resource management system can be structured as single or multiple resource manager processes spread across several systems – one can imagine a hierarchy of resource manager processes working with one another to manage the resources of a large distributed environment.

When an instrumented process is not meeting quality of service expectations, its coordinator will notify the resource management system to adjust resource allocations so that these expectations can be met. To do so, the resource management system must determine where in the distributed environment changes are needed, which resources should have their allocations changed, and how much the allocations should be changed. The adjustments carried out depend upon the application making the notification, the resources in question, and the underlying system and resource management schemes in use.

3.2.3 Detailed Example

In this section, we present an example showing how the general strategy of Section 3.1 is carried out through the components of the architecture described in this section. In this example, we will consider the instrumentation and use of a video application whose pseudo-code shown in Figure 3.

Suppose that the user specified a policy p that the application deliver video at a frame rate of 30 frames per second, plus or minus 2 frames. The application is started with its initial resource allocation. At this time, a coordinator c for the process is also created and a frame rate sensor r is initialized. At this point, the quality of service policy p is translated into initial thresholds for the frame rate sensor r. In this example, the policy from above will

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

Given: Video application v.
QOS policy p.

1. Perform initialization for v.
2. Initialize coordinator c.
3. Initialize frame rate sensor r with p.
4. Create window w for displaying video.
5. **while** (v **not done**) **do:**
6.   Retrieve next video frame f.
7.   Decode and display f in w.
8.   Execute $r \rightarrow probe$.
9. **endwhile**

Figure 3: Instrumentation Example

translate into a threshold specifying a target rate of 30 frames per second, an upper threshold of 32 frames per second, and a lower threshold of 28 frames per second.

Once initialized, the application retrieves video frames, decodes them, and displays them in a window w created by the application. After each frame is processed, a frame rate sensor probe is called, as shown on line 8 of Figure 3. The sensor r measures the time since the last probe invocation, and uses this value to determine the current frame rate, filtering out noise and unusual spikes in the measurements. If the given quality of service policy p is violated, the sensor r will notify the coordinator c. The coordinator c will, in turn, notify the resource management system of the situation to have resource allocations adjusted appropriately.

For example, if the delivered frame rate increased to 35 frames per second, the frame rate sensor r will generate a report to have the rate decreased, which would find its way to the resource management system. The resource management system determines a resource allocation reduction or redistribution to move the application's delivered quality of service into line with policy expectations in p, and makes the necessary adjustments. If, on the other hand, the delivered frame rate dropped to 25 frames per second, the frame rate sensor r threshold is violated and the sensor generates a report to have the frame rate increased. The resource management system determines a resource allocation improvement to move the application's quality of service into line with policy expectations p, and makes the necessary adjustments. This improvement could entail either the allocation of additional resources, or a redistribution of previously allocated resources.

4 Prototype Implementation

In this section, we describe a prototype system we have developed based on the generic architecture presented in the previous section for quality of service memory management.

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

4.1 Instrumentation Implementation

Instrumentation is provided to application developers through a C++ class library. This library contains a variety of standard sensors, actuators, and coordinators. The library can also be specialized to develop new classes of instrumentation components to meet the unique needs of specific applications. This is done through providing a variety of abstract classes suitable for specialization.

As noted in [13], instrumentation can be carried out in a variety of ways. Currently, most instrumentation is carried out by hand or using wrapper techniques, where standard functions or methods are replaced by instrumented versions. We have found that inexperienced users encounter few difficulties in correctly instrumenting their applications; nevertheless, we are actively investigating approaches to automate instrumentation to make the process even easier [18].

In the following sections, we examine implementation details of the various instrumentation components included in the instrumentation library.

4.1.1 Sensor Implementation

In the instrumentation library, several sensor classes are provided, including:

- Quality of service metric sensors. Among the metrics that can be measured by these sensors are video frame rates, transaction processing rates, transaction response times, computation rates, data loss, and video jitter.
- Buffer sensors. These sensors can be used to measure the characteristics of network and I/O buffers, including their capacity, current size, and recent trends. This information is useful in locating bottleneck nodes and resources in quality of service management.
- User hint sensors [14]. These sensors detect changes in user's usage of an application, useful in guiding resource management decisions for quality of service.
- Other management sensors. A variety of sensors for other purposes are also provided for fault detection, resource accounting, configuration recording, security violations and so on.

Probes for sensors are implemented as methods on these objects which are inserted into application source code to be executed along with the application code, or periodically through a timer. The sensors preprocess data to filter noise, spikes, and other transients from measurements to allow for smooth changes and graceful adaptations. For example, in our video frame rate sensor, filtering compensates for fluctuations in frame rates caused by variations in the video stream.

4.1.2 Actuator Implementation

In addition to sensors, the instrumentation library contains several actuator classes for application control, including:

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

- Quality of service adaptation actuators. These actuators allow applications to adapt to situations and change their behaviour when they cannot receive the resources they need to meet quality of service expectations.

- Process control actuators. These actuators allow processes to be terminated, suspended, and resumed.

Like sensors, actuators have probes as well to interface with code in the instrumented process as needed. In some cases, no probes are necessary as the actuators contain all the code needed to carry out their actions, as is the case with the process control actuators above. Other actuators, however, need probes to link into the original application to control its behaviour, as is the case with quality of service adaptation actuators. Developers register callback functions with actuators to be invoked when the actuator is called upon to carry out its action. For example, a video application may register an adaptation function with an adaptation actuator so that when an adaptation is necessary, the function is called to change characteristics of the video stream, such as the frame rate, frame size, compression ratio, colour scheme, and so on.

4.1.3 Coordinator Implementation

To coordinate instrumentation activities and interface with the outside world, the instrumentation library also contains several coordinator classes, including:

- Quality of service coordinators. These coordinators allow application quality to be controlled by the quality of service resource management system discussed in the following section.

- Logging coordinators. These coordinators allow application activity as recorded by sensors to be logged to a file or to a server over the network.

The above coordinators communicate with the outside world using a variety of different mechanisms, including files, sockets, semaphores, shared memory, message queues, and so on, depending on the circumstances and the communication required. The coordinator also filters and correlates sensor data as necessary. For example, a single quality of service problem might be detected simultaneously by several application sensors in place. Instead of reporting several problems to the resource management system, the quality of service coordinator processes the sensor data and generates only a single problem report.

Further implementation details of our instrumentation and code examples can be found in [13].

4.2 Resource Manager Implementation

As proof of concept, we have developed a simple resource management system capable of dynamically changing memory resident set sizes of instrumented processes. Our previous work in [15] details similar work on a CPU resource manager.

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

4.2.1 Existing Solaris 2.6 Memory Management

Before exploring the details of our prototype, we first give a brief overview of memory management in our initial target platform, Solaris 2.6. As mentioned in Section 2, Solaris uses two-handed clock selection for guiding memory management decisions. This approach treats all resident pages exactly the same, regardless of which processes currently own them. In this approach, the system page table is logically arranged as a circular list. A virtual clock hand scans through the circular list, clearing referenced bits for pages on its way. A fixed distance behind, a second hand follows the first hand checking the referenced bits. If the referenced bit for a page is still clear when the second hand reaches it, it has not been accessed recently, and is eligible for eviction. Otherwise, it has been accessed recently, and should remain in memory.

In Solaris 2.6 and later, the system page table is structured as a linked list of contiguous memory segments, to support the addition of memory to live systems. For two handed clock selection, a kernel process *pageout* calls interface functions to this linked list to traverse the page table and advance the hands of the virtual clock. The first hand clears hardware referenced bits maintained by the memory management unit; the second hand later synchronizes the hardware referenced bits (which may now be set once again) with those in the corresponding page table entries, to have the page table reflect the current hardware state. The second hand then checks these bits to decide if a page is eligible for eviction, according to the standard two handed clock approach. Eligible pages are added to a list of "free" pages, but are only actually evicted when another page needs to be paged in to handle a page fault. Pageout only runs when the size of the free list dips below system defined thresholds; it spends the rest of its time asleep, without carrying out two handed clock selection.

The exception to the above procedure is when pages are locked or pinned into memory. Such pages are designated as being unpageable, and are always resident in physical memory. Locked pages are essentially ignored by pageout when conducting page table scans. Typically, only vital kernel pages, such as the page table itself, are locked into memory. However, it is possible for processes with super-user privileges to do so as well, using library functions such as *mlock()*, *mlockall()*, and *plock()*.

4.2.2 Implementation Overview

Memory management in Solaris 2.6 is quite efficient, due to both its underlying clock selection approach, and its implementation. However, as stated in Section 2, the two handed clock approach is not the best at minimizing page faults. Furthermore, since it treats all pages equally regardless of process ownership, this approach is not suitable for quality of service management, where the needs of processes must be examined individually to meet expectations. The working set approach, however, is better suited to meeting the needs of individual processes, but incurs far too much overhead to be used on its own. A combination of the two approaches is necessary to extend Solaris 2.6 memory management to be able to support quality of service.

The key to doing this lies in the ability of Solaris to lock pages of processes into memory and make them unpageable. By selecting and locking critical pages of processes into physical memory, we can effectively augment the paging decisions already being efficiently made by the system. To determine which pages are most important to processes at any given time, we

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

can collect page reference information from kernel virtual memory management structures, including the system page table. This information need only be collected for processes with quality of service expectations, and those that are important to the user [14], which is a small fraction of the processes on a system. The collection of this information can be done at a low priority in the background, so that little overhead is generated. Based on this information, we can estimate the working sets of processes, while the system is still paging efficiently using two handed clock selection.

When a process is not meeting its quality of service expectations, and it is either page faulting heavily or memory is scarce on the system, increasing its memory resident size will likely alleviate the problem. Knowing the process's working set, we can lock in memory those pages that are most needed and used by the process to reduce its page faulting, and improve its quality delivery. Likewise, we can unlock pages when they are no longer in a process's working set or when quality of service expectations are exceeded. As a result, we can effectively implement our generic dynamic resource management approach from Section 3 for memory management. In doing so, we can ensure that processes have access to the memory they need when they need it on an individual basis, that memory is made available to other processes when no longer needed, and that this can be done in an efficient manner with little overhead, thereby meeting our quality of service requirements.

4.2.3 Implementation Details

The extensions to Solaris 2.6 memory management were implemented in a single user-level daemon process; no kernel modifications were required. This allows us to enable and disable memory management extensions at run time, and allows us to easily deploy it on any Solaris 2.6 system. The memory manager daemon must run with super-user privileges to get the necessary kernel access for retrieving process and page table information, locking and unlocking pages of memory, and a variety of other tasks. The structure of the memory manager daemon is shown in Figure 4. Its various components are discussed below.

- **Process Interface:** The process interface is used to interact with coordinators in instrumented processes. This includes registering processes with the memory manager for management, reporting violations of quality of service expectations, and so on. Processes communicate with the memory manager using semaphores and shared memory registered at the initialization of the processes. When the quality of service delivered by a process deviates from expectations, a sensor threshold in the process is violated. The sensor computes how far it is from its target level of quality, and passes this on to the coordinator for the process. The coordinator takes this value and signals the memory manager using semaphores to inform it of the need for a memory adjustment to meet expectations.

- **Memory Manager Controller:** The memory manager controller is in charge of control flow and message routing within the memory manager. This includes timing and synchronization of the tasks carried out by other components of the manager, including when and how often the tasks execute. It is also responsible for decisions made in managing memory, such as when to lock or unlock memory pages, how many pages to lock or unlock, and which pages to lock or unlock. These decisions are made

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

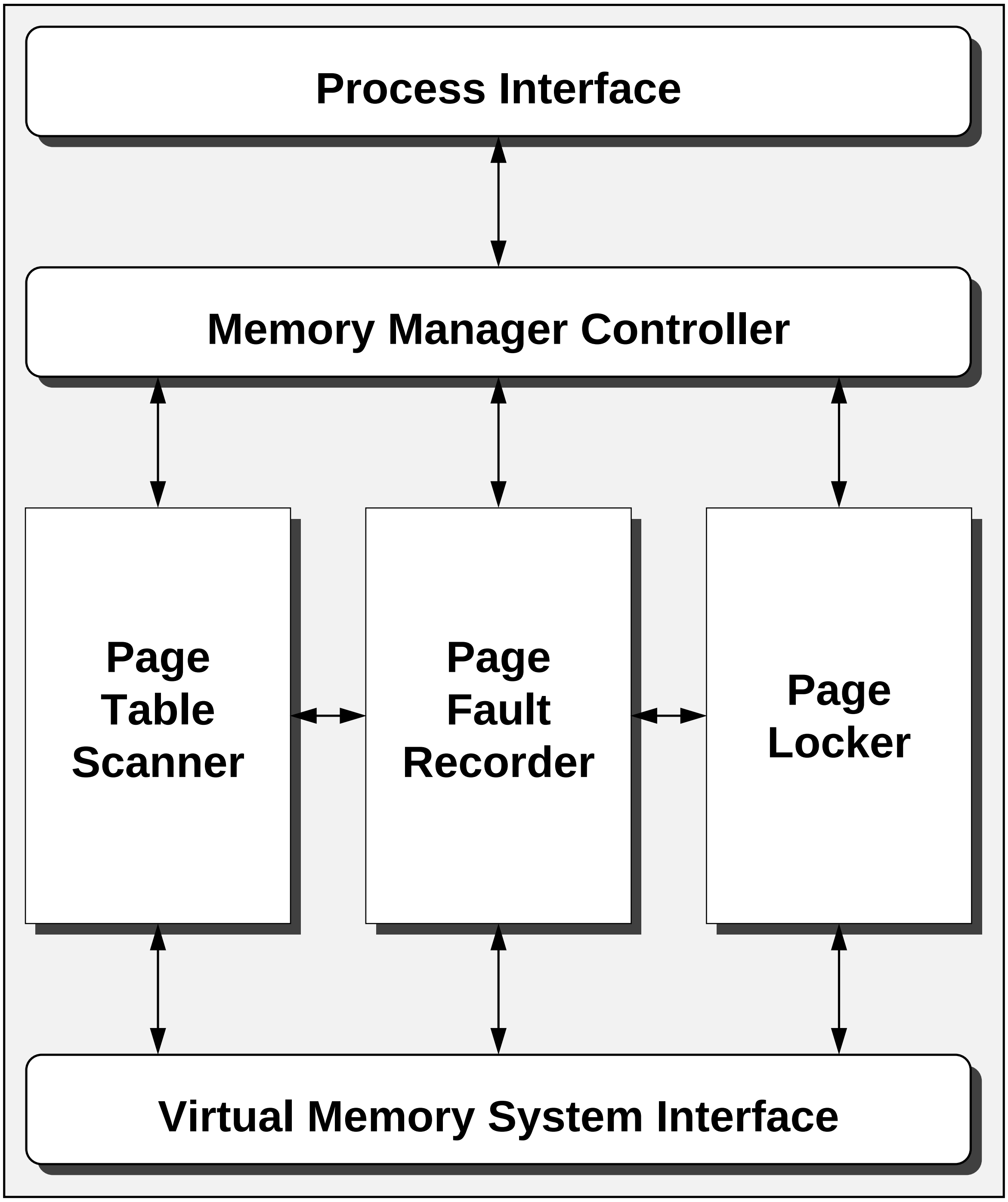

Figure 4: Memory Manager Details

in an intelligent fashion, based on prior actions in addition to the current state of the system and applications in question, so that the decision process can be fine tuned dynamically at run time.

For example, if the number of pages being locked or unlocked at once is too high, the application can "ping-pong" back and forth between quality that is either too high or too low – adjusting the number of pages locked or unlocked, as well as other decision parameters effectively eliminates this kind of problem. As another example, as available memory becomes scarce on the system, the controller can increase the rate at which page table and other virtual memory information is collected, to allow it to make quicker and more accurate memory management decisions. Likewise, if there is an abundance of physical memory available, quality of service violations due to memory shortages are not likely, so the rate of information collection can be decreased to reduce overhead and increase the overall efficiency of the system, much in the same way pageout works on Solaris systems.

- **Page Table Scanner:** The page table scanner is responsible for traversing the system page table to collect information on processes that have registered to the memory manager as having quality of service expectations needing to be enforced. As the page table is traversed, the scanner determines if each page is owned by a registered process. This check can be difficult as the Solaris virtual memory system was not designed to support it. Tracking page ownership in page structures would be complex and expensive, as it is possible for pages to be owned and used by several processes, leading to extra memory storage and time-consuming maintenance procedures. Since the two handed clock selection approach used in Solaris does not consider page ownership in its eviction decisions anyways, this information is not included in paging structures. Instead, page structures keep *vnode* references that interface with the file system to name memory objects and assist in paging operations to disk. By keeping track of *vnode* references contained by the various segments of a process's address space, we can match page *vnode* references while scanning the system page table and indirectly tie pages to the processes that own them.

  If a page is not owned by a registered process, it is ignored. Otherwise, the value of the page is calculated; the higher the page value, the more likely the page is in the current working set of the process, and the more valuable the page is to be locked into memory. Similarly, when pages need to be unlocked, those with the lowest value are selected first. As a page remains resident in physical memory, its value increases by a fixed amount depending on its free list and referenced bits. If a page's free list bit is set, the page has been marked for eviction and is currently on the free list. This means that the page is currently not very valuable to the process, and the page value maintained by the page table scanner should increase by only a small amount. If a page's free list and referenced bits are both clear, the page is more valuable to the process than the previous case because it has been referenced after those on the free list, but not recently. As a result, the scanner should increase the page's value by a moderate amount. If a page's referenced bit is set, the page has been accessed recently, and is quite valuable to the process, so its value should be increased by the scanner more than either of the two previous cases. When a page is no longer found to be resident in physical memory,

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

it has been paged out due to inactivity, so its value is decreased by a fixed amount by the scanner accordingly. Likewise, if a page has been locked into physical memory, its value is decreased by a fixed amount unless the page has been referenced. This is to prevent pages locked into memory from remaining locked when they are no longer being used, allowing the memory to be freed up for more valuable pages. As necessary, the page table scanner consults a number of process address space structures to augment the information collected from the system page table.

This approach to evaluating page values has several benefits. First, by calculating page values in this way, we can closely approximate the working set of a process at a relatively low cost, allowing the memory manager controller to make informed decisions to move quality of service delivery into line with expectations. Second, several of the parameters used in carrying out these calculations can be tuned at run-time to adjust page evaluation as necessary. Third, new memory management policies can be supported by this memory manager, simply by changing the way page evaluations are carried out, as this dictates which pages are locked and how resources are allocated. The study of new page evaluation mechanisms and the ability to completely change the mechanisms used at run-time is currently ongoing work.

- **Page Fault Recorder:** For each process registered with the memory manager, the page fault recorder measures its page fault rate over time as directed by the memory manager controller. This is done by retrieving resource usage statistics for the processes stored in the */proc* file system on Solaris systems. The page fault rate of an application cannot be measured directly, however. The page fault recorder can only retrieve the number of page faults that a process has incurred, and must determine the rate over time itself. In the future, statistical filtering and trend analyses could also be applied here to acquire additional information, if the memory manager controller requires it in making its memory management decisions.

- **Page Locker:** The page locker is in charge of locking and unlocking memory pages as instructed by the memory manager controller. It also maintains accounting information on which pages are locked, how many pages of lockable memory are available in the system, and so on. The Solaris virtual memory management system already maintains similar information, but does so with too many restrictions for our needs. For example, as virtual memory is allocated to processes, Solaris reduces the number of pages it considers lockable, to avoid page faulting in cases where too many pages are locked, or the locker is making uninformed locking decisions. Consequently, under ordinary conditions, Solaris would refuse many of the memory manager's lock requests unnecessarily.

  To overcome these problems, the page locker updates Solaris virtual memory management accounting structures so that Solaris is unaware of the page locking it is carrying out. Consequently, the page locker must also keep lock accounting information of its own to ensure that locking is done in a responsible manner. This includes ensuring that there is always a sufficient supply of pages on the system that are never locked to prevent physical memory from being completely filled with locked pages, which would effectively wedge the system. With this done, the page locker can carry out its task to

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

reduce page faulting in processes in need of quality of service improvements.

In times of extreme memory demand, it is possible for the page locker to run out of pages that it can lock for applications – demand of resources simply outstrips supply. In this kind of degraded mode of operation, there are several possible options. One is that all further locking requests are ignored, and the current allocations are maintained. Another is that resource allocations to all applications are scaled back to increase the supply of available memory. Another is to cancel the quality of service policies of some applications or force them to adapt to tighter resource restrictions, based on some kind of service differentiation mechanism. This issue is the subject of much ongoing work, and will be examined in future extensions to this work as well.

- **Virtual Memory System Interface:** The virtual memory system interface provides the memory manager with the mechanisms needed to interact with the underlying operating system. This includes kernel reading and writing access, as well as system calls to access kernel functions.

  Since user-level processes, such as the memory manager, can only call functions residing in the kernel through a system call, functions without a system call interface defined in the kernel's system call table cannot be called. Unfortunately, there are many such functions that the memory manager needs to call, to scan the page table (*page_first()*, *page_nextn()*) and lock memory in other address spaces (*as_ctl()*), for example. In these cases, system call interfaces for the functions are added at run-time to the system call table in unused slots, to allow the virtual memory system interface to access them.

  The virtual memory system interface also hides the details of the underlying operating system, which can help facilitate future efforts in porting the memory manager to other platforms. Such efforts are currently under way.

4.2.4 Memory Manager Operation

The memory manager daemon is usually started as part of the system boot process, although it can be started at any time. Once initialization completes, the memory manager is idle, waiting for the first instrumented process to register with it for management. Once the first process registers through the process interface, the memory manager controller launches threads to carry out page table scanning and page fault recording, and directs the page table scanner and page fault recorder to begin collecting the required information on the registered process. As additional processes register, the memory manager controller instructs the scanner and recorder to collect information on the new processes as well. Both the page table scanner and page fault recorder rely on the virtual memory system interface to read data from the kernel, as well as calling kernel system calls for them, whenever necessary in carrying out their tasks.

When the process interface receives a quality of service violation report from a registered process, the memory manager controller checks the page fault rate for the process from the page fault recorder and the amount of free memory on the system through the virtual memory system interface. If this information indicates that memory is not a problem with the process, no further actions are taken. (In the future, other resource managers, such as our CPU resource manager [15], could be contacted to provide the resource adjustments needed

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

to correct the problem.) If there is a memory problem, the controller retrieves a list of the pages most valuable to the process from the page table scanner, and instructs the page locker to ensure that these pages (and only these pages) are locked into memory through the use of the virtual memory system interface. Later, if the page table scanner determines that the locked pages are no longer valuable to the process, it has the page locker unlock them in a similar fashion.

As registered processes terminate, the page table scanner and page fault recorder stop collecting information on the terminated processes, as directed by the memory manager controller. When the last registered process terminates, the memory manager controller puts the scanner and recorder threads to sleep until they are needed again when new processes register in the future. The memory manager never shuts down itself, but it can be, if the need to do so arises.

5 Experimental Results and Experience

An initial set of experiments was conducted to evaluate the operation of the prototype instrumentation and memory manager discussed in the previous section. Based on the success of these tests, more rigorous experiments were planned and executed using our quality of service management testbed [16]. The results of these experiments proved to be quite interesting, demonstrating the utility, flexibility, and suitability to task of our work.

5.1 Video Conferencing Experiments

In these experiments, we evaluated the success of our prototype system in improving quality of service delivery through more effective memory management. The test application in these experiments was a simple video conferencing package [23]. To monitor its performance, and to allow quality of service violations to be reported, we added sensors to both the video transmitter and receiver to measure the video throughput, in frames per second. The resulting programs were executed on two of our UltraSparcs using Solaris 2.6 and our memory manager to assist in dynamically managing memory on the workstations. The UltraSparc 5 workstations are each equipped with a 269 MHz processor, 64 megabytes of main memory, and 192 megabytes of virtual memory, and are networked together by 10 megabit switched Ethernet. The transmitter and receiver both had a size of approximately 11 megabytes once loaded into memory, most of which caused by shared libraries linked in at run-time. We instructed the transmitter to send to the receiver 1800 frames of a 320×240 8-bit video stream captured in real-time from a camera at the department's operations counter, connected to a SunVideo board in the transmitter. We gave the application a quality of service policy dictating that the video be displayed at the receiver at a mean frame rate of 30 frames per second, allowing for a deviation of 2 frames per second from this target (giving a range of 28 to 32 frames per second).

Our experimentation consisted of executing the instrumented video conferencing application several times under a constant memory load generated by a memory load generator. The generator works by allocating the virtual memory that would remain after running the application, and using multiple threads to touch the pages randomly to induce some rather severe thrashing conditions. While this is an artificial workload, the conditions are rather

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

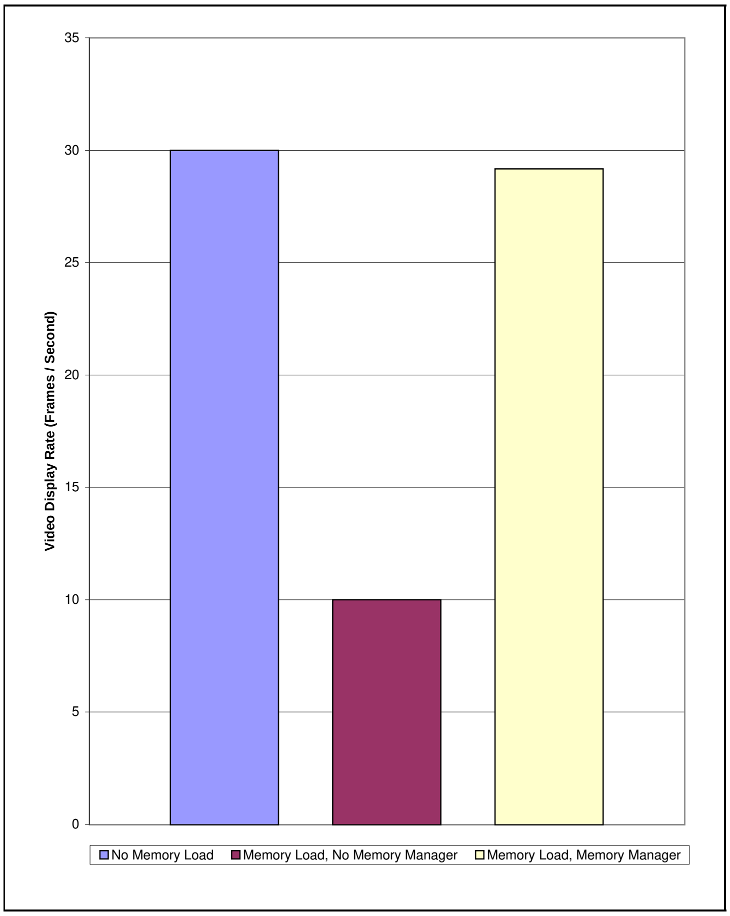

Figure 5: Video Conferencing Transmitter Analysis

extreme and provide a good stress test for the prototype under worst case working conditions. These experiments were repeated in the presence and absence of our memory manager for comparison purposes, and replicated five times for statistical confidence.

Figure 5 compares the mean completion time, in seconds, for the video conferencing application under a variety of conditions. Under no memory load, the video was displayed at an average rate of 30 frames per second, with virtually no perceptible jitter and around only 20 lost frames that were not displayed. When memory load was applied to the transmitter workstation, there was a significant degradation in performance. The mean frame rate dropped to 10 frames per second, with tremendous jitter and around 70 lost frames. When our memory manager was in place, there were substantial improvements in performance. The frame rate began at around 10 frames per second, as the memory manager acquired memory referencing behaviour of the application, but quickly increased to 30 frames per second after 2-3 seconds, for a mean rate of 29 frames per second. Likewise, jitter at the beginning of the video stream quickly disappeared and only 50 frames of video were dropped, most of which at the beginning of the stream.

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

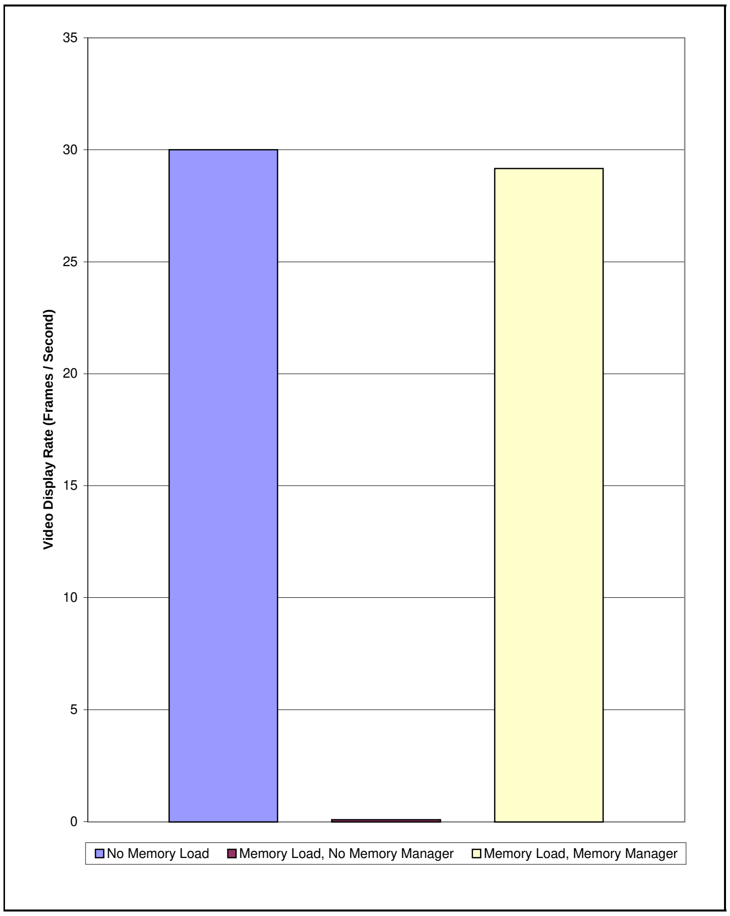

Figure 6: Video Conferencing Receiver Analysis

Figure 6 shows the results from a similar set of experiments, where memory load was induced on the receiver workstation instead of the transmitter workstation. Again, excellent quality was received when no load was applied to the system. When memory load was applied to the receiver, there was nearly 100% video frame loss – the application simply could not manage juggling buffers and displaying the video without the required memory resources, and consequently the frames had to be dropped. This resulted in a near zero mean frame rate for the video stream – unacceptable under any expectations. With our memory manager in place, once again there was a substantial improvement in performance. This time, only around 100 video frames were lost, almost all of which at the beginning of the video stream in the first 2-3 seconds when the memory manager was determining the application's memory referencing behaviour. After this point, video was delivered at the target rate of 30 frames per second, with nearly no perceptible jitter. This resulted in a mean frame rate of nearly 29 frames per second once again.

In both of the above cases, our memory manager significantly improved quality of service provisioning over standard Solaris memory management. This was not just because additional

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

resident pages were available to the application, but also because the most needed and most valuable pages were kept resident. If this was not the case, and the best page choices were not the ones selected, then application performance would have been worse, as additional and more costly page faults would occur to bring the best page choices into a smaller amount of available physical memory (the reduction being caused by poor choices locked into memory). Since application performance improved, however, we must conclude that the choices made by the memory manager were reasonably good ones.

Further, if our memory manager had the opportunity to observe the video conferencing application's memory references before the memory load was applied, which would more likely be the case in realistic, live systems, the memory manager could have reacted to the situation much faster, and the quality of service improvements would have been even more substantial. Preliminary experimentation to test this claim has been successful; more rigorous experimentation is currently under way.

5.2 Integer Sorting Experiments

In these experiments, we repeated the above style of experiments with a different kind of application. This time, a relatively simple data intensive task was used – an integer sorting application. As a test workload, we instructed the application to quick sort 8,192,000 randomly selected integers. This task was selected for experimentation because it is an easily controlled job that would require a great deal of memory, and, consequently, would be susceptible to memory deprivation. The size of the program was just over 32 megabytes once executing, most of which was required for storing the integers to sort.

To monitor the sorting application's performance, and to allow quality of service expectation violations to be reported, we again added instrumentation to measure its progress. The resulting program was executed on one of the Solaris 2.6 UltraSparc 5 workstations from the previous set of experiments and our memory manager to assist in dynamically managing memory on the workstation. Our experimentation consisted of executing the instrumented sorting application several times under a constant memory load generated by the memory load generator, as was done in the experiments above. These experiments were repeated in the presence and absence of our memory manager for comparison purposes, and replicated five times for statistical confidence.

Figure 7 compares the mean completion time, in seconds, for the integer sorting application under a variety of conditions. Under no memory load, the sorting process is complete in just under a minute. When the memory load is applied, performance drops substantially and the sorting process requires just over an hour and 45 minutes to complete when standard Solaris memory management is used without our memory manager in place. When our memory manager is executed, the sorting process is carried out in just less than 5 minutes with the same memory load applied – a substantial improvement over normal Solaris memory management. The extra time was required by the memory manager to determine the memory needs of the sorting application – in more realistic operating conditions, where the memory manager could see the referencing behaviour of the application before the memory load was applied, performance would be even better. As mentioned in the previous section, additional experimentation is under way to confirm this.

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

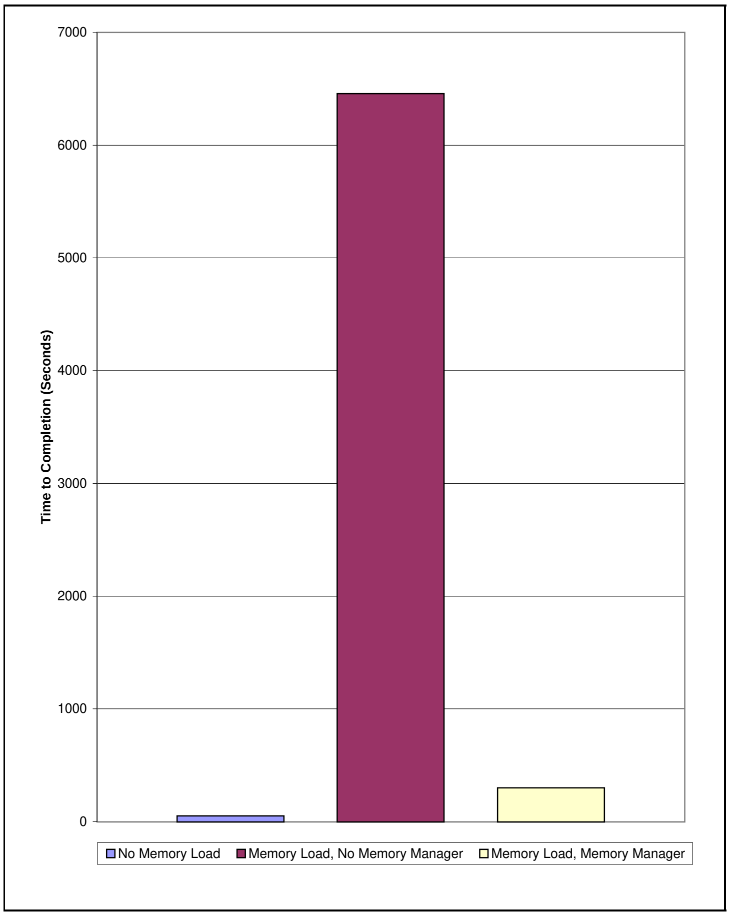

Figure 7: Integer Sort Analysis

5.3 Overhead Evaluations

As mentioned earlier, it is important for any resource manager to incur little overhead itself; if it consumes too many resources, the system might perform better without the resource manager in the first place. From measurements taken during experimentation, the overhead from our approach is minimal. An instrumented process on one of our UltraSparc systems requires approximately 400 microseconds more time to initialize itself and report to our memory manager. If the level of quality of service delivered meets expectations, one pass through the instrumentation code requires only 11 microseconds, on average. On the other hand, whenever a quality of service violation is detected, it takes on average 474 microseconds to report the violation to our memory manager, and to have it make appropriate adjustments by locking pages of process memory. This is acceptable since quality has already degraded at this point, and the additional time to correct the problem is necessary – the only alternative would be to allow to problem to persist, with quality continuing to be unsatisfactory to the user.

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

As the page table scanner collects page table information, only 30 microseconds are required on average to check the ownership of a given page, and compute and record its value for future use. It takes even less time for the page fault recorder to check and compute the current page fault rate for a process. As mentioned in Section 4, these tasks can be tuned at run-time to minimize overhead as they execute over time. Consequently, we can conclude that our approach to memory management is quite efficient in terms of processor consumption.

In terms of memory consumption, the instrumentation library only increases process size by 25 kilobytes. The memory manager itself requires less than 2 megabytes of memory at run-time, including all of its data. Memory is also conserved at run-time as the manager only locks pages valuable to executing processes with quality of service constraints. For example, the integer sorting application discussed in Section 5.2 had a size of more than 32 megabytes. During experimentation under load however, the memory manager never needed more than 10 megabytes of the application resident to maintain quality, and needed significantly less on average. As a result, we can see that our approach is efficient in terms of memory consumption as well.

6 Concluding Remarks

The work we describe here is part of ongoing work on addressing issues in quality of service management, and there are many complex issues yet to be resolved. We believe that the effective and efficient management of system resources is crucial to quality of service provisioning. Only a comprehensive system that considers all system resources, including memory, can be able to meet all of the needs and expectations of application users.

This paper focusses on the development of an approach to memory management to support this kind of quality of service provisioning. By using a dynamic resource management strategy tied to user expectations, we can ensure that processes have the appropriate pages of code and data in memory when needed, so that quality of service can be provided as necessary. At the same time, we can also ensure that as much memory as possible is available to other applications and that wastage is limited. Using a prototype system based on this approach to memory management, we have also been able to carry out a number of experiments and have realized several useful results, thereby demonstrating the utility of our work.

We have identified several avenues for continuing and future work in this area:

- We plan to continue the experimental validation of our system, and experiment with multiple applications under a wider variety of more realistic memory loads.
- We also plan to continue porting efforts to different platforms, including AIX, Windows NT, and Java.
- We need to extend our work to handle overload conditions when there simply are not enough resources to meet demand; we are currently examining service differentiation [1] and other approaches such as [24] as viable options. Utilizing actuators to facilitate quality of service adaptation and renegotiation is another interesting prospect.
- It is important to integrate this work with our previous work with CPU management, as well as investigate the management of other resources, possibly distributed across multiple hosts.

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

- Many applications have similar memory access patterns on each execution – it would be interesting to determine if memory reference behaviour and working set changes stored from one execution of an application could improve memory management decisions in subsequent executions.

- While the current prototype allows memory management to adapt over time, we need to fully explore how we can take advantage of this, as well as investigate how memory management policy itself can be changed at run-time.

- The work we have done to date is reactive – when quality of service violations occur, we correct them. Another approach that we are investigating is proactive quality of service similar to [11, 12], where potential problems are detected and handled before they actually occur.

Through this work, we have gained valuable insight into the quality of service management problem. With this experience, we can further pursue solutions that effectively and completely address the variety of quality of service problems that occur in today's computing environments.

About the Authors

Michael J. Katchabaw is a Ph.D. student in the Department of Computer Science at the University of Western Ontario. His research interests include quality of service management, distributed computing and multimedia, and electronic commerce.

Hanan L. Lutfiyya received her Ph.D. from the University of Missouri at Rolla in 1992. She is currently an Assistant Professor at the University of Western Ontario, London, Canada. Her research interests include management of distributed systems, software architecture, and electronic commerce.

Michael A. Bauer is Senior Director, Information Technology Services, at the University of Western Ontario. He is also a Professor in, and former Chair of, the Department of Computer Science. His research interests include distributed computing, applications of high-speed networks, and software engineering.

References

[1] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. *IETF RFC 2475*, December 1998.

[2] R. Carr and J. Hennessy. WSClock – A Simple and Effective Algorithm for Virtual Memory Management. *Proceedings of the Eight Symposium on Operating Systems Principles*, Pacific Grove, California, December 1981.

[3] H. Chu and K. Nahrstedt. Memory Management for Soft Real-Time Multimedia Applications. *Technical Report, University of Illinois at Urbana Champaign*, October 1997.

Technical Report #546, Department of Computer Science, The University of Western Ontario, London, Canada, November 1999

[4] F. Corbato. A Paging Experiment with the Multics System. *MIT Project MAC Report MAC-M-384*, May 1968.

[5] P. Denning. The Working Set Model for Program Behaviour. *Communications of the ACM, Volume 11, Number 5*, May 1968.

[6] M. Easton and P. Franaszek. Use Bit Scanning in Replacement Decisions. *RC-6192, IBM Research, Yorktown Heights, New York*, September 1976.

[7] J. Edwards and P. Cao. User-Oriented Resource Scheduling in UNIX. *Technical Report, University of Wisconsin, Department of Computer Science, CS-TR-96-1318*, 1996.

[8] M. Fabbi. Bandwidth Throwing – Yesterday's LAN Design Method. *Gartner Group Research Note*, December 1997.

[9] A. Fait and Z. Rosen. Experimental Studies of Access Graph Based Heuristics: Beating the LRU Standard? *Proceedings of the 1997 SIAM Symposium on Discrete Algorithms*, January 1997.

[10] G. Fine, P. McIsaac, and C. Jackson. Dynamic Program Behaviour Under Paging. *Proceedings of the ACM 21st National Conference*, May 1966.

[11] J. L. Hellerstein, F. Zhang, and P. Shahabuddin. An Approach to Predictive Detection for Service Management. *Presented at the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, Boston, Massachusetts, May 1999.

[12] C. Hood and C. Ji. Automated Proactive Anomaly Detection. *Proceedings of the Fifth International Symposium on Integrated Network Management*, May 1997.

[13] M. J. Katchabaw, S. L. Howard, H. L. Lutfiyya, A. D. Marshall, and M. A. Bauer. Making Distributed Applications Manageable Through Instrumentation. *Journal of Systems and Software 45 (1999)*, 1999.

[14] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. Using User Hints to Guide Resource Management for Quality of Service. *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1999.

[15] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. A Model for Resource Management to Support End-to-End Application-Driven Quality of Service. *Proceedings of the Tenth International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada, October 1998.

[16] M. J. Katchabaw, H. L. Lutfiyya, and M. A. Bauer. A Quality of Service Management Testbed. *Proceedings of the Third IEEE International Workshop on Systems Management*, Newport, Rhode Island, April 1998.

[17] R. Kath. The Virtual-Memory Manager in Windows NT. *Microsoft Developer Network Library Technical Article*, December 1992.

[18] S. Kaushal. Automating the Instrumentation in Distributed Applications. Master's thesis, The University of Western Ontario, December 1998.

[19] T. Kilburn, D. Edwards, M. Lanigan, and F. Sumner. One-level Storage System. *IEEE Transactions on Electronic Communications*, April 1962.

[20] P. Martin. Managing Database Server Performance within an Electronic Commerce Framework. *Presented at the 1999 International Symposium on Electronic Commerce*, Beijing, China, May 1999.

[21] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1996.

[22] Sun Microsystems Inc. madvise – Provide Advice To VM System. *Solaris 2.5 Reference Manual Answerbook*, Mountain View, California, 1996.

[23] Sun Microsystems Inc. *SunVideo User's Guide.* Mountain View, California, August 1994.

[24] T. Tan and W. Hsu. Scheduling Multimedia Applications Under Overload and Indeterministic Conditions. *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[25] U. Vahalia. *Unix Internals: The New Frontiers.* Prentice Hall, 1996.