15-712 Systems Final Report

Methods for Recognizing Service Quiescence

Gregory Hartman Jack Lin Michael Merideth

December 12, 2002

Abstract

Our motivation for this project is evaluating our hypothesis that if we had a variety of statistics concerning process resource consumption, we would be able to determine whether and when a process is quiescent.[1] We instrument the Linux operating system to allow us to gather relevant statistics, and write a second program, which we call QAnalyzer, that allows us to analyze these statistics, looking at variations in process resource consumption to determine whether and when a process is quicescent. Finally, to evaluate our hypothesis, we perform experiments in which we vary the behavior of the process under observation. Our results show both that we can determine when a process is not quiescent, and that certain classes of processes do post-processing when they would be expected to be quiescent.

1 Introduction

Before safely checkpointing the state of a service (that is, a server process providing an interface to an indeterminate number of clients), it must be ascertained that the service is quiescent. Without knowledge of the internal structure of a service, one would have difficulty in determining points of quiescence. For example, consider a simple client-server application, in which we treat the service as a black box. We might propose that the service would be quiescent when there are no outstanding client requests. For the purpose of simplifying this example, let us assume it is true that every client request is followed by a service response. Thus, we might propose monitoring the network connection at the service for incoming requests, and pairing these requests with responses. We might assume then that as soon as the service sent a response, if upon pairing this response with its request we determined that there were no additional requests outstanding, the service would be quiescent. However, this is an unsafe assumption. The service might, for example, issue a response and then do internal post processing of data it accessed.

In this paper we evaluate the hypothesis that by performing analysis on statistics of server utilization of a variety of resources, we could greatly increase our confidence that a service is indeed quiescent.

To gather the necessary statistics, we modify the kernel to track resource utilization on a per process basis. We gather the following information:

- CPU utilization
- Bytes sent over the network
- Packets sent over the network
- Number of page faults
- Number of disk blocks touched

In some applications, resource utilization fluctuates even in the case where the application truly is quiescent. We test our ability to determine quiescent points with our statistics by first writing small test-case applications that consume one or more system resources. We then explore detection of garbage collection in Java, and regular X11 application resource usage with emacs running against an X server. These applications consume resources even when they are quiescent.

Performance and resource consumption is a concern in nearly all operating system and distributed system research. The tools we have created to identify quiescent states may have broader application in characterizing the resource requirements of various servers. In turn, this could allow system administrators or automated systems to identify combinations of processes that would run well on a single server.

We ask the following questions during the evaluation:

- Using our techniques, can we see a significant difference between active and inactive applications?
- Can we detect post-processing in any of the applications?

In Section 2 we survey related work. In Section 3 we discuss our system design. Section 4 describes our testing methodology. Section 5 gives the results of our tests. Section 6 enumerates deficiencies in our current design and gives plans for future work. Section 7 concludes.

2 Related Work

2.1 Quiescence

Kramer and Magee [13] provide a fundamental definition of quiescence as the property that “[a] node is not within a transaction and will neither receive nor initiate any new transactions". They rely on application nodes to become quiescent when instructed to, and provide an algorithm for moving a group of nodes into a quiescent state within a bounded period of time. However, they cannot guarantee at runtime that any particular node has in fact complied with instructions.

Bidan et. al. [3] follow [13], but attempt to minimize impact on concurrency by reducing the number of nodes

[1] We define a quiescent process as one that is doing no processing.

that must be made quiescent at any one time. They do this by restricting their technique to systems built from multi-threaded nodes, and passivating only a part of each node. Their technique additionally loses some degree of generality by ignoring problems that would be caused by the presence of procedure call cycles.

Both PODUS [18] and ABACUS [1], which operate at the procedure granularity level, recognize quiescence by determining when a procedure is inactive. This method is not directly comparable to ours as it is more fine grained than our process based approach.

Gagliardi, et. al., [9] require that a module send notification, to a third party component, of any IPCs it is issuing. In this way, the third party can determine at any point in time which of the modules it is managing are quiescent. This system works on the assumption that when a module has no outstanding IPCs, it is indeed quiescent.

Tewksbury and Moser et. al., [19] [15] [16] define object quiescence as any state in which the object is not executing any of its methods. They outline two reasons why they need object quiescence in their system for performing live upgrades. First, their method involves object replication and consistent extraction of the state from multiple copies of a replicated object. Second, their method provides the ability to perform arbitrarily complex upgrades in which multiple objects are changed. They guarantee consistency by providing an atomic switch-over operation, which mandates the quiescence of all objects involved in the switch-over. They might benefit from our method of determining quiescence, as they state explicitly that they "cannot ensure the quiescence of an object which contains one-way, or asynchronous, operations unless the completion of a one-way operation is signaled somewhere to the infrastructure".

Hauptmann and Wasel [10] require that the programmer specify points of quiescence explicitly. In POLYLITH, Hofmeister and Purtilo [11] provide additional mechanisms for exporting the state of a process even if it is in the midst of a procedure call and even if procedure calls can be nested to arbitrary levels of depth. By not requiring that that programmer identify points in the code where a process is quiescent they reduce the conceptual burden. However, in order so that POLYLITH can restore the state of the process in another location, the programmer still must name explicit reconfiguration points.

Our concept of quiescence is closely related to those just discussed, i.e., a point in which it is safe to export the state of some module, thread, or procedure, etc. However, it is important to note that different authors in various communities may use the term quiescence in very different lights. For example, McKenney and Slingwine [14] are interested in quiescent threads in order to construct a lightweight locking policy for shared data. However, they more narrowly define a quiescent state of a thread as one in which the thread holds no locks for any shared data objects.

2.2 Resource Consumption

Chang et. al. [4] propose user-level sand-boxing to constrain application resource consumption. They use a combination of API interception and existing kernel statistics to limit the application's CPU, memory, and network usage. This paper also describes an algorithm to quickly determine that a process is waiting. This algorithm may be able to discover quiescent processes. We will be adding kernel statistics to track many network and disk resource consumption on a per application basis. This should reduce the need for API interception. In addition, our approach will allow system administrators to track resources for all of the applications on a system, not just the ones that are running in a sand-box.

Chang and Karamcheti [5] propose using these sandboxes to gather information to allow automatic run-time adaptation of applications. By gathering statistics for all of the processes on the system and making these statistics available to all applications, we may allow these applications to make better decisions.

Druschel and Banga [7] point out that the CPU resources consumed by network traffic are generally not associated with the application that caused the traffic. They propose a systems, called lazy receiver processing, to correct this problem and to provide better server throughput under high loads. We will not be able to address this concern in our implementation. As a result, we may overestimate the CPU utilization of some processes on systems that are experiencing high network traffic.

Banga et. al. [2] point out that many server systems need to track resources at a finer granularity than the process and thread level. They propose a new operating system abstraction called a resource container. They extend the lazy receiver approach to associate the kernel processing for network traffic with the appropriate container. Since quiescence happens at a process level, we do not need the fine-grained resource tracking that resource containers provide. However, we may be able to use resource-containers in the future to discover which clients are preventing processes from entering a quiescent state.

Jones et. al. [12] have implemented a system that tracks resources at the provider. Applications contact a planner to reserve resources. The planner, in turn contacts the provider. The provider tracks the actual resource utiliza-

tion, and contacts the planner when activities exceed their allocations. Our system does not need to reserve or allocate resources. However, we will be instrumenting the Linux kernel near the resource providers to track resource utilization. By tracking the extent of these modifications we could estimate the difficulty of porting this planning system to the Linux kernel.

Compton and Tennenhouse [6] attempt to gather CPU statistics to allow individual applications to shed load. When the system was implemented, it was difficult for applications to obtain system load statistics and the CPU utilization for the current process. In addition, when multiple applications were running on a system one application tended to consume the majority of the resources. At the end of the paper, they list information that would help the applications to make better decisions. We will gather statistics for many of the items on this list.

3 Design

Our test system can be divided into three primary components: the kernel modifications to implement resource counters, the process that gathers these counters at fixed intervals (QAnalyzer), and the test applications. In this section, we discuss the kernel modifications and QAnalyzer. In the next section, we describe the tests and test applications.

3.1 Kernel Enhancements

We discovered that there are several useful resource counters in the current Linux kernel. The number of page faults is already available through /proc/<PID>/stat. The minflt parameter gives the number of page faults that didn't result in a page being loaded into memory, and the majflt parameter gives the number of faults that required page loading. The CPU time is also available through this interface in the utime and stime parameters.

These parameters are stored in the task_struct. There are no parameters for the number of disk blocks touched, number of bytes of network traffic, or the number of network packets sent and received.

We have added the following values into the task_struct:

dskblks	Number of disk blocks accessed
cdskblks	Number of disk blocks accessed by children
roctnet	Number of octets (bytes on most systems) received for this process
rpktnet	Number of packets received for this process
soctnet	Number of octets sent for this process
spktnet	Number of packets sent for this process
croctnet	Number of octets received for the children of this process
crpktnet	Number of packets received for the children of this process
csoctnet	Number of octets sent for the children of this process
cspktnet	Number of packets sent for the children of this process

We encountered several issues as we added the new counters to the kernel. The c* counters in the parent process are updated when the child exits: parent->cminflt = child->minflt + child->cminflt. The man page for the /proc filesystem implies slightly different behavior. We have elected to implement the new counters so that they behave like the existing counters in the kernel.

We have added a new file, called "resuse", into the directories for each process. resuse contains the PID, minflt, majflt, utime, and stime parameters, along with the new parameters mentioned in the table above. All of these parameters are represented as space delimited 32 bit unsigned integers encoded in the ASCII character set. A single newline comes after the last parameter. We duplicate some of the parameters that are found in other proc files to make the implementations of QAnalyzer easier to build. Seeking resuse to offset 0 will cause the kernel to refresh resuse with new data.

Threads in Linux are assigned PIDs. As a result, each thread may have its own task structure in the kernel. There is no easy way to distinguish threads and regular processes from user space. The ps application does suppress threads by comparing the command line arguments and virtual memory size of each process to its parent. When all of these parameters match, ps assumes that the process is a thread. QAnalyzer should not have to duplicate this hack, so we added code to place the statistics for child threads on additional lines in the resuse file, following the line for the parent. However, there is a 4k limit on the file size in the kernel. This effectively limits the number of threads that can be returned to about 16. When this happens, the

last line of the file is "*". QAnalyzer can detect this condition by examining the return value from scanf. This modification causes the proc filesystem code to lock the task list and scan every task in the list to discover the threads. Other code in the Linux kernel uses the same approach to locate threads.

These modifications allow QAnalyzer to total the resources used for the process. The resources consumed by threads are counted as resources used by children, so QAnalyzer can accurately measure resource consumption for processes with many short-lived threads by examining the c* values of the initial thread of the process.

We discovered a bug in the counter handling in the Linux kernel. If a process is killed, the process's counters are added to its parents as expected. However, if the process has threads, only the time for the initial thread is added to the parent; the other threads are added to the $init$ task. This problem does not occur if threads exit before the parent process dies. We suspect that this bug is caused by the order of operations in Linux: when the signal is delivered to the parent, the threads are re-parented to the init process. When the signal is delivered to the threads, they die and add their time to the init process. This bug should be fixed to make the c* counters more reliable, but we did not attempt to fix it during the duration of this project.

We wanted to be certain that our patches were localized to a small set of files in the kernel. We especially wanted to avoid patching specific device drivers and filesystem, since patching in these areas may cause the counters to fail when the configuration of the test system changed. As a result, we have placed the increments for the disk based counters in the generic_file_read and generic_file_write functions.

Some specific filesystems do not use the generic_file routines. As a result, we will not see operations to these filesystems. In the 2.4.19 kernel, the following filesystem are affected: pipe, coda, devfs, hfs, ncpfs, ntfs, proc, intermezzo, jbd, and the openpromfs. In the last three, we will detect reads, but not writes. We suspect that some of these filesystems do not support writing.

We made small changes to the following files to implement the new counters:

fs/proc/array.c	Add code to return the values of the counters
include/linux/sched.h	Add the new counters to the task structure
kernel/exit.c	Increment the parent's statistics when a task exits
kernel/fork.c	Initialize the new statistics to zero
mm/filemap.c	Increment the counters for file operations
net/socket.c	Increment the counters for network operations

3.2 QAnalyzer

We have implemented QAnalyzer in C. QAnalyzer periodically gathers and accumulates observations, which it builds from the kernel enabled statistics. QAnalyzer employs interval timers to ensure that delays in sampling do not accumulate as QAnalyzer runs.

QAnalyzer records timestamps in its logs, and many of our test applications write output messages indicating when they are active and when they are idle. Since both processes use the same clock, we are able to compare our application's behavior with the information in the QAnalyzer logs. This would also allow us to detect when multiple intervals collapse into a single observation, though this has not occurred during our testing.

While refining our test plan, we realized that our planned statistical model could be implemented as a confidence interval for each of the resources we are monitoring (perhaps with an additional overall model to evaluate the significance of multiple resources varying together). This would be less expensive than, but equivalent to, hypothesis testing for each observation, and would allow us to run our quiescence analysis in real time in QAnalyzer. The reason for the efficiency improvement is that the components of each observation would merely need to be checked against their respective resource confidence interval.

4 Evaluation Framework

From the outset we had planned to empirically model the resource consumption of a quiescent process. We had expected that a process's resource consumption would fluctuate even while the process idled. This modeling stage became simplified when it became clear that many UNIX

processes consume no resources while idle. See the results section for a discussion of this topic.

We have written a suite of four test applications to consume specific resources. $spinner$ consumes user mode CPU time by looping over an increment operation on a variable. $client$ continually sends data via a socket connection to $server$ which reads the data. $reader$ reads a large file from disk. By monitoring the resource consumption of each of these programs, we are able to confirm that the resource counters in the kernel are working.

We also monitor the resource consumption of a number of real-world applications, including tcsh, emacs running against an X server, and the Java Virtual Machine (JVM). In doing so, we show that a single simple model of quiescence is not sufficient for every class of application.

Our JVM tests are an attempt to validate whether we can detect offline or post-processing activity not directly initiated by our own programs. By testing this, we demonstrate that internal knowledge of program behavior is not necessarily enough to determine quiescence if, for example, the program is tied to other processes.

The QAnalyzer timestamps aid us in evaluating the effectiveness of our approach by allowing us to compare observations with known points of quiescence, which we have our test programs signal upon entry and exit.

All along it has been an important goal to ensure the reproducibility of our results. For this reason we collected multiple datasets for test run of each process we monitored, and we varied QAnalyzer's sampling frequency for each of the many test runs we performed.

We ran these tests on RedHat Linux 8.0 and a modified 2.4.19 kernel. The test system had one 200 MHZ Pentium Pro processor, with 8k of Level 1 I cache, 8k of Level 1 D cache, 256K of Level 2 cache, and 256MB of 70ns RAM. ECC and write-back caching are enabled. The disk was a 4G Seagate ST34371W, connected to an Adaptec 2940 UW SCSI controller. The drive was partitioned with a 1G swap partition and 3G ext3 partition which is mounted as the root. The system was connected to the network, but is sitting behind a NAT/router.

This system initially exhibited some stability problems. When we ran non-stop kernel compiles with the unmodified RedHat kernel from the 8.0 distribution, the system spontaneously rebooted after about 12 hours. So, we first tested the modified kernel on a different system to avoid running into the known stability problems on our test system. The kernel ran for 18 hours while doing constant kernel compiles and running cat /proc/*/resuse once per second. No crashes, faults, or memory leaks appeared. Since moving the modified kernel to the test system, the test system has been entirely stable.

5 Results

Our experimental results show that certain quiescent processes consume no measurable quantities of the resources we track. For example, we were unable to detect any resource consumption in an idle UNIX tcsh shell process. However, this fact increases our confidence that when we observe fluctuations in resource consumption, we are, in fact, viewing an active process.

We present the results of controlled tests, in which we monitor the resource consumption of programs we wrote specifically to test our ability to track individual resource use.

We are able to distinguish between periods of activity and periods of quiescence in a Java process we wrote. We were also able to find an example of process post-processing, in the concurrent garbage collector of Sun's Java Runtime Environment (JRE), version 1.4.1.

5.1 Correctness of Method - Targeted Tests

Figures [1,2,3,4] show the results of our test suite of applications. Each of these tests was designed to test a specific type of of resource; our results show that we are able to detect this resource consumption.

Setting out in this project we hypothesized that quiescent UNIX processes exhibit a low, but non-zero, level of fluctuating resource consumption. As can be seen in the charts in this section, the quiescent shell process ($tcsh$), consumes none of our tracked resources at any rate we can detect. The result of this realization is that our model of a quiescent process is very simple: if we detect that a process is consuming any amount of resources, it is not quiescent; if a process is not consuming resources, is it quiescent.

5.2 Quiescent Processes that Consume Resources

When we expanded our tests to include an idle emacs process, we discovered that it did consume resources. First, it sent periodic traffic to the server to flash its cursor. In addition, it received events for mouse events over the emacs window.

We utilized Xnee to implement our test on emacs running against X server. Xnee records X11 protocol data such as Xevents from the local X server into log files and later, uses those recorded log files to replay those events.[2] The keyboard events are sent to whichever win-

[2]While we specified that Xnee record only the keyboard events, Xnee is also capable of capturing mouse events.

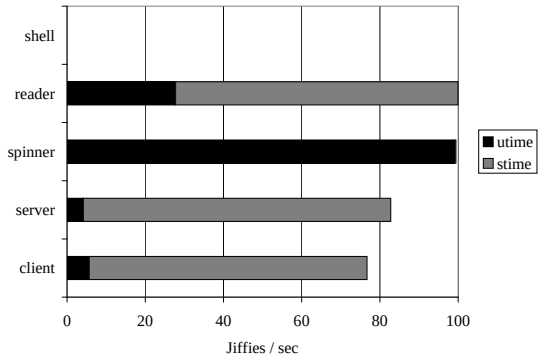

Figure 1: The average number of jiffies per second that each process was scheduled. The bars are stacked in this graph. There are 100 jiffies per second on the test system.

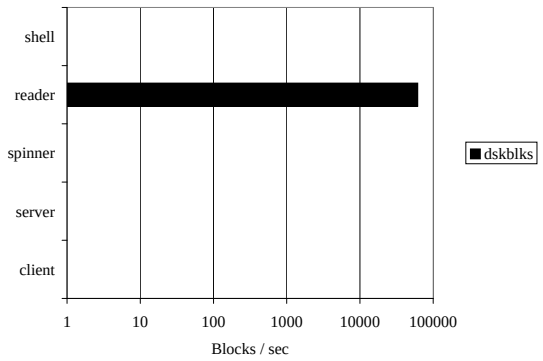

Figure 2: Graph of the number of blocks / second on a logarithmic scale.

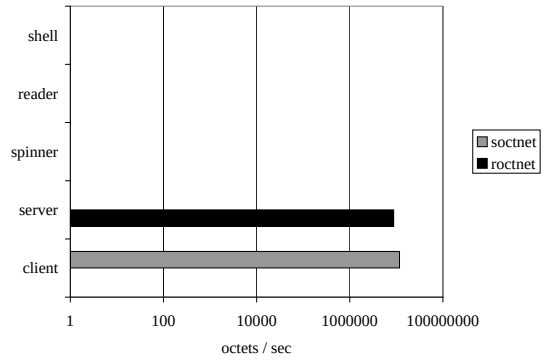

Figure 3: Graphs of the number of octets transmitted and received per second on a logarithmic scale.

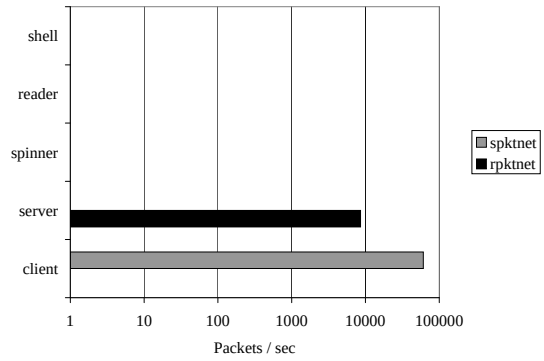

Figure 4: Graph of the number of packets sent and received per second on a logarithmic scale. The packet counters currently count system calls, not network packets.

dow is in focus; in our case, we clicked on the emacs window to bring it to the front in order to direct keyboard events there. In addition, we modified Xnee source code to output the system time stamps right before the keyboard events are replayed, so that we can compare periods of recorded program activity against the period of resource consumption output by the QAnalyzer.

Figures [5,6,7] show correspondences between known program activity periods and resource consumption, and illustrate the contrast between the resource consumption during idle periods with consumption during typing. It is interesting to note in Figure 6 that even when the program is quiescent, there is small, but non-zero, network resource consumption. The network bytes sent are due to the blinking cursor that appears in the emacs windows. Under this circumstance, it is still trivial to discern quiescence from activity by looking at the number of bytes sent, without even examining the other recorded resources such as the number of bytes received or utime.

5.3 Detecting Activity

Going into our evaluation, we shared the collective hypothesis that the Java Virtual Machine (JVM) garbage collector would run in the background, if given the chance. To test this, we designed and built a Java program that allocates memory, nullifies its reference to the memory, goes to sleep for a fixed interval of time, and then begins again. The program creates a time stamp when the process goes to sleep and another when it wakes. Comparing these time stamps to the time stamps corresponding to the QAnalyzer observations allows us to map observed activity to know periods of quiescence.

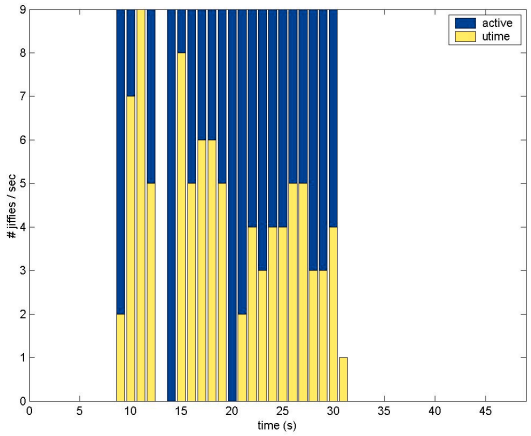

Figure 5: User mode CPU activity, showing with periods of activity (typing) indicated.

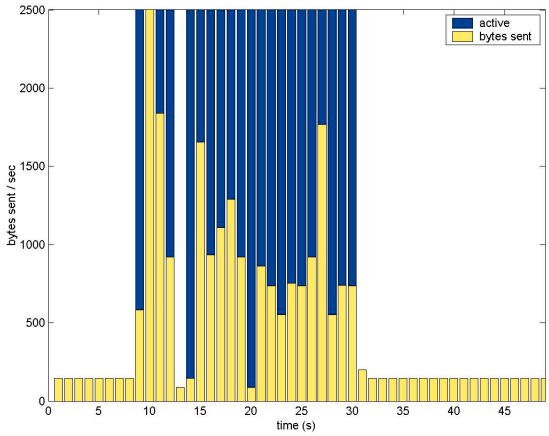

Figure 6: Sent bytes per second with activity (typing) period indicated. In is interesting to note the steady rate of bytes per second during inactive periods. This stream corresponds to the data required to notify the X server of the blinking cursor.

We monitored this program using version 1.3.1 of the Java Runtime Environment (JRE). Our QAnalyzer data indicated no activity outside of time periods that could not be accounted for by our programmed activity in our Java program. Somewhat surprised by this result, we began to look for explanations and found an article on "javacoffeebreak.com" [8] that appeared to confirm our results, stating: "In the Sun literature you'll find many references to garbage collection as a low-priority background process, but it turns out that this was a theoretical experiment that didn't work out. In practice, the Sun garbage collector is run when memory gets low." Certain that someone must have implemented a version of this "theoretical experiment", we began to search for alternate garbage collectors.

A Sun article [17] describes additional garbage collection policies added to Java version 1.4.1. Of particular interest is the new concurrent garbage collector, which runs in a background thread concurrently with the application thread(s). This collector must be explicitly enabled, which can be done with the *Xconcgc* runtime environment flag.

The results of analyzing our Java program again, using this concurrent garbage collector, are shown in Figure 8. These results still fail to show any JVM post-processing, as all observed activity can again be accounted for by the known activity of our program. Each sampling period containing observed activity (*utime, stime, and minflts*), also contains programmed activity, as determined by the direct output of the Java program (*activity-start, activity-end*).

While these results do not confirm our hypothesis that the JVM will garbage collect when the application thread(s) is quiescent, they do validate our primary hypothesis that QAnalyzer can use the kernel output statistics to recognize periods of programmed process activity. For confirmation of concurrent Java garbage collection, see the section of this paper concerning detecting post-processing.

5.4 Detecting Post-processing

We created a second Java program in which we were able to catch garbage collection activity in the process during time in which the code we wrote was supposed to be sleeping/quiescent. For this test we again used JRE 1.4.1 with the concurrent garbage collector.

While the post-processing activity, graphed in Figure 9, appears minimal, we are certain, for two reasons, that it is not an outlier. First, we repeated the test many times, each time getting very similar results. Second, we enabled JVM garbage collection verbose logging with the *verbose:gc* switch. The corresponding output showed garbage collection activity continuing for many seconds after the "Sleeping at:" message, seen in Figure 10, had been printed.

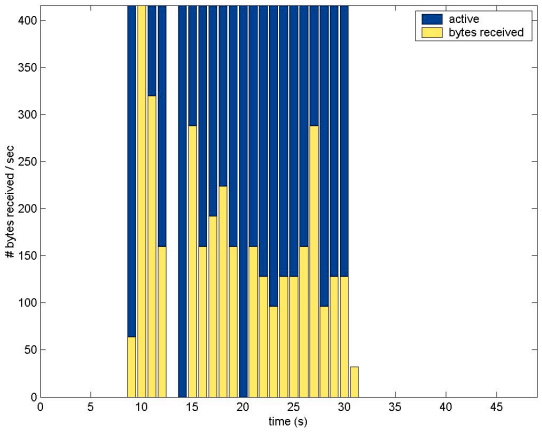

Figure 7: Received bytes per second with activity (typing) period indicated.

6 Future work

This project focuses on quiescence. However there may be broader applications for the resource counters that we are adding to the kernel. We have made several simplifications to the counters which would need to be addressed before broader application is practical:

- We should improve the resolution of the CPU counters in the system. The current counters have a resolution of .01s, which may not be high enough to detect activity some applications.
- We should implement LRP[7] to ensure that the processing time for network activity is attributed to the correct process.
- The disk counters should differentiate between accesses that hit the cache and accesses that result in a block read. This distinction is important from a resource management perspective, but either type of access indicates that the process is not quiescent.
- We should differentiate between network traffic that is local to the system and network traffic between systems.
- The packet counters currently count the number of system calls. However, a single system call could result in many packets being sent over the wire. Implementing packet counting will greatly expand the scope of the patches that we will make in the kernel.
- The network counters do not count network packets that do not carry data.
- We should create a system that can differentiate between TLB faults that are generated by TLB overflow and TLB faults that are generated by new requests for a page of memory. These modifications could be expensive in terms of system performance and will involve extensive modifications to the kernel.

The applicability of our method of detecting quiescence may not work for certain classes of applications, particularly those that consume substantial resources while quiescent. One test that we looked at was running a browser against the X server. When the browser is displaying pages with many animated GIFs, the network resources utilization is substantial. Based on network utilization alone, it might be difficult to discern quiescence from actual program activity. We would need to further examine this and similar classes of applications.

7 Conclusions

We have implemented simple kernel modifications that provide resource counters for unmodified user processes. We have also written QAnalyzer, a program that takes periodic samples from these counters and generates a log file. We demonstrate that different classes of UNIX processes would require different models of quiescence. Our testing results with this system indicate that the counters work, and that our method is able to detect activity in a wide range of processes.

8 Acknowledgments

We would like to thank Tim Halloran, who helped us to find and configure the test system.

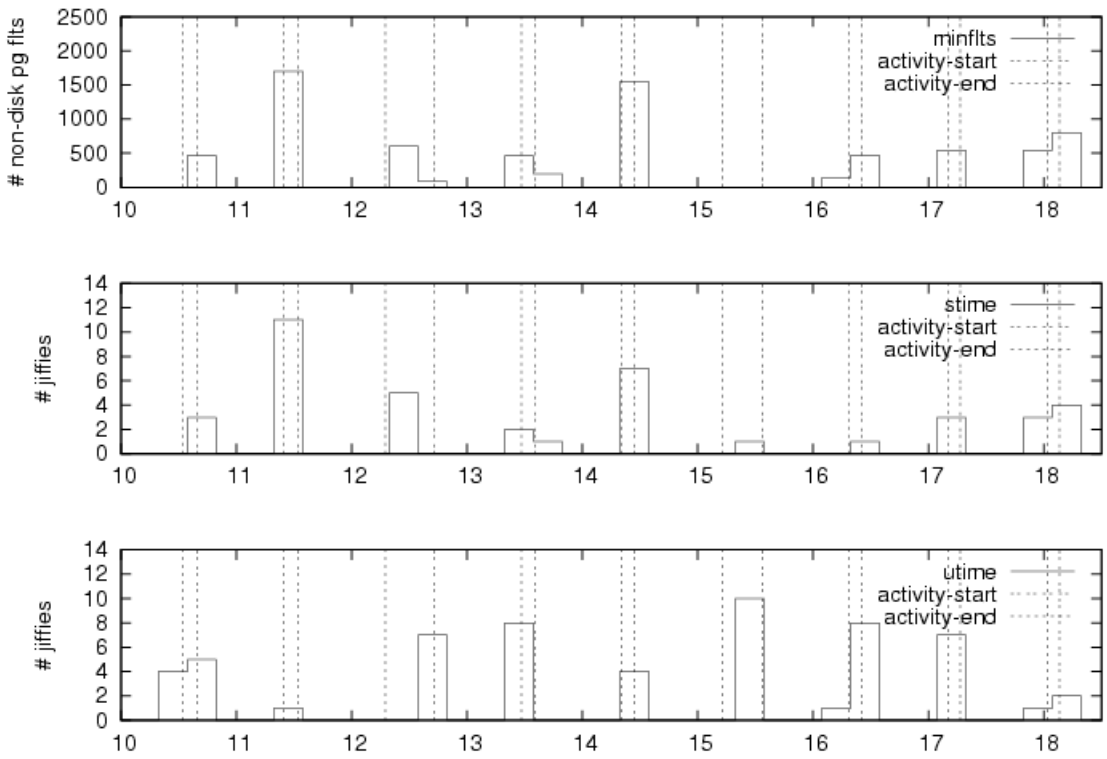

Figure 8: Periods of designed process activity are indicated between the *activity-start* and *activity-end* vertical dashed lines. The process spends more time sleeping than working, in an attempt to coax the garbage collector into motion. However, every period of observed activity can be accounted for by the designed activity. QAnalyzer is sampling at quarter-second intervals.

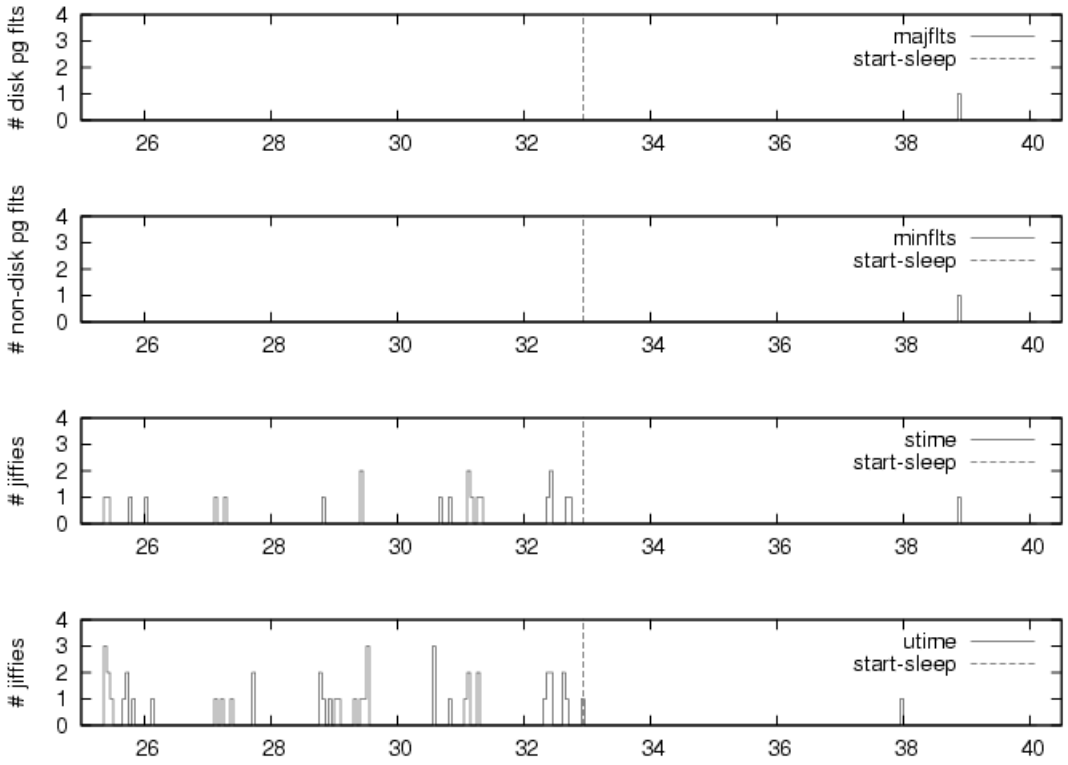

Figure 9: **Post-processing.** As can be seen at approximately 38 and 39 seconds, the garbage collector causes observed activity, despite the fact that the Java test program should be sleeping. Note the greater resource consumption before sleeping, which is due to our programmed activity in addition to garbage collection, while consumption during sleep is due only to non-programmed activity. QAnalyzer is sampling at one-twentieth of a second intervals.

```
private static int NUM_OBJECTS = 100000;
...
System.out.println("Starting at " + (new Date()).getTime());

{
    GarbageNew first = new GarbageNew();
    GarbageNew recent = first;
    int i = 0;

    try {
      for (; i<NUM_OBJECTS; i++) {
         recent.setNext(new GarbageNew());
         recent = recent.getNext();
      }
    }catch (OutOfMemoryError oe) {
         System.out.println("hit memory boundry at object "+ i);
    }

    recent.setNext(first);
}

System.out.println("Sleeping at " + (new Date()).getTime());
try{
    Thread.sleep(5000);
}catch(InterruptedException e){
    System.err.println("error sleeping:");
    return;
}
```

Figure 10: Code from the Java program that elicited post-processing in the garbage collector. We allocate a large number (100,000) of small objects and chain them together, finally creating a circular linked-list. We print out the time the program goes to sleep, so that we can determine which of the QAnalyzer observations occur while we are quiescent. In our test we limited the JVM maximum heap size to 1 megabyte, using the *Xmxn* flag.

References

[1] AMIRI, K., PETROU, D., GANGER, G. R., AND GIBSON, G. A. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX 2000 Annual Technical Conference* (2000), pp. 307–322.

[2] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation* (1999), pp. 45–58.

[3] BIDAN, C., ISSARNY, V., SARIDAKIS, T., AND ZARRAS, A. A dynamic reconfiguration service for CORBA. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems* (1998), IEEE Computer Society Press, pp. 35–42.

[4] CHANG, F., ITZKOVITZ, A., AND KARAMCHETI, V. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium* (2000).

[5] CHANG, F., AND KARAMCHETI, V. Automatic configuration and run-time adaptation of distributed applications. In *HPDC* (2000), pp. 11–20.

[6] COMPTON, C. L., AND TENNENHOUSE, D. L. Collaborative load shedding for media-based applications. In *International Conference on Multimedia Computing and Systems* (1994), pp. 496–501.

[7] DRUSCHEL, P., AND BANGA, G. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *Operating Systems Design and Implementation* (1996), pp. 261–275.

[8] ECKEL, B. Thinking in Java Excerpt : A bit about garbage collection. `http://www.javacoffeebreak.com/articles/thinkinginjava/abitaboutgarbage%collection.html`, December 2002.

[9] GAGLIARDI, M., RAJKUMAR, R., AND SHA, L. Designing for evolvability: Building blocks for evolvable real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium* (1996), pp. 100–109.

[10] HAUPTMANN, S., AND WASEL, J. On-line maintenance with on-the-fly software replacement. In *3rd International Conference on Configurable Distributed Systems* (1996), pp. 70 – 80.

[11] HOFMEISTER, C., AND PURTILO, J. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *Proceedings the 13th International Conference on Distributed Computing Systems* (1993), pp. 101–110.

[12] JONES, M., LEACH, P., DRAVES, R., AND J.S. BARRERA, I. Modular real-time resource management in the Rialto operating system. In *Proc. 5th Workshop on Hot Topics in Operating Systems* (1995), pp. 12–17.

[13] KRAMER, J., AND MAGEE, J. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering 16*, 11 (1990), 1293–1306.

[14] MCKENNEY, P., AND SLINGWINE, J. Read-copy update: Using execution history to solve concurrency problems. In *10th IASTED International Conference on Parallel and Distributed Computing Systems. (PDCS.98)* (1998).

[15] MOSER, L., MELLIAR-SMITH, P., NARASIMHAN, P., TEWKSBURY, L., AND KALOGERAKI, V. Eternal: Fault tolerance and live upgrades for distributed object systems. In *Proceedings of the DARPA Information Survivability Conference* (2000), pp. 184–196.

[16] MOSER, L. E., MELLIAR-SMITH, P. M., NARASIMHAN, P., TEWKSBURY, L., AND KALOGERAKI, V. The eternal system: An architecture for enterprise applications. In *International Enterprise Distributed Object Computing Conference* (University of Mannheim, Germany, 1999), pp. 214–222.

[17] NAGARAJAYYA, N., AND MAYER, S. Improving Java[tm] Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1. `http://wireless.java.sun.com/midp/articles/garbagecollection2/`, December 2002.

[18] SEGAL, M., AND FRIEDER, O. On-the-fly program modification: Systems for dynamic updating. *IEEE Software 10*, 2 (1993), 53– 65.

[19] TEWKSBURY, L., MOSER, L., AND MELLIAR-SMITH, P. Live upgrades of CORBA applications using object replication. In *Proceedings of the IEEE International Conference on Software Maintenance* (2001).