MEASURING THE IMPACT OF USABLE CONCURRENT FUNCTIONALITY AND THREADING ON TYPICAL GRAPHIC ARTS COMPUTATIONS

Eddie Jennings*, Ben Haley*, R.D. Holland*, and Tony Tassone*

Keywords: Computation, Graphics, Imaging, Multiprocessing

Abstract: The Windows NT™ operating system's support of multithreading and symmetric multiprocessing enables a new class of highly parallel software technologies. Software will soon be judged by performance gains achieved through parallelism. Available techniques for concurrent processing of typical graphic arts computational problems are presented. A new software technology called Usable Concurrent Functionality is described for presenting multithreaded applications to a computer user. A case study is presented showing the performance gains of image and color processing algorithms on single and multiple CPU computers. A standard benchmark for graphic arts software measuring operator productivity and throughput is proposed.

Introduction

The computer systems of today and tomorrow are challenging application software developers to invent new paradigms for presenting applications to the users of their products. The challenge lies in harnessing the raw horsepower of the computer architecture and the new features of the operating systems in ways that significantly increase the productivity of the user and the throughput of the software application (i.e., the work accomplished). Another challenge is to produce applications that will automatically scale in performance and throughput on introduced hardware innovations without significant re-design of those applications. Technologies such as symmetric multiprocessing (SMP) and threading are key components of producing such scalable software architectures.

In the Intel-based PC market, there are currently several dual Pentium based machines available (typically running Microsoft's Windows NT Workstation operating system). For servers, there are 2, 4, and 6 CPU (Pentium) machines also readily available (typically running Microsoft's Windows NT Server operating system). In fact, there is a

* Intergraph Corporation, Digital Imaging Department, Advanced Graphics and Development Systems Division, Mailstop IW17C4, Huntsville, AL 35894-0001, Email: rejennin@ingr.com

new category of machines known as *personal workstations*. A personal workstation is simply described as a computer built from standard PC components (CPU, RAM, disk controllers, graphics, network) that run standard operating systems and are capable of delivering the performance of traditional high-end UNIX workstations. Typically, the subsystems of a personal workstation are highly tuned (via careful system engineering) to produce workstation level performance. Dual-processor personal workstations are the most common today. The trend over the next 2 years is to the introduction of a new class of machine referred to as a *personal server*. A personal server is described as an extremely high performance machine capable of providing a dedicated service to the user (of a demanding application) while still filling the need of a workgroup server to other users. A typical graphic arts scenario would be an image processing/color correction/page assembly station running on a personal server that is also performing OPI picture replacement, printer management, and batch trapping for other users on the workgroup network. Personal servers will be differentiated from workstations by CPU horsepower (number of CPUs, clock speeds, etc.), peripheral capacity, and the use of a server-enabled operating system. The Windows NT scalable family of operating systems (Workstation and Server) is the logical choice for personal workstation and personal server machines. Its architecture was designed from the ground up supporting SMP machines and is itself fully threaded.

Workstations based on SMP technologies have existed under UNIX for several years. However, they never became mainstream for several reasons (cost, lack of SMP-enabled software applications, and the proprietary nature of workstations). The Open Software Foundation (OSF) operating system provided an SMP-enabled platform and has been well documented (Mui and Talbott, 1991).

The Macintosh and Power Mac platform that is so prevalent in the graphic arts is significantly behind in terms of SMP hardware architectures and operating systems. Norr (1995) recently reported that although the first Power Mac SMP machines (actually clones) have been announced for shipment in August 1995, Apple's plans for integrated SMP support within its operating system will most likely surface in late 1996 to mid-1997. Some interim solutions will be available from vendors to give applications access to multiple CPUs. However, these solutions will not be optimal.

Therefore, the fact that Windows NT is available today and was designed specifically for scalability in terms of SMP computer architectures coupled with the lack of true SMP support on the mainstream graphic arts platform leads the developers of serious applications (for which personal workstations and servers are intended) to Windows NT as the operating system of choice.

Threading Concepts

Concurrency is an attribute of a software design that allows a single application to execute independently in two or more code locations. Traditional multi-tasking operating systems allow concurrency between individual processes (i.e., programs).

Modern operating systems (such as the Windows NT family) are symmetric multiprocessing (SMP) and allow individual programs (processes) to be composed of threads. A *thread* is a unit of code in a process that can proceed independently of other threads. In simple terms a process is a task that is requested of the operating system while a thread is one independent subtask necessary to accomplish the overall task.

In an SMP operating system, a multi-tasking technique is employed when there are more threads waiting to run than there are CPUs available to run them. *Context switching* is used to switch the execution on a processor between the threads that are waiting to run. A thread's context is defined as a set of volatile registers representing the state of the processor, the user and kernel mode stacks used by the thread, and a private storage area (Custer, 1993). Context switching is the process of saving a running thread's context, loading the context of another thread, and beginning the execution of that thread. The saved context allows the original thread to return to execution at its next scheduled time slice.

Writing a multithreaded application is difficult. This is due to the fact that each thread within a process has access to that process's address space and resources. This can cause threads to "step on each other," produce random results, and produce hard to track software errors (such as deadlock). A key concept for providing order within a multithreaded application is *synchronization*. This provides the ability for a thread to wait for another thread to complete an operation before proceeding. Access to global data, peripherals, and application resources must be synchronized in a multithreaded application. To illustrate this concept, consider the following example. Thread A reaches a point where it needs to obtain a free memory buffer from the pool of previously allocated buffers. It determines that a buffer is available. It then reserves access to that buffer. To illustrate the potential problems in a multithreaded environment, assume that thread B also needs a memory buffer. If thread B looks to determine availability of a buffer after thread A has reserved its buffer, no problems would occur. However (due to context switching and parallel execution on multiple CPUs) it is possible that thread A could determine that buffer N is available. Then, thread B could also determine that N is available. Thread A reserves buffer N. Then, thread B also reserves buffer N. This represents an obvious serious flaw. The solution to the problem is to synchronize access to the data structure that controls the pool of memory buffers. In other words, only allow one thread to access this data structure at a time. This allows the sequence of reading the data structure (to determine availability of a buffer) and writing the data structure (to reserve a free buffer) to be indivisible (i.e., can't be interrupted by access from another thread). Synchronization is accomplished by such constructs as event objects, semaphore objects, code critical sections, and mutual exclusion.

Since a multithreaded application is difficult to build and most applications consist of many underlying libraries and services (some of which are not under the control of the primary application developers), the notion of code thread safety is important. A body of software is called *thread safe* if it is known to be able to be used within a multithreaded environment. In general, software is not thread safe if access to global data and resources

is not synchronized. Also, state driven software is generally not thread safe. Since the concepts of threading are relatively new, most existing software is not thread safe. This provides great difficulty in building applications for SMP architectures. In order to make a body of software thread safe, the developers must be concerned with the aspects of concurrency and learn to program "defensively." While this type of programming is well known to system programmers of operating systems and concurrent transaction based programs (databases, etc.), it is far from well known in the community of software application developers. Most developers have never been required to design and code software to avoid deadlock (as can easily happen in a multithreaded application when two or more threads need mutually exclusive access to two or more resources). Much education and retraining is necessary to build expertise so that truly concurrent applications can be built. Complicating this learning curve is the emergence of component software technologies (Adler, 1995) and applications composed of components largely unknown to the clients of the components. This makes the job of determining thread safety within an application a very difficult task. If components are truly unknown (in terms of their own thread safety), then localized threading within an application is the best that can be obtained.

The most common form of threading in use today is referred to as *black box threading*. In the black box approach, computationally difficult (i.e., time consuming) tasks are subdivided (i.e., multithreaded) in order to reduce the time required to execute those tasks. The typical case is to black box thread algorithms for better performance. This produces faster execution of tasks within an application but does not fundamentally alter the basic behavior of those applications. Other forms of threading include user interface threading and will be discussed in more detail later. Black box threading represents the most straightforward form of thread implementation and provides the simplest path to thread safety. In the next section, computational examples of black box threading applicable to the graphic arts are given.

Computational Concepts and Examples

The computational and threading techniques presented in this paper are applicable to a wide variety of problems. However, image processing is a discipline well suited to parallelism and is also very important within the graphic arts. Therefore, the remainder of this paper will focus on a class of image processing problems to demonstrate the concepts being conveyed.

Many traditional graphic arts image processing operations are applicable to the technique of black box threading mentioned above. Examples include unsharp masking, gradation, selective color correction, color separation, convolution filters, raster-based trapping, etc. Such algorithms are either pixel neighborhood independent (PNI) or pixel neighborhood dependent (PND). Examples of PNI operations are gradation, selective color correction, and color separation. These operations can be executed on a pixel without knowledge of the values of surrounding pixels. Examples of PND operations are unsharp masking, convolution filters, and trapping. In order to perform one of these

operations on a pixel, the surrounding pixel values must be accessed. In general, PND algorithms are more difficult to thread than PNI algorithms. In the following example, a convolution filter (PND algorithm) is demonstrated in terms of a possible threaded implementation. This implementation is provided for illustration purposes only and is not implied to be an optimal approach.

Recall that a convolution filter (Gonzalez and Wintz, 1987) is implemented digitally on an image in terms of a convolution matrix, M, of radius R (where R is typically 1, 2, or 3). Such a matrix is of dimensions (2R+1) x (2R+1). An image, I, can be notated as an N by M matrix of pixels, $P_{i,j}$. Let S(i,j) be the (2R+1) x (2R+1) subimage of I centered about pixel $P_{i,j}$ for values of i in {R+1,R+2, ..., N-R} and j in {R+1,R+2, ..., M-R}. The convolution of image I by matrix M is defined as an N by M image I' composed of pixels $P'_{i,j}$ where:

$$\begin{aligned} P'_{i,j} &= s\, f(S(i,j),M) + P_o \text{ for } i \in \{R+1,\dots,N-R\},\ j \in \{R+1,\dots,M-R\} \qquad (1) \\ &= P_{i,j}\text{, otherwise} \end{aligned}$$

and f is a pixel-valued function of two matrices defined as the sum of the component wise product of the entries of the matrices, s is a given scale factor, and P_o is a given pixel offset vector.

Now, assume that this convolution calculation is desired to be split into T threads. Putting this in context, typically N and M are large numbers (in the range of 500 to 4000), R is a small number (in the range of 1 to 3), and T is also a small number (in the range of 2 to 10). The following algorithm is based on splitting image I into T strips, $I_1, \dots, I_T$. Let N_T be $\lfloor N / T \rfloor$. Define strip I_i to be the N_T by M submatrix of I formed by taking rows starting at $(i-1)N_T+1$ in I (padding the last strip I_T if N is not an integral multiple of T). The basic idea is to process each strip independently in separate threads. However, the strips depend on each other at their boundaries (due to the PND nature of the convolution calculation). To solve this problem, define the boundary i (where $i \in \{1,\dots,T-1\}$) between strip i and i+1, B_i, to be the (2R+1) by M image consisting of the first R+1 rows of strip i+1 and the previous R rows of strip i. These boundary images B_i represent those pixels that are within the convolution matrix neighborhood for the edge pixels within each of the image strips. Note that the number of all pixels in all boundary images is a small percentage of the overall number of pixels in the original image I (for all realistic cases). For example, when N=M=2000 (a 2000x2000 pixel image), R=2 (5x5 convolution matrix), and T=4, the percentage of pixels in the set of three boundaries to the overall number of pixels in the image is only 0.75%. Figure 1 below depicts the strips of image I and the boundaries between those strips.

Now, the threaded algorithm can be summarized as:

1) Copy each of the boundary images into separate data structures.
2) Run the convolution calculation from equation (1) on the original image pixels corresponding to the copied boundary images.

3) Swap the contents of the copied boundary pixels with those calculated in step 2 above. This results in restoring I to its original state while storing the new values of pixels in I corresponding to the boundary images into separate data structures (for use later).
4) For each image strip I_i, create a thread to process the interior pixels of that strip (using equation (1)). The interior pixels are defined as rows R+1 through N_T - R.
5) Synchronize waiting on all threads to complete.
6) Copy the saved new boundary pixels back into their original locations from image I.

The bulk of the processing occurs in step 4 where the threading occurs. Improvements could be made in the above simplified algorithm by minimizing the amount of work done outside the threaded and synchronization steps.

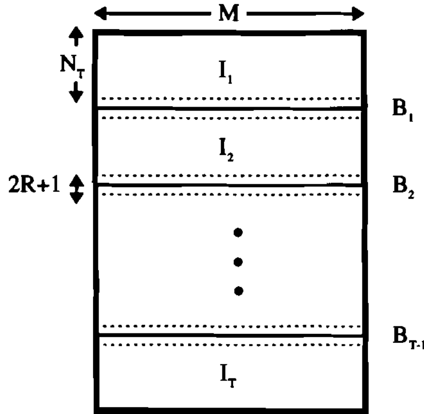

Figure 1. Image Strips and Boundaries.

Usable Concurrent Functionality

All of the threading concepts presented thus far have been black box in nature. The net result of these is to make the slower calculations in an application run faster on an SMP computer architecture. However, none of these techniques help the application user on a single-CPU machine. Also, none of these techniques represent a fundamental change in the computing paradigm for software applications. In this section, *Usable Concurrent Functionality (UCF)* is introduced as a new software technology that does represent a computing paradigm change. It addresses both single and multiple CPU machines in order to optimize user productivity and system throughput.

UCF is termed as usable due to its ability to present a highly concurrent software application to the user in an extremely intuitive and user friendly manner. As mentioned above, UCF (and threading in general) applies to a wide class of application problems. The examples given in this paper are all image processing in nature due to the relevance

to the graphic arts industry. Before describing the technical details and behaviors of a UCF system, the general needs of users and computing that lead to UCF is presented.

The traditional approach to the design of interactive software applications is based on processing the user's request immediately (tying up the computer resource until the request is completed). However, this approach is not well-suited for a typical user scenario. In this scenario, the user:

1) Thinks about the next task
2) Decides what to do
3) Tells the computer about the decision (i.e., what to do)
4) Waits while the computer works (processes)

Then, these four steps are repeated for the next task. This simple sequence of steps is shown in Figure 2 below. Steps 1, 2, and 3 are often accomplished while viewing the graphical user interface (GUI) of the application. If the GUI is not available during step 4, user time is wasted while waiting for the computer to finish processing the current task. Consider the advantage if the computer worked on accomplishing the current task while steps 1-3 are being performed by the user for the next task. In other words, the software application lets the user work at his / her pace and does not artificially limit that user to the pace of the computer. UCF is the embodiment of this concept.

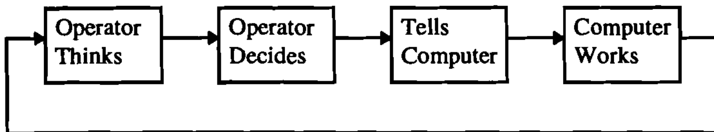

Figure 2. The Typical Scenario.

It is obvious that in many cases a user cannot "think / decide / tell" until the results of the previous task are complete. However, in some instances the user could "think / decide / tell" the computer about another unrelated task within the same application while the current task is processing.

Consider why this concept is important. For the casual user of a software application this is not vitally important. However, for the serious production user that constantly uses an application to produce results, any boost in productivity is important in many ways. First of all, consider the value of the user's time. The output of the user (i.e., the work accomplished by the user with a given software application) is proportional to the value of that user to his / her employer. In other words, the more work the user produces from the computer, the more return the employer gets on that user's salary and the investment in the computer and software. It is that type of user to which UCF systems is important. This type of user has three important attributes:

1) User time is important (costly).
2) Lengthy operations are typical (most tasks are time-consuming for the computer to accomplish).
3) Multiple jobs are always in progress (the user always has multiple jobs pending to be completed on the workstation).

UCF addresses users possessing these attributes. The following section describes the details and behavior of a UCF system.

UCF Details

UCF is an approach to designing and presenting a software application in a manner allowing the user to always interact with the application regardless of tasks in progress. A full implementation of UCF requires an application to be designed from the ground up with concurrency in mind. All contributing software must obey certain rules (or protocols) in order to participate in the UCF architecture. The following description is in terms of the graphical user interface (GUI) of an application implementing UCF.

Central to UCF is the ability to process data in the foreground or background under user control. The user can push any task to the background that requires more time than a time interval known as the user tolerance. The default value for user tolerance is 5 seconds (although it is configurable by the user). Any command initiated requiring more than the user tolerance to complete will (within a half a second of initiation) display a dialog box. This dialog box shows the current status of the command (a bar chart showing the percentage complete). It also contains two buttons: Cancel and To Background. The Cancel button allows the user to abort the command (with full cleanup of any partially accomplished results) for all operations for which it is feasible to implement such a cancellation feature (the majority of commands should support cancellation). The To Background button takes the currently executing command and pushes it to the background.

The To Background button is the default button on the dialog box (allowing the user to push the command to the background by simply pressing the Enter key). A preference is available that enables all such commands to be automatically pushed to the background (without explicit user action on a command-by-command basis). Once in the background, the user is free to work with the application. However, the data which is being operated on is locked until the command is complete. Users are able to access locked data when appropriate. The menus of the application reflect the legitimate operations that can be performed on a piece of data (an element) with executing (or pending) background tasks already specified. In some cases, a foreground task can be pushed to the background even when there are currently executing (or pending) tasks on an element. Therefore, some background tasks are queued to begin after completion of others.

In order to understand the types of tasks that are legitimate on locked data, all tasks to be performed are characterized as one of five types:

1) Immediate read - the data is used in read only mode and access to the data is part of the GUI of the command.
2) Deferred read - the data is used in read only mode and is not accessed during the GUI of the command.
3) Deferred write - the GUI specifies a change to the data without referencing the actual data. Processing the data is initiated automatically or by the user choosing an OK button from a dialog box.
4) Deferred write with GUI read - the GUI specifies a change to the data and references the actual data during the GUI (e.g., by providing a visual simulation of the results of the command). Processing of the data is initiated by selecting an OK button from a dialog box.
5) Immediate write - the data is directly written during the command's GUI.

In the above definitions, data refers to any entity that is perceived and interacted with by the user (e.g., images, other graphical elements, files on disk, etc.). The rules which dictate legitimate tasks on locked data include:

- When no background tasks are defined, any task can be initiated.
- When an immediate read or deferred read task is pending, any type of task other than an immediate write can be initiated. Write tasks are queued to begin only after the immediate read or deferred read completes. Read tasks are allowed to operate concurrently with the pending immediate read or deferred read task.
- When any form of a write task is pending, a deferred read or a deferred write can be initiated (and will be queued to begin only after the pending write task completes). In some cases, a deferred write with GUI read command is "demoted" to a deferred write command (the GUI read aspect of the command is disabled) and is also allowed to be initiated.

Certain classes of tasks are known to be a heavy burden on the computer. Therefore, no more than one (by default) task of a class can be executing concurrently. These classes of tasks are controlled by queues (processed in a first-in-first-out [FIFO] order by the UCF architecture). In reality, queues have properties which can be configured by the user to allow more than one task in a queue to execute concurrently.

Having described pushing tasks to the background, the next discussion covers how the user interacts with them and controls the application. This control is provided with a Background Manager command. This command displays a dialog box (implemented as a persistent dialog called a palette) that shows in a scrolling list all currently executing or pending tasks and queues. The information shown on this dialog is constantly updated. Optionally, this dialog can be dismissed by the user. The purpose of this dialog is to show the state of the system. The Background Manager dialog tells the user everything about what is going on in the application. Included in the scrolling list of tasks and queues is the status of the task, the name of the task (the command chosen by the user), a description of the data accessed by the task (e.g., file name, document name, thumbnail display of the

data, etc.), and the percentage complete. A preference is provided that enables the user to restrict the number of tasks allowed to execute concurrently (providing system tuning). The various states that a task can be in (and shown in the status field) are:

1) Processing - the task is currently executing.
2) Suspended - the task has been suspended by the user.
3) Ready - the task is ready to begin execution (it is not blocked by any prerequisite task) but has not yet started (e.g., there are too many tasks already executing).
4) Waiting - the task is blocked by another task (it is queued to begin only after a prerequisite task completes).
5) Aborting - the task received an abort but has not yet started cleaning up.
6) Cleaning up - the task is cleaning up from the abort request.
7) Completed - the task completed and will be removed from the Background Manager dialog on the next refresh.

Features are provided by the Background Manager to control the workload of the workstation. In particular, options are available allowing:

- A task to be suspended (execution of the task stops and waits for a resume or abort).
- A suspended task to be resumed.
- A task to be aborted with full cleanup (if applicable).
- All tasks to be paused (the execution of all tasks stop and wait for a resume or abort action to be initiated).
- All currently paused tasks to resume processing (with previously suspended tasks remaining suspended).
- A queue to be paused.
- A currently paused queue to be resumed.

In addition, a detailed view of any particular task can be displayed allowing more information and controls for the task. The most important piece of additional information on the detailed view dialog is a list of all tasks blocking that task (if any). A choice can be made so that the user is notified when the task completes (by a beep, a message displayed in the status bar, and / or displaying an alert dialog box). Lastly, the priority of the task can be established / changed. The standard priorities are labeled Very High, High, Medium, Low, and Very Low. In addition, two special priorities are available: Exclusive (meaning this task is the only background task allowed to run) and Intermittent (meaning this task only runs if no other background tasks are also running). An automatic priority boosting / lowering scheme is used to maintain the interactive performance of foreground tasks while background tasks are also running.

All of the Background Manager controls allow the user to regulate the execution of the tasks on the computer. Rush jobs can be accommodated with suspend / resume / abort features as well as priority setting. The Background Manager intuitively constrains the user from getting queued tasks out the correct order by using these controls. Tasks queued to operate on a piece of data must complete in the order they are submitted in order to get

the desired result. Independent tasks (not tied to the same piece of data) can complete in any order.

Another aspect of the Background Manager is that of an activity log. Tasks are able to post information to the activity log. The most common information posted is related to errors encountered in processing. The user can view a list of all activity log entries and can activate a detailed activity log for a particular task. In this way, a user can determine any information that a background task needed to convey during processing.

Issues with the implementation of UCF include what to do when a user attempts to exit the application while background tasks are executing. The developer must also take care of the case where the user attempts to shutdown the computer while background tasks are still running. Another area to be handled is error recovery. Background tasks that encounter an error (that could be recovered by interacting with the user) need to gracefully (and non-intrusively) request the attention of the user, allow correction of the problem, and then proceed. Implementing a UCF-enabled system can be quite complex. Adopting a strict object-oriented modeling and design methodology has proven to simplify this complex process (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen, 1991).

The basic goal of a UCF system is to prevent the user from having to wait on any operation to complete before continuing to work. For image processing systems, alternate approaches have been employed (such as recording image processing operations to a low-resolution file and batch processing the operations on the high-resolution image later). These types of deferred processing systems differ from the incremental computing approach of UCF. With UCF, it is possible for an operator to fully utilize most of the available computing cycles of a computer while accomplishing the end result sooner than with deferred processing systems. This is an important aspect of UCF given recent observations (Carriero, Freeman, Gelernter, and Kaminsky, 1995) that most desktop computers are idle much of the time. A high performance personal workstation or server should rarely be idle for optimum utilization of that computing resource. With UCF and this concept of incremental computing, the tasks are accomplished in less overall time without impairing the usability of the computer and application. This increases operator productivity and overall system throughput.

Case Study

A usability case study of an UCF-based image processing / color correction application that also used black box threading for all of its algorithms (developed as an R&D effort at Intergraph) was conducted to obtain quantitative timing results. The system used for this study was an Intergraph TD-4 personal workstation. The TD-4 used consisted of two 90 Mhz Pentium CPUs, a 512 Kbyte secondary cache, 112 Mbytes of RAM, a PCI-based fast SCSI-2 I/O system, two 1.1 Gbyte disk drives, a PCI-based Ethernet adapter, an Intergraph GLZ (OpenGL accelerated) graphics adapter with 24 Mbytes of video RAM, and a 21" monitor running (with the GLZ adapter) at a screen

resolution of 1280x1024. The operating system was Microsoft's version 3.5 of Windows NT Workstation.

The operations included in the case study were gradation change, unsharp masking, selective color correction, color cast removal, and defect removal cloning. In terms of the UCF architecture, each of these operations (except cloning) was implemented as a deferred write with GUI read. Each was demoted to a deferred write whenever a pending task was defined on the target image of the operation. The clone operation is an immediate write command. The images used in the case study were a combination of grayscale, RGB, and CMYK data types with in memory sizes ranging from 4 Mbytes to 50 Mbytes.

By isolating the performance improvements due solely to the black box threading of algorithms (via timing a dual CPU configuration vs. a single CPU configuration), it was determined that such algorithms represent an average improvement of 1.75 (single CPU time required divided by dual CPU time required) or phrased alternatively as a 75% improvement. Peak improvements were near 1.9 (90%). Obviously 2.0 (100%) represents the best possible ratio on a dual CPU machine achievable solely with black box threading. The average improvement of 1.75 was quite good considering areas of the system implementation that were not threaded (demand tile swapping, softedge masking, and copying of data between buffers). This result is consistent with the report by Norr (1995) stating that the manufacturer of a leading desktop imaging application reports that a second Pentium CPU generally yields a 60% to 90% increase in speed in that application. While the performance gains achieved by black box threading are impressive and important, they are not fundamental in nature. To see this, consider a dual CPU machine running a black box threaded application. As shown in this case study, it is possible to accelerate the application by a factor of almost two. However, that is the standard performance increase achieved by subsequent generations of microprocessors. Therefore, a dual CPU machine today is similar in performance to the next generation of a single CPU machine (when only black box threading is employed).

Timing the improvements due to the user interface threading of UCF turned out to be quite difficult. That is the reason a call for an industry standard benchmark for measuring operator productivity and system throughput is made in the next section of this paper. However, a few meaningful observations were made during the case study. For the operations tested, the "think / decide / tell" cycle represented an average of 27% of the time that was required to complete the operation. During the majority of this time, the computer is typically idle. In a UCF system, this idle time is virtually eliminated (since the computer is working on completing previous tasks while the operator is "thinking / deciding / telling" for the current task). In a traditional system, the operator is more subject to distractions causing even less utilized computer cycles. This occurs as the operator is waiting on the current task to finish before being able to "think / decide / tell" for the next task. If distracted while waiting, then the "think / decide / tell" cycle won't begin as early as possible. UCF addresses this by allowing the user to continue to work at all times. Figure 3 below depicts the increase in utilization of computer cycles from a

traditional system to a black box threaded system to a UCF system also using black box threading. In addition, consider an 8 hour shift where an operator is allowed a 45 minute meal break and averages a 15 minute break every two hours. This down time represents 22% of the available time in that shift. A trained operator of a UCF system can easily eliminate most of this down time by getting ahead of the application (by pushing tasks to the background) prior to each of these down times. In this scenario, very few of the available computing cycles would be wasted. Further gains could be realized by effective operators getting far enough ahead of the application so that a substantial portion of the following shift would be consumed by the computer catching up with the tasks pushed to the background. This could reduce labor costs within shops running multiple shifts or increase the throughput of shops running single shifts.

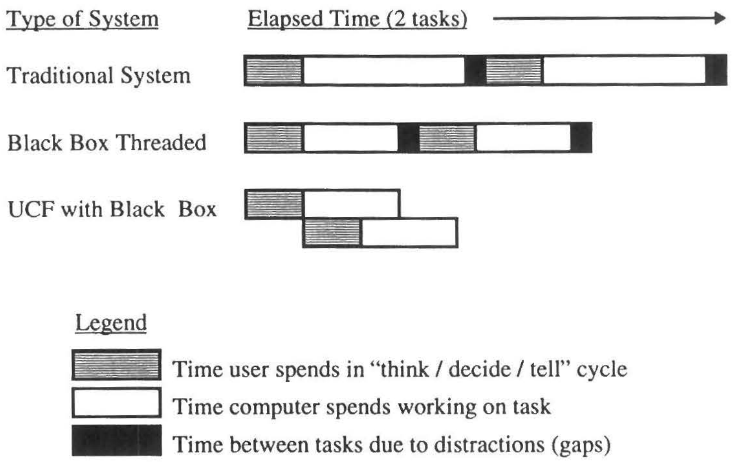

Figure 3. Computer Utilization Comparison.

The priority scheme implemented in the UCF-based system proved quite effective at allowing the interactive defect removal cloning to occur in the foreground while multiple tasks were allowed to execute in the background. The feeling to the operator was that the cloning was not perceptibly impaired by the tasks processing in the background. The automatic priority boosting / lowering of the UCF implementation for foreground / background tasks were key to this result.

Future Work - Proposed Benchmark

As mentioned above, testing and timing a UCF-enabled application to determine true performance improvements is difficult. It is well known within the computing industry that benchmarks are used to quantify performance of hardware platforms. However, the advent of commercially available SMP computers running SMP-enabled operating

systems and the emergence of software applications exploiting these computers and operating systems has created a new situation in the industry. This situation is that system performance is now very much a function of the architecture of the software application. A carefully threaded application employing black box threading and UCF can substantially out perform even the best black box threaded application (not to mention non-threaded applications). Also, a UCF application is also superior on a single CPU computer. However, these results cannot be effectively quantified by only measuring the speed of individual commands (operations). A more global measurement (i.e., a benchmark) is needed for this quantification.

The desired benchmark should attempt to measure the true operator productivity and system throughput of the combination of the software application, the hardware platform, and the underlying operating system. Examples of measures of system throughput of interest in the graphic arts include number of scans color corrected per shift, number of pages trapped and assembled per shift, number of PostScript files imaged per shift, etc. The result of the benchmark could be used to isolate the performance differences between competing software applications (by holding constant the underlying hardware and operating systems).

Conclusions

The future promises significant changes in the way software is developed for demanding disciplines. There is a long learning curve to be able to effectively implement a UCF like system and a moderate learning curve to be able to implement black box threading. As shown in this paper, the results for the user (and shop owner) are impressive when an SMP computer is coupled with the Windows NT operating system and a properly threaded application (that includes both black box threading and UCF). The software industry appears to be generally behind the capabilities of the hardware in terms of writing applications that can utilize most of the available computing cycles. The techniques presented in this paper prove to be effective at capturing the vast majority of these available cycles. The future holds an exciting time where computers are able to be upgraded with "banks" of additional CPUs to boost the performance of all properly written applications (lessening the need for dedicated special purpose accelerator cards in wide use today). Such upgrades will be viewed by tomorrow's user exactly like upgrades of adding more RAM or disks are viewed today. Technologies like UCF are important for making this vision of the future a winning proposition for the end user of software applications.

Acknowledgments

We thank our colleagues in the Digital Imaging Department of Intergraph for assisting in the preparation and production of this paper. In addition, we thank the executive management at Intergraph for allowing this information to be made public.

Literature Cited

Adler, R.
1995"Emerging Standards for Component Software", Computer (IEEE Computer Society), vol. 28, no. 3, pp. 68-77.

Carriero, N., Freeman, E., Gelernter, D., and Kaminsky, D.
1995"Adaptive Parallelism and Piranha", Computer (IEEE Computer Society), vol. 28, no. 1, pp. 40-49.

Custer, H.
1993"Inside Windows NT™" (Microsoft Press, Redmond, WA), pp. 90-104.

Gonzalez, R. and Wintz, P.
1987"Digital Image Processing" (Second Edition, Addison-Wesley Publishing Company, Reading, MA), pp. 81-90, 186-190.

Mui, L. and Talbott, S.
1991"Guide to OSF/1 - A Technical Synopsis" (O'Reilly & Associates, Inc., Sebastopol, CA), pp. 2-1 – 2-13.

Norr, H.
1995"Multiprocessing to define desktop in next PC wave", MacWeek (Ziff-Davis Publishing Co., New York, NY), vol. 9, no. 12, pp. 1, 84.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W.
1991"Object-Oriented Modeling and Design" (Prentice-Hall, Inc., Englewood Cliffs, NJ), 500 pp.