Towards Self-Optimizing Collaborative Systems

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

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

ABSTRACT
Two important performance metrics in collaborative systems are local and remote response times. Previous analytical and simulation work has shown that these response times depend on three important factors: processing architecture, communication architecture, and scheduling of tasks dictated by these two architectures. We show that it is possible to create a system that improves response times by dynamically adjusting these three system parameters in response to changes to collaboration parameters such as new users joining and network delays changing. We present practical approaches for collecting collaboration parameters, computing multicast overlays, applying analytical models of previous work, preserving coupling semantics during optimizations, and keeping overheads low. Simulations and experiments show that the system improves performance in practical scenarios.

Author Keywords
Performance; response times; optimization; processing architecture; communication architecture; scheduling.

ACM Classification Keywords
C.2.4 [Computer-Communication Networks]: Distributed Systems – Distributed Applications, Client/Server; C.4 [Performance of Systems] Performance Attributes.

General Terms
Performance; Experimentation.

INTRODUCTION
As the landscape of available collaborative applications expands, it is critical for an application to differentiate itself from the rest of the field in a useful fashion. An important differentiation factor for these systems is performance. If an application does not respond to user actions in a timely fashion or quickly notify users of actions of others, users may get frustrated and switch to a different application.

In general, in computer science, the performance of a system is a function of available resources. If resources are abundant, then the system always performs well. On the other hand, if resources are insufficient, then the system never performs well. These two boundary cases bracket the case in which resources are sufficient but scarce, called the window of opportunity [9]. In the window of opportunity, a system can have good performance, but new algorithms and implementations may be necessary to achieve it.

In this paper we focus on the window of opportunity for improving the performance of collaborative systems. We present a new collaborative framework that can take advantage of this opportunity and meet performance criteria better than existing systems without requiring hardware, network, or user-interface changes. Several performance metrics have been identified, such as local [14] and remote [4] response time, throughput [5], bandwidth [7], jitter [6], task completion time [2], and frame rate [16]. While all of them are important, our focus is on response times.

Previous work has shown that response times depend on three important factors: processing architecture, communication architecture, and scheduling of tasks dictated by these two architectures [2,3,10,11]. This work has developed theoretical analytical models and used simulations to validate these models. In this paper, we present a system that keeps track of all three of these factors and dynamically adjusts them to improve response times.

A flavor of the kind of improvements the system can provide is shown in Figure 1. It shows the response times from an actual collaborative session and that they are better with than without the system. In fact, the performance with the system eventually approaches the x-axis, which is the theoretical best performance where response times are zero. More importantly, these improvements are noticeable to users. Human-perception studies by Youmans [17] and Jay

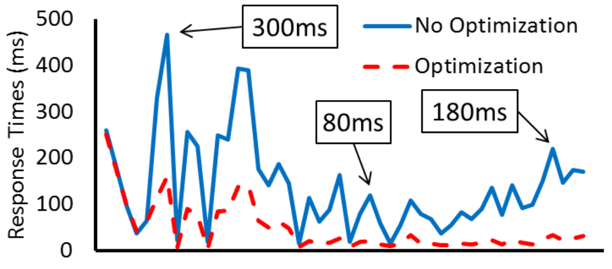

Figure 1. Response times during an actual collaboration session with and without the self-optimizing system

et al. [8] have shown that people can notice changes of 50ms in local and remote response times, respectively. As the response times in Figure 1 show, the system improves the response by as much as 300ms, which is noticeable.

The rest of this paper is organized as follows. First, we discuss background work that motivated the development of our system. We then present an issue based discussion of the implementation of the system. Following this, we present simulations and experiments that evaluate the benefits of the system in practical collaboration scenarios. Then, we discuss practical applications and broader impacts of the system. We end with conclusions and future work.

BACKGROUND WORK

Three important response time factors identified so far are processing architecture, communication architecture, and scheduling of tasks dictated by these two architectures.

Processing Architecture

Two main architectures have been used in the construction of collaborative systems [3]: 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. Each user interface is mapped to exactly one program component.

In the centralized (client-server) 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 (peer-to-peer) 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.

The traditional rule of thumb has been that the replicated architecture provides the best response times. While there are scenarios in which this rule is accurate, Chung [2] was the first to show, both analytically and experimentally, that (a) low network latency actually favors a centralized architecture and (b) asymmetric processing powers actually favor a centralized architecture.

Communication Architecture

Regardless of whether a centralized or the replicated architecture is used, master computers transmit commands to all other computers. If commands are large or the number of users is high, then 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. Junuzovic and Dewan [10] studied the use of multicast in realistic collaboration scenarios. They found that while multicast usually improves response times, it can harm them in some cases.

Scheduling Policy

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, while the communication architecture dictates the destinations to which a computer transmits commands.

The order in which a computer carries out the processing and transmission tasks impacts response times. Four single-core policies for scheduling these tasks have been identified by Junuzovic and Dewan [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, 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. As a result, a part of the transmission task can run earlier, thereby noticeably improving remote response times of some users.

On multi-core devices, intuitively the best response times will be obtained by executing the transmission and processing tasks in parallel on as many cores as possible. However, Junuzovic and Dewan [11] argue against this intuition. Specifically, they state that neither task should be scheduled on more than one core. Their reason for scheduling the processing task on one core is because in general the processing task is an application defined black box that cannot be parallelized by general frameworks. The reason for doing so with the transmission task is two-fold. First, using multiple cores for the transmission task makes it difficult to predict response times because the operating system can schedule the parallel send calls in an arbitrary order. Second, since the CPU is much faster than the network card, a single core saturates the network card. Thus there are no performance gains to using multiple cores to perform the transmission task.

They simulated performance with these policies in realistic scenarios. On single-core devices, they found that the (a) the lazy policy dominates the process-first policy and (b) none of the lazy, transmit-first, and process-first policies dominate each other. On multi-core devices, they found that (a) the parallel policy dominates all single-core policies and (b) using one core to perform both tasks can give performance that is equivalent to that of the parallel policy.

Summary

In summary, previous work has studied the impact of processing architecture, multicast, and scheduling policy on response times. Prior work shows that the combination of these three factors that provides the best response times depends on collaboration conditions. As these conditions can differ within and across collaborative sessions, it would be useful to develop a system that automatically optimizes the factors. In this paper, we present the first such system.

SYSTEM

In this first cut at the design and evaluation of a system that improves performance by automatically maintaining the processing architecture, communication architecture, and scheduling policy, we focused on 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.

The self-optimizing framework both shares applications and improves their performance. These two responsibilities are carried out by the sharing and optimization sub-systems.

Sharing Sub-System

The sharing sub-system is a reimplementation of Chung's approach [2]. In Chung's work, a part of the system, which we refer to as the client component, is logically situated between the user-interface and program components on each computer. The application is not aware of the system component: to the user interface, it appears to be the program component, and to the program component, it appears to be the user interface. The client components communicate directly with each other. They can be setup to enforce replicated or centralized architectures. In our system, we modified these components to also support multicast. We also modified them to support scheduling policy enforcement. When they intercept a command, they create separate threads for processing and transmitting the command. By controlling when and on what core these threads execute, they can enforce any scheduling policy.

Optimization Sub-System

While modifying the sharing aspects of Chung's system is a contribution, our main contribution is the optimization sub-system and the implementation issues it raises.

Driving Optimization Decisions

The main question with any optimization system is how it derives optimization decisions. In our case, the question translates to how it predicts the combination of response time parameters that give the best response times.

One approach is to use learned rules of thumb. Wolfe et al. [16] developed a system called Fiia that uses rules of thumb along with developer hints to automatically centralize or replicate parts of the application at runtime in order to improve response times and frame rates. As mentioned earlier, however, an issue with using rules of thumb is that they do not always accurately predict performance. For instance, Chung showed that there exist collaboration conditions that favor a centralized architecture for good response times, which contrasts the rule of thumb that the replicated architecture provides the best response times.

An alternative approach that always accurately predicts performance is to use an analytical model. Ideally, an analytical model should predict the impact on performance of processing architecture, communication architecture, and scheduling policy since these have been shown to be response time factors. It should also support an arbitrary number of users and take advantage of the latest hardware and software trends such as multi-core CPUs and non-blocking communication. The only model of which we are aware that satisfies all of these requirements was presented by Junuzovic and Dewan [11], which predicts response times in centralized and replicated architectures with an arbitrary number of users. It supports think times, multicast, multi-core processors, and non-blocking communication.

An analytical model usually has a set of assumptions that define the scenarios in which the model applies. Junuzovic and Dewan [11] make only one assumption: an application implements only mandatory coupling functionality and no optional functionality, such as awareness and concurrency control. This assumption is satisfied by the applications that we target, namely, PowerPoint, Checkers, and IM. In general, however, applications implement optional functionality. We will return to the issue of optimizing these applications in more detail in the discussion section.

To apply the Junuzovic and Dewan [11] model, one must first collect values for all of its parameters. Moreover, while the model predicts performance with multicast, it does not actually build a multicast overlay. Therefore, one must also compute a multicast overlay before invoking the model.

Collecting Values of Collaboration Parameters

The parameters include for each computer the processing and transmission times of input and output commands, the think times, and network latencies to all other computers. While most of these parameters are self-explanatory, the transmission times warrant a second look. The transmission of a command is done in two steps: first, the CPU queues the command for transmission by the network card, and then the network card transmits the command. Thus, the time required for both steps needs to be collected.

Parameter values are measured by a client component of the optimization sub-system running on each computer. These components record the processing and transmission times of each command, as well as its think time if the local user entered it. They also measure network latencies to other computers, as well as, report the performance data.

Once parameters are collected during a session, it pays to save them as this can reduce the time the system takes to perform the first optimization in future sessions. For instance, processing and transmission costs measured for a computer in one session can be used as cost estimates for that computer in future sessions. Further reuse is possible by grouping and saving these costs based on processor type. Think times can also be reused by grouping them on a user or application basis. With the reuse of values of these parameters, the system can perform an optimization in a future session as soon as it collects network latencies. Such reuse is also useful when latecomers arrive. Specifically, if costs have been gathered for the latecomer's computer type, then the system can optimize response times as soon as it receives latencies among the latecomer and other users.

Computing Multicast Overlays

The idea of multicast requires the construction, for each source of messages, a multicast overlay that defines the paths a message takes to reach its destinations. 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. In this first-cut at a self-optimization system that uses multicast, we did not want to develop a new multicast scheme. Instead, we chose an existing application-layer scheme: the HMDM algorithm presented by Brosh and Shavitt [1]. It is the only algorithm of which we are aware that accounts for application-layer transmission costs.

Applying a multicast algorithm requires the collection of its parameters. The HMDM scheme parameters include network latencies and transmission costs. Since these parameters are also collected for the analytical model, no additional data needs to be collected by our system in order to build a multicast tree using the HMDM scheme.

An important issue is how many multicast trees are computed. One option is to compute a multicast tree rooted at every user, but HMDM's $O(number\ of\ users^3)$ runtime makes this approach impractical in large scenarios. A more practical alternative is to compute a single multicast tree that is shared by all users. An issue with this option is that a command from any user but the one who is at the root of the tree must first reach the tree, which increases the response times. Our system can be configured to create an arbitrary number of trees. The default is one.

Applying the Analytical Model

Once a multicast overlay is created, the system can use the analytical model to predict performance. Unfortunately, it may not be able to predict performance for all combinations of response time factors using only the reported values. The reason is that some parameter values may be missing because not all computers report values of all parameters at all times. For example, in a centralized architecture, a slave does not report input processing costs, so the collected values are not sufficient for predicting the performance of an architecture in which the slave is a master.

When parameter values are missing, it is possible to estimate them using the values of other parameters. One approach is to assume that the input and output processing cost ratio is the same for all computers. Since there is always a master computer, the ratio can be computed. Consider the missing input processing cost of a slave. A master's processing cost ratio and the slave's output processing cost can be used to estimate the missing value. Other parameters can be estimated similarly.

Estimated parameter values may not reflect the actual parameter values, however, so they may result in optimizations that degrade performance. Such degradations are temporary. As the optimization system gathers data, it will eventually discard estimated values. In addition, by reusing computer costs across sessions, eventually, it will know most if not all parameter values at session start time.

While the analytical model is able to predict performance for different configurations of a system, it does not predict which configuration gives the best performance. The simplest approach for picking a configuration is to pick the one with the best average response times. However, this approach does not account for the fact that response times are inherently partially ordered and external criteria must be used to create a total order. In general, infinitely many external criteria exist and their application depends on the response time requirements. To satisfy the requirements, we rely on the notion of a response time function introduced by Junuzovic and Dewan [11], which is an expression of the requirements. Their function distinguishes between primary and secondary users and between local and remote response times. It accepts response times of configurations, a list of inputting users, and identities of all users as parameters and returns a ranking of configurations from best to worst.

Preserving Coupling Semantics During Optimizations

Once the system begins to perform an optimization, it is important to ensure that the switch is performed atomically from the perspective of commands already in the system or those entered during the change. Otherwise, a command may not reach all of its intended destinations or it may reach a destination multiple times. For example, during processing architecture change, a computer that changed from a master to slave may receive an input command, which would be inconsistent with the notion of replicated and centralized architectures. Also, during a communication architecture change, a command may reach a destination multiple times since a computer may forward commands to different destinations in the old and the new architectures.

In some cases, an optimization may be performed when the shared application is in a quiescent state, in which case distribution semantics are not impacted by the change. For example, a switch may happen during think times if they are high. Alternatively, the system could ensure a quiescent state by delaying an optimization until users take a break.

In general, however, an optimization may be triggered when the shared application is in a non-quiescent state. One approach to performing the optimization atomically is to suspend user inputs during the switch and wait for the distribution of commands already in the system to complete before starting the optimization. An issue with this solution is that response times of commands can be high if the switch time is long. A more intelligent solution is to run the old and new configurations in parallel and switch to the new configuration when it is fully deployed, which is done by Chung for processing architecture changes [2]. As this approach has been implemented previously for processing architecture changes, we did not duplicate it in our system. However, as discussed below, our system demonstrates that the general idea of concurrently running two architectures can be applied to other reconfiguration steps. Specifically, a compromise taken by our system is to break the optimization into three sequential atomic sub-steps, one for each response time factor, and suspend user inputs only during the processing architecture change. Since the processing architecture defines which computers process inputs, it must be deployed before the communication architecture as the latter dictates how inputs are distributed. The scheduling policy can be changed last as it is independent of the two architectures.

Our system first changes the processing architecture by reconfiguring the client components of the sharing system. To bring program components up to date on computers changing from a slave to a master, it uses Chung's solution [2] of replaying input commands to these components. To support the replay, the system logs input commands on master computers. It picks one old master at random to replay commands to new masters. During the change, the system suspends user inputs.

One issue when deploying a new processing architecture is deciding what communication architecture and scheduling policy should be used until they are also changed. In particular, since the new processing architecture redefines which users process input commands, a previous multicast architecture may distribute input commands to new slave computers, which is inconsistent with the notion of collaboration architectures. Therefore, when changing the processing architecture, the system changes the communication architecture to unicast. The system must also choose a scheduling policy to use. It simply keeps the old scheduling policy since, as mentioned above, task scheduling is independent of the processing architecture.

Once the new processing architecture is deployed, the system resumes user inputs and begins the communication architecture change. During this step, the system changes the communication pattern among the client components. To handle user inputs, the system continues to distribute them using the old architecture while it deploys the new one. The new communication architecture is activated only once it has been deployed on all computers, and it is immediately used for all new commands.

An important aspect of the system is that it does not discard the old architecture immediately after activating the new one. The reason is that when the new architecture is activated, there may be commands that have been only partially distributed using the old architecture. Thus, the old architecture continues to distribute these commands. To help each other decide which architecture to use, source computers tag each command with the version number of the architecture that should be used to distribute it. The net effect of using the old architecture for commands entered before the new architecture is activated is that from the perspective of commands, the change is atomic because a command is distributed using only the old or only the new architecture. Eventually, the old architecture can be torn down. To ensure that it is not removed prematurely, the tear down should be delayed by the maximum response time. In our experience, a delay of several minutes is sufficient.

Finally, the system performs the scheduling policy change. A scheduling policy change is always atomic from the perspective of user commands because scheduling policies do not determine the distribution of commands. However, using a mix of old and new policies may lead to performance degradations because the performance was not predicted for any mix of policies. Fortunately, it is a temporary degradation; once the computers switch to the new policy, the performance will improve as predicted.

Minimizing Overhead Impact on Performance

The optimization steps described above require both CPU and network resources. Therefore, care must be taken so that they do not negatively impact performance.

Two of these steps have to be executed by the client components on the users' computers. Only these components can accurately measure the model parameters, and only they can reconfigure themselves when an optimization is performed. The data collection and reporting overheads can negatively impact the response times of the local user, although in our experiments they were insignificant. If these overheads are an issue, they can be reduced by increasing (a) the number of commands a client waits before reporting data, (b) reducing the network latency polling frequency of a client, and (c) reducing the number of destinations in each latency poll.

The remaining steps are particularly CPU intensive. One reason is that they involve the execution of the multicast algorithm, which requires heavy computation when there are many users. Fortunately, these steps do not need to be executed by the client components. We encapsulate them into a component called the server component. An important issue is the location of the server. We do not centralize it on a user machine for fear of degrading the user's response times. An alternative is to distribute the server among multiple machines. While this decentralized approach reduces the performance impact on any single user, such approaches in general suffer from a distributed consensus problem. To avoid the consensus issue, we centralize the server, but on an infrastructure (non-user) computer, such as the one running the session manager. Such a computer exists even in the most highly distributed systems, and is often called the bootstrapping node [15]. An issue with this approach is that computing an optimization may take a long time when there are many concurrent sessions. Fortunately, this issue is temporary, as the system eventually finishes the computation. More importantly, it does not degrade response times during this time.

EVALUATION

So far, we have presented the self-optimizing system. An important question is whether it has practical benefits.

Experiments vs. Simulations

In computer science, it is common that the performance of a system is evaluated through experiments and simulations. Experiments measure the performance of a system in use, while simulations estimate the performance of a system by using an analytical model of the system. Simulations are more than a pure theoretical evaluation, however. While theoretical evaluations can give trends and implications, simulations can also provide quantitative results.

The choice between experiments and simulations involves a tradeoff. Although experimental results are arguably more believable than simulated results, simulations generally require fewer resources than experiments making them easier to run. In fact, when large scale experiments are not possible because of a lack of resources, simulations may still be possible. In this case, one way of reducing the "believability gap" is to validate subparts of the simulation with small-scale experiments.

Whenever possible, we performed experiments instead of simulations. Unfortunately, in our lab, we do not have a sufficient number of machines to perform large experiments – we only have ten. Although public clusters, such as PlanetLab and Amazon's EC2, provide access to a large number of machines, they do not offer sufficient control needed for performance experiments. For instance, there is no way to ensure that the same set of machines is always used. In addition, in PlanetLab, machine loads can vary across experiments because users share machines. Thus, for our large scale scenarios, we had to use simulations.

When we performed large-scale simulations, we also carried out smaller-scale experiments that were possible on our equipment to validate subparts of the simulations. We used a virtualization approach in which we treat each user's computer as a virtual computer that is mapped to a physical computer. One physical computer may have multiple virtual computers mapped to it. We added the virtualization functionality to our framework. It supports mapping up to one hundred users onto a single computer before memory becomes an issue. The performance data for users who are not mapped to a dedicated physical computer must be discarded because when multiple users are mapped to a single physical computer, timing measurements for them and thus any users downstream from them are not reliable.

Processing Architecture Automation Experiments

We used experiments to study the impact on performance of processing architecture changes in practical scenarios. To obtain realistic user commands and think times, we logged a collaborative Checkers game in which users play together against the computer. We chose this program because it is a computer-intensive task and its transmission costs are low, allowing us to validate the effect of processing time differences. In the experiments we conducted to gather these logs, two users played together against the computer and both users made Checkers moves. We assumed that the data in the logs is independent of the number of collaborators, their devices, and network latencies.

We used three computers, a Core2 2.0GHz desktop, a P4 1.7GHz desktop, and a P3 500MHz desktop, which have processing power differences that can be expected when users collaborate. We use the P3 desktop to simulate next-generation mobile devices and current generation netbooks. The computers are connected on a local LAN. Thus, the latencies between them were low (i.e., ~0ms).

To replay the logs, we added functionality to our system that enables us to replay previously recorded logs. 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 report the average performances.

We performed the following three-user experiment. Initially, two users are playing Checkers. User$_1$ is using the P4 and user$_2$ the P3 desktop. Suppose that the users are on the same LAN. Suppose also that after fifteen turns, user$_3$ joins using a Core2 desktop on the same LAN. We performed two sets of three experiments. In both sets, we used all three possible initial architectures: replicated, centralized on user$_1$'s P4 desktop, and centralized on user$_2$'s P3 desktop. In the first set of experiments, the optimization system was disabled so the architecture did not change during the session, while in the second set, it was enabled.

We did not provide any historical performance data to the optimization system; it collected all data from reports sent during the session. We configured (a) the client components to send reports after each command, (b) the clients to poll for network latencies every sixty seconds, and (c) the server component to perform optimizations every five commands. Finally, we used a total order function that ranks one system to be better than another if the former gives better response times to more users than the latter.

Architecture		$User_1$						$User_2$						$User_3$					
		1-5		6-20		21-48		1-5		6-20		21-48		1-5		6-20		21-48	
Initial	Opt?	Avg	Stdev	Avg	Stdev	Avg	Stdev	Avg	Stdev	Avg	Stdev	Avg	Stdev	Avg	Stdev	Avg	Stdev	Avg	Stdev
Cent on $User_1$	No	44.3	34.2	74.0	48.7	33.1	16.0	67.1	27.8	88.4	46.6	48.8	15.9			61.8	43.9	34.1	16.2
	Yes	44.8	33.8	75.4	48.9	19.2	6.25	64.0	30.8	87.4	47.7	29.6	7.16			63.0	46.6	16.9	6.42
Cent on $User_2$	No	128	91.3	217	139	98.9	49.3	131	89.0	221	139	104	49.3			174	121	98.1	49.1
	Yes	126	86.1	75.6	48.9	18.5	7.06	130	85.8	87.8	46.5	29.9	7.03			62.0	45.0	16.7	7.19
Rep	No	43.8	33.8	74.7	48.9	34.0	16.2	135	88.9	222	138	107	49.6			26.7	17.0	15.9	6.13
	Yes	43.9	33.6	75.3	48.4	19.4	6.45	133	86.3	88.9	46.2	29.1	6.64			62.6	44.3	17.1	6.73

Table 1. Response times (ms) measured during the Checkers architecture optimization experiment for commands before the first optimization (1-5), after the first but before the second optimization (6-20), and after the second optimization (21-48)

We expect two optimizations. We expect the first one to occur five moves into the game as the system gathers sufficient data by then. It should result in a switch to the centralized architecture in which the P4 is the master. The reason is that the P4 is more powerful than the P3. We expect the other optimization to happen after the twentieth move as this is five moves after $user_3$ joins, which means that sufficient data has been collected for $user_3$'s computer. It should result in a switch to the centralized architecture in which the Core2 is the master because it is the most powerful computer in the session. Specifically, we also expect that when the $user_3$ joins, the arrival does not trigger an optimization as there is no data for $user_3$'s computer. It should join as a slave of $user_1$. Since we expect optimizations after the fifth and twentieth commands, we present the average response times for commands one to five, six to twenty, and twenty-one to forty-eight (the end).

As Table 1 shows, in all cases except one, the performance is either better or no worse with than without optimization. As mentioned above, we consider a change of 50ms in response times significant. The highlighted cells in the table show noticeable improvements. Consider $user_2$'s response times. When the replicated architecture is used initially, the $user_2$'s average response times are improved by 133ms and 77.9ms after the first and second optimization, respectively.

The only case in which performance is worse with than without the self-optimizing system occurs for $user_3$ for commands six through twenty when the replicated architecture is used initially. This is actually expected. The reason is when the optimization system is running, by the time $user_3$ joins, the system had switched to the centralized architecture in which $user_1$'s computer is the master. Thus, when $user_3$ joins, $user_3$'s computer joins as a slave of $user_1$'s computer. On the other hand, when the system is not running, the architecture had not changed from replicated to centralized before $user_3$ joined. Thus, $user_3$'s computer would have joined as a master. Since $user_3$ has a more powerful computer than $user_1$, the architecture in which it is the master will offer better performance to $user_3$.

As described earlier, user inputs are paused during a processing architecture change. In our experiments, the pause did not negatively impact performance. The reason is that the maximum switch time, 360ms, was less than the minimum think time, 2134ms, so a switch could only happen either during think time or overlap with at most one command. The former was always true in our experiments.

System overheads also had no impact on response times. Such an impact would be betrayed in the response times for the first five commands. The reason is that in the case when our system is running, it does not change the architecture during the first five commands but it is still collecting data. As Table 1 shows, the response times for the first five commands with and without our system are the same.

Communication Architecture Automation Experiments

As mentioned above, the choice of unicast or multicast communication is important when the cost of transmitting commands is high. Thus, to study the effects of multicast on response times, we must consider large scale scenarios. Because large scale experiments were infeasible for us, we relied on simulations to evaluate the benefits of our system in large scale scenarios. Since simulations use an analytical model of a system to predict the performance of the system, the analytical model used in our simulations is the same as the model used by the self-optimizing system.

As was the case with experiments, we used realistic simulation data in the simulations. We considered a PowerPoint scenario in which the presentation is being given to 100, 200, 300, 400, and 500 audience members around the world. PowerPoint is a good choice of application for two reasons. It is perhaps a popular business collaborative application. Also, its transmission costs can be high, and thus multicast could help with its performance.

To obtain realistic PowerPoint commands and think times, 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 assumed that the data in the logs is independent of the number of collaborators, their devices, and network latencies.

To obtain the processing and transmission costs, we ran small scale distributed PowerPoint sessions with our system. We configured to system to just collect parameters without making optimization decisions. We then replayed the logs. To get costs for different machines, we repeated the procedure with four different computers: Core2 2.0GHz desktop; P4 1.7GHz desktop; P3 500MHz desktop; and 1.6GHz Atom netbook. As with the Checkers experiment, we removed all active processes that we could from the computers in order to reduce noise in the measurements.

Based on the published network latency data between 1740 computers [12], we set the network latencies between all users equal to those between a random subset of 100, 200, 300, 400, and 500 of the 1740 computers. Zhang et al. [18] showed that such subsets are representative of the entire set.

To create a multicast tree, we ran the HMDM algorithm with the latencies and measured transmission cost values.

Finally, we simulated the performance for a scenario in which (a) the centralized architecture is used, (b) the transmit-first policy is in effect, (b) the presenter is using a netbook, (c) the remaining users are using a random mix of netbooks and P3, P4, and Core2 desktops, and (d) a total order function that prioritizes maximum remote response times is used. We configured the system to use historical data and existing latencies instead of gathering them dynamically. We also configured it to begin performing the first optimization once the first command is replayed. We performed ten simulations and report the average results.

Table 2 shows the maximum remote response times as the number of users varied. The maximum remote response times increased much faster with unicast than with multicast as the number of users grew. As the number of users increased from 100 to 500, the maximum unicast remote response time grew by 5104ms, while the multicast remote response times grew only by 54.31ms. Also, the remote response times are noticeably better with multicast than with unicast for all sizes of collaborations: 1052ms better with 100 and 6102ms better with 500 users. The local response times are shown in Table 3. As Table 3 shows, with unicast, the local response time increased linearly with the number of users (427.6ms), while with multicast, it increased slightly (2.03ms). More importantly, local response times were significantly better with multicast than with unicast for all sizes of collaborations: 82.41ms better with 100 and 509.0ms better with 500 users. We did not find high maximum remote response time variability. The randomness in the simulations came mainly from randomly assigning realistic computer types to users and latencies among these computers, which mainly impact the multicast algorithm. The HMDM scheme was able to consistently navigate this randomness, which is a tribute to its design.

Num Users	100	200	300	400	500
Unicast	1651	2923	4227	5510	6756
Multicast	599.3	615.4	640.5	647.5	653.7

Table 2. Average maximum remote response times (ms)

Num Users	100	200	300	400	500
Unicast	164.2	271.2	378.1	485.0	591.9
Multicast	80.85	82.00	82.16	82.94	82.88

Table 3. Average local response times (ms)

The simulation results show that the system can noticeably improve performance through multicast. Whether the system would actually deploy multicast depends on the response time function. If the function favors configurations that minimize the maximum local or remote response times, then the system would deploy multicast. Also, although multicast appears to always be better than unicast, unicast is still useful. Unicast is easy to deploy and maintain while multicast requires computation for both. Thus, unicast can be the default option that is overridden by multicast only when it is noticeably worse than multicast.

Through limited experiments made possible by our virtualization approach described earlier, we compared the simulated and experimental values for a scenario with 2 and 100 users. We found that in all but one case, the simulated value was within 10ms of the measured value. The case it was not within 10ms was the maximum remote response time in the 100 user scenario, where it was 79.1ms lower. However, the measured value was 1415ms. Thus, the error, though higher than the noticeable threshold, was less than 6%. Even with the error, the response time trends predicted in the simulations were observed in the experiments.

Scheduling Policy Automation Experiments

We also measured the impact on response times of automating the scheduling policy maintenance. Because of space restrictions, we only give a summary of the results.

We wanted to verify that our system choses the same scheduling policies as those predicted by Junuzovic and Dewan simulations [11]. They found that on single-core devices, the lazy, transmit-first, and concurrent policies do not dominate each other. Hence, the one that optimizes response times depends on the users' response time requirements. They also show that on multi-core devices, the parallel policy dominates all single-core policies, although in some cases using a single-core policy on a single-core can give performance that is equivalent to that when using multiple cores with the parallel policy.

As for the communication architecture results, we used the simulation-and-validation approach. We showed that on single-core devices, our system chooses the scheduling policy that best meets response time requirements. In particular, if the response time function favors a system that (a) provides the best local response times, our system deploys the lazy policy, (b) provides the best remote response times to as many users as possible, our system switches to the transmit-first policy, and (c) improves as many remote response times as possible without noticeably degrading the local response times, the system deploys the lazy policy. In all simulations, the remote response time requirements would have been satisfied noticeably worse with other policies. Moreover, we show that on multi-core devices, the system uses the parallel policy only if it gives significantly better performance than all of the single-core policies. These results confirm simulations from prior work.

DISCUSSION

In this section, we discuss the broader impact and issues with the adoption of the self-optimization system.

Beyond the Scope

It is important to note that the limitations of the model used by our system are not limitations of the system. The implementation issues we presented are orthogonal to the choice of analytical model. Regardless of the model, it is necessary to gather its parameters, handle nuances in its application, and compute multicast overlays. The remaining issues, namely, the steps required to keep overheads low and preserve coupling semantics, are model independent.

On the other hand, the scope of applications whose performance can be improved by our system is dictated by the scope of applications supported by the analytical model it uses to predict performance. Since the applications in our driving problems had only coupling functionality, it was sufficient for our system to use a model that accounted only for coupling commands. As new performance models are created that relax this assumption, they can be used by our system to expand its application scope. For instance, new models that account for costs of consistency maintenance and conflict resolution could be used in our system to improve performance in scenarios where users generate a high rate of small actions, such as real-time editing systems.

Even with the current model, our system can still be useful for applications that have optional functionality. For instance, World of Warcraft has concurrency control. Thus, our system with the current model may not correctly predict performance when conflicts occur and may even degrade it. However, the model still applies to, and our system will still improve, the performance of, non-conflicting commands. Hence, if our system happens to degrade performance of conflicting commands, the degradation has to be weighed against the improved performance of non-conflicting commands. This is important given that the majority of commands do not conflict. In fact, sometimes conflicts do not occur at all. For instance, they did not happen in the collaborations we logged. Also, in some scenarios, social protocols prevent conflicts or users do not care about them.

In the scenarios we tested, users took turns to play a game and broadcasted a presentation. Ideally, we should also evaluate the system for scenarios with co-authoring actions, such as co-creating a presentation. Unfortunately, such logs are not publicly available, and we did not have an opportunity to observe and log any such sessions ourselves.

Immediate Applicability

An important question is whether performance of collaborative systems is an issue today. Consider Citrix GoToMeeting, Cisco Webex, and Microsoft Live Meeting, three of the most popular collaborative systems. Websites for all three products contain instructions on how to improve performance. Moreover, numerous unofficial websites offer help to users who are suffering from poor mouse and keyboard response times in these systems. Thus, performance of systems today does indeed matter.

We have shown that a window of opportunity for improving performance exists in some practical scenarios assuming that 50ms is a noticeable threshold. While 50ms is the only noticeable threshold reported by prior work, it was not studied using PowerPoint, Checkers, or Instant Messaging, so it is possible that it may not apply to these applications. More studies are needed to resolve this issue. Also, there is no guarantee that a noticeable improvement is always possible. We have shown that it is possible when resources are somewhat stressed. An important question is whether these conditions occur in practice.

Even if a noticeable performance improvement is possible, an important question is whether the complexity of the optimization system is worth it. As with any complex system, it has development and deployment costs, but it is not all or nothing. The components for optimizing each of the three factors are logically independent, and it is possible to implement some but not all of them to get some but not all of the benefits. The amount of implementation effort is proportional to the benefits it provides. For instance, applications typically support either centralized or replicated semantics. To get the full benefit of our system, they can be updated to support both architectures, or they stay unchanged but still get the benefit of our system's communication architecture and scheduling policy automation. This is consistent with the philosophy in successful commercial software, which provides features that are not used all the time. Also, in our case, the extra functionality does not have a performance overhead, as we see above, and unlike several commercial systems, no user overhead, as the user-interface is not changed. In general, implementation overhead is less important than performance or user overhead, as it does not deteriorate the end-user experience. Moreover, such a system has to be implemented and tested only once for all applications.

Additional complexity arises from the sharing sub-system, as the component situated between the user-interface and program components has to be implemented for each application. However, Chung [2] has already shown that its implementation is straightforward for applications with well-defined user-interface and program components and whose source code is available. In addition, we were also able to implement it for a black-box application, which, as a result, does not satisfy these requirements – PowerPoint. PowerPoint provides a COM interface. To bridge Java and COM, we used J-Integra, and then we were able to build the client component for PowerPoint. The process took two weeks. The majority of the time was spent on investigating COM-Java bridges. Thus, building the component for other COM applications will take less time. We anticipate similar investments for applications with other types of interfaces.

Future Applicability

An important question is whether our system will be useful in the future. For instance, as processor and network speeds improve, it may seem that processing and transmission costs will decrease. As a result, the choice of processing and communication architectures may not matter. However, historically, increasing processor and network speeds have resulted in more complex applications with increasing resource requirements. Also, on mobile devices, the hardware capabilities are developing less rapidly because of power conservation issues. Thus, architectures will continue to matter. Similarly, as devices become multi-core, it may seem that the choice of scheduling policy will not matter. Multi-core processors, however, are less power efficient, which is an issue for mobile devices. For this reason, many mobile devices still use a single-core processor. Thus, scheduling policies will continue to matter.

CONCLUSION

This paper makes two main contributions. First, it shows that it is possible to develop a self-optimizing system for collaborative applications that uses an analytical model to drive optimizations. It also presents new implementation issues relevant to all future self-optimizing frameworks. In addition, the presented system can be used as an instructional tool for teaching students about collaborative systems. In particular, students can use the system to experience the impact on response times of processing architecture, multicast, and scheduling policy. Currently, we are incorporating it into a graduate collaborative systems course and plan to make it available for download.

It will be useful to (a) extend the design space of applications that can benefit from our system; (b) improve the performance of massive online virtual worlds, such as Second Life; and (c) perform user studies to evaluate the perceived benefits of our system in actual collaborations. It is also important to study multi-pronged solutions to user experience issues caused by high response times. For instance, our work improves user experience by reducing response times. An orthogonal approach, taken by Savery and Graham in the TimeLines model [13], is to adjust processing of commands in a manner that masks them. Neither approach is perfect: ours does not reduce response times to zero and theirs does not perfectly mask them. It would be useful to first reduce responses times with our system and then use TimeLines to mask them as this may give a better user experience than with either system alone. It would also be interesting to add performance parameters that capture human factors as users can work around some performance issues. While functional, these workarounds may prevent optimal use of a system and may open other, potentially hidden, opportunities for improvement.

ACKNOWLEDGEMENTS

This research was funded in part by an NSERC scholarship, a Microsoft Research fellowship, and NSF grants IIS 0712794 and IIS-0810861.

REFERENCES

1. Brosh, E. and Yuval, S. Approximation and heuristic algorithms for minimum-delay application-layer multicast trees. *IEEE INFOCOM*, 2004.
2. Chung, G. Log-based collaboration infrastructure. *Ph.D. Dissertation, UNC Chapel Hill*, 2002.
3. Dewan, P. Architectures for Collaborative Applications. *CSCW: Trends in Software*, 7, 1999, edited by Beaudouin-Lafon, M.
4. Ellis, C.A, Gibbs, S.J., and Rein, G. Groupware: some issues and experiences. *ACM CACM*, 34, 1 (Jan 1991).
5. Graham, T.C.N., Phillips, W.G., and Wolfe, C. Quality analysis of distribution architectures for synchronous groupware. *CollaborateCom*, 2006.
6. Gutwin, C., Dyck, J., and Burkitt, J. Using cursor prediction to smooth telepointer actions. *ACM GROUP*, 2003.
7. 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.
8. 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).
9. Jeffay, K. Issues in multimedia delivery over today's internet. *IEEE Multimedia Systems*. Tutorial. 1998.
10. Junuzovic, S. and Dewan, P. Multicasting in groupware? *IEEE CollaborateCom*, 2007.
11. Junuzovic, S. and Dewan, P. Scheduling in variable-core collaborative systems. *ACM CSCW*, 2011.
12. p2pSim: a simulator for peer-to-peer protocols. http://pdos.csail.mit.edu/p2psim/kingdata. Mar 4, 2009.
13. Savery, C. and Graham, T.C.N. It's about time: confronting latency in the development of groupware systems. *ACM CSCW*, 2011.
14. Shneiderman, B. Response time and display rate in human performance with computers. *ACM CSUR*, 16, 3 (Sep 1984).
15. Wikipedia. Bootstrapping Node, Dec 27, 2009.
16. 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.
17. Youmans, D.M. User requirements for future office workstations with emphasis on preferred response times. *IBM United Kingdom Laboratories* (Sep 1981).
18. 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.