NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
NWU-CS-04-28
February 20, 2004

A Framework And Toolkit For Understanding User Comfort With Resource Borrowing

Ashish Gupta, Bin Lin, Peter Dinda

Abstract

Resource borrowing is a common underlying approach in grid computing and thin-client computing. In both cases, external processes borrow resources that would otherwise be delivered to the interactive processes of end-users, creating contention that slows these processes and decreases the comfort of the end-users. How resource borrowing and user comfort are related is not well understood and thus resource borrowing tends to be extremely conservative. To address this lack of understanding, we have developed a sophisticated distributed application for directly measuring user comfort with the borrowing of CPU time, memory space, and disk bandwidth. Using this tool, we have conducted a controlled user study with qualitative and quantitative results that are of direct interest to the designers of grid and thin-client systems. In this report, we describe the system in detail and related implementation issues. We also discuss other factors related to the project and also details of the controlled study. A separate paper documents the results of this controlled study in more detail.

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, and EIA-0224449. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

Keywords: user studies, desktop grids, cycle stealing, applications studies, grids, I-O and file systems, performance modeling, resource management or scheduling

Contents

1 Introduction **3**
1.1 Studying the user comfort-resource borrowing relationship . . . 3
1.2 Factors affecting the results . . . 4
1.3 The chain of events leading to user discomfort . . . 5
1.4 Related work . . . 5

2 System design **7**
2.1 Monitoring . . . 7
2.1.1 The PDH library in Windows . . . 8
2.1.2 Monitoring processes . . . 8
2.1.3 Recording the context and context history . . . 8
2.2 Emulating resource borrowing . . . 9
2.2.1 The testcase model . . . 9
2.2.2 Testcase generation . . . 9
2.2.3 The CPU exerciser . . . 10
2.2.4 The memory exerciser . . . 15
2.2.5 The disk exerciser . . . 15
2.2.6 The network exerciser . . . 17
2.3 User feedback techniques . . . 18
2.4 The client/server architecture . . . 19
2.4.1 The protocol . . . 20
2.5 The client interface . . . 20

3 Controlled study **22**
3.1 Terminology used . . . 22
3.2 The experimental setup . . . 22
3.2.1 Hardware . . . 22
3.2.2 Invitation and interaction with subjects . . . 23
3.2.3 Testcases . . . 24
3.2.4 Overview of factors used in the controlled study . . . 25
3.3 Control study results . . . 25
3.3.1 What level of resource borrowing leads to user discomfort for a significant fraction of users? . . . 25
3.3.2 How does the level depend on which resource or combination of resources is borrowed? . . . 27

3.3.3 How does the level depend on the user's context? 29
3.3.4 How does the level depend on the user, factoring out context? 29
3.3.5 How does the level depend on the time dynamics of resource borrowing? 31

4 Programming issues **33**
4.1 System information monitoring 33
4.2 Displaying an __int64 value in Borland C++ 33
4.3 Handle GUI only in the main thread 34
4.4 Changing thread priorities 34
4.5 Working with global hotkeys 34
4.6 Programming the system tray icons 35
4.7 Client-server problems 35
4.8 System detection code 35
4.8.1 CPU detection 35
4.8.2 Memory detection 35
4.8.3 Disk detection 36
4.8.4 Network detection 36
4.8.5 System detection 36
4.8.6 Operating system detection 36
4.9 Generating the GUID 36

5 Conclusions and future work **37**
5.1 Advice to implementors 37
5.2 Conclusions 37
5.3 Internet-wide study 38
5.4 Possible interaction between different exercisers 38
5.5 Resource control using human feedback 38

A Publicity **39**
A.1 Inviting controlled study participants 40

1

Introduction

Recently, many applications have emerged which employ a new method of distributed computing: utilizing the vast resources of desktop PCs around the world, taking advantage of the fact that their resources are usually under-utilized [18, 6, 1] by borrowing them for their own computations. Examples in scientific computing include Condor [17, 10], Entropia [4], SETI@Home [26], Protein Folding at Home [16], DESChall [5], and the Google Toolbar [11]. Examples in peer-to-peer content distribution systems include commercial tools such as Kazaa [24] and Gnutella [20], as well as academic projects [25, 21, 13, 12]. Some of these systems are deployed on millions of Windows hosts. This model is promising because of the large number of Desktop PCs in use.

Current systems are very conservative in how they borrow resources because if they cause the user to feel the machine is slower than is desirable, the user is likely to disable them. For example, the default behavior in Condor, Sprite [7], and SETI@Home is to execute only when they are quite sure the user is away, when the screen saver has been activated. Other systems run at a very low priority, or they simply ask the user to specify constraints on resource borrowing, something that few ordinary users understand. If less conservative resource borrowing does not lead to significantly increased user discomfort, the performance of systems like Condor, Sprite and SETI@Home could be increased through techniques such as linger-longer scheduling [22].

Our work examines to what extent we can borrow a PC's resources, even while it is in use, without resulting in the slowdown we cause dramatically impacting the foreground user's interactivity with the system—how conservative must we be in resource borrowing? Can we borrow resources without affecting user's experience of the PC? If so, how much can we borrow? Which resources? Does the type of user matter? The type of PC? The application context? Surprisingly, there are few if any qualitative or quantitative answers to these questions, or even measurements that could be used to address the questions.

1.1 Studying the user comfort-resource borrowing relationship

In response to the lack of information, we have developed a system, the Understanding User Comfort System (UUCS). UUCS is a distributed Windows application similar to SETI@Home in design. A UUCS client emulates resource borrowing of CPU, memory and disk on the user's machine in a controlled manner encoded in a *testcase* provided by a server. The user operates the machine as normal, but if his interactivity gets affected, he may express discomfort by clicking on an icon or pressing a hot-key. Resource borrowing stops immediately if discomfort is expressed or when the testcase is finished. The point of discomfort, if

any, is recorded along with contextual information. By analyzing the results of applying a particular set of testcases to a particular set of users, we can qualitatively and quantitatively characterize user comfort. The entire process essentially consists of four steps:

1. Monitor current resource contention (Section 2.1)
2. Emulate resource borrowing (Section 2.2)
3. Get user feedback (Section 2.3)
4. Record the feedback levels along with other useful data

Repeating these experiments many times, with different users and scenarios, can give us a quantitative relationship between resource borrowing and user comfort. Based on this, a reliable model can be developed to help in intelligent borrowing of resources without affecting the user's interactivity.

A PC has multiple resources (CPU, disk, memory, network) that may be borrowed, cause slowdown, and thus affect user interactivity. How these resources affect the user may depend on many other factors like the application being used by the user etc. For example, a word processor application may not be significantly affected by disk borrowing in the background, whereas some disk-intensive applications like a backup utility may be much more sensitive. In this project we study the effects of CPU, disk, and memory resources. These resources can also be borrowed in a combined manner, and studying the combined effects is more challenging.

1.2 Factors affecting the results

Modeling user comfort can be complex. There is a cause–effect relationship between loss of interactivity and user comfort. This relationship may be affected by many factors, some of which we discuss in this section.

1. **Resource** : The user's sensitivity to resource borrowing may depend on the resource that is being borrowed, and it may depend on the application (or context, see below) being run. In a computationally intensive application such as a ray tracer, borrowing more memory may not have as much impact as borrowing CPU resources.
2. **Context** : The level of interactivity that we must maintain depends on the application that the user is currently using. While using a highly interactive application, such as playing Quake, small delays can contribute significantly to user discomfort. In more basic applications like word processing, the demands on interactivity and resources are lower, and thus more resource borrowing is possible compared to more interactive applications. While borrowing resources, one will have to keep user's current context in mind.
3. **System** : The hardware configuration details like the processor speed and amount of physical memory could impact the permissible amount of resource borrowing.
4. **User** : Discomfort caused by loss of interactivity may also vary over individuals. We want to study to what extent comfort levels vary across individuals and groups. These differences could also arise from user's computing experience thus shaping his expectations from a PC.

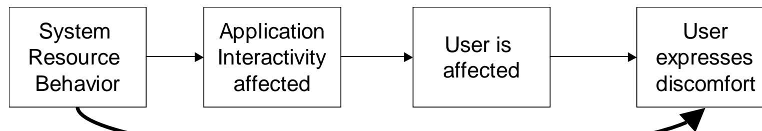


Figure 1.1: The chain of events leading to user discomfort

5. **Time dynamics** : There is an interesting phenomenon called the *Frog in the Pot*. If a frog is put in a pot of boiling water, it will immediately jump out. However, if we ramp up the temperature of the water slowly, the frog just sits there, not reacting to the change. Humans may also be able to tolerate slowly increasing resource borrowing more readily than abrupt changes. More generally, the reaction of the user could depend based on the manner, over time, in which the resources are borrowed. We use the concept of testcases to control resource borrowing in different ways to study time dynamics. Each testcase corresponds to a unique resource borrowing profile. The concept of testcases is discussed in more detail in section 2.2.1.

1.3 The chain of events leading to user discomfort

There are four steps in which changes in system resource usage propagate to user discomfort (Figure 1.1). We are interested in studying the relationship between resource borrowing and user discomfort—the end points of the chain. Our tools allow us to do stimulus-response studies where resource borrowing provides the stimulus for the user's response of discomfort. It is also feasible to develop models for other stages, however. Another interesting area of study would be to examine the relationship between interactivity and resource borrowing. This excludes the user from the system and would involve studying the action-response relationship mathematically. For example, in word processing, the time delays could be monitored between user-initiated actions and the corresponding response. Making such a monitoring and measuring system would be a challenging task in itself.

1.4 Related work

Work in related areas has concentrated on the impact of latency on user-perceived utility of the system [15, 8]. Computer-related frustration in humans has also been studied. A study at MIT [14] shows that user emotional states like frustration and anxiety can be significantly affected by the computer environment and effective measures can be taken to relieve them. It further shows by controlled experiments that effective measures do in fact change levels of user frustration. Another study at the University of Michigan [19] explored the design of effective feedback mechanisms for user frustration and how this data can be used for redesigning and adapting systems. Specifically, it discusses user interface design issues, how poor interfaces can frustrate user and incorporating user irritation feedback for better interface design. Users were asked to rate a variety of frustration feedback devices like squeezable mouse, web forms and a Frustometer consisting of a simple on-screen slider. Frustometer was found to be most accessible, being the most straightforward interface, and consequently seemed to require less cognitive feedback when reporting

feedback. This indicates that the user feedback mechanism needs to be kept simple. Within the systems community, related work has examined the performance of end-user operating systems using latency as opposed to throughput [9], and suggested models for interactive user workload [2]. However there exist no quantitative, empirical measurements that could be used to answer our questions.

2

System design

Here we describe the design and implementation of the Understanding User Comfort System. The goal of the system is to study the relationship between borrowing various resources in different ways and user comfort. Developing the system was a challenging task, requiring various levels of Windows systems programming and unexpected issues. The development environment chosen for the application is Borland C++ Builder Professional 6.0. As described in Section 1.1, the entire process involves four essential steps. To recapitulate:

1. Monitor contention and system information
2. Emulate Resource Borrowing
3. Get User Feedback
4. Record the feedback levels along with other useful data

We discuss these four steps in detail, especially the interesting portion of developing the resource borrowing modules, as well as monitoring all kinds of data we could extract from the system.

2.1 Monitoring

In this section, we explain what is monitored, the reasons for monitoring those items, and the techniques used to monitor them. Monitoring is important for getting a snapshot of the system when the user expresses his feedback. The monitored information helps us to analyze the resource contention which led to the user feedback, as well as analyze contextual information which could help us gain further insight into user behavior. The data which we currently monitor is:

1. Utilization values of the three system resources: CPU, memory and disk
2. Current running process list and list of their parameters
3. The current application which the user is interacting with (also referred to as the **context**)
4. A history of user contexts: This stores the foreground process usage history of the user. The following is an example of such a history:

```
Chicago ''L''.org: News Briefs - Mozilla Firebird,20
The Apple Store (U.S.) - Mozilla Firebird,10
Apple - Mozilla Firebird,1
Apple - iTunes - Mozilla Firebird,10
Apple - iTunes - Overview - Mozilla Firebird,6
```

The user context helps us to understand how interactivity depends on the application which the user is interacting with. In our study, we find that the acceptable level of resource borrowing greatly depends on the current user context. The process list helps us to examine details of the applications running, which could be helpful in some circumstances. For example, if the user is running a resource intensive application in the background, it may also affect the levels of resource borrowing.

Monitoring is done periodically, once per second. The snapshot is appended to an internal buffer that is later saved on to the results file when the user gives feedback.

2.1.1 The PDH library in Windows

PDH stands for Performance Data Helper and is an interface for querying performance data on Windows NT/2000/XP machines. Windows NT collects volumes of information such as process lists and the number of disk reads per second. PDH.DLL provides a nice API wrapper around all the work of obtaining and interpreting performance information. Using the concept of counters, the user can request that the PDH library monitor certain information that can be later queried. The system is described by various objects like CPU, Memory, Disk, etc, and each object has associated counters like CPU Utilization in the last 1 second, Disk read/write bytes in last 1 second etc.

In UUCS we monitor the counters for the CPU, disk and memory. PDH also allows us to monitor process information, but we found this to be inefficient. Instead, we used another method to monitor all the processes in the system.

2.1.2 Monitoring processes

We need a fast way to enumerate the entire process list, without any penalty on the system resources. We found an undocumented system call, `_ZwQuerySystemInformation()` (Section 4.1). This system call efficiently enumerates the processes along with related process parameters.

2.1.3 Recording the context and context history

We also need to acquire the current foreground application that the user is interacting with because the interactivity of the system is likely to depend on it.

Windows provides the following API functions for getting this information:

```
GetForegroundWindow()
GetWindowText()
GetParent()
```

For an active dialog, we follow the chain of parent pointers originating from the active dialog. This gives us a complete hierarchy of application windows that the user is interacting with.

Name	Description
$step(x,t,b)$	contention of zero to time b, then x to time t
$ramp(x,t)$	ramp from zero to x over times 0 to t
$log(r)$	logarithm of linear ramp with slope of r
$exp(\tau)$	exponential with time constant τ
$sin(x,f)$	sine wave of frequency f and amplitude $x+x$
$saw(x,f)$	sawtooth wave of frequency f and amplitude $x+x$
$expexp(\mu_{as},\mu_{ar},\mu_{ss},\mu_{sr})$	Poisson arrivals of exponential-sized jobs (M/M/1)
$exppar(\mu_{as},\mu_{ar},\alpha_{ss},\alpha_{sr})$	Poisson arrivals of Pareto-sized jobs (M/G/1)

Figure 2.1: Testcases.

2.2 Emulating resource borrowing

Resource borrowing is emulated by separate resource "exercisers" for the CPU , disk and memory resources. These exercisers run in the background and can borrow resources in a controlled manner. The control information is specified in exerciser testcases which describe in detail how each resource should be borrowed over time. In this section we describe the design of testcases, and the exercisers of CPU, memory and disk.

2.2.1 The testcase model

Testcases encode the details of resource borrowing for various resources. A testcase consists of a unique identifier, a sample rate, and a collection of exercise functions, one for each resource that will be used during the execution of the testcase (the *run*). An exercise function is a vector of values representing a time series sampled at the specified rate. Each value indicates the level of contention (the extent of resource borrowing, described in Sections 2.2.3, 2.2.5, and 2.6) for a resource at the corresponding time into the testcase. For example, consider a sample rate of 1 Hz, and the vector $[0, 0.5, 1.0, 1.5, 2.0]$ for the CPU resource. This exercise function persists from 0 to 5 seconds from the start of the testcase. From 3 to 4 seconds into the testcase, it indicates that contention of 1.5 should be created and subsequently 2.0 in the next second.

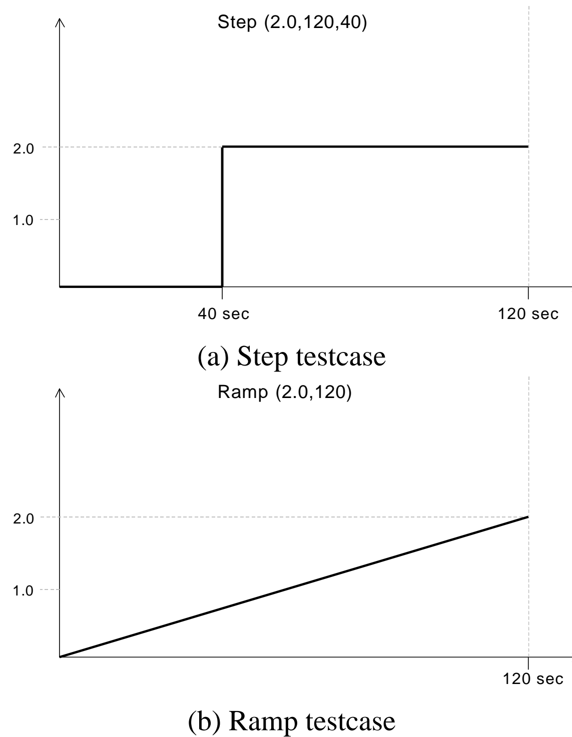
2.2.2 Testcase generation

Our testcase tools let us generate testcases of many different kinds, as summarized in Figure 2.1. In our controlled study, we use a small set of $step$ and $ramp$ testcases with different parameters. Figure 2.2 shows examples for $step(2.0, 120, 40)$ and $ramp(2.0, 120)$ respectively. We use a text database of all the testcases which is manipulated by a set of tools we have developed in Perl. These tools enable us to create, merge, append and plot different types of testcases. In our Internet-wide area study (Section 5.3), we currently have over 2000 testcases. We generate a set of testcases for each type, with different parameters and resource combinations. Multiple resources are borrowed simultaneously with different borrowing characteristics for each resource. For example:

```
make_testcase.pl tracefile 125  120 0.5 "cpu=(ramp 4)" "mem=(step 60 0.5)"
```

will generate a testcase (number 125) of 60 samples each for the CPU and memory resources (120 seconds, 0.5 Hz). The CPU contention will rise linearly from 0 to 4 during those 120 seconds, while the memory contention will rise to 0.5 at time 60s).

For the most past, Figure 2.1 is self explanatory, but some additional detail is necessary for the $expexp$ and $exppar$ testcases. $expexp$ simulates an M/M/1 queue of service rate 1 under processor sharing.



(a) Step testcase

(b) Ramp testcase

Figure 2.2: Step and ramp testcases.

The queue depth at the sample times is the output. This is the amount of contention that is produced at that time. The arrival rate begins at μ_{as} and increases at a rate of μ_{ar} over the duration of the testcase. Similarly, the mean job size starts at μ_{ss} and grows at a rate of μ_{sr} over the life of the testcase. $exppar$ follows the same idea, except here the job sizes are chosen from a Pareto distribution with α starting at α_{ss} and increasing at a rate α_{sr} over the life of the testcase.

Using our testcase generation tools, we created several testcase suites. Four suites consisting of ramps, steps, and blank testcases were used in the controlled study described later. In addition, we created a large suite consisting of all of the different kinds for our Internet-wide study. Because the $expexp$ and $exppar$ generators are not ergodic, we generate over 30 different instances for each set of parameters.

2.2.3 The CPU exerciser

Windows Scheduling and notion of CPU contention

Windows 2000/XP uses a priority driven, preemptive scheduling algorithm. A running job is assigned a time quantum, but size of the time quantum varies. Unlike Unix, what is scheduled is threads, not processes, and in general, no consideration is given to what process the thread belongs to. This means that a process with many runnable threads would receive much more CPU time than a process with only one runnable thread. Threads are assigned priorities from 0 to 31. Within a priority level, the Windows scheduler follows time-sliced, round-robin scheduling. Each thread is allocated certain time quanta for execution by the CPU. In Windows NT, the size of the quantum is around 12 ms.

We use contention average, a measure of contention, to create CPU contention through purely

application-level means. Under Windows, the contention average is the average of the numbers of threads in the operation system's ready-to-run queue. If large number of threads are contending for the CPU, the fraction of time allocated to an individual thread will drop and the thread will perceive slowdown. Thus, in our Understanding User Comfort Client, we load the CPU by creating additional threads for execution on the CPU. The larger the number of threads, the less time allocated to the user applications.

Note that Windows has built-in mechanisms to boost the priority levels of interactive applications and thus it may run at a higher priority than our threads for periods of time when its priority level is boosted. Our aim is to study how background loads affect the interactivity of the user with these applications.

The exerciser

The CPU exerciser runs in the background exercising the CPU according to the exercise function specified in the testcase. As noted previously, a testcase is a time series indicating the desired contention values sampled at a given rate. Currently we use the sampling rate of 1 Hz in all of our testcases. Our exerciser supports fractional contention as well, which is important in studying finer effects of CPU contention on interactivity of other applications. We can then control the CPU contention precisely. This means that a contention of 4.5 translates to 4 threads executing all the time and one thread intermittently computing and sleeping for equal amounts of time. Full threads are easier to design and are just infinite loops performing some mathematical operations. However, creating fractional contention on the CPU is more challenging.

To create fractional contention, we need to balance the compute times and sleep times (when the thread is not executing) precisely for a thread. Our approach to create fractional contention is to have two functions, `sleep()` and `work()`. `sleep()` sleeps for a given time x and `work()` executes on the CPU for the same time x. The value of x is computed by careful calibration as described below. We call these functions randomly with a probability distribution to match the amount of CPU contention required. To generate the contention of 0.6 we call the `work()` function with a probability of 0.6 and the `sleep()` function with a probability of 0.4.

However, there are issues involved with the `Sleep()` function in the Windows API. The `Sleep()` function is inaccurate for small values of time and the sleep time also depends on number of other threads executing on the processor. Therefore to create a precise contention on the CPU requires careful calibration of our `sleep()` and `work()` functions. We want these functions to consume equal amount of time per call. If this is ensured, then desired fractional contention can be created by calling these functions with the right probability values.

Calibration consists of two phases. The first phase is for `sleep()` and the second phase is for `work()`. In the first phase, we need to first find out reliable values of sleep. The Windows API `Sleep()` functions takes a parameter which indicates the time to sleep, in milliseconds. In practice, small values of this parameter do not result in accurate sleep times. Therefore in calibration we search for a reliable `Sleep()` time parameter which sleeps for the time specified. We start with a small time parameter x and record the time taken for the Sleep(x) to return, say y. For this, we use high resolution timing functions present in Windows API, `QueryPerformanceCounter()` and `QueryPerformanceFrequency()`. Using these functions we can get timing up to microsecond level of precision. If the difference in x and y is large, we increment x by 1 and try again. We select a value for which the error in `sleep()` function is within a small percentage of the Sleep time requested. In practice we found that this the right value of x depends on the OS as well as the underlying machine configuration.

In the second phase, we need to find out the number of loop iterations (of a simple mathematical computation), which will keep the CPU busy for the same time as the sleep parameter calibrated in the

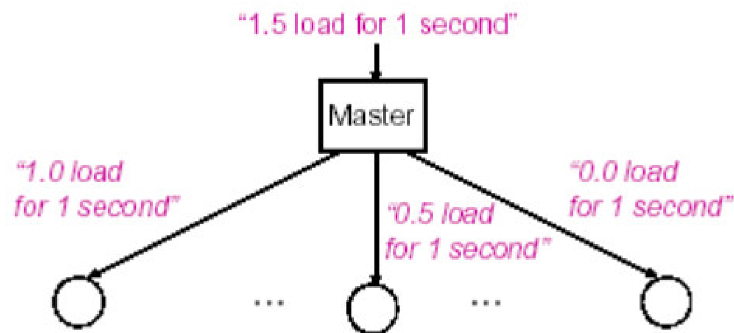


Figure 2.3: Example of working threads to produce contention

first phase. For this we use a exponential back-off technique to arrive at a precise value for the loopcount parameter. It is important that the calibration be done when the system is idle, to get accurate values. After calibration, we have two useful values: *sleeptime* and *loopcount* such that Sleep(*sleeptime*) and looping *loopcount* times take equal amount of times to complete.

Finally, to create fractional contention, we simply loop for the calibrated loopcount with the required probability and sleep for the rest of the time. This achieves the required fractional contention. Using the above techniques, we can then produce contention by using a family of full-running thread threads with one fractional thread (Figure 2.3). Figure 2.4 shows some exercise functions produced by the fractional thread. We can control the CPU contention with good accuracy.

Measuring and verifying CPU contention

The PDH Library in Windows provides queue length counters that can be used to verify the CPU loads produced by our exerciser function. We first developed a simple application QAverage that samples this counter with exponentially distributed time intervals that are then averaged to give a time average of the CPU Contention (PASTA principle). However, we found that these counters are not representative of the system. Two full threads would increase the counter value by 4 or 5. The values returned by this counter seem to be of little use. A detailed study [3] concludes that measured processor queue length counter is of little use in understanding Windows NT system performance from an application point of view. This paper further states that the statement in Microsoft documentation (and similarly repeated in some third party literature) that a "processor queue" value of two threads generally indicates processor congestion is inaccurate and misleading.

To overcome this hurdle, we decided to develop a method to measure CPU contention on our own. The contention on the CPU is correlated with the CPU utilization of a given thread. This information can be used to get an estimate of the CPU contention. Our application Contention Meter measures contention by comparing the CPU allocation to a thread in idle state vs. the CPU loaded state. It first calibrates itself by running a loop for a given number of iterations and records the time taken for the execution of the loop. Let this time be t_1. This must be then when the system is idle. Then, the program is run in a continuous loop, in which it reruns the previously executed loop and records the new time taken by this loop. Let this time be t_2. Then the CPU utilization for the thread relative to the idle state of the CPU is $\frac{t_1}{t_2}$. Assuming that t_1 is the time taken when the thread has 100% of the CPU, CPU contention is then equal to $\frac{t_2}{t_1}$. The contention

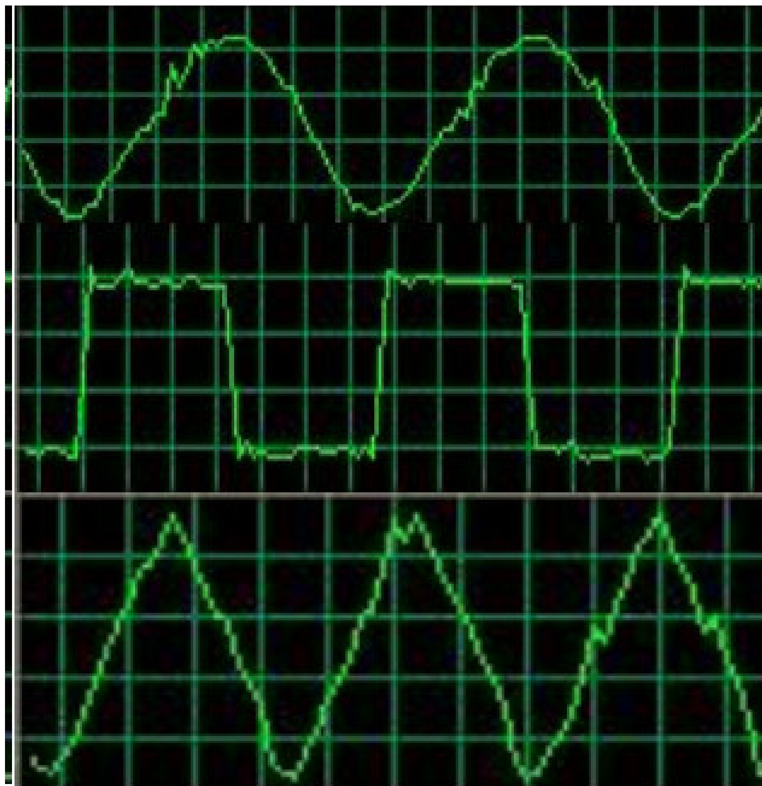


Figure 2.4: Example outputs of different CPU exerciser functions

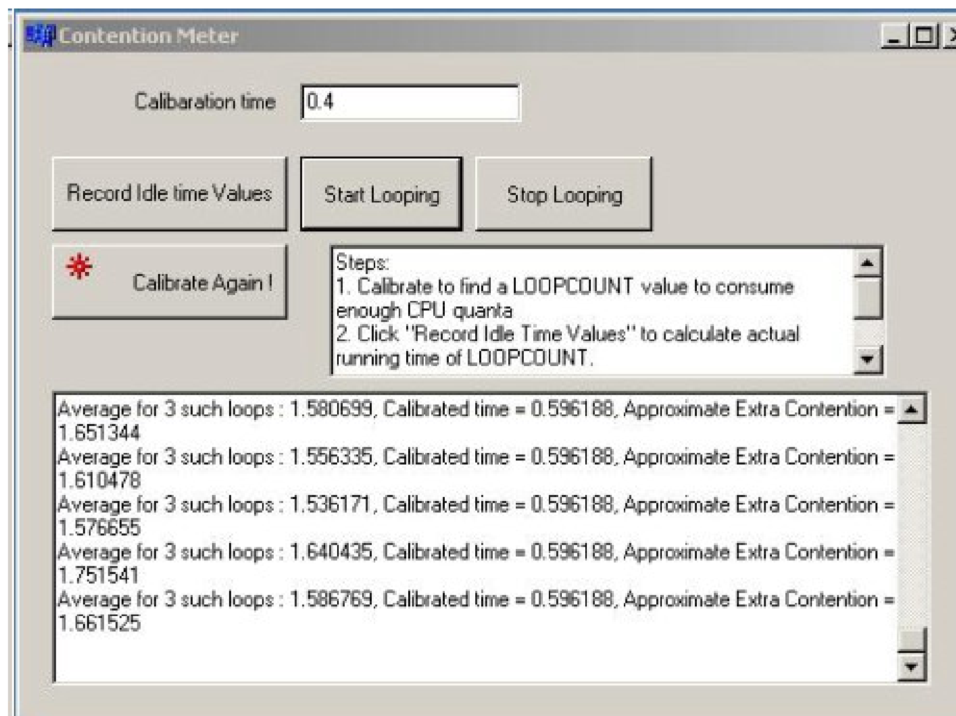


Figure 2.5: The Contention Meter application

meter displays this ratio as the contention of the CPU.

We then used this tool to measure and verify the contention produced by CPU exercisers. We found close correlation between contention values reported by the Contention Meter vs. the contention that the exercisers were attempting to produce. For example, Figure 2.5 shows the contention reported by Contention Meter when the exercisers were told to produce a contention of 1.6. The contention measured by the contention meter oscillates around 1.6.

Scheduling priority of the CPU exerciser

Windows does scheduling at the thread level and allows assignment of priorities to the different threads. Lower priority threads are not executed if any higher priority thread is awaiting execution. Deciding the priority of the CPU exerciser threads was also an issue. There are three possibilities with different consequences.

1. **Below Normal** The CPU exercisers would run only when no other thread is running. This is similar to running in the IDLE mode. Here the resource exercisers would not affect the application threads. Since we want to study the interaction of resource borrowing with user interactivity, we do not use this priority level.

2. **Above Normal** The CPU exerciser threads would be given higher priority than the application threads which mostly run under the Normal priority. If a CPU exerciser thread runs full time, other applications would be stuck and there is complete loss of interactivity. This is also not an option.

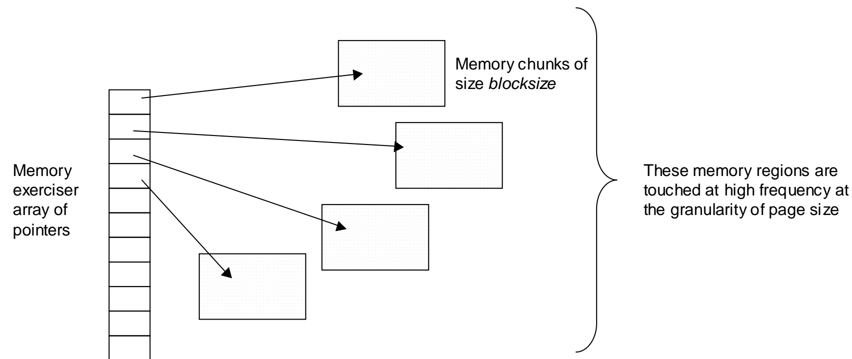


Figure 2.6: The memory exerciser. Here a total of blocksize*4 memory is allocated.

3. **Normal** Here the CPU exerciser threads are treated at the same level as other application threads and here the affect of CPU threads on other applications can be studied. Thus, the CPU exerciser threads are run under normal priority in Windows.

2.2.4 The memory exerciser

The memory exerciser is much simpler in design. It simply allocates a given amount of physical memory, avoiding having the allocated memory paged to disk. Contention is interpreted as the fraction of physical memory it should attempt to allocate. To borrow the desired amount of physical memory, it keeps a pool of allocated pages equal to the size of physical memory in the machine and then touches the fraction corresponding to the contention level with a high frequency, making its working set size inflate to that fraction of the physical memory.

It is important to ensure that the memory exerciser doesn't itself consume a lot of CPU cycles as it touches a large number of pages at a high frequency. We use the dynamic memory allocation features of C++ to allocate memory. We found that the amount of CPU cycles consumed by the memory exerciser depends on the unit allocation $blocksize$. To allocate a given amount of memory, we keep an array of pointers and then bind the required number of pointers to newly allocated regions of size $blocksize$. This is done at a regular frequency. Figure 2.6 shows the memory structures used by the memory exerciser. To ensure low CPU usage, a $blocksize$ of 1 MB was chosen. It was found that a $blocksize$ of 64K produced significant CPU contention. Surprisingly, lowering the $blocksize$ to around 64K - 512 bytes eliminated the CPU contention. This is almost certainly due to the dynamic memory allocation strategies implemented in the particular C++ run-time (malloc blocksize effects).

We avoid contention levels greater than one because this immediately results in thrashing which is not only very irritating to all users (as it affects interactivity drastically), but also very difficult to stop punctually.

2.2.5 The disk exerciser

Here we borrow the contention concept from the CPU exerciser. We create threads to produce contention. There are two kinds of threads: a full working thread and a fractional thread.

In a full working thread, we continuously keep writing the disk. In a fractional thread, we need to balance the writing times and sleep times (when no writes are issued) precisely for the thread. Our approach

to creating fractional contention is very similar to the one we used in CPU exerciser. We have two functions, `sleep()` and `write()`. `sleep()` sleeps for a given time x and `write()` writes disk for time x. We call these functions randomly with a probability distribution to match the amount of disk contention required. To generate the contention of 0.6 we call the `write()` function with a probability of 0.6 and the `sleep()` function with a probability of 0.4.

However, there are issues involved with the `sleep()` function in the Windows API as we mentioned before. We want `sleep()` and `write()` functions to consume an equal amount of time. For this, we use high resolution timing functions `QueryPerformanceCounter()` and `QueryPerformanceFrequency()` to calculate the average time of each disk write with a fixed data chunk size. Using these functions we can get timing up to a millisecond level of precision. In the fractional thread, we then use the calculated sleep time in `sleep()` in the fractional thread. The following is pseudocode for the fractional disk writing thread:

```
while(1)
{
        if (rand_number() < fraction_number)
        {
            disk_seek (file, random_place);
            write_disk (file, chunk_of_data);
        }
        else
        {
            sleep (sleep_time);
        }
}
```

There are three levels of disk caches we need to deal with: application level, OS kernel level and hard disk level. Data writes issued from application level can be sped up any level of caching. Furthermore, sometimes the writes we issue will not write disk at all or write disk in an unpredictable way. For example, if the OS or disk detects that two consecutive writes are for the same sector, only one disk write will be actually occur. Another issue is that the disk itself can queue, reorder, and merge writes to reduce or eliminate seeks and rotational latency. When writing to the cache on the disk, it is possible to achieve very high bandwidth, particularly in bursts (try `hdparm -t` versus `-T` on Linux to get a sense of this, or a tool like `bonnie++`).

To make application-level writes directly and precisely produce raw disk writes, we need to counteract the caches. First,we use the `CreateFile()` function with the `FILE_FLAG_NO_BUFFERING` parameter to create a large file in disk. The size of the file is the minimum of twice the physical memory size or the available disk space size. The `FILE_FLAG_NO_BUFFERING` parameter turns off the OS level cache. Second, we use the `FlushFileBuffers()` function immediately following each write (`WriteFile()`), which forces data to be immediately flushed to disk. Third,we randomly seek in the file before each `WriteFile()`, which makes the probability of two consecutive writes to the same sector very small.

There are two kinds of contention involved in our exerciser: the contention for seeks, and the contention for disk bandwidth. With very large write sizes, contention will be dominated by contention for disk bandwidth. If the user application issues small writes, they will not be much affected since their predominant cost is a seek. With very small write sizes, the contention will be dominated by how many seeks/second the disk can do, but there will be plenty of bandwidth left over. In that case, if the user

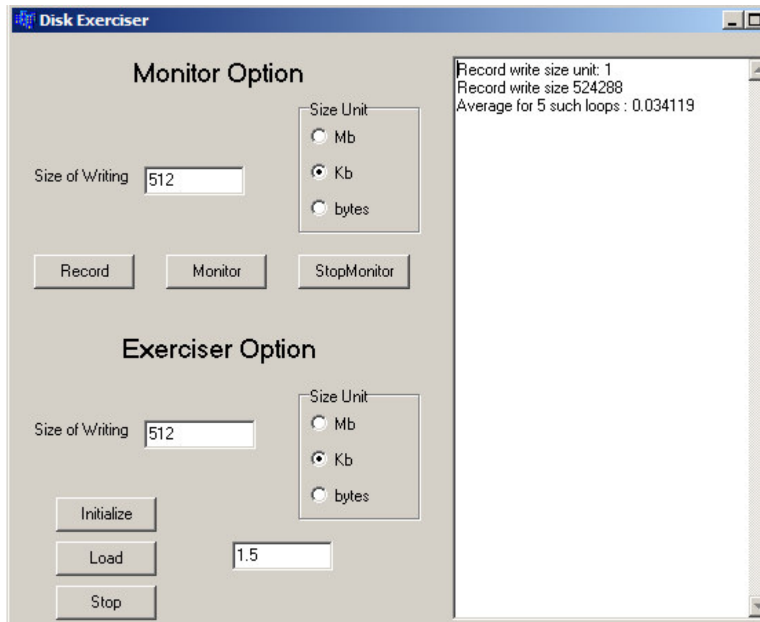


Figure 2.7: Disk Contention Monitor

application issues large writes, they will be minimally effected. Because we seek to affect all applications, we have the exerciser issue writes of randomly chosen sizes. In this way, the exerciser affects both large and small writes/reads made by other programs. In the disk exerciser, we set the `NumberOfBytesToWrite` in `WriteFile()` to be a random number between 0 and 4 MB. (In the fractional thread, the number is a fixed 512 KB)

Another issue is that since we open the disk space file with the `FILE_FLAG_NO_BUFFERING` flag set, for the random seek and random write size to work properly, the file pointer must be set to sector-aligned positions and the write size must be an integer number of sectors. A sector-aligned position/number is a position/number that is a whole number multiple of the volume's sector size. We obtain a volume's sector size by calling the `GetDiskFreeSpace()` function.

We developed our own disk contention monitor to measure disk contention on our own. The contention on the disk is correlated with the disk writing time (how long it takes to write a given number of bytes) of a given thread. This information can be used to get an estimate of the disk contention. Our application measures contention by comparing the disk writing time of a thread in the idle state versus the loaded state (when the exerciser is running). It first calibrates itself by running a loop for a given number of iterations and records the time taken for the execution of the loop. Let this time be t_1. This must be done when the system is idle. Then, the program is run in a continuous loop in which it reruns the previously executed loop and records the new time taken by this loop. Let this time be t_2. The contention introduced is then $t_2/t_1 - 1$ The contention meter displays this ratio as the contention of the disk. Figure2.7 is a screenshot of the initial state of the disk contention monitor.

2.2.6 The network exerciser

The goal of the network exerciser is to affect the user's network traffic and control the available bandwidth, while minimally affecting other machines in the network. We firstly built a network exerciser that used

sockets to send UDP packets. The exerciser can control the packet sending rate while targeting one machine, broadcasting or multicasting. We evaluated the UDP exerciser with different irritation parameters (Kbytes/sec). The goal was to slow down the network interface, as measured by downloading a file from a local server using Flashget. As we increased the exerciser transfer rate, the time taken to transfer the file increased. (The normal transfer time for the file was recorded as less than 10 seconds, file size is 25,000,005 bytes) However, the transfer speed depended on the destination for the UDP packets. Destinations can be of 4 types:

1. **127.0.0.1 / local IP address** : The idea here was to keep the network stack and the device driver busy without emitting packets. Unfortunately, this doesn't affect the transfer rate at all, even if we have a thread to receive the artificial traffic. No network bandwidth is consumed, which is also shown in the Windows XP task monitor. It looks like local adapter just discards loop-back packets.

2. **Specific host on LAN** : This slows down the transfer, but is it is very hard to control the impact. For a 5000 kbps sending rate, the time was around 25 seconds. The transfer time increases slowly as the exercise rate increases, until sending rate reached 7000 kbps, at which point the transfer time is 2:11.

3. **Multicast**: This has the most impact on the transfer speed. When a sending rate of 5000 kbps was used, the transfer time is 21 seconds. At 6000 kbps sending rate, the transfer time fluctuates a lot, from 32 seconds to a very long time. At 6500 kbps, the transfer is almost stopped. We cannot use multicast because of the impact on every host in the LAN.

4. **External host** : When we sent packets to external host, there was no effect on file transfer rate. This is still mysterious to us, as the network card is still engaged in sending the UDP packets. Why should targeting a specific host on LAN and external host give totally different results? A possible explanation is that the packets are taking a different route through the switched LAN. Perhaps when we talk to an external machine, via the router, the packets are not interfering with our transfer.

Following the UDP exerciser, we built another which emitted Ethernet packets exerciser using the WinPcap[1] library. Here the idea was to directly inject Ethernet packets with the destination address being the same as the source address on the interface. We hypnotized that this should pass the packet directly to the device driver to be sent. If the driver was not too clever, and the network adapter was not too clever, it would send the packets we inject. The first switch it encountered would send it right back. This way we would impact the network only as far as that first switch. However, we did not get the expected results. This does not seem to affect the transfer rate at all and again no network bandwidth is consumed.

At this time we have not developed an adequate network exerciser, although we continue to seek ways of doing so.

2.3 User feedback techniques

Feedback is essential for the user to indicate his discomfort due to resource borrowing. It should be easy enough for the user to express. We have two methods: a keyboard hotkey or right clicking on a system tray icon in Windows. Note that this is binary feedback, either Yes or No. Interestingly, extensive research has been conducted on various ways to express frustration feedback to the PC [14]! Other methods include analog methods of expressing discomfort, which allow the user to also include the level of discomfort, thus

[1] http://winpcap.polito.it/

fine-tuning his feedback. Advantage of this method is that it would allow us to develop a more sophisticated model of user comfort with resource borrowing. However, it would also require more participation from the user, making fewer users interested in trying our software. Based on results from our current study, other methods may also be considered for feedback in the future.

2.4 The client/server architecture

The Understanding User Comfort System employs a client-server system. This allows us lot of flexibility in transferring testcases and results between the client and server. The client is the application on the user's end which does all the resource borrowing tasks. The server is responsible for sending new testcases to the client and also for receiving back results from the client. A client-server system allows us some flexibility:

1. Testcases need not be packaged with the client. This allows us to withdraw some testcases or add new testcases to the testcase database which the clients can download wherever they may be installed.

2. All the results can be gathered at the server, which facilitates organizing and analyzing the results.

3. It also allows us to control parts of the behavior of the client like wait time between testcases, user idle time parameters, etc.

The client can be configured to periodically **hotsync** with the server. Hotsync refers to the downloading of new testcases by the client and transferring the results back to the server. The following is a brief description of both the processes

Downloading Testcases: The server is designed to send a random set of testcases from its testcase repository. The client decides how many testcases it wants to receive, which is a configurable parameter on the client side. Thus each client receives a different set of testcases from the server.

The server picks up testcases randomly from its repository using a exponentially distributed random seek distances in the file. It starts from the beginning and seeks using the exponential distribution. λ, the mean value of the exponential distribution is chosen as $\frac{totalnumberoftestcases}{numberoftestcasesrequestedbyclient}$. The client on receiving the new set of testcases, merges them with its existing set of testcases maintaining a sorted order. This is implemented using a simple merging algorithm that also eliminates any duplicates received from the server.

Sending Results: The client sends back the following information to the server:

- The results file which logs all the user feedbacks along with the monitored information.

- The system configuration file if it hasn't been sent before

- The config file which contains the configuration information about the client. This is only referred to in case of any discrepancy faced later on.

The server is multi-threaded to handle multiple client connections simultaneously. It was tested for robustness with 50 clients connecting simultaneously. Several bugs were discovered and fixed during this stress test.

Receiving and Sending Testcases	
Client Action	Server Action
GET TESTCASE <number of testcases requested> CLOSE	 <Send required number of testcase to client> CLOSE
Notes	
Client will disconnect before any connection.	Send testcases randomly from testcase database.
In case of any exception, the client disconnects and aborts from the operation.	

Sending Results to the Server	
Client Action	Server Action
POST RESULTS ID (Wait for timestamp) <Send latest results> CLOSE	 <Send latest Timestamp to the client> <Receive and append results to client file> CLOSE
Notes	
ID refers to a globally unique ID of the client	Server appends the results to the client's results file.
The client only needs to send the results recorded after the timestamp.	

Figure 2.8: Protocol for client-server communication

2.4.1 The protocol

The client/server system uses a HTTP like text based protocol. There are two sets of commands, for retrieving testcases and sending results. These protocols are summarized in Table 2.8.

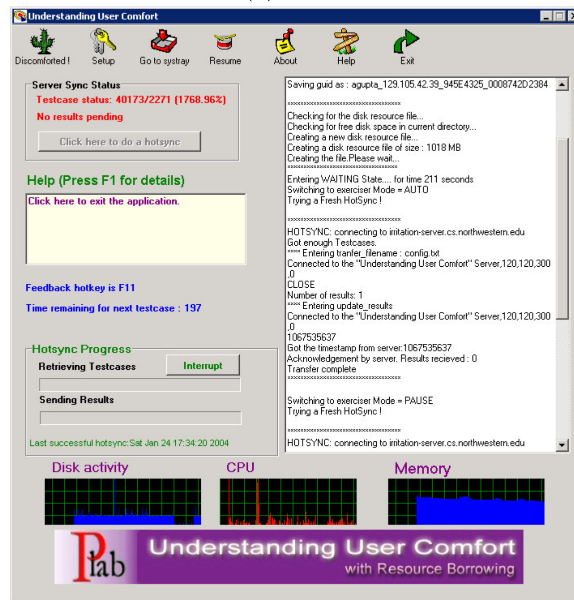
Figure 2.8 does not show the protocol for transferring system information and config information, which is similar to the results protocol. The server logs every activity to a file. In case of any discrepancy, the file can be referred to. It also records all the client connects, thus acting as a log for server traffic.

2.5 The client interface

Figure 2.9 shows the graphical interface of the UUCS client. The most basic interface is the tray interface (Figure 2.9(a)), in which a user can only express discomfort, either by clicking on the tray icon or by pressing a hot-key (F11 here). The remainder of the interface can be disabled, and is disabled in our controlled study. If it is enabled, the user can see a popup menu (Figure 2.9(b)), or pop up a detailed view (Figure 2.9(c)) of the operation of the application.

(a) Tray interface

(b) Menu



(c) Application

Figure 2.9: Client interface. The menu and full application interface can be disabled.

3

Controlled study

As the second stage of our user studies (after a beta test), we conducted a study which involves users participating in experiments with some pre-specified controls. The purpose of the controlled study is to answer some of the questions which we want to study and the data which we have got from this study has indeed given us some clear answers to some questions we had posed earlier.

3.1 Terminology used

User The test subject

Application The program the user runs

Task Period The duration of a task

Task Some operation we ask the user to accomplish using an application. A task is timed and should exceed the length of a task period in other words, even for power users, they should not ever finish the task in the task period.

User Comfort Client Our software for inducing and measuring user discomfort

Testcase Period The duration of a testcase. The Task Period is an integral multiple of the Testcase Period.

Run Execution of resource borrowing in the client during a task, using some set of exercise functions.

3.2 The experimental setup

Doing a controlled study gives us some advantages, mainly in helping us control various factors which provide more focus in answering certain questions. Designing the controlled study was tricky and mistakes in choosing the wrong controls can be costly later on. We also learned from our experience, and would improve the experiments if repeated.

3.2.1 Hardware

Two machines were used for conducting control experiments in two separate private environments. Their hardware configuration is summarized in Figure 3.1.

Machine Configuration	
Hardware Configuration	2.0 GHz P4, 512 MB, 80 GB. Dell Optiplex GX270, 17 in monitor
Operating System	Windows XP
Applications Installed	Word 2002, Powerpoint 2002, IE 6, Quake 2

Figure 3.1: Machine configuration.

3.2.2 Invitation and interaction with subjects

An advertisement for the controlled study was prepared and posted in the Computer Science Department and in the Main Engineering Building at Northwestern University (named Tech). Email announcements were also sent to mailing lists. Each subject was asked to fill in a questionnaire with following questions:

1. Email address
2. Gender, age, occupation
3. Overall, do you consider yourself a beginning, typical, or power user of computers?
4. Overall, do you consider yourself a beginning, typical, or power user of MS Windows, or have you never used it before ?
5. For each of our Applications (Word, Powerpoint, Internet Explorer and Quake III), do you consider yourself a beginning, typical, or power user, or have you never used it before?

During the experiment, each user was given written instructions [1], after which they started the tasks described below. Each task was timed using a custom-made software timer which runs in the background and pops up at the end of 16 minutes.

The 33 users in our study consisted primarily of graduate students and undergraduates from the Northwestern engineering departments. Anecdotal evidence suggests that this group is more sensitive to resource borrowing than others. Each user was given $15 for participating. The duration of the study for each user was 84 minutes. The user:

1. Filled out a questionnaire as described above. The key questions were user self-evaluations as “Power User”, “Typical User”, or “Beginner” for use of PCs, Windows, Word, Powerpoint, Internet Explorer, and Quake. (5 minutes).
2. Read a one page handout (5 minutes).
3. Acclimatized themselves to the performance of our machine by using the above applications (10 minutes).
4. Performed the following tasks:
    - Word processing using Microsoft Word (16 minutes): Each user typed in a non-technical document with limited formatting.

[1] These instructions can be found at http://comfort.cs.northwestern.edu

No.	Resource	Type	Word Parameters	Powerpoint	Internet Explorer	Quake
1	CPU	Ramp	7.0,120	2.0,120	2.0,120	1.3,120
2	Blank					
3	Disk	Ramp	7.0,120	8.0,120	5.0,120	5.0,120
4	Memory	Ramp	1.0,120	1.0,120	1.0,120	1.0,120
5	CPU	Step	5.5,120,40	0.98,120,40	1.0,120,40	0.5,120,40
6	Disk	Step	5.0,120,40	6.0,120,40	4.0,120,40	5.0,120,40
7	Blank					
8	Memory	Step	1.0,120,40	1.0,120,40	1.0,120,40	1.0,120,40

Figure 3.2: Testcase descriptions for the 4 tasks (given in random order).

- Presentation making using Microsoft Powerpoint (16 minutes): Each user duplicated a presentation consisting of complex diagrams involving drawing and labeling from a hard copy of a sample presentation.
- Browsing and research with Internet Explorer (16 minutes): Each user was assigned a news web site and asked to read the first paragraphs of the main news stories. Based on this, they searched for related material and saved it. This task involved multiple application windows.
- Playing Quake III (16 minutes): Quake III is a well known first-person shooter game. There were no constraints on user's gameplay.

As the user performed the tasks, the UUCS client executed in the background and ran specific testcases. It recorded all the system and contextual information as well as the user feedbacks, which were later used to generate the results.

3.2.3 Testcases

We designed a set of primary testcases for each task that would help us answer some specific questions we have. Testcases were either of the type **ramp** or **step**. We had 8 testcases of duration two minutes each as the primary testcases for each task, and some extra testcases in case the user exhausted these. They are run consecutively for each 16 minute task.

Calibration: Each task has different resource requirements, and the regions of resource usage where interactivity is affected may be different for each task. For example, in Word very high values of CPU Contention (around 3-5) are needed to affect interactivity, whereas in Quake CPU Contention values in the region of 0.2 to 1.2 are enough to affect the interaction drastically. Therefore, careful calibration is required to choose the parameters for the testcases for each task to observe any phenomena of interest. The calibration was done by using the applications, running a large number of testcases with different parameters, and then selecting those testcases which affected interactivity. However, these judgments are subjective in nature and therefore prone to error. Incorrect calibration can result in regions of operation in which the user is seldom discomforted or always discomforted. It is also important to choose the value of step testcases carefully, as they help us in understanding the time dynamics of the relationship between resource borrowing and user discomfort, the frog in the pot hypothesis (Section 1.2). By studying the user feedback values from both, we can judge whether a ramp can lead to higher levels of resource borrowing compared to a step.

The testcase description for the control study by each task are summarized in Figure 3.2, with additional details below:

- Testcases 1, 3, 4 test for the level at which user discomfort is incurred
- Testcases 1 versus 5, 3 versus 6, 4 versus 8 are tests for frog-in-pot
- Testcases 2, 7 (and irritation expressed during sleep periods of all the others) test for the background level of irritation

3.2.4 Overview of factors used in the controlled study

As described earlier the controlled study allows us to control various factors which can in turn help us find answers to our questions directly. Some of the factors which were fixed and hence acted as controls are:

1. Context: The applications and their durations are pre-defined, thus letting us study the effects of context.
2. Hardware and the software environment: The hardware was same for the two PCs used, thus eliminating any biases due to hardware. The effects due to hardware cannot be studied in this. The OS (Windows XP) and installed software were also same.
3. Testcases: Using the same set of testcases for each task allowed us to study affect of specific patterns of resource borrowing. Testcases were either of type ramp or step with different parameters. Using careful calibration (Section 3.2.3), we can also study the frog in the pot hypothesis (Section 1.2).

3.3 Control study results

We now address the questions posed in the introduction using empirical cumulative distribution functions and informal factor analysis. We describe the results below, along with additional observations.

3.3.1 What level of resource borrowing leads to user discomfort for a significant fraction of users?

From the perspective of an implementor this is a key question. We can answer this question using cumulative probability distributions (CDF) derived from running our ramp testcases, aggregated across contexts to convey a general view of each resource.

Figures 3.3-3.5 show CDFs for CPU, memory and disk aggregated over all the tasks. The horizontal axis is the level of contention for each resource. The vertical axis is the cumulative fraction of users discomforted. As the level of borrowing increases, users' interactivity is increasingly likely to be affected. This is the *discomfort region*. Some users do not become discomforted in the range of levels explored. We refer to this as the *exhausted region.* Each graph is labeled with the number of runs that ended in discomfort ($DfCount$) and exhaustion ($ExCount$). There is also some probability that a user will feel discomforted even when no resource borrowing (blank testcase) is occurring. We refer to this as the *noise floor* and it is reflected in Figure 3.10.

To make our discussion easier, we derive three metrics from the CDFs. The first is f_d, the fraction of testcases which provoke discomfort,

$$f_d = \frac{DfCount}{DfCount + ExCount}$$

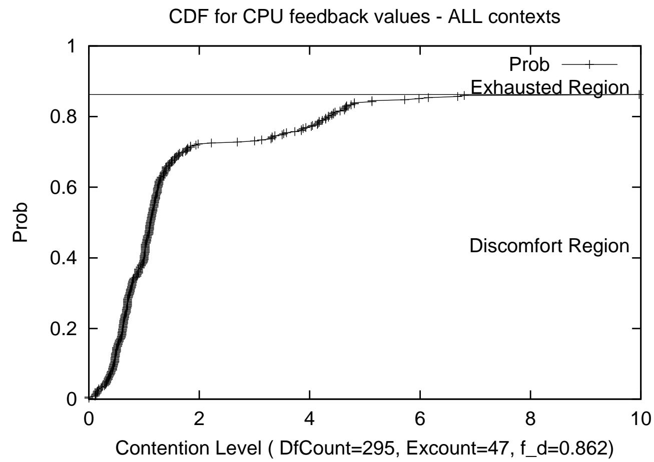


Figure 3.3: CDF of discomfort for CPU.

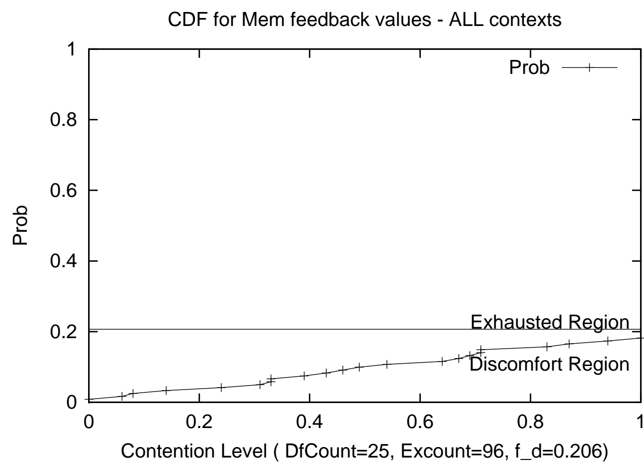


Figure 3.4: CDF of discomfort for Memory.

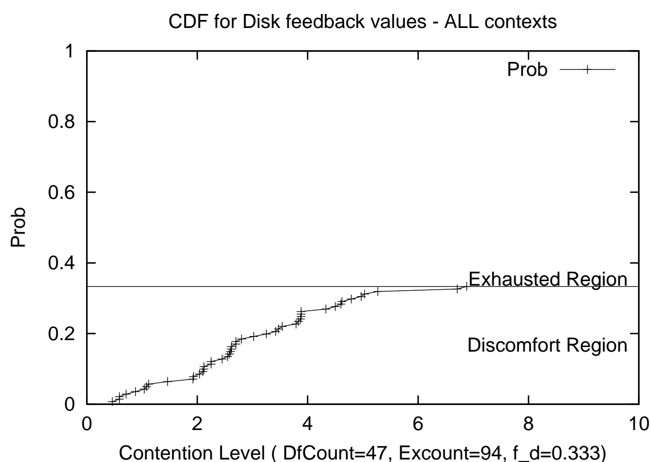


Figure 3.5: CDF of discomfort for Disk.

A low value of f_d indicates that the range of contention applied in that context for resource borrowing doesn't affect interactivity significantly.

The second metric is $c_{0.05}$, the contention level that discomforts 5% of the users. This is the 5th-percentile of the CDFs. This value is of particular interest to implementors as it provides them with a level that discomforts only a tiny fraction of users. Any other percentile can also be found from these CDFs.

The third metric is c_a, the average contention level at which discomfort occurs. This is useful in comparing classes of users. Figures 3.7, 3.8, and 3.9 show the three metrics.

Figure 3.3 shows the CDF for CPU borrowing. Notice that even at CPU contention levels of 10, more than 10% of users do not become irritated. More importantly, we can reach contention levels of 0.35 while irritating fewer than 5% of the users ($c_{0.05,cpu} \simeq 0.35$). This corresponds to consuming 35% of the processor when there are no competing threads.

Figure 3.4 shows the CDF for memory. Notice that almost 80% of users are unfazed even when nearly all their memory is consumed ($f_d = 0.21$). Furthermore, aggregating over the four contexts, it appears we can easily borrow 33% of memory on these common PCs while irritating fewer than 5% of users ($c_{0.05,memory} \simeq 0.33$) in general.

Figure 3.5 shows the CDF for disk bandwidth. Almost 70% of users are comfortable even with seven competing tasks ($f_d = 0.33$). Furthermore, we can easily execute a single disk writing task, capable of consuming the whole disk bandwidth if run alone, while irritating fewer than 5% of the users ($c_{0.05,disk} \simeq 1.11$). We found this result remarkably counterintuitive as we ourselves tend to become uncomfortable when large amounts of unexplained disk I/O occurs on our desktops. The Dell machines we used for the study are remarkably quiet and have very dim disk lights. We suspect that it is the limited feedback about disk activity that leads to users accepting far higher amounts of disk contention than they otherwise might.

3.3.2 How does the level depend on which resource or combination of resources is borrowed?

Figure 3.12 (located at the end of this chapter) shows the CDF for each context and resource pair. Consulting its columns as well as the aggregated CDFs shown earlier clearly shows the strong dependence on the type of resource. Within the contention levels explored by the ramp testcases for each resource, users are much

	CPU	Memory	Disk	Total
Word	L	L	L	L
Powerpoint	M	L	L	M
IE	M	M	H	M
Quake	H	M	M	H
Total	M	L	L	

Figure 3.6: User sensitivity by task and resource (Low, Medium, High).

	CPU	Memory	Disk
Word	0.71	0.00	0.10
Powerpoint	0.95	0.07	0.17
IE	0.75	0.30	0.61
Quake	0.95	0.45	0.29
Total	0.86	0.21	0.33

Figure 3.7: f_d by task and resource.

	CPU	Memory	Disk
Word	3.06	*	3.28
Powerpoint	1.00	0.64	3.84
IE	0.61	0.31	2.02
Quake	0.18	0.08	0.69
Total	0.35	0.33	1.11

Figure 3.8: $c_{0.05}$ by task and resource. (* indicates insufficient information)

	CPU	Memory	Disk
Word	4.35 (3.97,4.72)	*	4.20 (1.89,6.51)
Powerpoint	1.17 (1.11,1.24)	0.64 (0.21,1.06)	4.65 (3.67,5.63)
IE	1.20 (1.07,1.33)	0.55 (0.39,0.71)	3.11 (2.69,3.52)
Quake	0.64 (0.58,0.69)	0.55 (0.37,0.74)	1.19 (0.86,1.52)
Total	1.47 (1.31,1.64)	0.58 (0.46,0.71)	2.97 (2.54,3.41)

Figure 3.9: c_a by task and resource, including 95% confidence intervals. (* indicates insufficient information)

more tolerant with borrowing of memory and disk. This observation is qualitative as the testcases for each resource are different, but within the levels explored this holds true.

The varying tolerance by resource also shows up in our aggregated f_d, c_5 and c_a metrics, the column totals of Figures 3.7, 3.8, and 3.9. An important point to note is that the high f_d value of CPU (0.86), does not mean that the probability of discomforting users by borrowing CPU is 0.86. This probability depends on the contention. To determine this probability, a level must be chosen and the CDFs consulted as described in the previous section.

3.3.3 How does the level depend on the user's context?

In Figure 3.12, we see dramatic differences in the reactions to resource borrowing between different contexts. Consulting the rows illustrates this. It is clearly the case that the user's tolerance for resource borrowing depends not only on the resource, but also on what the user is doing.

The totals row of Figure 3.9 shows the average level at which discomfort occurs for the CPU contention for the four tasks. For an undemanding application like Word, the CPU contention can be very high (> 4) without significant affecting interactivity. However, with finer-grain interactivity, as in Powerpoint and Quake, the average level is much lower. This is likely due to the more aggressive CPU demands of these applications. Still, for the most aggressive application, Quake, results show that a thread with contention of nearly 0.2 can still run with a low probability of discomfort.

We used the same testcase for memory in all four tasks, growing the working set from zero to nearly the full memory size. The effect of memory borrowing is minimal in the case of Word (no discomfort recorded) and Powerpoint. IE and Quake are much more sensitive to memory borrowing, with more instances of discomfort ($f_d = 0.3$ and $f_d = 0.45$, respectively). For IE and Quake, value of $c_{0.05,mem}$ is 0.31 and 0.08 respectively, meaning that Quake users become discomforted at much lower levels. It appears that once office applications like Word and Powerpoint form their working set, significant portions of the remaining physical memory can be borrowed with marginal impact. This seems to be less true for IE and Quake, where their memory demands may be more dynamic.

Disk bandwidth can be borrowed with little discomfort in typical office applications. In Word and Powerpoint, the fraction of testcases ending in discomfort was small ($f_d = 0.01$ and $f_d = 0.17$ respectively), in the wide range covered by the testcases. IE and Quake are more sensitive. For identical disk testcases, we find that IE is more sensitive ($f_d = 0.61$). This may be expected as IE caches files and users were asked to save all the pages, resulting in more disk activity.

Figure 3.10 shows that users express feedback even when there is no testcase running. We note that users exhibit this behavior only in IE and Quake. Quake is a very demanding application in which jitter quickly discomforts users. There are sources of jitter on even an otherwise quiescent machine. Discomfort in IE depends to some extent on network behavior.

Figure 3.6 summarizes our judgment of user sensitivity to resource borrowing by resource and task. Note that the totals are not derived from the columns but represent overall judgments from the study of the CDFs (Figure 3.12).

3.3.4 How does the level depend on the user, factoring out context?

Users' comfort with resource borrowing depends to a small extent on their perceived skill level. We asked our users to rate themselves as {Power User, Typical User, or Beginner} in each of {PC Usage, Windows, Word, Powerpoint, IE, and Quake}.

Total		
	Non-Blank testcases	Blank
Discomforted	295	33
Exhausted	47	212
MS Word		
Discomforted	48	0
Exhausted	20	59
Prob of discomfort from blank testcase		0.00
MS Powerpoint		
Discomforted	71	0
Exhausted	4	60
Prob of discomfort from blank testcase		0.00
Internet Explorer		
Discomforted	50	14
Exhausted	17	50
Prob of discomfort from blank testcase		0.22
Quake		
Discomforted	126	19
Exhausted	6	43
Prob of discomfort from blank testcase		0.30

Figure 3.10: Breakdown of runs.

Application	Resource	Background	Rating Pair	P-value	Avg. Contention Difference
Quake	CPU	PC	Power, Typical	0.006	0.176
Quake	CPU	Windows	Power, Typical	0.031	0.137
Quake	CPU	Quake	Power, Typical	0.001	0.224
Quake	CPU	Quake	Typical, Beginner	0.031	0.139
IE	Disk	Windows	Power, Typical	0.004	1.114
IE	Mem	Windows	Power, Typical	0.011	0.354

Figure 3.11: Significant differences based on user-perceived skill level.

We compared the average discomfort contention levels for the different groups of users defined by their self-ratings for each context/resource combination using unpaired t-tests. In some cases, we found significant differences, summarized in Figure 3.11. The largest differences were for the combination Quake/CPU. For example, a Quake Power User will tolerate 0.224 less CPU contention than a Quake Typical User at a significance level (p-value) of 0.001. Even the users' self-rating for general Windows and PC use can lead to interesting differences in their tolerance. For example, for CPU, the differences between discomfort levels for Power and Typical users are quite drastic with $p = 0.002$ (PC Background) and $p = 0.010$ (Windows Background). Applications which have higher resource requirements show greater differences between user classes. However, our results are preliminary here and will improve with our Internet-wide study.

These results expose the psychological component to comfort with resource borrowing. Experienced or power users have higher expectations from the interactive application than beginners. When we borrow resources it may be helpful to ask the user to rate himself.

3.3.5 How does the level depend on the time dynamics of resource borrowing?

For this question, we have only preliminary results. We tested the frog in the pot hypothesis as described earlier. We paired ramp and step testcases in our study to explore if a similar phenomenon might be true of user comfort with resource borrowing—that a user would be more tolerant of a slow ramp that a quick step to the same level. We did observe the phenomenon in Powerpoint/CPU—the majority of users (96%) tolerated higher levels in the ramp testcase with a contention difference of 0.22 (averaged) with a p-value of 0.0001.

Our Internet-wide study (Section 5.3) is intended to address the question of time dynamics and of raw host speed more carefully. [2]

[2] Please refer to http://comfort.cs.northwestern.edu for more information on the Internet-wide study

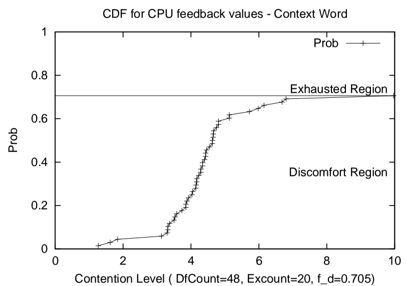
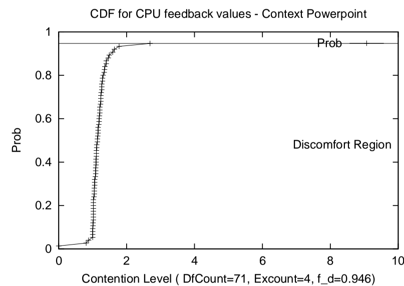
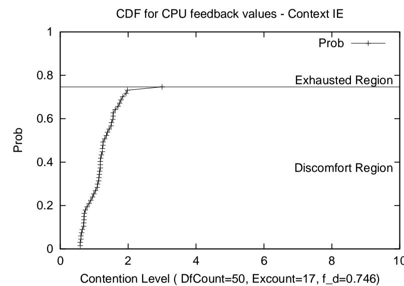
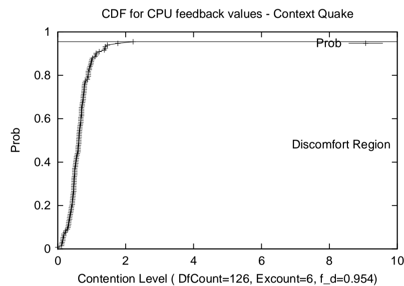
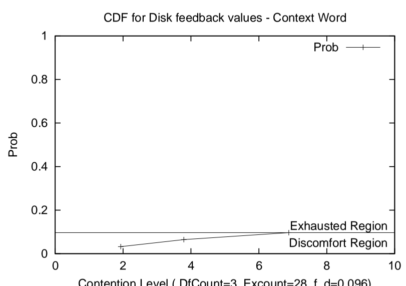
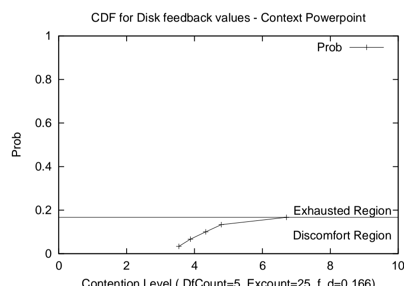
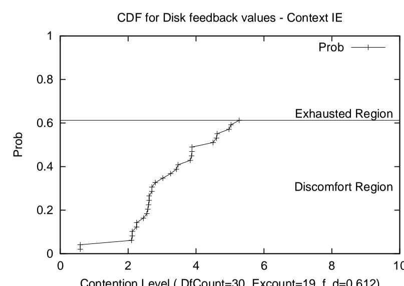
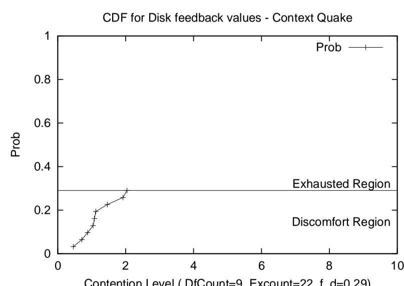
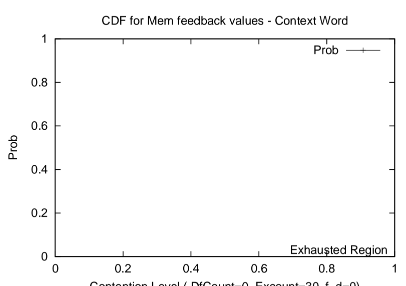
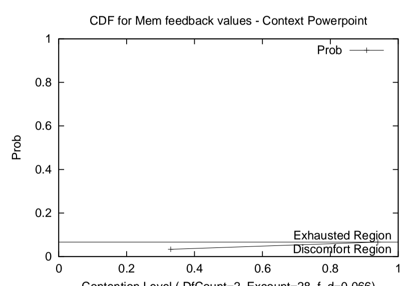
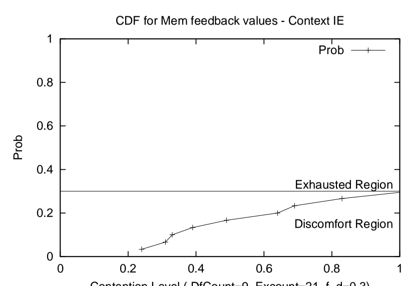
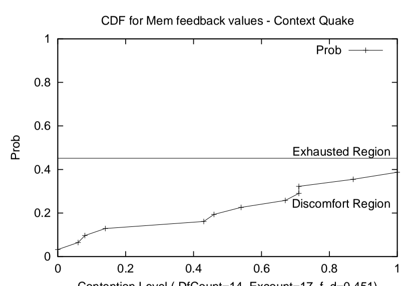
	Word	Powerpoint	Internet Explorer	Quake
CPU				
Disk				
Mem				

Figure 3.12: CDFs by resource and task.

4

Programming issues

In this section we discuss some programming issues we faced while developing the Understanding User Comfort client and server. Some of these are specific to the development environment we have used (Borland C++ Builder Professional 6.0).

4.1 System information monitoring

Process monitoring using PDH puts a heavy load on the CPU. However, the Windows Task Manager does this quite efficiently. We searched for more efficient means to enumerate all the processes on Windows and came across an undocumented System call `_ZwQuerySystemInformation()`, which does this efficiently on Windows NT/2000/XP systems. Please refer to the code for the complete source code for enumerating processes and related information.

4.2 Displaying an __int64 value in Borland C++

This was needed to handle large disk sizes and other 64 bit structures. The following code fragment does the trick.

```
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  // ULARGE_INTEGER is a __int64 value.
  ULARGE_INTEGER fbatc, tnob, tnofb;

  GetDiskFreeSpaceEx("c:", &fbatc, &tnob, &tnofb );

  char a[40] = { 0 };
  sprintf(a, "%I64Ld", tnob.QuadPart);

  ShowMessage(a);
}
```

4.3 Handle GUI only in the main thread

We observed that showing and hiding forms or dialogs in other threads besides the main GUI thread led to unexpected exceptions. It is advisable to have all the GUI handling code in the main thread itself.

4.4 Changing thread priorities

Threads are scheduled to run based on their scheduling priority. Each thread is assigned a scheduling priority. The priority levels range from zero (lowest priority) to 31 (highest priority). Only the zero-page thread can have a priority of zero. (The zero-page thread is a system thread responsible for zeroing any free pages when there are no other threads that need to run.). The system treats all threads with the same priority as equal. The system assigns time slices in a round-robin fashion to all threads with the highest priority. If none of these threads are ready to run, the system assigns time slices in a round-robin fashion to all threads with the next highest priority.

The priority of each thread is determined by the following criteria:

- The priority class of its process
- The priority level of the thread within the priority class of its process

The priority class and priority level are combined to form the base priority of a thread.

The following code illustrates how to change the priority class for a process and then the priority for a particular thread belonging to that process. Put

```
SetPriorityClass( GetCurrentProcess(), REALTIME_PRIORITY_CLASS );
SetThreadPriority( GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL );
```

before the code you want to boost and

```
SetPriorityClass( GetCurrentProcess(), NORMAL_PRIORITY_CLASS );
SetThreadPriority( GetCurrentThread(), THREAD_PRIORITY_NORMAL );
```

after the block—this will bring your application to a normal status.

4.5 Working with global hotkeys

The C++ Builder VCL Hotkey component does not handle system wide hotkeys. Windows API Calls `RegisterHotKey()` and `UnRegisterHotKey()` do the trick. They take modifiers and the hotkey. Modifiers can be changed logically using `MOD_CONTROL`, `MOD_ALT` and `MOD_SHIFT` operators. To set a key like F11 as the hotkey, use `VK_F11` as the hotkey without any modifiers. F12 cannot be registered as a hotkey.

For handling the hotkey message, a message handler for the `WM_HOTKEY` needs to be added to the Form class. `TMessage` is the Windows Message Type and `GlobalHotKey()` is the function which is called on receiving this event:

```
BEGIN_MESSAGE_MAP
                 MESSAGE_HANDLER(WM_HOTKEY,TMessage,GlobalHotKey);
END_MESSAGE_MAP(TForm)
```

4.6 Programming the system tray icons

We have used the C++ Builder TrayIcon component which automatically minimizes the application to a system tray icon. After it has been minimized, it can handle regular events like any other control, like right clicks. It also supports popup menu on specified input. The `OnClick` event is used to allow the user to express feedback using a click on the system tray icon.

4.7 Client-server problems

We used the Indy Socket Library[1] available for C++ Builder to implement our client and server. The client showed many unexpected problems while running. There were numerous exceptions of the type "Socket operation on non-socket error". The lesson we learned is that every client socket function must be enclosed within a `try () catch(...)` structure to catch all exceptions. Please refer to the client code for exact code used for this.

Also, the `Connected()` method should be called before attempting to use a connection between the client and a server.

4.8 System detection code

The following describes how we can detect and record the detailed hardware and system information of a user machine.

4.8.1 CPU detection

CPU information like processor type, processor model, numbers of processors and so on are all detected. For example, we get information like "Intel x86, Family 15, Model 2, Stepping 7, 2000 MHz, MMX". Since the frequency of CPU is an important factor considering resource borrowing, we use the following four methods together to detect and record it, in case some method does not work in certain system:

- Calculate the CPU frequency dynamically
- Assembly code
- Querying Windows registry
- msinfo32.exe tool under Windows

4.8.2 Memory detection

We can detect detailed memory information like physical memory size, virtual memory size, page file size, page file number and so on. We use two methods together.

- Windows API (e.g. `GlobalMemoryStatus()` and `GetSystemInfo()`)
- msinfo32.exe tool under Windows

[1] http://www.nevrona.com/Indy/

4.8.3 Disk detection

We can detect detailed disk information of user machine. All disks (physical and virtual) and partitions information will be detected. Here we also use Windows API (e.g. `GetDiskFreeSpace()`) and the msinfo32.exe tool methods together to ensure correctness.

4.8.4 Network detection

We make use of the Windows network utilities, ipconfig.exe and route.exe, to detect all network (physical and virtual) related information.

4.8.5 System detection

By using msinfo32.exe tool, we can detect various hardware system information, for example:

```
System Name MINET-4
System Manufacturer     Dell Computer Corporation
System Model    OptiPlex GX260
System Type     X86-based PC
BIOS Version/Date   Dell Computer Corporation A02, 7/25/2002
SMBIOS Version 2.3
```

4.8.6 Operating system detection

Using the Windows API (e.g. `GetVersionEx()`) and the msinfo32.exe tool, we also detect detailed OS.

4.9 Generating the GUID

A Globally unique identifier (GUID) is needed to uniquely identify the client on the server side. GUIDs help in keeping the information received from the various clients separate from each other. The client GUID is creating using four values: the group id specified in the client setup, the IP address of the client, calling `UuidCreateSequential()` and extracting the MAC address part and the Disk Volume Serial Number got from Windows API call `GetVolumeInformation()`.

The entire string returned by `UuidCreateSequential()` is not used, because it was found to change its value with time. The MAC address portion is the only part that is fixed, so we extract that.

5

Conclusions and future work

5.1 Advice to implementors

Based on our study, we can offer the following guidance to implementors of distributed computing and thin-client frameworks.

- Borrow disk and memory aggressively, CPU less so.
- Build a throttle [23]. Your system can benefit from being able to control its borrowing at a fine granularity similar to the UUCS client.
- Exploit our CDFs (Figures 3.3-3.5) to set the throttle according to the percentage of users you are willing to affect. As we collect more data, the CDF estimates will improve.
- Know what the user is doing. Their context greatly affects the right throttle setting.
- Consider using user feedback directly in your application.

5.2 Conclusions

We have described the design and implementation of a system for measuring user comfort with resource borrowing, as well as a carefully controlled study undertaken with the system. The end result has three components. First, we provided a set of empirical cumulative distribution functions that show how to trade off between the level of borrowing of CPU, memory, and disk resources and the probability of discomforting an end-user. Second, we describe how resource type, user context (task), and user-perceived expertise affect these CDFs. Finally, we have made initial observations on how the time dynamics of resource borrowing affect the level.

Surprisingly, disk and memory can be borrowed quite aggressively with little user reaction, while CPU can also be borrowed liberally. Our observations formed the basis of advice for the implementers of distributed computing and thin-client frameworks. We are currently exploring how to use user feedback directly in the scheduling of these frameworks and applications.

5.3 Internet-wide study

Our controlled study at Northwestern helped us to answer and address many of the questions we raised in the introduction. We are now expanding both the scale and the questions through an Internet-wide study open to all participants. Any individual with a Windows computer is welcome to visit http://comfort.cs.northwestern.edu to download and run a copy of the UUCS client. The client is configurable by the user, including privacy options. We currently have about 100 users and are looking for many more.

In the Internet-wide study, the user uses the computer as normal. When user indicates any discomfort, the client records the context, the running processes, the contention levels, and other data similar to the controlled study. We plan to use this data to create better estimates for the aggregated resource CDFs (Figures 3.3-3.5), better understand the effect of context and the effect of the raw performance of the machine which was not studied in our controlled study.

5.4 Possible interaction between different exercisers

Each resource exerciser could affect the operation of other exercisers. Since disk exerciser threads also consume CPU, they could also contribute to CPU contention and thus affect the CPU results. The amount of interaction depends on the number of disk contention threads running. Beyond a certain number of threads, the disk threads may start having significant effects. This sort of interaction is one area of future study. CPU exercisers could also affect the disk exercisers, by not allowing the disk exercisers enough CPU time to do their disk processing.

5.5 Resource control using human feedback

The bulk of our work has focused on characterizing user comfort statically, with the goal of finding "do not exceed" limits on resource borrowing. However, we are also interested in using user comfort feedback directly, to dynamically find the limits and track them as the change with time. In this vein, we are exploring the use of the "one button" feedback approach to control the limit, trading off between the average time between button presses and the amount of CPU that can be consumed by the distributed application that is contending with the user.

Appendix A

Publicity

Website: The project website is located at *http://comfort.cs.northwestern.edu*. The website was designed to attract users to give an introduction to the project and also to attract people to the study.

Posters: Two posters have been designed. First poster was designed for the beta study and introducing people in the Computer Science Department to the project. The second poster is a call for participation for everyone to participate in the Internet-wide study.

The following is the text for the second poster which serves as a brief introduction the project:

CALL FOR PARTICIPATION
UNDERSTANDING USER COMFORT
WITH RESOURCE BORROWING

What is our Project about ?

As you may already be aware, today's desktop computer resources can be used for useful background computation like protein folding (folding@home), drug research and others (seti@home). Such computation is mostly done when the system is idle. Our goal is to find out how much of the resources can be borrowed from users' non-idle PC, still without affecting their work. This could result in tremendous benefits for distributed computing projects. It could also help us understand how much computing power is actually needed by individuals in their day-to-day tasks, which could be used to provide cheaper computing power to the users.

How can you help ?

We have developed an application that emulates resource borrowing. You can help us by installing this application and giving feedback whenever you feel that your PC's performance is below your expectations. On expressing feedback, it releases all borrowed resources and the PC immediately returns to normal.

The application runs in the background and feedback can be expressed simply by pressing a key or right-clicking the mouse. The installation is intrusion-free and doesn't affect your registry or any system files, and the application can be uninstalled easily and cleanly. Your feedback will give us good insights into understanding user behavior and contribute to our research.

You can download the application from http://comfort.cs.northwestern.edu/software.html. Please read the brief help file for a quick intro to using the application

A.1 Inviting controlled study participants

For publicizing our control study, we used posters and email. The following message was sent to different groups. The poster has similar text.

```
USER STUDY PARTICIPANTS WANTED!
$15 for your time!

Help advance the state of the art!

We are working to understand user irritation with resource borrowing in
computer systems.   We seek members of the Northwestern community
(faculty, staff, students) to participate in a user study to that end.
To participate, please email irritation@cs.northwestern.edu.   We will
arrange an appointment with you that will take roughly 1.5 hours.  You
```

will simply use a Windows computer for several tasks while we make it slower and faster in different ways. You'll just press an ''irritation button'' when the computer annoys you. That's it. We will pay you $15 for your time.

Bibliography

[1] ANDERSON, T. E., CULLER, D. E., AND PATTERSON, D. A. A case for networks of workstations. *IEEE Micro* (February 1995).

[2] BHOLA, S., AND AHAMAD, M. Workload modeling for highly interactive applications. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1999), pp. 210–211. Extended version as Technical Report GIT-CC-99-2, College of Computing, Georgia Tech.

[3] BOLKER, E. D., DING, Y., FLYNN, W., SHEETZ, D., AND SOMIN, Y. Interpreting Windows NT processor queue length measurements. In *Proceedings of the 31st Computer Measurement Group Conference* (December 2002), vol. 21, pp. 759–770.

[4] CHIEN, A. A., CALDER, B., ELBERT, S., AND BHATIA, K. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing 63*, 5 (2003), 597–610.

[5] CURTIN, M., AND DOLSKE, J. A brute force search of des keyspace. *;login:* (May 1998).

[6] DINDA, P. A. The statistical properties of host load. *Scientific Programming 7*, 3,4 (1999). A version of this paper is also available as CMU Technical Report CMU-CS-TR-98-175. A much earlier version appears in LCR '98 and as CMU-CS-TR-98-143.

[7] DOUGLIS, F., AND OUSTERHOUT, J. Transparent process migration: Design alternatives and the Sprite approach. *Software Practice and Experience 21*, 7 (July 1991), 1–27.

[8] EMBLEY, D. W., AND NAGY, G. Behavioral aspects of text editors. *ACM Computing Surveys 13*, 1 (January 1981), 33–70.

[9] ENDO, Y., WANG, Z., CHEN, J. B., AND SELTZER, M. Using latency to evaluate interactive system performance. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (1996).

[10] FREY, J., TANNENBAUM, T., FOSTER, I., LIVNY, M., AND TUECKE, S. Condor-g: A computation management agent for multi-institutional grids. In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC 2001)* (2001), pp. 55–66.

[11] GOOGLE CORPORATION. Google compute. http://toolbar.google.com/dc/.

[12] HUA CHU, Y., RAO, S., SHESHAN, S., AND ZHANG, H. Enabling conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of ACM SIGCOMM 2001* (2001).

[13] JANNOTTI, J., GIFFORD, D., JOHNSON, K., KAASHOEK, M., AND JR., J. O. Overcast: Reliable multicasting with an overlay network. In *Proceedings of OSDI 2000* (October 2000).

[14] KLEIN, J. T. Computer response to user frustration. Master's thesis, Massachusetts Institute of Technology, 1999.

[15] KOMATSUBARA, A. Psychological upper and lower limits of system response time and user's preferance on skill level. In *Proceedings of the 7th International Conference on Human Computer Interaction (HCI International 97)* (August 1997), G. Salvendy, M. J. Smith, and R. J. Koubek, Eds., vol. 1, IEE, pp. 829–832.

[16] LARSON, S. M., SNOW, C. D., SHIRTS, M., AND PANDE, V. S. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. In *Computational Genomics*, R. Grant, Ed. Horizon Press, 2002.

[17] LITZKOW, M., LIVNY, M., AND MUTKA, M. W. Condor —a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS '88)* (June 1988), pp. 104–111.

[18] MUTKA, M. W., AND LIVNY, M. The available capacity of a privately owned workstation environment. *Performance Evaluation 12*, 4 (July 1991), 269–284.

[19] REYNOLDS, C. J. The sensing and measurement of frustration with computers. Master's thesis, Massachusetts Institute of Technology Media Laboratory, 2001. http://www.media.mit.edu/~carsonr/pdf/sm_thesis.pdf.

[20] RIPEANU, M., FOSTER, I., AND IAMNITCHI, A. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal 6*, 1 (2002).

[21] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (2001).

[22] RYU, K. D., AND HOLLINGSWORTH, J. K. Fine-grain cycle stealing for networks of workstations. In *Proceedings of ACM/IEEE SC98 (Supercomputing '98)* (November 1998), pp. 801–821.

[23] RYU, K. D., HOLLINGSWORTH, J. K., AND KELEHER, P. J. Effi cient network and I/O throttling for fi ne-grain cycle stealing. In *Proceedings of Supercomputing '01* (November 2001).

[24] SHARMAN NETWORKS. The Kazaa Media Desktop. http://www.kazaa.com.

[25] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM 2001* (2001), pp. 149–160.

[26] SULLIVAN, W. T., WERTHIMER, D., BOWYER, S., COBB, J., GEDYE, D., AND ANDERSON, D. A new major seti project based on project serendip data and 100,000 personal computers. In *Proceedings of the Fifth International Conference on Bioastronomy* (1997), C. Cosmovici, S. Bowyer, and D. Werthimer, Eds., no. 161 in IAU Colloquim, Editrice Compositori, Bologna, Italy.