Thread-level Parallelism and Interactive Performance of Desktop Applications

Krisztián Flautner
manowar@engin.umich.edu

Rich Uhlig
richard.a.uhlig@intel.com

Steve Reinhardt
stever@eecs.umich.edu

Trevor Mudge
tnm@eecs.umich.edu

University of Michigan
1301 Beal Ave.
Ann Arbor, MI 48109-2122
+1-734-764-0203

Intel Microprocessor Research Lab
5350 NE Elam Young Parkway
Hillsboro, OR 97123
+1-503-696-3154

Abstract

Multiprocessing is already prevalent in servers where multiple clients present an obvious source of thread-level parallelism. However, the case for multiprocessing is less clear for desktop applications. Nevertheless, architects are designing processors that count on the availability of thread-level parallelism. Unlike server workloads, the primary requirement of interactive applications is to respond to user events under human perception bounds rather than to maximize end-to-end throughput. In this paper we report on the thread-level parallelism and interactive response time of a variety of desktop applications. By tracking the communication between tasks, we can focus our measurements on the portions of the benchmark's execution that have the greatest impact on the user. We find that running our benchmarks on a dual-processor machine improves response time of mouse-click events by as much as 36%, and 22% on average—out of a maximum possible 50%. The benefits of multiprocessing are even more apparent when background tasks are considered. In our experiments, running a simple MP3 playback program in the background increases response time by 14% on a uniprocessor while it only increases the response time on a dual processor by 4%. When response times are fast enough for further improvements to be imperceptible, the increased idle time after interactive episodes could be exploited to build systems that are more power efficient.

1. Introduction

Does multiprocessing make sense on the desktop? There is anecdotal evidence regarding the positive effect of multiprocessing on the "responsiveness" of interactive applications. Intuitively, the premise makes sense: sudden bursts of background activity can be handled concurrently with the foreground task and individual processes can be sped up if they are composed of multiple threads. In this paper we investigate whether multiprocessing can indeed affect the user-perceived response time—the time it takes for the computer to respond to user initiated events—of interactive desktop applications. The primary questions that we deal with are the following:

- How much do threads run concurrently in interactive desktop applications?
- Does concurrency translate into improved interactive performance (response time)?

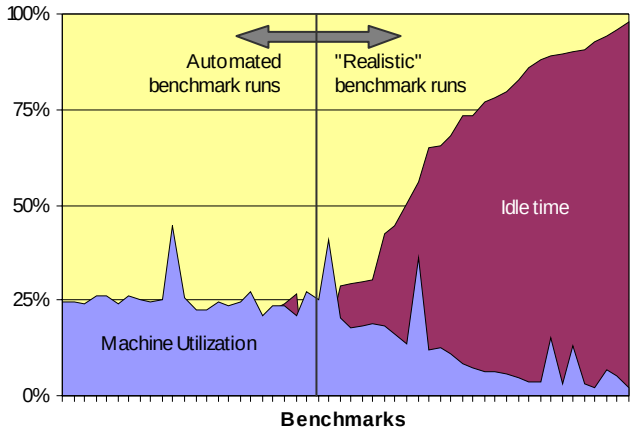
These questions are particularly important for processor designers who are considering techniques that exploit thread-level parallelism, such as simultaneous multithreading (SMT) [13] and single chip multiprocessing (CMP) [7]. To date, most research in this area has used either synthetic workloads (e.g., concurrently running multiple SPEC benchmarks) or server workloads [1][10][12]. Our results show that the performance characteristics of these benchmarks are very different from those of desktop workloads. Although our measurements were made on a multiprocessor, we look at thread-level parallelism at the system (application and OS) level, not at the microarchitecture level. Thus our results are not particular to any specific architecture for exploiting thread-level parallelism.

Over the past years, multiprocessors have moved from the server segment to workstation users and are now entering the desktop arena as well. Recently, Apple Computer made dual PowerPC based machines standard across most of its desktop PowerMac line [16]. Moreover, processors capable of executing multiple instruction streams concurrently will be readily available in the near future. SMT- and CMP-based products have already been announced [3][4][15] and the cost of existing multichip multiprocessors have been decreasing steadily [16].

Figure 1 illustrates the machine utilization and idle time characteristics of 52 desktop workloads across three operating systems (Windows NT, BeOS, and Linux) running on a quad-processor machine (using data from [6]). Machine utilization is a measure of how effectively a machine's computing resources are exploited and is 100% if all processors

ASPLOS 2000 **Cambridge, MA** **Nov. 12-15 , 2000**

FIGURE 1. Machine utilization and idle time of 52 workloads



The figure shows machine utilization and idle time of 52 workloads on a quad-processor machine. The automated benchmarks were driven by a GUI automation tool (e.g., Visual Test), while the realistic workloads were run by a human.

in the system are utilized. It would be difficult to make a case for multiprocessing for desktop workloads based on the presented machine utilization data. Very few of the workloads exceed 25% machine utilization, suggesting that for the most part, only one out of the four available processors is exercised. Some of the benchmarks exhibit even lower utilization in the 5% to 10% range, suggesting that a single processor is more than powerful enough for running these workloads. What is the need for multiprocessing when a single processor seems to be adequate?

The problem with the machine utilization metric is that it weighs all parts of the benchmark's execution equally. From the metric's point of view, generating a page of output on the screen is as important as the idle period when the user is consuming the data (think time) and the processor is doing no useful work. We define idle time as the percentage of time all processors in the machine are idle simultaneously. Machine utilization can only be used to accurately measure the effects of concurrent execution if idle time during the benchmark run is close to zero.

As the figure shows, the amount of idle time in desktop applications can be very large. In some cases idle time amounts to more than 90% of total execution time. This high ratio should not be unexpected since interactive applications run at the rate at which the user interacts with them, which is determined by human cognition and motor skills and includes significant think time. The high proportion of idle time can be obscured by the use of automated benchmarks, such as Sysmark 98 [17] or Winstone 99 [18], that perform each operation as soon as the previous operation completes without taking think time into account. The automated benchmarks in Figure 1 have an average of 12% idle time versus 64% for the realistic benchmark runs.

To get around the limitations of the machine utilization metric, we define a new metric called thread-level parallelism (TLP). TLP is the machine utilization over the non-idle portions of the benchmark's execution. This definition sidesteps the problems of the original metric and allows us to use more realistic desktop workloads that can include a lot of idle time. Intuitively, TLP is a metric of speedup due to concurrent execution on the non-idle portions of the workload (Section 3).

The most relevant metric for interactive applications is not the overall throughput but response time: the amount of time it takes for the computer to respond to a user initiated event. These periods are also referred to as interactive episodes. We focus our measurements on the interactive episodes by tracking communication between tasks in the kernel (Section 3.1). Figure 2 illustrates a sample TLP trace where the ranges corresponding to interactive episodes have been highlighted.

Our response time measurements are detailed in Section 4. We find that, while desktop applications can incur more than 90% idle time during execution, running our benchmarks on a dual-processor machine provides a 22% average improvement in application response times. In Section 4.4, we investigate the effects on response time of a concurrently running MP3 playback application. Here, the dual-processor machine improves response time by 29% on average. Thus, multiprocessing on the desktop can be a viable means of improving the user experience.

2. Previous work

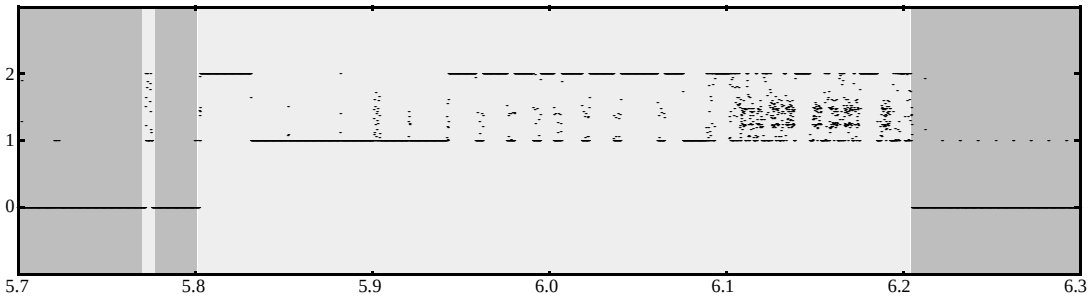
Various papers have dealt with the characterization of desktop applications [9][2][5]. In [5], Endo et al. performed a detailed analysis of interactive performance in a uniprocessor Windows NT environment. Our methodology has been influenced by their design choices and their definitions of think and wait time.

Hauser et al. [8] have approached the role of threads in interactive systems by analyzing the design patterns in two threaded object-oriented environments. Their analysis focused on the use of threads for program structuring instead of run-time statistics. However, their conclusion that most threads are used for programmers' convenience and few for exploiting concurrency is echoed by our results.

In their study of the characteristics of desktop applications [9] Lee et al. observed that most of the instructions are executed from a single dominant thread. In our experience TLP can vary greatly based on the choice of OS and workloads. In this study we show that multiprocessing can have beneficial effects even on standard desktop workloads.

Our previous investigations into the concurrency characteristics of desktop applications have provided a high-level view of a broad set of workloads on three operating systems: Windows NT, Linux, and BeOS [6]. These results showed that while most workloads under BeOS and Windows NT use a relatively large number of threads, the actual concurrency derived from them is limited and heavily dependent on the workload and the operating system. In this study we expand on these results by focusing in more detail on interactive applications under Linux. On a previous version of Linux that we studied, applications exhibited very little concurrency (TLP of 1.0-1.13). This was partially due to the fact that many of the applications did not use kernel threads to work around reentrancy bugs in the C library, and to the heavy use of the global kernel lock in the 2.2.13 kernel. However, less than a year later, due to a more recent

FIGURE 2. Segment of the TLP trace of the Ghostview benchmark



This figure contains a portion of the TLP trace of the Ghostview benchmark. The light areas correspond to the interactive episodes. In this trace, there are two interactive episodes: a short one from 5.772s to 5.775s and a long one from 5.802s to 6.205s. The y axis corresponds to the measured TLP, while the x axis specifies the elapsed time in seconds.

kernel (2.3.99-pre3) and an updated C library, the average TLP of our Linux benchmarks has increased to 1.27.

3. Metrics and methodology

The principle metrics that we are interested in are idle time (Idle), thread-level parallelism (TLP), and response time (T_R). Idle time is the fraction of measurement time when all of the processors in the system are executing the idle task. Thread-level parallelism, on the other hand, is a measure of how many threads are executing concurrently when the machine is not idle. These two quantities provide a more accurate insight into workload characteristics than machine utilization (MU) alone. While machine utilization can give an accurate picture of the concurrency of the system if idle time is close to zero, it can obscure the presence of concurrent execution in the case of interactive applications, where idle time is high. Response time (T_R) is the length of time between the initiation and completion of an interactive event, which we also refer to as the length of an interactive episode.

The following equations precisely define machine utilization (MU), the TLP metric and their relationship.

$$MU = \frac{\sum_{i=1}^{n} c_i i}{n} \qquad \text{(EQ 1)}$$

We use the variable c_i to denote the fraction of time that exactly i threads execute concurrently. The value of i ranges from 0 to n, where n is the number of processors in the machine. Equation 1 shows the formula for machine utilization, while Equation 2 shows the computation for TLP. The variable c_0 represents the fraction of time that the machine was idle. Correspondingly, idle time is defined as the fraction of time when all processors in the machine were idle simultaneously.

$$TLP = \frac{\sum_{i=1}^{n} c_i i}{1 - c_0} \qquad \text{(EQ 2)}$$

Equation 3 relates machine utilization to TLP, which in essence is the machine utilization over the non-idle portions of the program execution. Note that the result is scaled by n since machine utilization ranges from 0 to 1 while TLP ranges from 1 to n.

$$TLP = \frac{nMU}{1 - c_0} \qquad \text{(EQ 3)}$$

In certain cases, we use a subscript to show the range on which our metrics were computed. In particular, TLP_{ie} gives TLP for the interactive episodes and TLP_{run} shows the thread-level parallelism for the entire benchmark.

Our measurement technique relies on intercepting thread switch events in the OS kernel and keeping an accurate trace of the executing threads on all of the CPUs in the system. Since the timestamp counters are synchronized across all processors in the machine, we use the time stamps associated with thread switch events to compute how much the processors in the system execute concurrently. We have confirmed that the counters under Linux are synchronized to within 100 cycles, well below the microsecond resolution we desired. This methodology allows us to monitor the execution of all threads in the system, not just the ones in the running benchmark, and takes all scheduling and synchronization overhead into account.

Figure 3 shows the hardware and software configuration of our benchmarking environment.

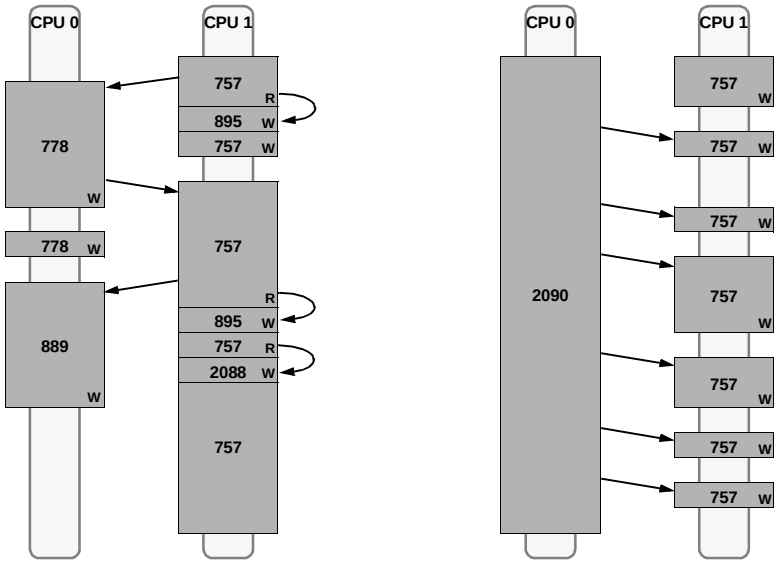
3.1 Detecting interactive episodes

The beginning of an interactive episode is initiated by the user and is usually signified by a GUI event, such as

FIGURE 3. Benchmarking environment configuration

Hardware configuration	Software configuration
Dell Precision WorkStation 410	Linux Mandrake 7
Two 450Mhz Pentium II	Modified 2.3.99-pre3 kernel
512K L2 Cache	XFree86 3.3.6
512M RAM	Helix GNOME 1.2
Matrox Millennium II AGP 4M	glibc 2.1.3 C library

FIGURE 4. Trace fragments illustrating tasks and communication events



Two typical trace fragments are shown in this figure from the Ghostview benchmark. The first picture shows communication events between tasks after a mouse button was pushed, while the picture on the right corresponds to a complex image being rendered on the screen. The letter R to the right of the pid shows that the task was preempted (the task is ready to execute), a W indicates that it is waiting for an event and gave up time on its own (it is waiting for an event to complete). The arrows indicate the communication flow between the tasks. Task pids in the figure correspond to the following: 757 - X server, 778 - sawmill (window manager), 895 - gnome-terminal, 889 - tasklist_applet, 2088 - Ghostview (gv), 2090 - Ghostscript (gs). Note that Ghostview uses Ghostscript to render pdf and postscript data. In these examples, when a task is waiting for an event, it is blocked in the select or the poll system call.

pressing a mouse button or a key on the keyboard. Finding the end of an episode is more difficult since there is no event that automatically gets generated when the computer is done responding. One approach is to assume that user initiated events are CPU bound and to define the end of an episode as the beginning of a relatively long idle section [11]. The length of an interactive episode is thus the elapsed time between a user initiated event (e.g., a mouse click) and the beginning of the next idle period that is longer than a predefined threshold. There are two problems with this approach:

- Episodes that are I/O bound may be terminated prematurely if the wait time exceeds the idle threshold.
- There is a significant latency between the end of an interactive episode and its classification, complicating online use of episode information (e.g., for scheduling).

We developed a more robust episode detection mechanism to alleviate these problems. To find interactive episodes, we keep track of the set of tasks that communicate with each other as a result of a user-initiated GUI event.

The start of an interactive episode is signified by the GUI controller (X server in our case) sending a message through a socket to another task. When this happens both the GUI controller and the receiver of the task are added to what we refer to as the task set of the episode. If the members of the task set communicate with non-member tasks, then those tasks are also added. The end of the episode is reached when all the following conditions are met for tasks in the task set:

- No tasks are executing.
- Data written by the tasks have been consumed.
- No task was preempted the last time it ran (i.e., all gave up time on their own by blocking in a system call).
- No tasks are blocked on I/O.

Figure 4 illustrates two typical trace fragments from the Ghostview benchmark. In the first case, four processes communicate with the server as the result of a mouse-click event. Note that when a task gives up time it usually does so in the poll or select system calls which signifies that the task is ready to process more data.

The second trace fragment illustrates the interaction between the X server and a client that is continuously sending data to be displayed. In this case, the Ghostscript renderer is running continuously on CPU 0 and sends the data to the display whenever it has completed rendering a segment. The communication is unidirectional and asynchronous and the resulting parallelism is asymmetric. In our experiments symmetric utilization of both processors was rare (i.e., when the TLP is 2 for an extended period of time).

Most applications under UNIX communicate using sockets, signals, and pipes. In particular, the X server uses sockets to communicate with its clients. We do not track interactions via other methods such as System V IPC and shared memory since our benchmarks do not use them. By

TABLE 1. Linux benchmark descriptions and characteristics

Benchmark	Version	Description	Dual processor			Uniprocessor
			TLP_{ie}	TLP_{run}	$Idle_{run}$	$Idle_{run}$
Acroread	4.0	Acrobat PDF file viewer	1.20	1.19	88%	87%
FrameMaker	5.5.6beta	Document editor	1.35	1.33	93%	93%
Ghostview	3.5.8	PostScript and PDF file viewer	1.42	1.39	84%	84%
GIMP	1.1.22	The GNU Image Manipulation Program	1.26	1.24	88%	84%
Netscape	4.7	Web browser	1.34	1.28	90%	89%
Xemacs	21.1 patch 8	Text editor	1.26	1.21	93%	92%
		Average	**1.31**	**1.27**	**89%**	**88%**

tracking the communications between the tasks, we are able to determine which tasks have an effect on interactive performance. Unlike other operating systems (e.g., Windows NT), Linux does not differentiate between threads and processes. Threads are implemented using regular processes and the `clone` system call. We use the name "task" as a synonym for both threads and processes.

The implementation that performs the tracking is as non-invasive as possible. The difficulty was not in the actual implementation but in finding all the parts of the kernel that needed to be tracked. Currently we track communications through the following system calls:

> kill, pread, pwrite, read, readv, recv, recvfrom, rcvmsg, send, sendmsg, sendto, write, writev

We instrumented each of these system calls to emit a trace of the signals, inodes, and sockets that they are accessing. The socket information is output instead of the inode number, when a socket is accessed through an inode. To be able to match read and write requests through socket pairs, we use the socket's pair (`sock->sk->pair`) on a write and the read socket itself on a read event. Currently we track only communications through UNIX sockets since this is the only socket type that is local to the machine. One could extend this methodology to track communications through other types of sockets if the communicating programs are all local to the machine. However, we have seen no need for this extension so far.

The primary reason for tracking signals is that the thread library (LinuxThreads) uses signals to implement synchronization between threads. By looking at the signal activity we can determine how threads communicate through condition variables, mutexes, and locks. The two functions that needed to be instrumented are `handle_signal` and `send_sig_info`. An alternative to this approach would have been to instrument the thread library; however, our current approach is more generic and has lower overhead.

To determine when tasks are blocked on I/O, we instrumented the `schedule` function to record the reason why it was called. If it is called from a part of the kernel that is related to I/O (such as the read and write system calls), then we assume that the task is blocked while waiting for an I/O event to complete. Since there is no predefined way in Linux to find which system call caused a transition to the kernel, we instrumented key system calls to put their id in a field of the executing task's `task_struct`. Once execution gets to the `schedule` function, our code looks at this field and outputs the task's reason for giving up time.

An attractive feature of our methodology is that the ends of interactive episodes can be found immediately, without having to wait first for an arbitrary amount of time to elapse. This information could be used on line by the kernel to make better scheduling and service quality decisions.

3.2 Benchmarks

Table 1 gives a short description of our benchmarks and summarizes their high-level characteristics. The data presented in this paper are averages of seven benchmark runs in each configuration. All benchmarks were run by a live user. While we aimed to repeat each run as accurately as possible, there are slight variations between the runs. All the significant events (e.g., mouse clicks, text entry) were performed in the same order during each benchmark run. However, the exact path of mouse movement (and therefore the interactive episodes corresponding to them) and the amount of time between events varies from one run to the other.

All our applications show significant amounts of TLP and a high fraction of idle time. It is likely that the idle time of the application executing on an actual user's desktop would be higher since although we interacted manually, we made no efforts to consume all the information presented by the program. The overall TLP of our applications are similar to what we measured during the interactive episodes. However, in all cases the TLP in interactive episodes was higher than the average for the entire run of the benchmarks.

4. Response time results

One way of quantifying an application's performance is to measure the time it takes to complete a run of the benchmark. This approach works for throughput-oriented benchmarks, however it runs into difficulties when one tries to measure interactive applications. Nonetheless, benchmarks such as Sysmark 98 [17] and Winstone 99 [18] attempt to quantify the performance of interactive applications by turning them into throughput-oriented benchmarks. They accomplish this by using a software driver that clicks

TABLE 2. Response time on a dual-processor and a uniprocessor machine

Benchmark	Average response-time improvement of mouse-click events	Selected episodes					
		Episode description	Response time (T_R) improvement	Dual processor		Uniprocessor	
				TLP_{ie}	T_R (sec)	Measured T_R (sec)	Predicted T_R (sec)
Acroread	15%	Displaying successive pages of a pdf file.	14%	1.21	0.119	0.138	0.144
			17%	1.21	0.125	0.150	0.151
			12%	1.15	0.231	0.261	0.264
FrameMaker	22%	Visually manipulating a FrameMaker document.	30%	1.43	0.296	0.425	0.422
			20%	1.29	0.040	0.050	0.051
			25%	1.41	0.022	0.029	0.031
			27%	1.38	0.021	0.029	0.029
Ghostview	34%	Displaying successive pages of a pdf file.	36%	1.51	0.223	0.346	0.336
			19%	1.29	0.455	0.562	0.586
			36%	1.53	0.225	0.352	0.345
			30%	1.47	0.403	0.578	0.593
			32%	1.52	0.331	0.484	0.502
GIMP	19%	Pixelize	18%	1.37	0.456	0.553	0.623
		Motion blur	8%	1.15	1.206	1.312	1.390
		Sharpen	20%	1.33	0.408	0.511	0.541
		Laplace edge-detect	15%	1.19	0.982	1.156	1.164
		Undo	22%	1.38	0.118	0.152	0.164
Netscape	21%	Displaying simple HTML pages from a machine-local web server.	24%	1.34	0.252	0.331	0.338
			25%	1.42	0.096	0.127	0.137
			20%	1.31	0.064	0.079	0.084
			28%	1.44	0.085	0.118	0.123
Average	**22%**						

through these applications as quickly as possible and by measuring the length of the end-to-end execution.

The problem with this approach is that in interactive applications not all parts of the program's execution are equally important. The relevant metric is improvement in response time (the time it takes to respond to a user-initiated event) of critical episodes [5]. This time has also been called "wait time," to refer to the fact that during these periods the user is actively waiting for the computer to complete the task.

This section presents measured response times from uni- and dual-processor machines. The first subsection compares the performance of individual interactive episodes while the second analyzes and compares aggregate statistics from full benchmark runs. Section 4.3 further analyzes these results in the context of user-perceptible improvement. Finally, Section 4.4 repeats the initial experiments with an added active background process (an MP3 player).

4.1 Individual episodes

We have noted that while the runs of our benchmarks are similar to each other, they are not completely identical. This poses problems when we attempt to compare two different runs to each other. We cannot just compare the average lengths of interactive episodes from one run to the other, since the set of episodes in each run could be slightly different. We overcame this problem by comparing only individual episodes that occur after the same mouse click in each trace. These episodes are more repeatable than those caused by other events, such as focus changes, and tend to have longer response times.

To correlate mouse-click events with interactive episodes we modified the X server to write an entry into the trace every time a mouse button is pressed. The postprocessor can then correlate interactive episodes that occur after a marker. Table 2 summarizes the results of the response time measurements for these episodes. The first column includes the overall response time improvement for the mouse-click episodes in the benchmark, while the rest of the columns show the collected data and the response time improvement for a few selected episodes. Note that, since our Xemacs

TABLE 3. Episode distribution (dual processor)

Benchmark	[0ms, 1ms)		[1ms, 10ms)		[10ms, 100ms)		[100ms, inf)	
	% of episodes	% of time	% of episodes	% of time	% of episodes	% of time	% of episodes	% of time
Acroread	92.69%	5.75%	4.11%	3.52%	1.13%	11.89%	2.08%	78.85%
FrameMaker	72.10%	2.87%	17.60%	6.98%	8.58%	42.11%	1.72%	48.04%
Ghostview	89.87%	2.24%	6.73%	2.22%	0.76%	6.7%	2.64%	88.85%
GIMP	87.93%	2.7%	10.19%	5.89%	0.32%	0.64%	1.57%	90.77%
Netscape	89.98%	3.56%	8.62%	13.88%	0.98%	31.43%	0.42%	51.13%
Xemacs	65.01%	4.78%	34.36%	86.01%	0.63%	9.21%	0%	0%

workload was driven solely by keyboard interactions, we were unable to compute response time improvement for it.

The results indicate that on our benchmarks response time improved on a dual-processor machine by an average of 22% (36% in the best case, 8% in the worst). The average TLP for the episodes is 1.31, higher than the average for the complete benchmarks, which is 1.27. Idle time during the interactive episodes in all cases was zero or very close to it (a few tenths of a percent).

To check our results we used the dual-processor numbers to estimate the uniprocessor run-time (Equation 4) and then checked them against actual measured values. DP refers to the response time (T_R) and idle time (T_{Idle}) on a dual processor, while UP refers to the same measurements on a uniprocessor machine.

$$T_{R(UP)} = (T_{R(DP)} - T_{Idle(DP)})TLP + T_{Idle(DP)} \qquad \text{(EQ 4)}$$

The equation scales the non-idle portions of the episode by the measured TLP and assumes that no scaling occurs on the idle portions. This simple model predicts the uniprocessor episode lengths to within 4% on average (one run had an error of 11% and for all others, error was under 7%). Given that we made no special provisions to reduce experimental variations (e.g., by turning off background daemons) and that all traces were driven by a real user (instead of an automated script), we think that the error is within a reasonable margin.

Most Linux applications are not threaded; concurrency emerges from simultaneously running multiple processes. The only applications from our benchmarks that actually used threads (through the LinuxThreads API) were Netscape and GIMP. These applications derived some TLP by running intra-application threads concurrently. However, most of the TLP was achieved by running the application thread concurrently with the user interface threads: mostly with the X server but also with the other GUI tasks (such as the window manager, desktop applets, etc.). This contrasts with our experience under Windows NT, where application threads ran concurrently primarily with threads from the System process, which includes device drivers and other operating system threads [6].

4.2 All interactive episodes

In the previous section we looked at select episodes that come after mouse clicks to figure out the response time improvement. These episodes usually represent the heavyweight episodes during the benchmark runs and, while these episodes usually make up the largest percentage of time during the run, the number of short episodes dominates.

Table 3 shows the episode length distribution of our benchmarks. Due to the large variance of episode lengths, we separated the results into four categories. For each category, the number of episodes that fall into it are given (percentage of episodes) along with the total amount of time spent (as a percentage of total time in all interactive episodes). While the majority of the episodes are very short and are in the few tenths of a millisecond range, only a

TABLE 4. TLP and episode length distribution (dual processor)

Benchmark	[0ms, 1ms)		[1ms, 10ms)		[10ms, 100ms)		[100ms, inf)	
	TLP_{ie}	avg. length	TLP_{ie}	avg. length	TLP_{ie}	avg. length	TLP_{ie}	avg. length
Acroread	1.38	0.25	1.19	3.47	1.20	42.76	1.18	153.66
FrameMaker	1.38	0.30	1.20	2.94	1.36	36.42	1.37	207.72
Ghostview	1.30	0.23	1.18	3.10	1.33	83.39	1.43	315.85
GIMP	1.43	0.17	1.22	3.28	1.35	11.49	1.26	328.83
Netscape	1.70	0.07	1.16	2.73	1.41	55.59	1.33	210.60
Xemacs	1.87	0.07	1.24	2.52	1.14	14.68	N?A	N/A

TABLE 5. Episodes above the perception threshold

Benchmark	100ms threshold				50ms threshold			
	Dual processor		Uniprocessor		Dual processor		Uniprocessor	
	% time	# episodes	% time	# episodes	% time	# episodes	% time	# episodes
Acroread	79%	8	85%	9	89%	9	90%	9
FrameMaker	48%	2	49%	2	69%	5	75%	6
Ghostview	89%	12	95%	15	96%	15	96%	15
GIMP	91%	9	92%	9	91%	9	92%	9
Netscape	51%	4	63%	7	72%	10	73%	10
Xemacs	0%	0	0%	0	0%	0	0%	0

small portion of the time is spent in episodes corresponding to them. This makes sense given the orders-of-magnitude variance in episode lengths. Examples of the short episodes include:

- Moving the mouse and updating its position.
- Updating the appearance of the cursor.
- Handling window focus changes.
- Handling keyboard events.

Towards the right hand side of the table is the data corresponding to the heavyweight episodes that were the subject of our investigations in the previous section. Since in the majority of our benchmarks most of the time is spent in executing these kinds of episodes and most of them fall above the perception threshold of the user, these are of primary importance for speeding up.

Table 4 gives the TLP and average length of interactive episodes. The most significant observation is that in all cases TLP is higher in the interactive episodes than the average for the entire run of the benchmark (see Table 1). This matches with our observation in the previous section. Moreover, in all cases the interactive episodes appear to be very CPU bound, with zero or close zero idle time.

Short episodes (less than one millisecond long) tend to have the highest TLP. This should not be surprising since these episodes usually perform an update of a few GUI objects on the screen, which requires the tight interaction of both the X server and the client. The TLP in the most used category varies from one benchmark to the other. It is never smaller than the overall average for the program but in some cases it is smaller than the average for the interactive episodes.

While most time is spent executing episodes that fall in the tenth of a second to over a second range, some last for only a fraction of a millisecond. With episodes that are so short, the question comes up whether there is any perceptible improvement in responsiveness using two processors.

4.3 The perception threshold

We have shown that TLP can be exploited successfully to reduce the response time of interactive applications. Once the response time reaches a certain threshold, the user is not able to detect any further improvement. What exactly that threshold is depends on the event type and can vary from one user to another. While its actual value is hard to quantify, the perception threshold sets an upper bound for the required performance.

Determining the exact length of an interactive episode can also be problematic. Does an episode begin when the user clicks the mouse button, or when that event is delivered by the X server? We have taken the position that interactive episodes begin when the event is delivered, since the X server may need to wait for additional events—such as extra mouse clicks to distinguish a double-click from single-click or for a button release—before delivering the event. Our measurements show that the delay between the hardware event and event dispatch can vary from a few tenths to hundreds of milliseconds. When considering an appropriate perception threshold for a user, this extra delay may need to be accounted for. We do not address this problem in this paper.

Table 5 shows the number and fraction of time spent in episodes that are above the perception threshold. The fraction of time is expressed as the percentage of time in all interactive episodes. Data is computed for two threshold values, which were selected based on data from [2]. Given either threshold, most of the time is spent executing episodes that fall above the perception threshold. Based on this data, FrameMaker, Netscape, and Xemacs are the most responsive applications. This correlates well with our experience; all three of these applications seemed to be very responsive and we could not experience any qualitative difference between the dual-processor and uniprocessor runs of these applications. We must note, however, that while interactive episodes in Netscape were under the perception threshold when accessing a web server on the local machine, accessing servers on the Internet would certainly show more episodes in the perceptible range due to network latency. However, network latency is not something that multiprocessing in the client can reduce.

While exploiting TLP reduces the average length of the interactive episodes, it only causes a shift of an interactive episode from above to below the perception threshold, if its length on a uniprocessor does not greatly exceed the threshold. None of the benchmarks had a significant shift in the number of episodes when the perception threshold is set to 50ms, and only a few of our benchmarks' episodes moved

TABLE 6. Response time improvement on dual-processor and uniprocessor machines with MP3 playback in background

Benchmark	TLP_{ie}	TLP_{run}	Response time improvement	Response time increase due to MP3 playback	
				Dual processor	Uniprocessor
Acroread	1.25	1.19	23%	4%	15%
FrameMaker	1.40	1.20	29%	1%	13%
GhostView	1.46	1.34	38%	4%	10%
GIMP	1.32	1.23	23%	4%	14%
Netscape	1.39	1.24	31%	5%	16%
Xemacs	1.35	1.18	N/A	N/A	N/A
Average	**1.36**	**1.23**	**29%**	**4%**	**14%**

below the cutoff at the 100ms threshold (Acroread, GhostView, and Netscape).

4.4 Effects of background activity

To round out our investigations, we wanted to know what happens when a background process is executing along with the interactive application. To gain some insight into such workloads, we repeated our experiments with an MP3 player running in the background. We used a very simple MP3 player called mpg123 (version 0.59r) along with the esd sound daemon. The player lacks a graphical interface and only does music playback—no visual effects. This application is light-weight and exhibits very little concurrency. We measured 1.02 TLP and 95% idle time when running music playback by itself.

The results are presented in Table 6. The response times are averages over all mouse-click events, as in Table 2. The performance improvements due to two processors is more significant than in our previous measurements. The average improvement is 29%, in contrast to 22% without the background task. On a dual processor the work required for MP3 playback is mostly absorbed by the extra processor. However, on a uniprocessor the extra work cannot be off-loaded and must be performed during the critical path, thus extending the lengths of the interactive episodes. Compared to our previous results, the dual-processor episode lengths are increased by an average of 4%, while the uniprocessor episode lengths are increased by an average of 14%.

The average TLP within the interactive episodes increases to 1.36 while the average for the entire benchmark run decreases to 1.23. The trend is the same as in our previous measurements without MP3 playback. However, in this case the difference between TLP_{ie} and TLP_{run} is significantly greater (the average TLP_{ie} is 1.31 and TLP_{run} is 1.27 when there is no MP3 playback in the background). The reason for the greater difference is that MP3 playback is periodic and has no inherent concurrency (it is not threaded, just a single task), which affects TLP in two ways:

- It reduces idle time and the new non-idle portions have a TLP of one.
- On existing non-idle periods, it increases TLP.

Since interactive episodes have very little idle time, TLP goes up due to the concurrently running MP3 playback process. On the non-interactive portions, the background application reduces idle time and replaces the idle thread with a single running application. Since all of our benchmarks are dominated by idle time, the TLP_{run} of the applications is the same or lower with MP3 playback in the background than without.

5. Conclusions and future work

The fundamental question that this paper attempts to illuminate is whether it is beneficial for a desktop user to use a multiprocessor machine for everyday tasks. We have shown that existing Linux workloads exhibit thread-level parallelism, which translates into improvement of the user-perceived response time of the applications. Using two processors instead of one is a straightforward way to reduce execution length in the critical path in our benchmarks by 8% to 36% (22% on average). Moreover, these improvements represent 16% to 72% of the maximum reduction achievable on a dual-processor machine (50%). Using two processors can thus be an effective and efficient way of improving interactive performance.

The average response time improvement on a dual-processor machine increases to 29% with an MP3 player executing in the background. Although the extra processor eliminates most of the overhead of a background task, it does not absorb it all. The average length of an interactive episode increased by 4% due to audio playback (vs. 14% on the uniprocessor). This result is consistent with the level of TLP we found in interactive applications: we should expect the response time to remain unchanged only if the second processor is completely unused by the foreground application.

For most of our applications, using more than two processors is not likely to yield great improvements. This conclusion is supported by our previous experience on a quad-processor machine, where the only workloads that had a TLP of 2 or more were hand-parallelized or were batch jobs [6]. Our current results show that most workloads have TLP under 1.4, which implies that increasing the number of processors would only be beneficial in less than 40% percent of the time (i.e., the proportion of the episode where TLP is 1

is not reduced). Even by making the optimistic assumption that in all these episodes four threads could run concurrently 40% of the time, we can only expect an overall speedup of 20% on a quad-processor over a dual-processor machine.

In our opinion, our results indicate that the Linux kernel and associated software have come of age as an SMP platform. While thread-level parallelism can certainly be further improved by removing uses of the global kernel lock, we believe that the emphasis should now be on application writers to refactor their programs with multithreading in mind. Multiprocessing would become even more compelling if TLP could be increased above 1.5.

Historically, uniprocessor performance has doubled every 18 months. Given a TLP of 1.5, this means that the lead time of a dual processor over an equivalent-performing uniprocessor is about three quarters of a year. Given that most interactive episodes in our measurements were shorter than 500ms, in about three years these episodes will fall under the 100ms perception threshold on a dual-processor machine and in about four years it will be sufficiently fast on a uniprocessor. On the other hand, the software four years from now will likely increase its processing requirements.

In this paper we focused on the performance effects of multiprocessing on interactive applications. However, at some point in the future the decision whether to have multiple thread contexts in hardware may have to be made on some factors other than performance, such as energy and power efficiency. If response time is below the threshold of user perception, one can reduce energy and power consumption by running each CPU at a lower frequency and voltage, without degrading the user experience. Exploiting TLP to reduce critical paths could enable further frequency and voltage reductions.

Dynamic voltage and frequency scaling [11][14] also requires algorithms that determine a priori which episodes are fast enough and how fast to execute them. This is a direction of our ongoing research. Our methodology of tracking communications between tasks can be used to identify the performance critical parts of the workload and to estimate the required level of performance.

In our experience most of the concurrency was achieved by overlaying the execution of the GUI controller (X server) with an application task that is communicating with the GUI. We have noted that the utilization of processors is usually unbalanced. This leaves some room for software designers to repartition interfaces in order to more efficiently utilize the hardware. In particular, a higher level API for rendering images in the X server could improve the balance between the server and the clients. A more general approach to balancing would be to run the CPUs in the system at different levels of performance depending on the particular workload. This optimization would increase TLP and decrease energy consumption.

6. Acknowledgments

This work was supported by an Intel Graduate Fellowship, by an equipment grant from Intel, and by DARPA contract number F33615-00-C-1678.

References

[1] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Berghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[2] J. B. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Seltzer, and M. D. Smith. The Measured Performance of Personal Computer Operating Systems. *Proceedings of the 15th ACM Symposium on Operating System Principles*, pp. 299-313, December 1995.

[3] K. Diefendorff. Power4 Focuses on Memory Bandwidth: IBM Confronts IA-64, Says ISA Not Important. Microprocessor Report, Volume 13, Number 13, October 6, 1999.

[4] K. Diefendorff. Compaq Chooses SMT for Alpha: Simultaneous Multithreading Exploits Instruction- and Thread-Level Parallelism. Microprocessor Report, Volume 13, Number 16, December 6, 1999.

[5] Y. Endo, Z. Wang, J. B. Chen, and M. I. Seltzer. Using Latency to Evaluate Interactive System Performance. *2nd Symposium on Operating Systems Design and Implementation*, pp. 185-199, October 1996.

[6] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism of desktop applications *Proceedings of Workshop on Multi-threaded Execution, Architecture and Compilation*, Toulouse, France, January 2000.

[7] L. Hammond and K. Olukotun. *Considerations in the Design of Hydra: a Multiprocessor-on-a-Chip Microarchitecture.* Stanford University Technical Report No. CSL-TR-98-749.

[8] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using Threads in Interactive Systems: A Case Study. *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 94-105, December 1993.

[9] D. C. Lee, P. J. Crowley, J. Baer, T. E. Anderson, and B. N. Bershad. Characteristics of Desktop Applications on Windows NT. *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[10] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[11] T. Pering, T. Burd, and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. *Proceedings of International Symposium on Low Power Electronics and Design 1998*, pp. 76-81, June 1998.

[12] J. S. Seng, D. M. Tullsen, and G. Z. N. Cai. Power-Sensitive Multithreaded Architecture. *Proceedings of International Conference on Computer Design 2000*, September 2000.

[13] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 206-218, June 1995.

[14] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. *Proceedings of the First Symposium of Operating Systems Design and Implementation*, November 1994.

[15] *Microprocessor Architecture for Java Computing*. http://www.sun.com/microelectronics/MAJC, Sun Microsystems, 1999.

[16] Press release: Apple Debuts New PowerMac G4s with Dual Processors. http://www.apple.com/pr/library/2000/jul/19g4.html

[17] http://www.bapco.com/sys98k.htm

[18] http://www.zdnet.com/zdbop