Scheduling in Variable-Core Collaborative Systems

Sasa Junuzovic
Microsoft Research
Redmond, WA 98052, USA
sasa.junuzovic@microsoft.com

Prasun Dewan
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
dewan@cs.unc.edu

ABSTRACT
The performance of a collaborative system depends on how two mandatory collaborative tasks, processing and transmission of user commands, are scheduled. We have developed multiple policies for scheduling these tasks on computers that have (a) one processing element on the network interface card and (b) one or more processing cores on the CPU. To compare these policies, we have a developed a formal analytical model that predicts their performance. It shows that the optimal scheduling policy depends on several factors including the number of cores that is available. We have implemented a system that supports all of the policies and performed experiments to validate the formal model. This system is a component of a self-optimizing scheduler we have developed that improves response times by automatically choosing the scheduling policy based on number of cores and other factors.

Author Keywords
Optimization; scheduling; multi-core; analytical model.

ACM Classification Keywords
C.2.4. [Computer-Communication Networks]: Distributed Systems – distributed applications, client/server; C.4 [Performance of Systems] Performance Attributes.

General Terms
Algorithms, Measurement, Performance, Experimentation.

INTRODUCTION
Performance is a critical success factor in the world of collaborative applications. If an application does not respond to actions by a user quickly or notify the user of actions of others in a timely fashion, the user may get frustrated and quit the session.

Several performance metrics have been identified, such as local [14] and remote [4] response times, throughput [5], jitter [7], bandwidth [8], and task completion time [2]. While all of them are useful, we focus on response times. Response times depend on a variety of factors such as processing [3] and communication [10] architecture and scheduling of collaborative tasks [11]. We focus on the scheduling of tasks for processing and transmission of commands, both of which are necessary for coupling.

Previous work has identified and evaluated several different policies for scheduling these tasks [11]. However, it has assumed that a single processing element is used to perform all aspects of these tasks. We have developed four variations of these policies, which leverage parallelism on modern devices offered by multiple CPU cores and the processor on the network interface card. To compare these four policies, we have developed a formal evaluation model, which predicts the local and remote response times of these policies. This model is a component of a self-optimizing scheduler we have developed that improves response times by automatically choosing the scheduling policy based on number of cores and other factors.

A flavor of the performance improvements the optimizer can provide is given in Table 1. The table shows average response times of five users with and without the optimizer. For four of them, performance is better with than without the system, by 99ms, 56ms, 101ms, and 99ms. More importantly, the improvements are noticeable to the users. In particular, human-perception studies by Youmans [16] and Jay et al. [9] have shown that people can notice changes of 50ms in local and remote response times, respectively.

The rest of this paper is organized as follows. First, we consider background work that forms the basis of our research. Then, we discuss parallelism in modern machines and aspects of it that a scheduling system can exploit. Following this, we present four parallelizing scheduling policies and the analytical model for them. Next, we outline implementation of our system and describe the simulations and experiments we performed to evaluate it. Then, we present discuss the broader impact of our work. Finally, we end with conclusions and directions for future work.

BACKGROUND WORK
The tasks in a collaborative application are determined by the functionality it provides. By definition, all collaborative

Response Time (ms)	$User_1$	$User_2$	$User_3$	$User_4$	$User_5$
Non-optimized	459	426	473	510	564
Optimized	360	370	372	411	623

Table 1. Average response times of five users in a collaborative session with and without optimized scheduling

CSCW 2011, March 19–23, 2011, Hangzhou, China.

applications must couple the users. By extension, the tasks that provide coupling are mandatory. There are two such tasks, those for processing and transmitting user commands. In general, an application may have tasks related to awareness, concurrency control, and consistency maintenance mechanisms. While all of these mechanisms are important, they are in general optional. Our work focuses on the scheduling of the mandatory tasks.

Because our work focuses on the mandatory tasks, the degree to which it applies in applications with optional functionality is limited. For instance, World of Warcraft has concurrency control. Thus, our work may not correctly predict its performance when conflicts occur. However, when there are no conflicts, our work may still apply. We will return to the issue of scheduling optional tasks in more detail in the discussion section. Meanwhile, we focus on scheduling of the mandatory tasks whose nature depends on the processing [3] and communication [10] architectures.

Processing Architecture

Two main processing architectures have been used in the construction of collaborative systems: centralized (client-server) replicated (peer-to-peer). In both cases, it is assumed that the shared application is logically divided into separate user-interface and processing components. The user-interface component transforms user input into input commands and sends these commands to the program component. Conversely, it processes output commands that it receives from the program component and transforms the result into updates to the display. The program component processes user input by converting input commands to output commands. The user-interface component is replicated on each user's computer and allows a user to manipulate application state not shared with the other users. The program component is logically shared by all users and may be physically centralized or replicated, depending on the processing architecture. Regardless, each user interface is mapped to exactly one program component.

In the centralized architecture, all of the user-interface components are mapped to a single program component running on one of the user's computers. The computer running the program component is called the master and all of the other computers are called slaves. In the replicated architecture, each user-interface component is mapped to the program component running on the local computer. Whenever a master receives a command from the local user, it sends the command to all of the other computers, thereby ensuring the program components on different masters are kept in sync.

Communication Architecture

Regardless of which processing architecture is used, the master computers transmit commands to all other computers. If commands are large or the number of users is high, the transmission costs can be high.

An important question when transmission costs are high is whether a master computer uses unicast or multicast to communicate with other computers. The idea of multicast requires the construction, for each source of messages, a multicast overlay that defines the paths a message takes to reach the destinations. In this paper, we make two assumptions regarding multicast. First, because IP-multicast is not widely deployed, we assume an application-layer multicast in which end-hosts form the overlay. Second, as in peer-to-peer file sharing systems, we assume that only the users' computers can be used in the overlay.

Scheduling

Both the processing and the communication architecture mandate specific tasks that the users' devices must perform. The processing architecture determines which computers process input commands in addition to processing outputs. The communication architecture, on the other hand, dictates the destinations to which a computer transmits commands. The number of such destinations determines the significance of the transmission cost. In the unicast case, this cost is negligible on the destination computers.

The order in which a computer carries out the processing and transmission tasks can impact response times. Four main policies have been identified by Junuzovic and Dewan to schedule these tasks [11]. Three of these are straightforward: (a) process-first, which completes the processing task before starting the transmission task, (b) transmit-first, which does the reverse, and (c) concurrent, which creates a separate thread for each of these tasks and schedules these threads in a round-robin fashion.

The fourth, called lazy, [11] gives precedence to the processing task, but delays its execution and allows the transmission task to run during this delay if the resulting increase in local response times is not noticeable. The reason for delaying processing is that a part of the transmission task can run earlier, thereby noticeably improving remote response times of some users. By definition, lazy dominates process-first. Thus, we ignore process-first in the rest of the related work discussion.

Junuzovic and Dewan have developed a formal model to compare the other three policies [11]. Using the model, they show that each of these three policies can offer some users noticeably better response times than the other two policies. Their model assumes (a) a single-core CPU, (b) no concurrent commands, (c) no type-ahead, (d) no batch transmission, and (e) blocking communication. In blocking communication, after a CPU thread sends a message to the network interface card, it bocks until the message is transmitted to the network.

We extend prior work in four ways. We identify parallelism in modern computing devices that a scheduling system can use. We motivate and present four scheduling policies that take advantage of this parallelism. Three of these are existing policies in which blocking communication is

replaced with non-blocking while still using a single processing core. The fourth is a variation of the concurrent non-blocking policy in which two instead of one CPU cores are used. We develop an analytical model that extends the previous model by accounting for the parallelism. Finally, we present a self-optimizing scheduler that automatically chooses the optimal scheduling policy.

SOURCES OF PARALLELISM

Modern computing devices have multiple general-purpose and specialized processing units that can be used in parallel. For example, multi-core CPUs have become common. The parallelism they offer can significantly improve performance. As a result, many applications that were optimized for the traditional single-core CPUs are now being redesigned to leverage multi-core CPUs.

Devices today also have a specialized processor on the network interface card that is capable of executing most of the network stack. For instance, some network cards execute the full TCP/IP protocol. With such cards, the CPU is relieved of dealing with the TCP/IP algorithm, and thus, has more time to perform other tasks. The parallel processing offered by the network card processor is useful for any distributed system as some of the transmission task can be offloaded onto the network card. Non-blocking communication was invented for this reason.

Graphical processing units (GPUs) can also perform some tasks on behalf of the CPU. Since GPUs are not general-purpose, they cannot execute all tasks. However, for tasks they can execute, such as array operations, they offer excellent performance – much better than that of a CPU. Thus, a useful idea is to offload computation from the CPU to the GPU, which is often done in graphics.

One of the purposes of this paper is to encourage designers of collaborative systems to think about and exploit the parallelism in modern devices. For example, collaborative tasks can be executed in parallel on multi-core devices. Also, while non-blocking communication is used in some systems, the benefits it offers were never formally studied or systematically exploited. As mentioned above, Junuzovic and Dewan [11] assumed blocking communication in their analytical model, which logically groups the CPU and the network card processor into a single processing unit. Also, it would be useful to use GPUs for performing parts of input processing, not just output rendering, tasks.

EXPLOTING PARALLELISM

In this paper, we exploit parallelism in two ways. We model and analyze the effects of executing the processing and transmission tasks in parallel on the CPU cores and the network card processor. We leave the use of the GPU and other processors as future work.

Non-Blocking Communication

In order to take advantage of non-blocking communication, we must dissect the procedure a computer takes to transmit a command to a destination. The procedure has two steps. First, the CPU reads command data from memory and then writes it to a location in memory which it shares with the network card. Second, the network card reads the data from the shared memory location and then transmits it by placing it on the physical wire. Thus, there are two transmission tasks: the CPU and the network card transmission task. The difference between the blocking and non-blocking communication is whether or not the CPU transmission task blocks until the network card transmission task completes.

Typically, the network card processor is slower than the CPU. In all of our experiments, the CPU transmission task completed at least two times and usually more than ten faster than the network card transmission task. Thus, it may seem that non-blocking communication should always be used because it allows the CPU to perform other tasks while the network card is transmitting. However, a subtle but an important issue arises when a large command is transmitted to a large number of destinations in a non-blocking fashion – the network card buffers can overflow! Hence, there are scenarios in which blocking communication should be used instead of non-blocking communication. In this paper, we assume that network card buffer overflow does not occur.

Since the CPU transmission times are dominated by those of the network card, it may seem unnecessary to consider the impact of CPU transmission on performance. Traditionally, the impact has, in fact, been neglected. The transmission time of a command has been calculated as simply "size of command/bandwidth," which considers only the network card. This calculation is invalid for end-user computers because it does not account for the CPU transmission costs. In particular, before the command reaches the network interface, the operating system must traverse the network stack and copy data buffers along the way, which takes time. Moreover, the operating system must perform these steps for each destination. Study by Abdelkhalek et al. [1] of the server for Quake, a popular multi-player game, found that CPU transmission costs can be significant in practice. They found that the server spent 50% of CPU time on transmitting commands to clients.

Formal Analysis

We have developed a formal analytical model that accounts for non-blocking communication. It supports centralized and replicated processing architectures with unicast and multicast communication architectures. For a given processing and communication architecture combination, the model predicts response times of different scheduling policies. We do not derive the complete model here because of space limits; it is presented in the doctoral dissertation on this work [12]. Instead, we give enough detail to show its benefits compared to the Junuzovic and Dewan model [11].

Let us consider the remote response times of a slave user for command entered by the master user in a centralized-multicast architecture. As we are currently analyzing the parallelism between the CPU and the network card, suppose

that all devices have single-cores. To reach a particular user's computer, which we refer to as the destination computer, the command must travel from the source to the destination along some path. The path may consist of additional computers. We refer to the source computer and these additional computers as intermediate computers. The terms destination and intermediate are relative to a particular path. An intermediate computer on one path is a destination computer on a different path as all users see the output of an input command. Let π denote the path from the source to the destination, m the number of computers on the path, and $\pi_k, 1 \leq k \leq m$, the k^{th} computer on the path π, where π_1 is the source and π_m the destination.

The remote response time of command i to computer j along path π is given by

$$remote_{i,j} = \sum_{k=1}^{m-1} d(\pi_k, \pi_{k+1}) + \sum_{k=1}^{m-1} int(i, \pi_k) + dest(i, \pi_m)$$

where $d(\pi_k, \pi_{k+1})$ is the network latency between the k^{th} and $k+1^{st}$ computers on path π, $int(i, \pi_k)$ is the delay command i experiences on the k^{th} intermediate computer on the path, and $dest(i, \pi_m)$ is the delay command i experiences on the destination computer. The destination and intermediate computers contribute different delays because the former contributes to the remote response time of the local user while the latter contribute to the remote response time of a remote user. This results in a fundamental difference between the equations for the intermediate and destination computers. In the case of an intermediate computer, we must determine when the computer transmits to the downstream computer. In the case of the destination computer, we must determine when the input and output processing complete. While the network latencies are independent of scheduling policy and whether or not non-blocking communication is used, the intermediate and destination delays are not. We present the benefit of non-blocking communication with the transmit-first policy; the full model shows the benefit for all policies.

Consider first the transmit-first delay of the destination computer π_m when blocking communication is used. In general, a destination computer may also have to forward commands as it may be an intermediate computer on a different path. Thus, the delay of the destination computer depends on the number of computers to which it forwards commands because it must first transmit a command to all of them before processing it. When blocking communication is used, transmitting a message to a destination means having to place the entire message on the physical wire. Thus, the transmit time is equal to the CPU transmit time plus the network card transmit time. We denote the CPU and network card output transmit times of computer j by ${x_{CPU}}_{i,j}^{OUT}$ and ${x_{NIC}}_{i,j}^{OUT}$, respectively. Thus, the destination delay is given by

$$dest_{i,\pi_m}^{BLOCK}(i, \pi_m) = \left({x_{CPU}}_{i,\pi_m}^{OUT} + {x_{NIC}}_{i,\pi_m}^{OUT}\right) * fan_{\pi_m} + p_{i,\pi_m}^{OUT}$$

where $p_{i,j}^{OUT}$ and fan_j denote the time computer j requires to process output to command i and the number of destinations to which it forwards commands.

When non-blocking communication is used, the CPU and the network card work in parallel once the CPU transmits to the first destination. The above equation, however, does not capture the parallelism. In particular, with non-blocking communication, the destination delay does not depend on the network card transmission time. The delay is given by

$$dest_{i,\pi_m}^{NONBLOCK}(i, \pi_m) = {x_{CPU}}_{i,\pi_m}^{OUT} * fan_{\pi_m} + p_{i,\pi_m}^{OUT}$$

As the two equations show, non-blocking delay is ${x_{NIC}}_{i,\pi_m}^{OUT} * fan_{\pi_m}$ less than the blocking delay. Thus, if the total network card transmission time is high, then non-blocking communication will offer noticeably better response times than blocking communication.

Consider now the transmit-first delay of intermediate computer π_k. In this case, the delay is equal to the time the computer requires to transmit the command to π_{k+1}, the next computer on the path. When blocking communication is used, the delay is given by

$$int_{i,\pi_k}^{BLOCK}(i, \pi_k) = \left({x_{CPU}}_{i,\pi_k}^{OUT} + {x_{NIC}}_{i,\pi_k}^{OUT}\right) * pos(\pi_k, \pi_{k+1})$$

where $pos(\pi_k, \pi_{k+1})$ denote the position of computer π_{k+1} in computer π_k's list of destinations.

When non-blocking communication is used, the delay is lower because the CPU is faster than the network card. Thus, once the CPU transmits to the first destination, the time the CPU requires to transmit to the remaining destinations will overlap with the network card transmission time. Thus, the non-blocking delay is given by

$$int_{i,\pi_k}^{NONBLOCK}(i, \pi_k) = {x_{CPU}}_{i,\pi_k}^{OUT} + {x_{NIC}}_{i,\pi_k}^{OUT} * pos(\pi_k, \pi_{k+1})$$

As the intermediate delay equations show, the non-blocking delay is ${x_{NIC}}_{i,\pi_m}^{OUT} * (pos(\pi_k, \pi_{k+1}) - 1)$ lower than the blocking delay. Thus, if the number of destinations is large, the time the CPU requires to transmit a command to all of them can be high. In this case, the non-blocking delay is noticeably better than the blocking delay.

The above discussion illustrates two results. First, the existing model cannot be used to accurately predict response times when non-blocking communication is used. Second, the new model is capable of making response time predictions with non-blocking communication. From now on, we assume non-blocking communication in all policies.

Parallel Multi-Core Scheduling Policy

As the above discussion shows, when non-blocking communication is used, only partial control of scheduling is possible. Specifically, the network card transmission task is not schedulable – it always follows the CPU transmission task. Fortunately, this does not affect the control over the scheduling of CPU tasks.

As mentioned before, prior work has studied four different single-core scheduling policies: process-first, transmit-first, lazy, and concurrent. Each of these policies has a multi-core equivalent. Their pseudo-code for a centralized-multicast architecture is given in Figure 1. Regardless of policy, the master must process an input before processing and transmitting the corresponding output. Thus, the first step is always to process the input on all cores if the computer is the master. To process and transmit the output, the multi-core equivalents of the sequential schemes simply use all cores for one and then for the other task. The lazy policy is more complex. It delays processing while the delay is not noticeable. During the delay, it transmits. If transmission does not complete before processing starts, it is completed after processing completes. Thus, the multi-core equivalent of lazy delays the processing task on all cores. Finally, the multi-core equivalent of the concurrent policy uses half of the cores for one and half of the cores for the other task.

There are two important issues with multi-core scheduling policies. The first issue is whether a task can be parallelized on multiple cores. From a general framework perspective, the processing task is opaque – hence, the policies cannot parallelize it explicitly. The CPU transmission task, on the other hand, is parallelizable by nature. For example, two cores can transmit to two different destinations in parallel. More importantly, as the communication architecture defines the transmission tasks on each computer, the scheduling policy used by the framework can split the transmission task among multiple cores.

The second issue is whether or not it is beneficial to parallelize a task on multiple cores. From the point of view of predicting performance, it is in fact not beneficial to parallelize the transmission task. The reason is that when multiple cores perform a send call, the operating system can service these calls in an arbitrary order. Thus, there is no guarantee of the order in which the operating system will queue messages for transmission by the network card. As a result, it is difficult to predict when a message is actually transmitted to, and therefore received by, a destination. In addition, parallelizing the transmission task is also not beneficial from the perspective of optimizing response times. As one core is sufficient to saturate the network card, executing the transmission task in parallel on multiple cores will not improve remote response times.

It is still beneficial to carry out the processing and transmission tasks in parallel on different cores. The reasons are that with that policy (a) the CPU transmission times does not impact the response times to the local user and (b) the processing time does impact the response times of the remote users to which the computer transmits. We refer to such scheduling as the parallel multi-core scheduling policy. It is like the multi-core concurrent policy except that it uses only one core for the processing task and only one core for the transmission task.

Since we cannot explicitly parallelize the processing task and parallelizing the transmission task makes it difficult to predict performance, our self-optimizing system does not support the multi-core versions of the process-first, transmit-first, and lazy policies. The only multi-core policy it supports is the parallel policy.

Formal Analysis

Our formal model is able to account for the parallelism offered by multiple cores. As above, due to space constraints, we present only a flavor of it. We give sufficient detail to demonstrate the benefits of the multi-core parallel policy over other (single-core) policies.

Let us consider again the remote response times of a slave user for commands entered by the master user in a centralized-multicast architecture. Suppose that each machine has a multi-core CPU. Moreover, suppose that non-blocking communication is used. As explained earlier, the response time is a sum of the network delays, intermediate computer delays, and the destination delay along the path from the source to the destination. The network delays are independent of scheduling policy, while the intermediate and destination computer delays are not.

Consider first the parallel policy delay of the destination computer π_m. With the parallel scheduling policy, the transmission and processing tasks are performed by different cores. Therefore, the delay is a function of only the processing times and is given by

$$dest_{i,j}^{PARA}(i, \pi_m) = p_{i,\pi_m}^{OUT}$$

Compared to the transmit-first destination delay with non-blocking communication derived earlier, the parallel delay is $x_{CPU_{i,\pi_m}^{IN}} * fan_{\pi_m}$ lower, which is the time the CPU requires to forward the command. If the time the CPU requires to forward a command is than greater than the noticeable threshold, then the transmit-first delay is noticeably worse than the parallel delay. In fact, the parallel delay is equal to the minimal possible destination delay, as the destination computer must process the output.

Multi-Core Process-First	Multi-Core Transmit-First	Multi-Core Lazy	Multi-Core Concurrent

`if master then proc input on N cores`
`pro output on N cores`
`trans output on N cores`
	`if master then proc input on N cores`
	`trans output on N cores`
	`proc output on N cores`
		`if master then proc input on N cores`
		`trans output on N cores while delay < noticeable`
		`proc output on N cores`
		`trans output on N cores`
			`if master then proc input on N cores`
			`trans output on N/2 cores AND proc output on other N/2 cores`

Figure 1. Pseudo-code for multi-core process-first, transmit-first, lazy, and concurrent scheduling policies on N-core processors

Consider the parallel policy delay of the intermediate computer π_k, which is equal to the time computer π_k requires to forward the command to π_{k+1}, the next computer on the path. In this case, the parallel delay is equal to the transmit-first delay since in both cases, the processing task does not interfere with the transmission task. Thus, the parallel delay is given by

$$int_{i,j}^{PARA}(i,\pi_k) = x_{CPU}{}_{i,\pi_k}^{OUT} + x_{NIC}{}_{i,\pi_k}^{OUT} * pos(\pi_k,\pi_{k+1})$$

As was the case with the destination delay, the parallel intermediate delay is as good as any single-core policy delay. In fact, the full model also shows that it can be significantly better when processing times are high. Moreover, the parallel intermediate delay is equal to the minimum possible delay because an intermediate computer must transmit the command to the next destination, and the delay captures the time requires for the transmission. Thus, overall, the model predicts that when multiple cores are available for scheduling, the parallel multi-core scheduling policy should always be used.

IMPLEMENTATION

The model has identified several factors that impact the response times provided by a scheduling policy, including CPU processing and transmission times, network card transmission times, network latencies, and number of cores. As these factors can change both during and between sessions, the optimal scheduling policy may also change. For this reason, we implemented a scheduler that can react to changes in the parameter values and automatically switch to the optimal policy. We do not have space to present its full implementation; instead, we focus on one key issue and its solution – other issues and the rationale for them are given in the dissertation on this topic [12].

A key issue when comparing predicted performances for the scheduling policies is how the system decides which policy is optimal. The simplest approach is to define the optimal policy as the one whose average response time is noticeably better than that of all others. One issue with this approach is that there is no way to distinguish between response times of different users, which may be important. For instance, some users, and hence their response times, may be more important than others.

The problem with the simplistic comparison arises because the response times are inherently partially ordered and external criteria must be used to create a total order. In general, infinitely many external criteria exist. We focus on only two: favoring important users and favoring local or remote response times. Their exact application depends on users' response time requirements. Thus, what is needed is a user-defined function that accepts as parameters the response times of all policies, the list of inputting users, and the identities of all users, and returns a total performance order. Then, one policy is defined to be better than another if the function ranks the former higher than the latter.

EVALUATION

The model, policies, and the scheduler we have presented lead to two important questions. First, does the choice of scheduling policy noticeably impact performance in practical scenarios? Second, does the scheduler choose the optimal policy in these scenarios? To answer these questions, we must evaluate the model and the scheduler.

Simulations vs. Experiments

In general, in computer science, there are two possible ways to evaluate a system: simulations and experiments. Simulations estimate the performance of a system by using a model of the system. Compared to theoretical applications of the model, which give only trends and implications, simulations predict quantitative results in practical scenarios. Experiments, on the other hand, measure the actual performance of the system while it is being used.

Ideally, experiments should be used to evaluate a system. Two issues may make it impractical to run experiments. One issue is repeatability: it may not be possible to ensure the same conditions across experiments. The other issue is resource availability: some experiments require a large number of computers that may not be available. When it is not practical to run experiments, simulations should be used instead. As they rely on models to simulate results, it is important to verify that these models accurately represent the system being evaluated. Once the simulations are validated, they may be used even when experiments are practical. The reason is that, in general, it is easier and quicker to setup and run simulations than it is experiments.

Simulations

To show practical impacts of the choice of scheduling policy and the benefits of the self-optimizing scheduler, we had to use simulations. The reason is that the choice of scheduling policy matters only when transmission costs are high, which implies large scenarios. Unfortunately we do not have enough computers to perform large scale experiments – we only have ten in our lab. One solution is to use public clusters, such as PlanetLab and Amazon's EC2. However, these clusters do not provide controls necessary for repeatability. For instance, the loads on the machines vary because they are shared with other users of the cluster, and there is no way to ensure always using use of the same working set of machines.

It may seem that we need to perform two different sets simulations, one to evaluate the impact of scheduling policies on response times and another to test whether or not the scheduler switches to the optimal policy. However, in this case, it is possible to perform both simultaneously. The reason is that the scheduler must anyway evaluate the impact of scheduling policy on response times before it decides which policy to use. Therefore, as we present the results of the simulations, we first present the impact of scheduling policy on response times and then how the scheduler chooses the optimal policy.

Simulation Setup

To simulate performance in practical circumstances, we consider a scenario in which a PowerPoint presentation is being given to a large audience around the world. PowerPoint is a good choice of application for two reasons. First, it is perhaps the most popular business collaborative application today. Second, it has high transmission costs, which we need for our simulations, as mentioned above.

To obtain realistic processing and transmission costs, we identified user-commands in logs of actual PowerPoint use. We analyzed recordings of two presentations. These recordings contain actual data and users' actions – PowerPoint commands and slides. We assume that the data and users' actions in the logs are independent of the number of collaborators and response times. We used these commands to perform small scale distributed PowerPoint experiments by replaying commands using built-in replay capabilities in our scheduler. We configured to system to collect parameters but not make any optimization decisions. After each experiment, we obtained the processing and transmission costs from the performance data file output by our system. To get costs for different machines, we repeated the procedure with four different source and destination computers: Core2 2.0GHz desktop; P4 1.7GHz desktop; P3 500MHz desktop; and 1.6GHz Atom netbook. For fear of having our measurements affected by other applications, we removed as many active processes as possible on each computer, which is a common approach in experiments comparing alternatives. Nevertheless, as LAN delays and CPU loads vary during an experiment, we performed each one ten times and averaged the values.

Based on the published network latency data between 1740 computers [13], we set the network latencies between all users equal to those between a random subset of the published data. One issue with randomly selecting the subset is whether the subset preserves properties, such as triangle inequality and latency distributions, of the entire set. Zhang et al. [17] analyzed random subsets taken from latencies measured between 3997 computers and found that they were representative of the overall measurements.

As the lazy policy requires response time thresholds in order to decide how long processing can be delayed, we used the noticeable thresholds of 50ms, which we discussed earlier, for both local and remote response times.

In the simulations, we consider a centralized-multicast architecture in which the presenter's computer is the master. Also, we consider a multicast tree in which the presenter's computer and a small subset of the other computers forward commands to all of the remaining computers. These remaining computers are evenly divided among the source and the forwarders. The source sends commands to the forwarders, and then the source and the forwarders send commands to their respective destinations. We fixed the latencies among the six forwarding computers to be low (i.e., 0ms). Such a communication architecture is similar to the one used by Webex. In Webex, a user joining a session connects to one of several infrastructure computers. These computers, which are connected by high-bandwidth low-latency connections, serve as forwarders of commands. In our scenario, the presenter's and the forwarding computers are like the dedicated Webex forwarders; however, unlike Webex forwarders, which only forward commands, they also process commands. Last, the computers are all using non-blocking communication.

Multi-Core Results

To evaluate the benefit of the self-optimizing system on multi-core computers, we simulated a scenario in which (a) there are 1200 users, (b) five computers in addition to the source act as forwarders, (c) the source and forwarders send commands to 199 other computers each, (d) all users are using Core2 desktops. The results are best understood as response time differences between policies. Thus, we compare lazy, transmit-first, and process-first policies against the parallel policy.

The remote and local response times are shown in Table 2 and Table 3, respectively. The parallel policy provides noticeably better remote response times than the process-first policy to 999 of the 1199 audience members by as much as 311ms, while the remote response time differences to the remaining 200 users are not noticeable. Moreover, compared to the remote response times of the transmit-first policy, the parallel policy provides noticeably better response times to only five users (69.3ms) and provides unnoticeably different response times to the rest. The five users whose response times are improved are the forwarders. This makes sense as the processing on these computers is delayed until transmission completes when the transmit-first policy is used, while the processing is not delayed when the parallel policy is used. Finally, the parallel policy remote response times are the same as those of the lazy policy for 1198 users and not noticeably different the remaining user. The reason is that when non-blocking communication is used and the lazy policy delays processing, the network card did not catch up complete transmitting while the CPU was processing the command – when transmission resumed, the network card was still busy with previous transmissions. Thus, the pause in the transmission did not affect remote response times of the downstream users. As for the local response times, those of the parallel policy are as good as, not noticeably better, and noticeably better than those of process-first, lazy, and transmit-first policies, respectively.

Overall, the response times show a counter-intuitive result. When leveraging the network card processor, using a single core to perform the CPU tasks can provide unnoticeably different performance compared to using multiple cores even if processing and transmission times are high. Thus, in some scenarios, a single-core policy should be used when it offers the same performance as the parallel policy because all but one core will be free to perform other tasks.

The self-optimizing scheduler uses the predicted values to choose the policy that best meets the response time requirements. As mentioned above, these response time requirements are captured by the total order function. Based on the above results, regardless of the response time requirements, the scheduler would choose the lazy policy.

Single-Core Results

To evaluate the benefit of the self-optimizing system when devices only have single-cores, we simulated the same scenario as in the multi-core case with the following changes: (a) there are 600 instead of 1200 users, (b) the source still sends commands to five forwarders, (c) the source and the forwarders sends commands to 99 instead of 199 other computers each, (d) the presenter is using a netbook, and (e) the remaining users are using a random mix of netbooks or P3 and P4 desktops. As before, the results are best understood as differences in response times. Thus, we compare the lazy policy response times against those of the other three single-core policies.

The remote and local response time results are given in Table 4 and Table 5, respectively. Three main points can be extracted from the tables. First, compared to the lazy policy, the transmit-first and concurrent policies provide better remote response times to 407 users by as much as 240ms, which is noticeable. Moreover, compared to the process-first policy, the lazy policy provides better response times to all but one user (604ms), which is also noticeable. By extension, the transmit-first and concurrent policies also offer noticeably better remote response time to all but one user compared to the process-first policy. Second, the local response times of the lazy policy are noticeably better than those of transmit-first and not noticeably worse than those of process-first (although they are not noticeably better than the concurrent local response times, they are in other simulations which we do have room to show). Third, for five users, the lazy remote response times are noticeably better (158ms) than those of transmit-first and concurrent. It turns out that four of these five users are one of the forwarders. Therefore, compared to the transmit-first and concurrent policies, the lazy policy is more fair to users whose computers are doing most of the work.

Response Time Differences (ms)		Transmit-First – Lazy	Process-First – Lazy	Concurrent – Lazy
Max Difference		1.79	9.88	0
Min Difference		69.3	311	28
Num Users for Who the Difference is:	0	0	1198	407
	5	999	0	5
	0	0	0	187
	1194	200	1	0

Table 2. Remote response time (ms) differences between the lazy and concurrent, process-first, transmit-first policies.

Lazy	Transmit-First	Process-First	Concurrent
9.88	78.9	9.88	59.7

Table 3. Local response times (ms) of lazy process-first, transmit-first, and concurrent policies.

The simulation results show that scheduling policies impact response times in at least some practical scenarios. Thus, depending on the total order function, the scheduler can choose the scheduling policy that provides the best response times. For instance, if the total order function chooses the system that provides the best absolute local response times, the scheduler would choose the process-first scheduling policy. However, if the function chooses the system that provides the best remote response times, the scheduler would choose the transmit-first (or concurrent) the concurrent scheduling policy. Finally, if the function chooses a system that provides the best remote response times without increasing the local response times more than what is noticeable, our scheduler would use the lazy policy.

Validating Simulations

As mentioned above, it is important to verify that the simulations are accurate. The validation requires running experiments and simulations for the same scenario and comparing the results. Unfortunately, as mentioned above, we cannot run experiments for large scenarios. This was the reason to use simulations in the first place. It seems that this is "a chicken and an egg" problem. Fortunately, even with as few as ten machines that we have, we can run some limited experiments. To do so, we use a virtualization-like approach in which we treat each user's computer as a virtual computer that is mapped to one of the physical computers. One physical computer may have multiple virtual computers mapped to it. In fact, in experiments involving hundreds of users, there can be a large number of virtual computers mapped to a single physical computer since we have only ten of them. Using the limited experiments made possible with the virtualization approach, we were able to compare the simulated and actual performance for on both single-core and dual-core machines. We found that in all cases, the scheduling policy chosen by the scheduler was also the one that best satisfied the user's response time requirements.

DISCUSSION

The evaluation of our system shows its practical benefits in realistic scenarios. It proves that for some applications, the self-optimizing scheduler is able to meet response time

Response Time Differences (ms)		Transmit-First – Parallel	Process-First – Parallel	Lazy – Parallel
Max Difference		158	604	158
Min Difference		-240	36	-240
Num Users for Who the Difference is:	407	0	407	1198
	5	598	5	0
	187	0	187	0
	0	1	0	1

Table 4. Remote response time (ms) differences between the parallel and concurrent, process-first, transmit-first policies.

Parallel	Transmit-First	Process-First	Lazy
107	177	58	118

Table 5. Local response times (ms) of parallel, process-first, transmit-first, and lazy policies.

requirements better than existing systems. We do not claim that it does so for all applications. Developing such a framework and for all types of applications is difficult. The reason is that any framework designed to optimize other systems must be able to handle all salient features of these systems. Therefore, such a framework must be more complex than the systems it reconfigures. Since the design space of collaborative applications is complex, it is beyond the scope of a single work to create an optimizer for all of them. A practical alternative is to focus an important subset of applications, which is the approach we took.

Scope

Our work is motivated by three driving problems: a distributed PowerPoint presentation; a collaborative Checkers computer game; and instant messaging. They are important examples of real collaborative scenarios. Distributed presentations are becoming common, instant messaging is pervasive, and collaborative games, such as checkers, chess, and online poker, are extremely popular. In fact, by itself, distributed presentations are an important scenario as an entire industry has been created around them.

Beyond the Scope

The instances of target scenario applications that were available to us provided only coupling. In general, applications may have other functionality. While our model accounts only for coupling tasks, it can still be useful for predicting performance of application that have other tasks.

Some applications, called thick-client systems [5], support local-only (or syntactic sugar) commands. An example is a game that lets users modify non-shared aspects of the game world. Our model can indirectly account for local-only commands as the cost of executing them is reflected in the cost of processing the shared commands. For example, if a user is entering more local-only commands than other users, then it appears to the model that the user has a computer that is slower than that of other computers. In addition, the model also handles awareness commands. From a general framework perspective, these commands are indistinguishable from shared user commands. By extension, the self-optimizing scheduler works for applications with awareness and local-only commands.

Our work also accounts for some consistency maintenance schemes. For instance, operation transforms mechanisms may transform an incoming command to maintain consistency across computers. Our work can account for the time required for the transform by including that time in the processing time of the command.

Our work can also indirectly apply to systems that transmit commands in batches even though our model assumes no batching. Two minor adjustments are required. First, transmission time is assigned to be equal to 1) zero for a command whose transmission is delayed because of batching and 2) the sum of transmission times of all batched commands for a command whose input triggers the batch transmission. Second, the remote response time of a command is adjusted by adding to it the time its transmission was delayed by the batching algorithm.

Our model, however, does not account for concurrency control commands. The reason is that these mechanisms impose their own scheduling schemes. For instance, some schemes delay processing of a command until they can ensure that the command does not conflict with other commands. Others schemes handle a conflict after it occurs by undoing and then re-execute commands. These extra scheduling steps do not necessarily imply that our system will hurt performance when concurrency control mechanisms are in place. Further study is needed to determine what actually happens. Moreover, if it turns out that our system degrades performance of conflicting commands, the degradation has to be weighed against the improved performance of non-conflicting commands. This is especially important given that majority of commands do not conflict. In fact, in many scenarios, conflicts do not occur at all. For instance, they did not happen in the collaborations we logged. Also, in some scenarios, users do not care if conflicts occur or they rely on social protocols prevent them [6], so there is no need to handle them.

Future Applicability

Poor performance of collaborative systems today does not necessarily imply poor performance tomorrow. An important question is whether the self-optimizing scheduler will be useful in the future. For instance, it may appear that the choice of scheduling policy will not matter as all devices are becoming multi-core, in which case we the parallel policy can always be used. However, multiple cores are less power efficient which causes problems on mobile devices. As proof, consider the latest mobile devices, such as Apple's iPhone 4 and iPad or Motorola's Droid X. They have single-core processors. Moreover, even if a device has multiple-cores, using one core may be sufficient to perform collaboration tasks, leaving the other cores free to perform other tasks. Our scheduler will choose a single-core policy in this case. Thus, choice of single-core scheduling policies will continue to matter.

Noticeable Response Time Differences

When choosing the optimal policy, we use 50ms as a noticeable threshold because prior work has shown that it is noticeable. Also, it is the only noticeable threshold that has been reported. However, none of the prior work that studied these thresholds used distributed PowerPoint, distributed Checkers, or Instant Messaging. Hence, the results may not apply to these applications. More studies are needed to resolve this issue. For now, we are operating on the principle that when all else is equal, a system with response times that are 50ms better than those of others is superior.

There is no guarantee that a noticeable improvement is always possible. Specifically, scheduling policy changes offer performance benefits when network and processor

resources are somewhat stressed. An important question is whether these conditions exist in all scenarios. While we expect that they do arise, further analysis of network and processor bottlenecks in real collaborative sessions is required to determine whether they actually occur.

To the Developers

We are not aware of exiting tools that improve response times by changing scheduling policies. Nevertheless, developers can still benefit from our work. They can use timing loops to measure all parameters in the model and use the model to predict response times for scheduling policies we studied. Then, they can enforce the policy that gives optimal performance by manually setting thread core/CPU affinities and controlling the order in which tasks execute.

CONCLUSION

This paper makes several contributions. At the highest level, it shows that it is possible to develop a self-optimizing scheduler for collaborative systems. It presents an analytical model that can be used to guide optimization decisions. The model reflects software and hardware realities better than previous models by accounting for non-blocking communication and multi-core processors. The paper also motivates and presents four scheduling policies that take advantage of parallelism. Three of these are existing policies in which blocking communication is replaced with non-blocking while still using a single processing core. The fourth is a variation of the concurrent non-blocking policy in which two instead of one CPU cores are used. Moreover, it shows that versions of the process-first, transmit-first, and lazy policies that use more than one processing core are not necessary. Finally, it presents new implementation issues that must be tackled in the creation of any self-optimizing systems.

In the future, we plan to extend the design space of applications that can benefit from our scheduler. We will study the interplay of the scheduler and concurrency control scheduling schemes. In addition, we are interested in optimizing performance in virtual worlds, such as Second Life. We also plan to study and evaluate scheduling policies for collaborative systems that use other specialized processors, such as GPUs. Moreover, we plan to continue to study avenues for self-optimization. It would be interesting to implement systems that self-optimize the communication architecture. Also, Wolfe et al. [15] have a system that automatically at runtime switches processing architectures to improve performance. It would be useful to create a single system that optimizes processing and communication architectures, as well as, scheduling policies.

AKNOWLEDGEMENTS

This work was funded in part by an NSERC scholarship, a Microsoft Research fellowship, and NSF grants IIS 0312328, IIS 0712794, and IIS 0810861. Also, the reviewers' comments substantially improved the paper.

REFERENCES

1. Abdelkhalek, A., Bilas, A., and Moshovos, A. Behavior and performance of interactive multi-player game servers. *IEEE ISPASS* (2001), 137-146.
2. Chung, G. Log-based collaboration infrastructure. *Ph.D. Dissertation, UNC Chapel Hill* (2002).
3. Dewan, P. Architectures for collaborative applications. *Trends in Software: Computer Supported Co-operative Work*, (1999). 165-194.
4. Ellis, C.A, Gibbs, S.J., and Rein, G. Groupware: some issues and experiences. *ACM CACM,* 34, 1 (Jan 1991), 39-58.
5. Graham, T.C.N., Phillips, W.G., and Wolfe, C. Quality analysis of distribution architectures for synchronous groupware. *CollaborateCom* (2006), 1-9.
6. Greenberg, S. and Marwood, D. Real time groupware as a distributed system: concurrency control and its effect on the interface. *ACM CSCW* (1994), 207-217.
7. Gutwin, C., Dyck, J., and Burkitt, J. Using cursor prediction to smooth telepointer actions. *ACM GROUP* (2003), 294-301.
8. Gutwin, C., Fedak, C., Watson, M., Dyck, J., and Bell, T. Improving network efficiency in real-time groupware with general message compression. *ACM CSCW* (2006), 119-128.
9. Jay, C., Glencross, M., and Hubbold, R. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM TOCHI,* 14, 2 (Aug 2007), Article 8.
10. Junuzovic, S. and Dewan, P. Multicasting in groupware? *IEEE CollaborateCom* (2007), 168-177.
11. Junuzovic, S. and Dewan, P. Lazy scheduling of processing and transmission tasks in collaborative systems. *ACM GROUP* (2009), 159-168.
12. Junuzovic, S. Towards self-optimizing collaboration systems. *Ph.D. Dissertation, UNC Chapel Hill* (2010).
13. p2pSim: a simulator for peer-to-peer protocols. http://pdos.csail.mit.edu/p2psim/kingdata, Mar 4, 2009.
14. Shneiderman, B. Response time and display rate in human performance with computers. *ACM CSUR, 16, 3* (Sep 1984), 265-285.
15. Wolfe, C., Graham, T.C.N., Phillips, W.G., and Roy, B. Fiia: user-centered development of adaptive groupware systems. *ACM Symposium on Interactive Computing Systems* (2009), 275-284.
16. Youmans, D.M. User requirements for future office workstations with emphasis on preferred response times. *IBM United Kingdom Laboratories* (Sep 1981).
17. Zhang, B., Ng, T.S.E, Nandi, A., Riedi, R., Druschel, P., and Wang, G. Measurement-based analysis, modeling, and synthesis of the internet delay space. *ACM Conference on Internet Measurement* (2006), 85-98.